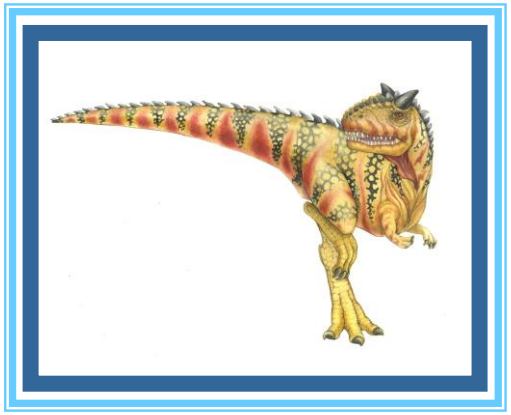
# Chapter 5: Synchronization



# Chapter 5: Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Synchronization Examples

# Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem

# Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

increment : เพิ่มค่าขึ้น
decrement: ลดค่าลง

# Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
         while (count == BUFFER_SIZE)
                        ; // do nothing
              buffer [in] = nextProduced;
              in = (in + 1) % BUFFER_SIZE;
              count++;
}
```

# Consumer

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

                    /*  consume the item in nextConsumed
}
```

# Race Condition

Recall that count is a shared variable

- count++ could be implemented as  statement of the producer.

```
register1 = count          P1
register1 = register1 + 1  P2
count = register1          P3
```

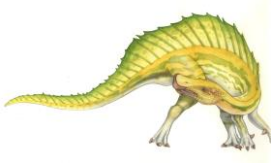
- count-- could be implemented as  Statement of the Consumer

```
register2 = count          C1
register2 = register2 - 1  C2
count = register2          C3
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

interleaving: การแทรกสลับการทำงานของชุดคำสั่ง



# Solution to Critical-Section Problem

A Solution must satisfy the following 3 requirements

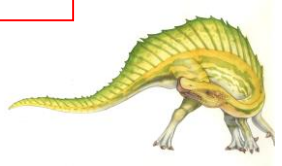
1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections (การห้ามอยู่พร้อมกัน)
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely (มีความก้าวหน้า)
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted (รอคอยอย่างมีขอบเขต)
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Each process may consume different amount of time in execution of it critical section

Each process has a segment of code, called a critical

Critical section: เขตวิกฤต คือพื้นที่ที่process แต่ละตัวสามารถทำการปรับปรุง เปลี่ยนแปลงค่าตัวแปรต่างๆ ของprocess โดยไม่มี process อื่นเข้ามาเกี่ยวข้องในพื้นที่นี้

exist: ยังปรากฏอยู่ , คงอยู่ postponed: ปฏิเสธ
indefinitely : ไม่แน่นอน granted:ได้รับการอนุญาตแล้ว



# Peterson's Solution

- Two process solution (เป็นวิธีที่ใช้กับ 2 process)
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process $P_i$ is ready!

# Algorithm for Process $P_i$

Initialization
boolean flag[i] - flag[i] = flag[j]

มุ่งประเด็นไปที่ ช่วงเวลาหนึ่งมี 2 process เท่านั้น

How can this Peterson's solution meet 3 requirement (p.5.8) of a sulution to the critical

See my note

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

$P_i$ does nothing and wait for $P_j$ compleat หรือการที่ $P_i$ ติดลูป(loop)

i: current process
j: other process

# Synchronization Hardware

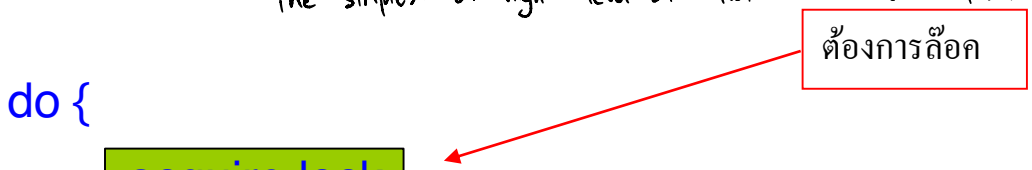
- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Special H/W instruction that allow

Uniprocessor : โปรเซสเซอร์ตัวเดียว

# Solution to Critical-section Problem Using Locks

The simplest of higher-level s/w tool to solve the critical-section problem is the mutex lock.




```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```


# Semaphore

A more robust tool that can behave similary to a mutex lock (p.s.ม)

- Synchronization tool that does not require busy waiting (ไม่ต้องการการรอคอยที่มาก)
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal() การเข้าแบบไม่จำกัดเวลา
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations มี 2 การดำเนินการที่ทำกับSemaphore

  - **wait (S)** {
    ```
    while S <= 0
        ; // no-op
    S--;
    ```
    }

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

Initialization
S >= 1

Do nothing since the critical section is locked

wait to test.

signal to increment.

# Semaphore as General Synchronization Tool

S in [-infinity , + infinity]

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  S in [0,1] only initially
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;   //  initialized to 1
do {
    wait (mutex);
        // Critical Section
    signal (mutex);
        // remainder section
} while (TRUE);
```

***wait* (mutex):**

**while mutex≤ 0 do *no-op*;**
**mutex--;**

wait to test รอไปเรื่อยๆ ถ้าเป็น 0 ถ้า 1 ไป

***signal* (mutex):**

**mutex++;**

signal to increment

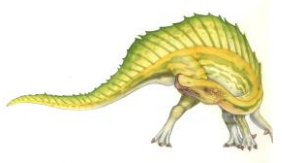
# Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

# Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue. (ให้เข้าไปรอในคิวยังไม่ทำงาน)
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue. (นำออกจากคิวเพื่อรอทำงาน)



# Semaphore Implementation with no Busy waiting (Cont.)

Intialization
S = 1

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

ค่า value อาจติดลบได้แสดงให้เห็นว่ามี process รอคอย semaphore

A linked list is used as a waiting queue of semaphon S.

- Implementation of signal:

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

There's still some process in the waiting queue of semaphon S.

Semaphon S. 1 S2. ตัวอย่างการใช้

# Deadlock and Starvation

การใช้ **semaphore** อาจทำให้เกิดเหตุการณ์ขึ้นได้ ดังนี้

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

รอให้มี process หนึ่งทำคำสั่ง signal ก่อนจึงจะทำสั่ง wait ได้

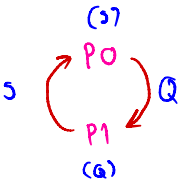
| $P_0$ | $P_1$ |
|---|---|
| wait (S); | wait (Q); |
| wait (Q); | wait (S); |
| . | . |
| . | . |
| . | . |
| signal (S); | signal (Q); |
| signal (Q); | signal (S); |

S = 1 - 1 = 0

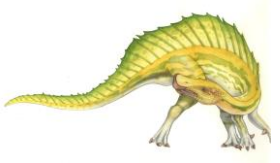
Q = 1 - 1 = 0

Q = 0 Thus, P0 must wait for P1 to release Q

S = 0, Thus, P1 must wait for P0 to release S.

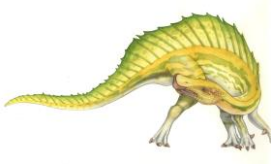



- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process



# Classical Problems of Synchronization

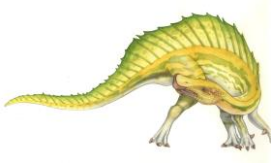
- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem



# Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value N.

Producer & consumer



# Bounded Buffer Problem (Cont.)

wait &

- The structure of the **producer process**

Initialization:
int N = 5
semaphore

```
do {
        // produce an item in nextp
        wait (empty);
        wait (mutex);
        // add the item to the buffer
        signal (mutex);
        signal (full);
} while (TRUE);
```

empty = 5 → 4 → 3 → 2 → 1 → 0

Empty --

The 1st producer lock the execution of critical section (code of buffer updating)
mutex = 1 → 0

mutex = 0 → 1

release the lock
the 1st producer

Full ++

Full = 0 → 1 → 2 → 3 → 4 → 5.

# Bounded Buffer Problem (Cont.)

- The structure of the **consumer process**

```
do {
        wait (full);
        wait (mutex);

            // remove an item from buffer to nextc

        signal (mutex);
        signal (empty);

            // consume the item in nextc

} while (TRUE);
```

full = 5 → 4 → 3 → 2 → 1 → 0

Full --

mutex = 1 → 0

The 1st consumer lock the execution of critical section (code for buffer updating)

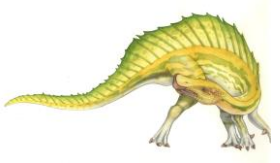
mutex = 0 → 1

The 1st consumer releases lock

Empty ++

Critical section

empty = 0 → 1 → 2 → 3 → 4 → 5



# Readers-Writers Problem

-ใช้ข้อมูลร่วมกัน ผู้อ่านสามารถอ่าน (Reader) ข้อมูลร่วมกันได้หลายๆ คน

-ผู้เขียน (Writer) 1 คน สามารถเขียนข้อมูลได้ ณ ช่วงเวลาหนึ่ง โดยไม่มีผู้เขียนคนอื่นมาใช้ข้อมูลร่วม และห้ามผู้อ่านมาอ่านขณะที่เขียนอยู่

อาจทำให้เกิดปัญหา **Starvation** ได้ทั้ง ฝั่งผู้เขียน และฝั่งผู้อ่าน

- 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
- Shared Data
  - Data set
  - Semaphore **mutex** initialized to 1
  - Semaphore **wrt** initialized to 1
  - Integer **readcount** initialized to 0

ป้องกันตัวแปร readcount (ผู้อ่าน)

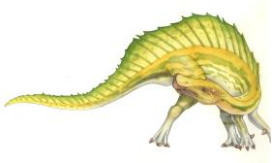
ป้องกันผู้เขียน

# Readers-Writers Problem (Cont.)

- The structure of a **writer process**

```
do {
        wait (wrt) ;

            // writing is performed

        signal (wrt) ;
} while (TRUE);
```

- ผู้อ่านคนแรก และคนสุดท้าย จะต้องใช้ตัวแปร wrt เพื่อทำให้การทำงานประสานกันได้กับผู้เขียน

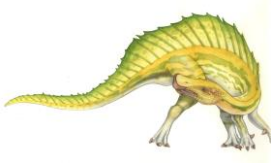


# Readers-Writers Problem (Cont.)

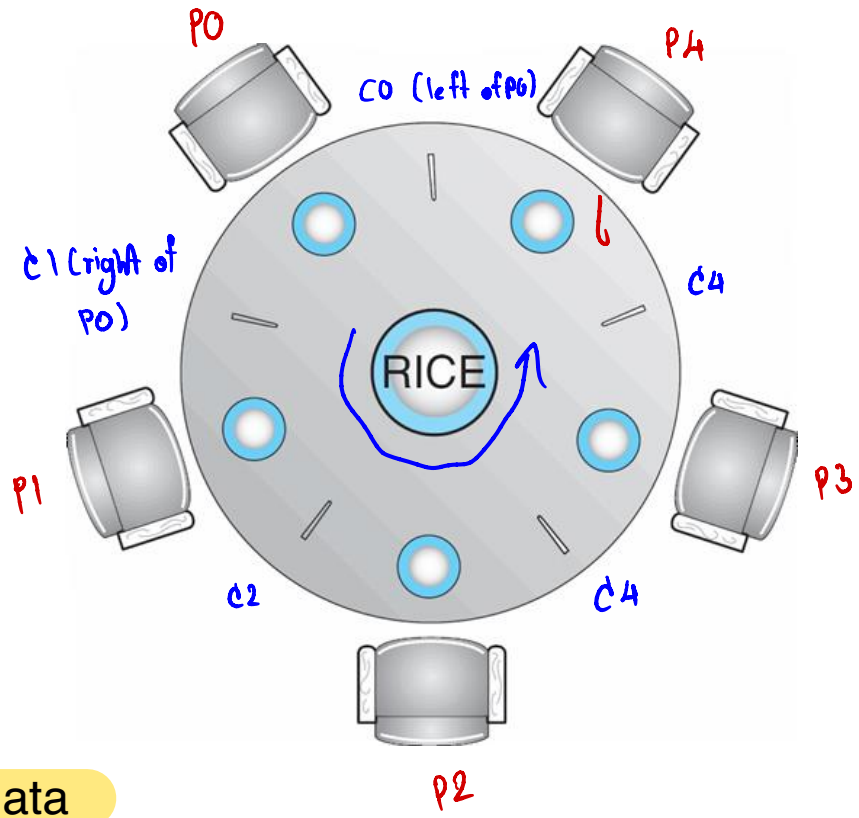
- The structure of a **reader process**

```
do {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)
        wait (wrt) ;
    signal (mutex)

    // reading is performed

    wait (mutex) ;
    readcount  - - ;
    if (readcount  == 0)
        signal (wrt) ;
    signal (mutex) ;
} while (TRUE);
```

หากผู้อ่านมีมากกว่า 1 คน ถ้าผู้เขียนกำลังทำงานอยู่ ผู้อ่านคนที่ 2 จะรออยู่ โดยการตรวจสอบตัวแปร mutex

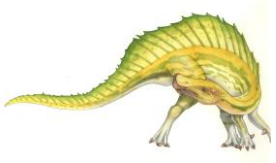


# Dining-Philosophers Problem




- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

C0 .. C4



# Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

i in [0...4]

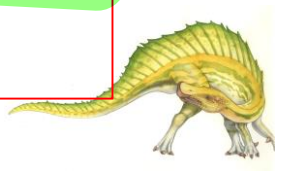
```
do {
        wait ( chopstick[i] );          lock Pi's left chopstick
        wait ( chopStick[ (i + 1) % 5] );   lock Pi right chopstick

        // eat

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

        // think                        Release Pi S right Chopstick

} while (TRUE);
```

หยิบตะเกียบ ใช้ operation Wait

วางตะเกียบ ใช้ operation Signal

สิงหนาท 7/9/66.

อาจเกิดปัญหา Deadlock ได้ หาก ทุกคนหิวพร้อมกัน แล้วหยิบตะเกียบข้างซ้าย เหมือนกันหมด



# Dining-Philosophers Problem (Cont.)

- อาจเกิดปัญหา Deadlock ได้ หากทุกคนหิวพร้อมกัน แล้วหยิบตะเกียบข้างซ้ายเหมือนกันหมด

วิธีแก้ไขเพื่อเลี่ยงการเกิด Deadlock

* มีนักปราชญ์นั่งโต๊ะได้ไม่เกิน 4

* กำหนดให้จะหยิบตะเกียบได้ตะเกียบด้านซ้ายและขวาต้องว่างทั้งคู่ (ขณะอยู่ใน Critical-Section

* ใช้การสลับกัน เช่น ให้คนเลขคี่หยิบซ้ายก่อน ข้างขวา และให้คนเลขคู่ หยิบขวาก่อน ข้างซ้าย

** อาจเกิดปัญหา starvation ได้หากแก้ไขไม่รัดกุม **

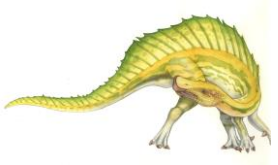
# Problems with Semaphores

- incorrect use of semaphore operations:

  (การใช้ **operation** ของ **semaphore** ที่ไม่ถูกต้อง)

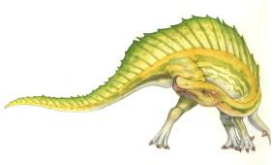
  - signal (mutex) …. wait (mutex) ทำให้ไม่เกิดคุณสมบัติ Mutual exclusion
  - wait (mutex) … wait (mutex) ทำให้เกิดปัญหา Deadlock ได้เพราะไม่มีใครปลดล๊อค
  - Omitting of wait (mutex) or signal (mutex) (or both)

    มีการละเลยการใช้ operation wait() หรือ signal() หรือทั้งคู่ จึงทำให้กลไกการทำงานของ semaphore ไม่เข้าจังหวะกัน



# Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads xI libary



# Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
  for a short critical to access the shared data
- Uses condition variables and readers-writers locks when longer sections of code need access to data
  See example on p.5.34
  For a long critical section to access the shared data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

# Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

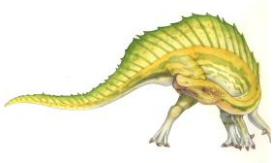
Spinlock = a thread waits in a loop or spin unit the lock is available (released)

For efficiency, the kernel ensure that a thread will never be preempted while holding a spinlock

For thread synchronization, a dispather object may use diff mechanism including mutex lock,

# Linux Synchronization

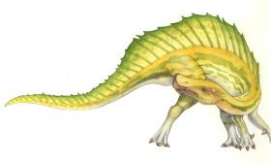
- Linux:
  Prior to version 2.6, Linux was a non-preemptive kernel i.e. process running
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks



# Pthreads Synchronization

Many system that implement Pthreads also provide Semaphore
Semaphore are not part of the POSIX

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks



# End of Chapter 5