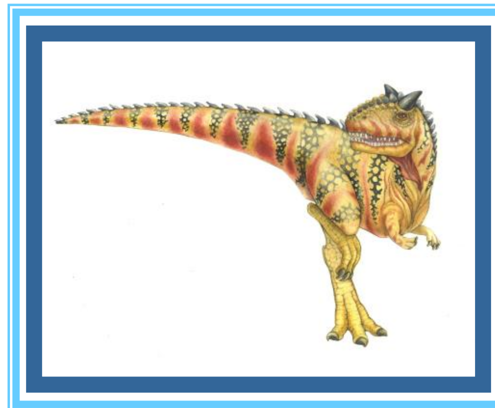
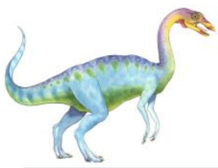
Chapter 5: Synchronization

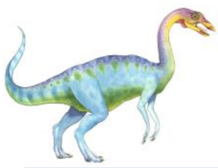


Silberschatz, Galvin and Gagne ©2018



Chapter 5: Synchronization

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



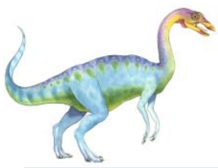
Objectives

- 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

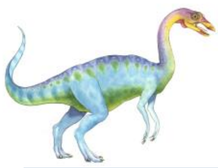
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having 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

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

- 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}

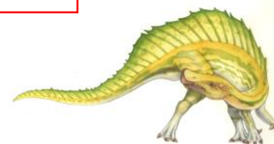
interleaving: การแทรกสลับการทำงานของชุดคำสั่ง

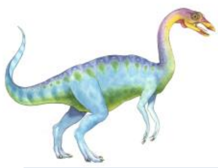
Solution to Critical-Section Problem

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

Critical section: **เขตวิกฤต** คือพื้นที่ที่process แต่ละตัวสามารถทำการปรับปรุง เปลี่ยนแปลงค่าตัวแปรต่างๆ ของprocess โดยไม่มี process อื่นเข้ามาเกี่ยวข้องในพื้นที่นี้

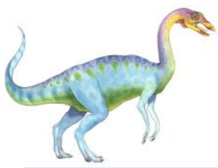
exist: ยังปรากฏอยู่ , คงอยู่ postponed: ปฏิเสธ
indefinitely : ไม่แน่นอน granted: ได้รับการอนุญาตแล้ว





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!

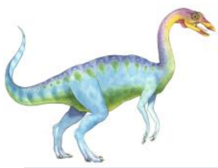


Algorithm for Process P_i

มุ่งประเด็นไปที่ ช่วงเวลาหนึ่งมี 2 process เท่านั้น

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

i: current process
j: other process

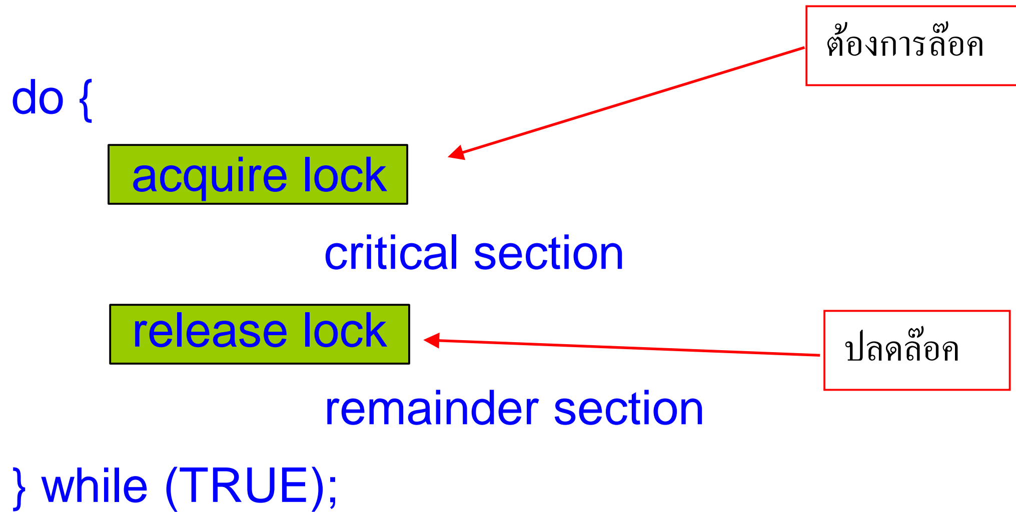


Synchronization Hardware

- 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

Uniprocessor : โปรเซสเซอร์ตัวเดียว

Solution to Critical-section Problem Using Locks



Semaphore

- Synchronization tool that does not require busy waiting (ไม่ต้องการการรอคอยที่มาก)
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations มี 2 การดำเนินการที่ทำกับSemaphore
  - **wait (S)** {
    ```
    wait (S) {
          while S <= 0
              ; // no-op
            S--;
     }
    ```
  - **signal (S)** {
    ```
    signal (S) {
       S++;
    }
    ```

Semaphore as General Synchronization Tool

- 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
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

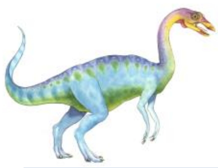
```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
       // Critical Section
    signal (mutex);
        // remainder section
} while (TRUE);
```

***wait* (mutex):**

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

***signal* (mutex):**

```
        mutex++;
```



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.)

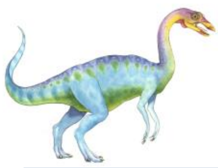
- Implementation of wait:

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

ค่า value อาจติดลบได้แสดงให้เห็นว่ามี process รอคอย semaphore

- Implementation of signal:

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

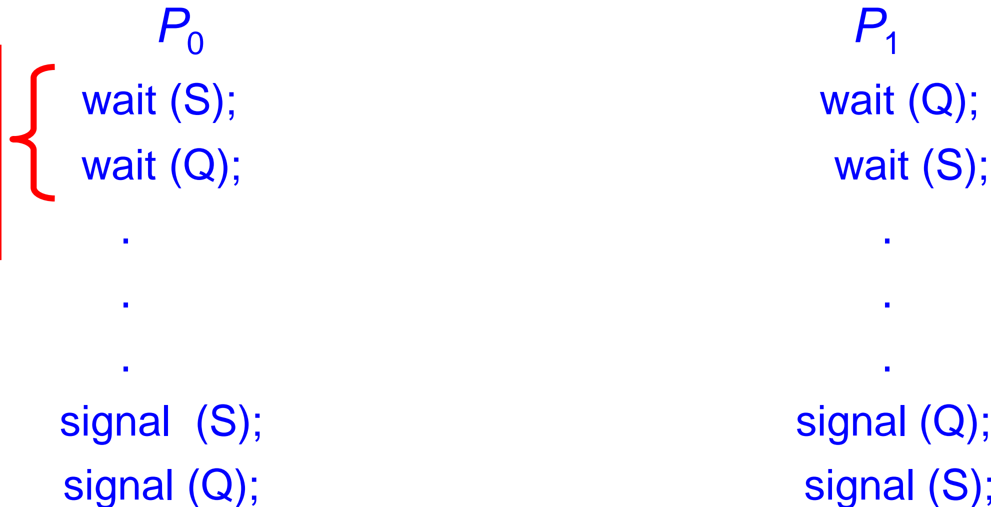


Deadlock and Starvation

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 หนึ่ง ทำคำสั่ง signal ก่อน จึงจะทำสั่ง wait ได้

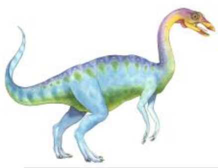


```
P0                      P1
wait (S);               wait (Q);
wait (Q);               wait (S);
.                       .
.                       .
.                       .
signal  (S);            signal (Q);
signal (Q);             signal (S);
```

- 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

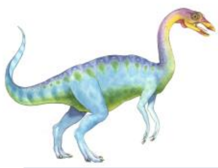
Classical Problems of Synchronization

- 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.



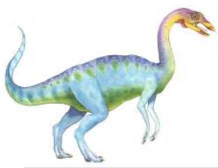
Bounded Buffer Problem (Cont.)

- The structure of the **producer process**

```
do  {

            //   produce an item in nextp

        wait (empty);
        wait (mutex);

            //  add the item to the  buffer

         signal (mutex);
         signal (full);
    } while (TRUE);
```

Empty --

Full ++



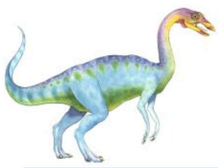
Bounded Buffer Problem (Cont.)

- 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 --

Empty ++



Readers-Writers Problem

-ใช้ข้อมูลร่วมกัน ผู้อ่านสามารถอ่าน (Reader) ข้อมูลร่วมกันได้หลายๆ คน

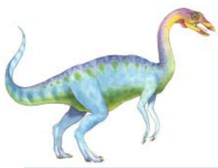
-ผู้เขียน (Writer) 1 คน สามารถเขียนข้อมูลได้ ณ ช่วงเวลาหนึ่ง โดยไม่มีผู้เขียนคนอื่นมาใช้ข้อมูลร่วม และห้ามผู้อ่านมาอ่านขณะที่เขียนอยู่

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

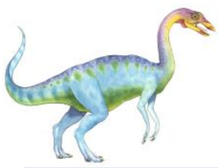


Readers-Writers Problem (Cont.)

- The structure of a **writer process**

```
do {
        wait (wrt) ;

            //    writing is performed

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

- ผู้อ่านคนแรก และคนสุดท้าย จะต้องใช้ตัวแปร wrt เพื่อทำให้การทำงานประสานกันได้กับผู้เขียน

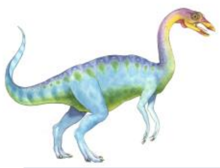


Readers-Writers Problem (Cont.)

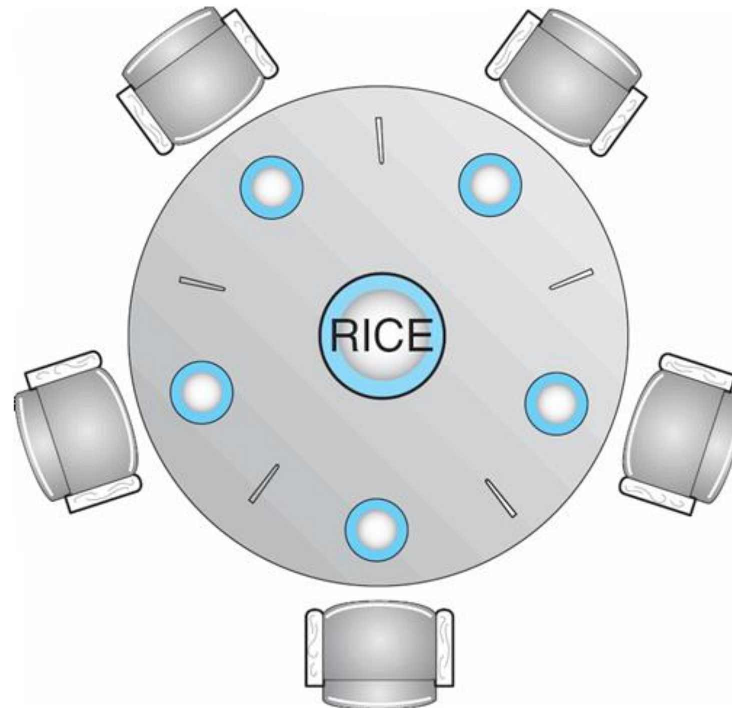
- The structure of a **reader process**

```
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

Dining-Philosophers Problem (Cont.)

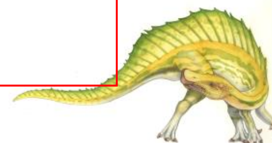
- The structure of Philosopher *i*:

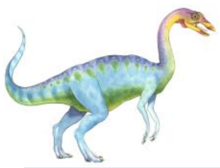
```
do {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            // eat

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

            // think

} while (TRUE);
```

หยิบตะเกียบ ใช้
operation Wait

วางตะเกียบ ใช้
operation Signal

อาจเกิดปัญหา
Deadlock ได้ หาก
ทุกคนหิวพร้อมกัน
แล้วหยิบตะเกียบข้างซ้าย
เหมือนกันหมด





Dining-Philosophers Problem (Cont.)

- อาจเกิดปัญหา Deadlock ได้ หากทุกคนหิวพร้อมกัน แล้วหยิบตะเกียบข้างซ้ายเหมือนกันหมด

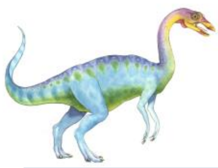
วิธีแก้ไขเพื่อเลี่ยงการเกิด Deadlock

* มีนักปราชญ์นั่งโต๊ะได้ไม่เกิน 4

* กำหนดให้จะหยิบตะเกียบได้ตะเกียบด้านซ้ายและขวาต้องว่างทั้งคู่ (ขณะอยู่ใน Critical-Section

* ใช้การสลับกัน เช่น ให้คนเลขคี่หยิบซ้ายก่อน ข้างขวา และให้คนเลขคู่ หยิบขวาก่อน ข้างซ้าย

** อาจเกิดปัญหา starvation ได้หากแก้ไขไม่รัดกุม **



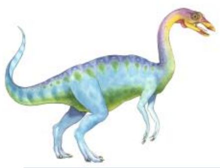
Problems with Semaphores

- incorrect use of semaphore operations:

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

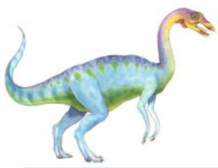
  - signal (mutex) …. wait (mutex) ทำให้ไม่เกิดคุณสมบัติ Mutual exclusion
  - wait (mutex) … wait (mutex) ทำให้เกิดปัญหา Deadlock ได้เพราะไม่มีใครปลดล็อค
  - Omitting of wait (mutex) or signal (mutex) (or both)

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



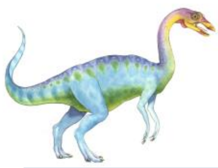
Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads



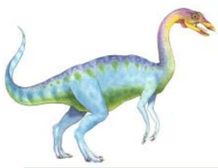
Solaris Synchronization

- 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
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock



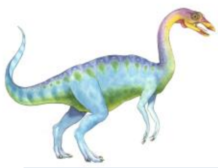
Windows XP Synchronization

- 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



Linux Synchronization

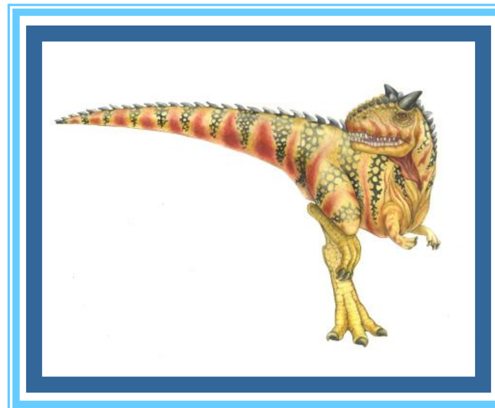
- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks



Pthreads Synchronization

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

End of Chapter 5



Silberschatz, Galvin and Gagne ©2018