

Chapter 5: Synchronization

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

2

6

1

3

5

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

Operating System Concepts - 10th Edition

Silberschatz, Galvin and Gagne 2218

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: เพิ่มทำขึ้น decrement: กลทำกล

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++;
}

Operating System Concepts = 10° Edition 5.5 Silberschatz, Galvin and Gagne 02015

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

interleaving: การแกรกสนักกรรท์รานของจุดค์กซึ้ง

Coperating System Concepts - 10m Edition

Silberschatz, Galvin and Gagne 62018
```

Solution to Critical-Section Problem 1. Mutual Exclusion - If process P<sub>i</sub> is executing in its critical section, then no other processes can be executing in their critical sections (การห้ามอยู่พร้อม 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: เพลริลดุล คือทั้นที่ที่process แต่ละตัวสามารถทำการปริบปรุง เปลี่ยนแปลงทำตัวแปรทำงๆ พองprocess โดยไม่มี process อื่นเช่นมาเกี่ยวข้องในที่นที่นี้ exist: ยังปรากฏอยู่ , คงอยู่ postponed: ปฏิเสธ indefinitely : ไม่แน่นอน granted:ได้รับการอนุญาตแล้ว

8

10

Peterson's Solution

Two process solution (ឃើមទីមើល 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[t] = true implies that process P<sub>i</sub> is ready!

Algorithm for Process P<sub>i</sub>

บุ๋งประเด็นไปที่ ช่วงเวลาหนึ่งมี 2 process เท่านั้น

do {

flag[i] = TRUE;

turn = j;

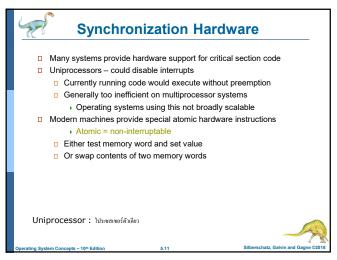
while (flag[j] && turn == j);

critical section
flag[i] = FALSE;

remainder section
} while (TRUE);

i: current process
j: other process
j: other process

9



```
Semaphore

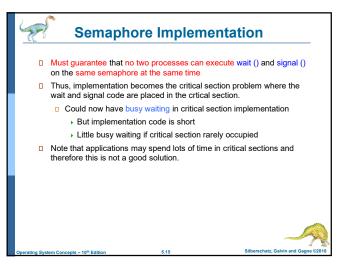
Synchronization tool that does not require busy waiting (ให้ดีอการการวอดองที่มาก)
Semaphore S – integer variable
Two standard operations modify S: wait() and signal()
Originally called P() and V()
Less complicated
Can only be accessed via two indivisible (atomic) operations ขึ 2 การต่าเน็นการที่ทำกับSemaphore
wait (S) {
while S <= 0
; // no-op
S--;
}
signal (S) {
S++;
}
Operating System Concepts − 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne €2018
```

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 wait (mutex):
| do {
wait (mutex);	while mutex≤ 0 do no-op; mutex--;	
// Critical Section	signal (mutex);	mutex++;
// remainder section	signal (mutex):	mutex++;
while (TRUE);	while (TRUE);	mutex++;
Counting System Concepts - 10™ Edition	Silberschatz, Galvin and Gagne Cottes	

13 14



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

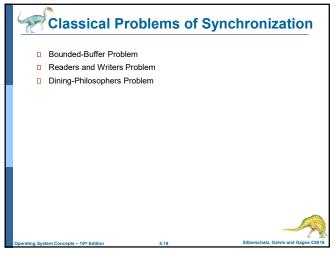
Two operations:
block – place the process invoking the operation on the appropriate waiting queue. (ให้เข้าโปรดในสิวธ์โปรทำงาน)
wakeup – remove one of processes in the waiting queue and place it in the ready queue. (ป่าออกรากกับก็อวอทำงาน)

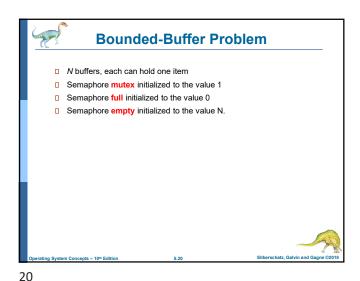
16

18

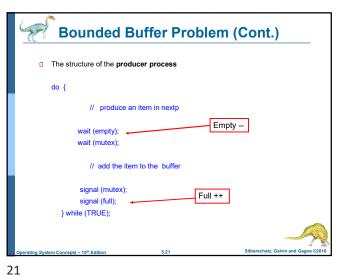
15

**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 wait (S); wait (Q): ทำคำสั่ง signal ก่อน wait (Q); wait (S); จึงจะทำสั่ง wait ได้ signal (S); signal (Q); 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





19



**Bounded Buffer Problem (Cont.)** ☐ The structure of the consumer process Full -wait (full): wait (mutex); // remove an item from buffer to nexto signal (mutex); signal (empty); Empty ++ // consume the item in nexto } while (TRUE);

22

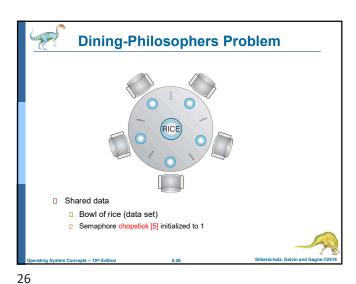
```
Readers-Writers Problem
       -ใช้ข้อมูลร่วมกัน ผู้อ่านสามารถอ่าน (Reader) ข้อมูลร่วมกันได้หลายๆ คน
       -ผู้เขียน (Writer) 1 คน สามารถเขียนข้อมูลได้ ณ ช่วงเวลาหนึ่ง โดยไม่มีผู้เขียนคนอื่นมาใช้ข้อมูลร่วม และ
       ห้ามผู้อ่านมาอ่านขณะที่เขียนอยู่
        อาจทำให้เกิดปัญหา Starvation ได้ทั้ง ฝั่งผู้เขียน 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
                               ป้องกันตัวแปร readcount (ผู้อ่าน)
    Semaphore mutex initialized to 1

    Semaphore wrt initialized to 1

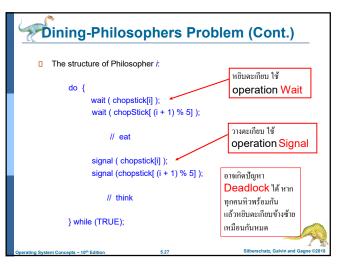
    □ Integer readcount initialized to 0
```

**Readers-Writers Problem (Cont.)** ☐ The structure of a writer process ผู้อ่านคนแรก และคนสุดท้าย จะต้องใช้ตัวแปร wrt เพื่อทำให้การทำงานประสานกันได้กับ do { wait (wrt); // writing is performed signal (wrt); } while (TRUE);

```
Readers-Writers Problem (Cont.)
☐ The structure of a reader process
                                                    หากผู้อ่านมีมากกว่า 1 คน ถ้า
                                                    ผู้เขียนกำลังทำงานอยู่ ผู้อ่านคน
             wait (mutex);
                                                    ที่ 2 จะรออยู่ โดยการตรวจสอบ
             readcount ++
             if (readcount == 1)
                                                    ตัวแปร mutex
                       wait (wrt) ;
             signal (mutex)
                 // reading is performed
              wait (mutex) ;
              readcount - -
             if (readcount == 0)
                       signal (wrt);
              signal (mutex);
        } while (TRUE);
```



25



Dining-Philosophers Problem (Cont.)

- อาจเกิดปัญหา Deadlock ได้ หากทุกคนหิวพร้อมกับ
แล้วหยิบตะเกียบข้างซ้ายเหมือนกันหมด

วิธีแก้ไขเพื่อเลี่ยงการเกิด Deadlock

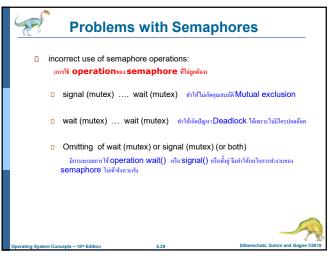
\* มีนักปราชญ์นั่งโต๊ะได้ไม่เกิน 4

\* กำหนดให้จะหยิบตะเกียบได้ตะเกียบด้านซ้ายและขวาด้องว่างทั้งคู่ (ขณะอยู่ใน Critical-Section)

\* ใช้การสลับกับ เช่น ให้คนเลขคี่หยิบซ้ายก่อน ข้างขวา และให้คนเลขคู่ หยิบขวา ก่อน ข้างช้าย

\*\* อาจเกิดปัญหา Starvation ได้หากแก้ไขไม่รัดกุม \*\*

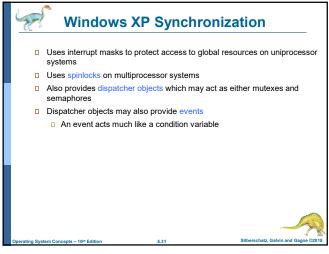
27 28



Synchronization Examples

Windows XP
Linux

Operating System Concepts – 10th Edition 6.30 Silberschatz, Galvin and Gagne 62018



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

Operating System Concepts – 10<sup>th</sup> Edition

5.32

Silbarschatz, Galvin and Gagne (2018)

