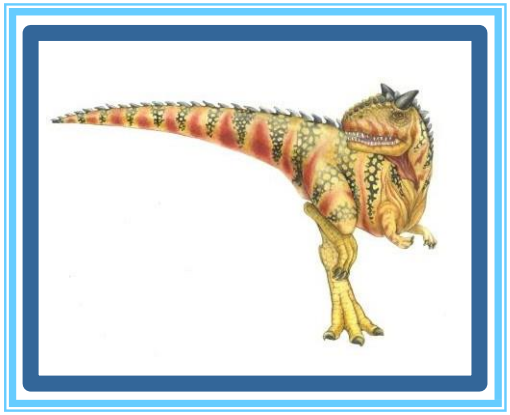
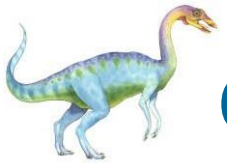
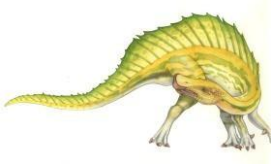
Chapter 5: Synchronization

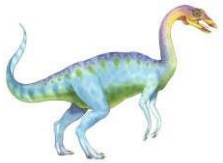




Chapter 5: Synchronization

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





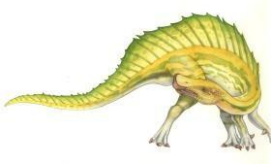
Objectives

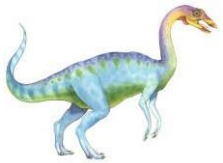
เพื่อแนะนำปัญหาส่วนวิกฤต ซึ่งสามารถใช้วิธีแก้ปัญหาเพื่อให้มั่นใจว่าข้อมูลที่ใช้ร่วมกันมีความสอดคล้องกัน

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data

เพื่อนำเสนอโซลูชันซอฟต์แวร์และฮาร์ดแวร์ของปัญหาส่วนวิกฤติ

- To present both software and hardware solutions of the critical-section problem





Background

การเข้าถึงข้อมูลที่ใช้ร่วมกันพร้อมกันอาจส่งผลให้ข้อมูลไม่สอดคล้องกัน

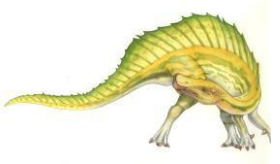
- Concurrent access to shared data may result in data inconsistency

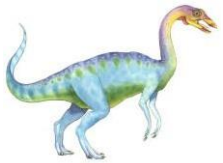
การรักษาความสอดคล้องของข้อมูลจำเป็นต้องมีกลไกเพื่อให้แน่ใจว่าการดำเนินการตามกระบวนการความร่วมมือเป็นไปอย่างเป็นระเบียบ

- 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.

  สมมติว่าเราต้องการนำเสนอวิธีแก้ไขปัญหาผู้บริโภค-ผู้ผลิตที่เติมเต็มบัฟเฟอร์ทั้งหมด เราสามารถทำได้โดยนับจำนวนเต็มเพื่อติดตามจำนวนบัฟเฟอร์เต็ม เริ่มแรก จำนวนจะถูกตั้งค่าเป็น 0 โดยผู้ผลิตจะเพิ่มขึ้นหลังจากที่สร้างบัฟเฟอร์ใหม่และจะลดลงโดยผู้บริโภคหลังจากที่ใช้บัฟเฟอร์

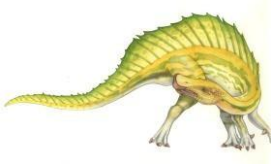
increment : เพิ่มค่าขึ้น
decrement: ลดค่าลง

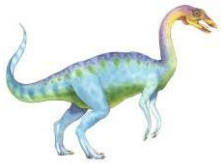




Producer

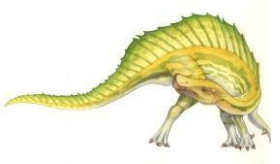
```
while (true) {
        ผลิตรายการและใส่ใน nextProduced
       /*  produce an item and put in nextProduced  */
        while (count == BUFFER_SIZE)
                        ; // do nothing
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
}
```

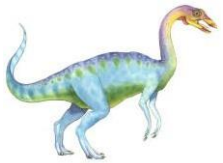




Consumer

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

                  /*  consume the item in nextConsumed
                  ใช้รายการใน 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
```

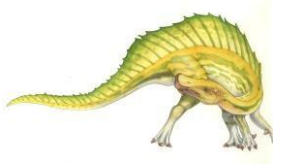
- count-- could be implemented as (Statement of the Consumer)

```
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



แนวทางแก้ไขปัญหาส่วนวิกฤต

Solution to Critical-Section Problem

การยกเว้นร่วมกัน - หากกระบวนการ Pi กำลังดำเนินการในส่วนที่สำคัญ จะไม่มีกระบวนการอื่นใดที่สามารถดำเนินการในส่วนที่สำคัญได้

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)

   ความคืบหน้า - หากไม่มีกระบวนการใดดำเนินการในส่วนวิกฤติและมีกระบวนการบางอย่างที่ต้องการเข้าสู่ส่วนวิกฤติ ดังนั้นการเลือกกระบวนการที่จะเข้าสู่ส่วนวิกฤติถัดไปจะไม่สามารถเลื่อนออกไปได้อย่างไม่มีกำหนด

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)

   การรอแบบมีขอบเขต - ต้องมีขอบเขตตามจำนวนครั้งที่กระบวนการอื่นได้รับ

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)

   อนุญาตให้เข้าสู่ส่วนที่สำคัญของตน หลังจากที่กระบวนการได้ส่งคำขอเพื่อเข้าสู่ส่วนที่สำคัญและก่อนที่จะได้รับคำขอนั้น

   สมมติว่าแต่ละกระบวนการดำเนินการด้วยความเร็วที่ไม่ใช่ศูนย์

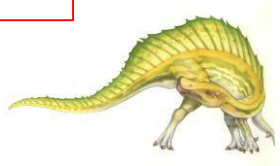
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

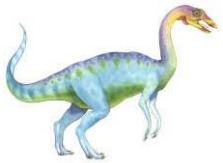
   ไม่มีสมมติฐานเกี่ยวกับความเร็วสัมพัทธ์ของกระบวนการ N

Each process has a segment of code, called a critical

Critical section: **เขตวิกฤต** คือพื้นที่ process แต่ละตัวสามารถทำ การปรับปรุงเปลี่ยนแปลงค่าตัวแปรต่างๆ ของprocess โดยไม่มี process อื่นเข้ามาเกี่ยวข้องในพื้นที่นี้

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





Peterson's Solution

โซลูชันสองกระบวนการ

- Two process solution (เป็นวิธีที่ใช้กับ 2 process)

สมมติว่าคำสั่ง LOAD และ STORE เป็นแบบอะตอมมิก นั่นคือไม่สามารถขัดจังหวะได้

- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.

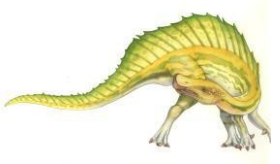
กระบวนการทั้งสองใช้ตัวแปรสองตัวร่วมกัน:

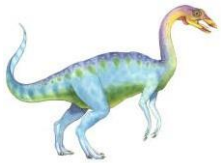
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]

การหมุนของตัวแปรบ่งชี้ว่าตาของใครที่จะเข้าสู่ส่วนวิกฤต

- 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!

อาร์เรย์แฟล็กใช้เพื่อระบุว่ากระบวนการพร้อมที่จะเข้าสู่ส่วนวิกฤติหรือไม่ flag[i] = true หมายความว่ากระบวนการ Pi พร้อมแล้ว!





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 solution to the critical

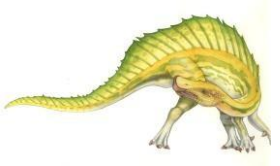
See my note

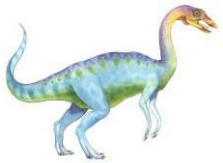
```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section
} while (TRUE);
```

ส่วนที่เหลือ

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

i: current process
j: other process





Synchronization Hardware

หลายระบบให้การสนับสนุนฮาร์ดแวร์สำหรับโค้ดส่วนสำคัญ

- Many systems provide hardware support for critical section code
- Uniprocessors – สามารถปิดการใช้งานการขัดจังหวะได้
- Uniprocessors – could disable interrupts
  - รหัสที่รันอยู่ในปัจจุบันจะดำเนินการโดยไม่มีการจองล่วงหน้า
  - Currently running code would execute without preemption
  - โดยทั่วไปแล้วระบบมัลติโปรเซสเซอร์ไม่มีประสิทธิภาพเกินไป
  - Generally too inefficient on multiprocessor systems
    - ระบบปฏิบัติการที่ใช้สิ่งนี้ไม่สามารถปรับขนาดได้ในวงกว้าง
    - Operating systems using this not broadly scalable

เครื่องจักรสมัยใหม่ให้คำแนะนำพิเศษเกี่ยวกับฮาร์ดแวร์ของอะตอม

- Modern machines provide special atomic hardware instructions
  - อะตอม = ไม่หยุดชะงัก
  - Atomic = non-interruptable

ทดสอบคำในหน่วยความจำและตั้งค่า

  - Either test memory word and set value
  - หรือสลับเนื้อหาของคำในหน่วยความจำสองคำ
  - Or swap contents of two memory words

Special H/W instruction that allow

Uniprocessor : โปรเซสเซอร์ตัวเดียว

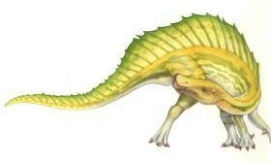
Solution to Critical-section Problem Using Locks

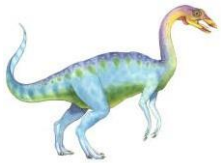
The simplest of higher-level s/w tool to solve the critical-section problem is the mutex lock.

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

ต้องการล๊อค

ปลดล๊อค





Semaphore

เครื่องมือการซิงโครไนซ์ที่ไม่ต้องรอยุ่ง

- Synchronization tool that does not require busy waiting (ไม่ต้องการการรอคอยที่มาก)
- Semaphore S – integer variable (เซมาฟอร์ S – ตัวแปรจำนวนเต็ม)
- Two standard operations modify S: wait() and signal() (การดำเนินการมาตรฐานสองประการแก้ไข S: wait() และ signal())
  - Originally called P() and V() (เดิมเรียกว่า P() และ V())
- Less complicated (ซับซ้อนน้อยลง)
- Can only be accessed via two indivisible (atomic) operations (สามารถเข้าถึงได้ผ่านการดำเนินการที่แบ่งแยกไม่ได้ (อะตอมมิก) สองรายการเท่านั้น) มี 2 การดำเนินการที่ทำกับ Semaphore

- **wait (S)** {

```
    while S <= 0
        ; // no-op
    S--;
}
```

- **signal (S)** {

```
    S++;
}
```

เซมาฟอร์เป็นเครื่องมือการซิงโครไนซ์ทั่วไป

Semaphore as General Synchronization Tool

การนับเซมาฟอร์ – ค่าจำนวนเต็มสามารถครอบคลุมโดเมนที่ไม่จำกัด

S in [-infinity , + infinity]

- Counting semaphore – integer value can range over an unrestricted domain

เซมาฟอร์ไบนารี - ค่าจำนวนเต็มสามารถอยู่ในช่วงระหว่าง 0 ถึง 1 เท่านั้น สามารถนำไปใช้ได้ง่ายกว่า

- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement

S in [0,1] only เท่านั้น

เรียกอีกอย่างว่าล็อค mutex

  - Also known as mutex locks

สามารถใช้เซมาฟอร์นับ S เป็นเซมาฟอร์ไบนารีได้

- Can implement a counting semaphore S as a binary semaphore

จัดให้มีการยกเว้นร่วมกัน

- 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*;

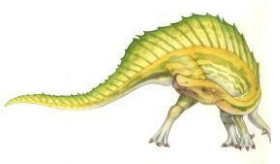
mutex--;

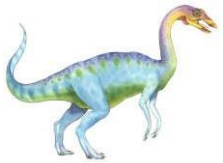
wait to test รอไปเรื่อยๆ ถ้าเป็น 0 ถ้า 1 ไส

***signal* (mutex):**

mutex++;

signal to increment





การใช้งานเซมาฟอร์

Semaphore Implementation

ต้องรับประกันว่าไม่มีสองกระบวนการใดที่สามารถดำเนินการรอ () และส่งสัญญาณ () บนเซมาฟอร์เดียวกันในเวลาเดียวกันได้

ไม่สามารถทำพร้อมกันได้

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time

Since both operation update a semaphore (s.)

ดังนั้นการนำไปปฏิบัติจึงกลายเป็นปัญหาส่วนวิกฤตโดยที่รหัสการรอและสัญญาณถูกวางไว้ในส่วนวิกฤต

- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.

ตอนนี้อาจยุ่งกับการรอคอยในการใช้งานส่วนสำคัญ

  - Could now have busy waiting in critical section implementation

แต่รหัสการใช้งานนั้นสั้น

    - But implementation code is short

ไม่รู้ว่ารอนานแค่ไหน

การรอยุ่งนิดหน่อยหากส่วนสำคัญไม่ค่อยมีคนอยู่

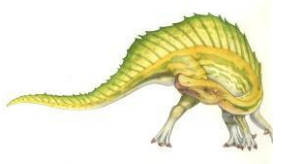
รอนานกับคนที่ใช่ไหม

    - Little busy waiting if critical section rarely occupied

โปรดทราบว่าแอปพลิเคชันอาจใช้เวลามากในส่วนที่สำคัญ ดังนั้นนี่จึงไม่ใช่วิธีแก้ปัญหาที่ดี

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

ถ้าสั้น โค้ดมันสั้น แต่ว่า busy waiting เกิดขึ้น



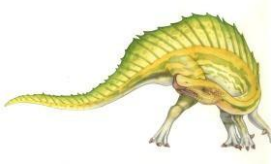
การใช้งาน Semaphore โดยไม่ต้องรอยุ่ง
Semaphore Implementation with no Busy waiting

แต่ละเซมาฟอร์จะมีคิวรอที่เกี่ยวข้องกัน แต่ละรายการในคิวรอจะมีรายการข้อมูลสองรายการ:

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

สองการดำเนินงาน:

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



การใช้งาน Semaphore โดยไม่ต้องรอยุ่ง

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 อาจติดลบได้แสดงให้เห็นว่า มี process รอคอย semaphore

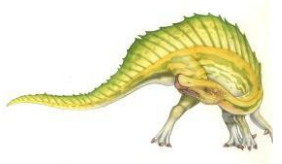
A linked list is used as a waiting queue of semaphon S.

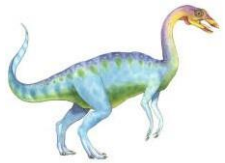
- Implementation of signal:

```
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. ต่อกันอย่างไร





การหยุดชะงักและความอดอยาก

Deadlock and Starvation

การใช้ semaphore อาจทำให้เกิดเหตุการณ์ขึ้นได้ ดังนี้

การหยุดชะงัก – สองกระบวนการขึ้นไปกำลังรออย่างไม่มีกำหนดสำหรับเหตุการณ์ที่อาจเกิดจากกระบวนการรอเพียงกระบวนการเดียวเท่านั้น

- 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

ให้ S และ Q เป็นสองเซมาฟอร์เริ่มต้นที่ 1

รอให้มี process หนึ่งทำคำสั่ง signal ก่อน จึงจะทำคำสั่ง wait ได้

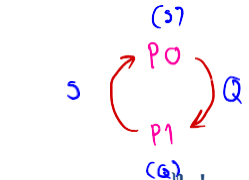
P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

S = 1 - 1 = 0

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

Q = 1 - 1 = 0

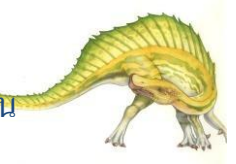
S = 0, Thus, P1 must wait for P0 to release S.



ความอดอยาก - การปิดกั้นอย่างไม่มีกำหนด กระบวนการอาจไม่ถูกลบออกจากคิวเซมาฟอร์ที่ถูกระงับไว้

- 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

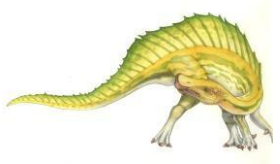
การผกผันของลำดับความสำคัญ - ปัญหาการจัดกำหนดการเมื่อกระบวนการที่มีลำดับความสำคัญต่ำกว่าค้างล็อคที่จำเป็นสำหรับกระบวนการที่มีลำดับความสำคัญสูงกว่า

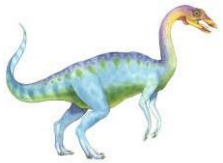


ปัญหาคลาสสิกของการซิงโครไนซ์

Classical Problems of Synchronization

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



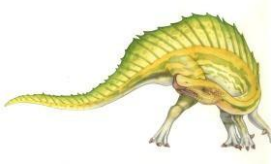


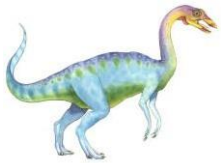
Bounded-Buffer Problem

N บัฟเฟอร์ แต่ละรายการสามารถเก็บได้หนึ่งรายการ

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

Producer & consumer





Bounded Buffer Problem (Cont.)

wait &

โครงสร้างของกระบวนการผลิต

- The structure of the **producer process**

Initialization:
int N = 5
semaphore

```
do {
        // สร้างรายการใน nextp
        // produce an item in nextp

        wait (empty);
        wait (mutex);

        // add the item to the buffer
        // เพิ่มรายการลงในบัฟเฟอร์

        signal (mutex);
        signal (full);

} while (TRUE);
```

empty = 5 → 4 → 3 → 2 → 1 → 0

Empty --

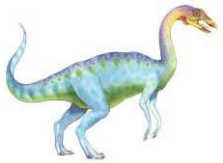
The 1st producer lock the execution of critical section (code of buffer updating)
mutex = 1 → 0

mutex = 0 → 1

release the lock
the 1st producer

Full ++

Full = 0 → 1 → 2 → 3 → 4 → 5.



Bounded Buffer Problem (Cont.)

โครงสร้างของกระบวนการผู้บริโภค

- 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)

ลบรายการออกจากบัฟเฟอร์ไปยัง nextc

mutex = 0 → 1

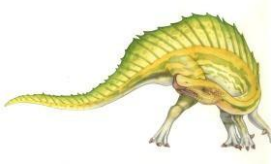
The 1st consumer releases lock

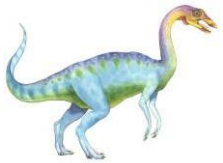
Empty ++

Critical section

empty = 0 → 1 → 2 → 3 → 4 → 5

ใช้รายการใน nextc





Readers-Writers Problem

-ใช้ข้อมูลร่วมกัน ผู้อ่านสามารถอ่าน (Reader) ข้อมูลร่วมกันได้หลายๆ คน
-ผู้เขียน (Writer) 1 คน สามารถเขียนข้อมูลได้ ณ ช่วงเวลาหนึ่ง โดยไม่มีผู้เขียนคนอื่นมาใช้ข้อมูลร่วม และห้ามผู้อ่านมาอ่านขณะที่เขียนอยู่

อาจทำให้เกิดปัญหา Starvation ได้ทั้งฝั่งผู้เขียน และฝั่งผู้อ่าน

ชุดข้อมูลจะถูกแชร์ระหว่างกระบวนการต่างๆ ที่เกิดขึ้นพร้อมกัน

- A data set is shared among a number of concurrent processes
  ผู้อ่าน – อ่านเฉพาะชุดข้อมูล พวกเขาไม่ได้ทำการอัพเดตใด ๆ
  - Readers – only read the data set; they do **not** perform any updates
  นักเขียน - สามารถอ่านและเขียนได้
  - Writers – can both read and write

ปัญหา – อนุญาตให้ผู้อ่านหลายคนอ่านพร้อมกัน นักเขียนเพียงคนเดียวเท่านั้นที่สามารถเข้าถึงข้อมูลที่ใช้ร่วมกันได้ในเวลาเดียวกัน

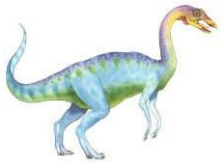
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time

ข้อมูลที่ใช้ร่วมกัน

- Shared Data
  - Data set
  เซมาฟอร์ mutex เริ่มต้นเป็น 1
  - Semaphore **mutex** initialized to 1
  เซ็นมาสำหรับ mutex เหตุการณ์เป็น 1
  - Semaphore **wrt** initialized to 1
  จำนวนการอ่านจำนวนเต็มเริ่มต้นเป็น 0
  - Integer **readcount** initialized to 0

ป้องกันตัวแปร readcount (ผู้อ่าน)

ป้องกันผู้เขียน



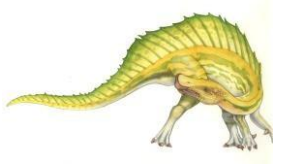
Readers-Writers Problem (Cont.)

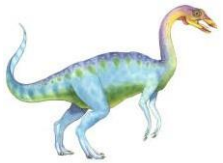
โครงสร้างของกระบวนการเขียน

- The structure of a **writer process**

```
do {
        wait (wrt) ;
                // writing is performed
        signal (wrt) ;
} while (TRUE);
```

- ผู้อ่านคนแรก และคนสุดท้าย จะต้องใช้ตัวแปร wrt เพื่อทำให้การทำงานประสานกันได้กับผู้เขียน





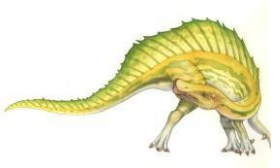
Readers-Writers Problem (Cont.)

โครงสร้างของกระบวนการอ่าน

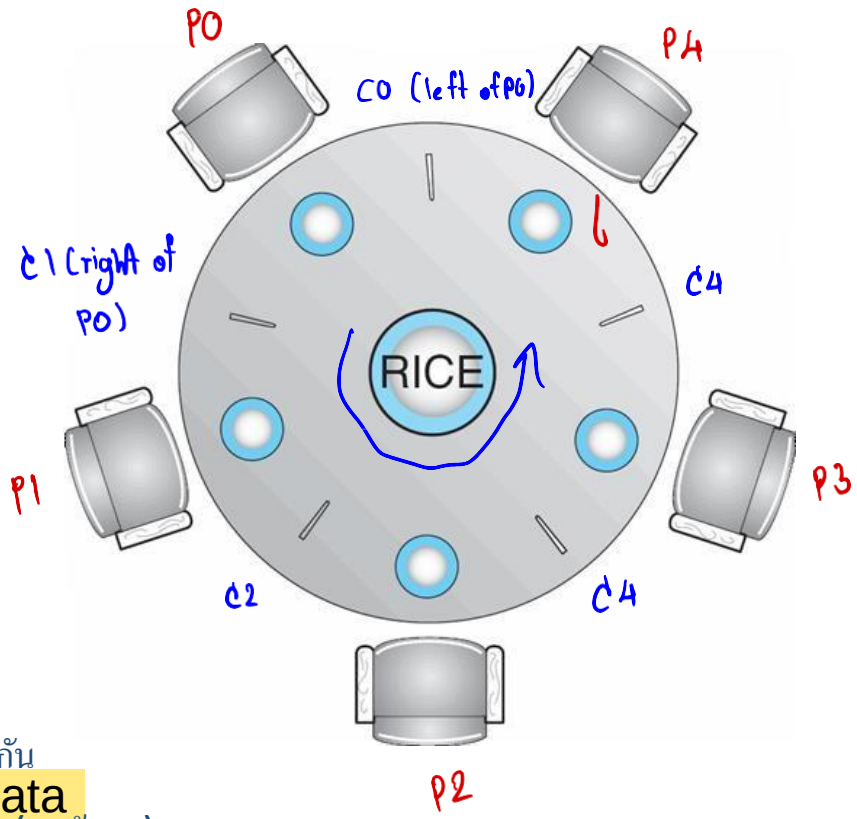
- The structure of a **reader process**

```
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);
```

หากผู้อ่านมีมากกว่า1 คน ถ้าผู้เขียนกำลังทำงานอยู่ ผู้อ่านคนที่ 2 จะรออยู่ โดยการตรวจสอบ ตัวแปร mutex



Dining-Philosophers Problem



ข้อมูลที่ใช้ร่วมกัน

- Shared data

ชามข้าว (ชุดข้อมูล)

  - Bowl of rice (data set)

ตะเกียบเซมาฟอร์ [5] เริ่มต้นเป็น 1

  - Semaphore chopstick [5] initialized to 1

C0 ... C4

Dining-Philosophers Problem (Cont.)

โครงสร้างของปราชญ์

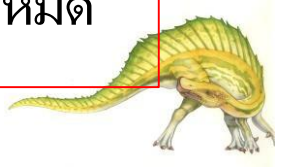
- The structure of Philosopher *i*:

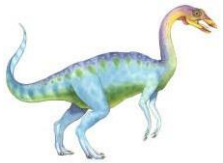
```
                                          i in [0...4]
do {
        wait ( chopstick[i] );            lock Pi's left chopstick
        wait ( chopStick[ (i + 1) % 5] );  lock Pi right chopstick

        // eat

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

                                          Release Pi S right
                                                  Chopstick
        // think

} while (TRUE);
```

หยิบตะเกียบ ใช้ operation Wait

วางตะเกียบ ใช้ operation Signal

อาจเกิดปัญหา **Deadlock** ได้ หากทุกคนหิวพร้อมกันแล้วหยิบ ตะเกียบข้างซ้ายเหมือนกันหมด





Dining-Philosophers Problem (Cont.)

- อาจเกิดปัญหา Deadlock ได้ หากทุกคนหิวพร้อมกัน แล้วหยิบตะเกียบข้างซ้ายเหมือนกันหมด

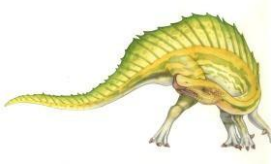
วิธีแก้ไขเพื่อเลี่ยงการเกิด Deadlock

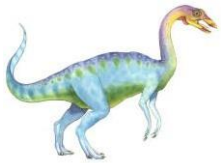
* มีนักปราชญ์นั่งโต๊ะได้ไม่เกิน 4

* กำหนดให้จะหยิบตะเกียบได้ตะเกียบด้านซ้ายและขวาต้องวางทั้งคู่ (ขณะอยู่ใน Critical-Section

* ใช้การสลับกัน เช่น ให้คนเลขคี่หยิบซ้ายก่อน ข้างขวา และให้คนเลขคู่ หยิบขวาก่อน ข้างซ้าย

** อาจเกิดปัญหา starvation ได้หากแก้ไขไม่รัดกุม **





ปัญหาเกี่ยวกับเซมาฟอร์ส

Problems with Semaphores

การใช้การดำเนินการเซมาฟอร์ไม่ถูกต้อง

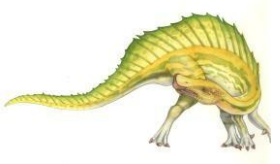
- incorrect use of semaphore operations:

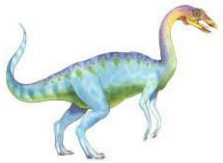
  (การใช้ operation ของ semaphore ที่ไม่ถูกต้อง)

  - signal (mutex) …. wait (mutex) ทำให้ไม่เกิดคุณสมบัติ Mutual exclusion
  - wait (mutex) … wait (mutex) ทำให้เกิดปัญหา Deadlock ได้เพราะไม่มีใครปลดล๊อค

    การละเว้นการรอ (mutex) หรือสัญญาณ (mutex) (หรือทั้งสองอย่าง)
  - Omitting of wait (mutex) or signal (mutex) (or both)

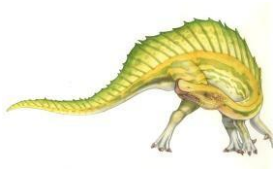
  มีการละเลยการใช้ operation wait() หรือ signal() หรือทั้งคู่ จึงทำให้กลไกการทำงานของ semaphore ไม่เข้าจังหวะกัน

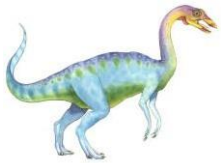




Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads xI libary





การซิงโครไนซ์โซลาริส

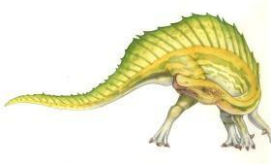
Solaris Synchronization

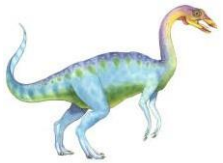
ใช้การล็อคที่หลากหลายเพื่อรองรับการทำงานหลายอย่างพร้อมกัน มัลติเธรด (รวมถึงเธรดแบบเรียลไทม์) และการประมวลผลหลายตัว

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
  ใช้ mutexes ที่ปรับเปลี่ยนได้เพื่อประสิทธิภาพในการปกป้องข้อมูลจากส่วนของโค้ดแบบสั้น
- 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

ใช้ตัวแปรเงื่อนไขและตัวอ่านและผู้เขียน
จะล็อกเมื่อส่วนที่ยาวขึ้นของโค้ด
จำเป็นต้องเข้าถึงข้อมูล

ใช้ประตูหมุนเพื่อเรียงลำดับรายการเธรดที่รอรับ mutex แบบปรับได้
หรือการล็อคตัวอ่าน-ผู้เขียน





การซิงโครไนซ์ Windows XP
Windows XP Synchronization

ใช้มาสก์ขัดจังหวะเพื่อปกป้องการเข้าถึงทรัพยากรทั่วโลกบนระบบตัวประมวลผลเดียว

- Uses interrupt masks to protect access to global resources on uniprocessor systems

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

- Uses spinlocks on multiprocessor systems

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

ยังมีอ็อบเจ็กต์โปรแกรมเลือกจ่ายงานซึ่งอาจทำหน้าที่เป็น mutexes และเซมาฟอร์ด้วย

- Also provides dispatcher objects which may act as either mutexes and semaphores

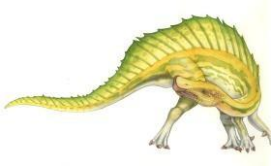
For thread synchronization, a dispather object may use diff mecheanin including mutex lock,

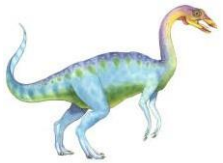
ออบเจ็กต์ Dispatcher อาจจัดให้มีกิจกรรมด้วย

- Dispatcher objects may also provide events

เหตุการณ์ทำหน้าที่เหมือนกับตัวแปรเงื่อนไข

  - An event acts much like a condition variable





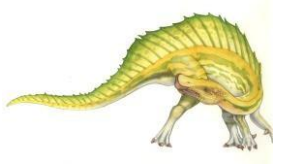
การซิงโครไนซ์ลินุกซ์

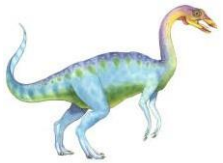
Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections ก่อนเคอร์เนลเวอร์ชัน 2.6 ให้ปิดใช้งานการขัดจังหวะเพื่อใช้ส่วนวิกฤตแบบสั้น
  - Version 2.6 and later, fully preemptive
    เวอร์ชัน 2.6 และใหม่กว่า ยึดเอาเสียก่อนโดยสมบูรณ์

ลินุกซ์จัดให้

- Linux provides:
  - semaphores
  - spin locks





การซิงโครไนซ์ Pthreads

Pthreads Synchronization

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

Pthreads API ไม่ขึ้นกับระบบปฏิบัติการ

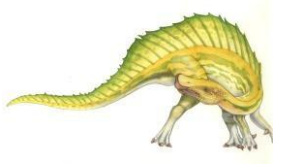
- Pthreads API is OS-independent

มันมี:

- It provides:
  - mutex locks
  - condition variables

ส่วนขยายที่ไม่สามารถพกพาได้ ได้แก่:

- Non-portable extensions include:
  - read-write locks
  - spin locks



End of Chapter 5