บทที่ 5
การประสานกระบวนการ
(Process Synchronization)

ในการทำงานของระบบคอมพิวเตอร์ที่มีโปรเซสหรือกระบวนการหลาย ๆ กระบวนการเข้าทำงานพร้อมกัน (Concurrent Process) ซึ่งเป็นพื้นฐานของการทำงานในระบบหลายโปรแกรม หรือที่เรียกมัลติโปรแกรมมิ่ง (Multiprogramming) และเป็นหน้าที่ของระบบปฏิบัติการที่จะต้องควบคุมการดำเนินงานของกระบวนการเหล่านั้นให้สามารถทำงานร่วมกันอย่างสอดคล้องกัน โดยไม่มีปัญหาใด ๆ เกิดขึ้น ระบบที่มีหลายกระบวนการทำงานพร้อม กันจะประกอบด้วยกลุ่มของกระบวนการ ที่เข้ามาอยู่ในระบบเพื่อประมวลผลดังต่อไปนี้

- กระบวนการของระบบ (OS Process) จะมีหลายกระบวนการ เมื่อมีกระบวนการของระบบเหล่านี้เข้ามา ซีพียูจะต้องทำคำสั่งของระบบก่อน
- กระบวนการของผู้ใช้ (User Process) หน้าที่หลักของซีพียู คือดำเนินการประมวลผลกระบวนการของผู้ใช้ เมื่อมีกระบวนการของผู้ใช้เข้ามา ซีพียูจะทำคำสั่งของผู้ใช้

ซีพียู สามารถดำเนินการประมวลผล หรือเรียกว่า “ดำเนินการ” ทั้งกระบวนการต่าง ๆ ของระบบ และกระบวนการของผู้ใช้ ได้อย่างพร้อม ๆ กันโดยที่ไม่มีปัญหา ดังนั้นในการทำงานของระบบเพื่อให้แน่ใจว่ามีการจัดลำดับในการดำเนินการ กระบวนการเหล่านั้นจะต้องมีกลไกในการจัดการให้แต่ละกระบวนการสามารถทำงานได้อย่างสอดคล้องกันหรือเรียกว่าการประสานกระบวนการ (Process Synchronization) กระบวนการเหล่านั้นต้องสามารถติดต่อสื่อสารกันได้

ความเป็นมา

กระบวนการที่ประสานการทำงานสอดคล้องกัน จะเห็นได้ชัดเจนสำหรับกระบวนการที่มีการทำงานในลักษณะกระบวนการของผู้สร้าง (Producer Process) และกระบวนการของผู้ใช้ (Consumer Process) โดยที่กระบวนการของผู้สร้างมีหน้าที่สร้างข้อมูลขึ้นมา และกระบวนการของผู้ใช้มีหน้าที่ใช้ข้อมูลนั้น ดังตัวอย่างต่อไปนี้

- การพิมพ์ข้อมูลที่เครื่องพิมพ์

กระบวนการของผู้สร้าง (Producer Process) คือโปรแกรมมีหน้าที่ประมวลผลให้ได้ข้อมูลออกมา

กระบวนการของผู้ใช้ (Consumer Process) คือเครื่องพิมพ์ มีหน้าที่นำข้อมูลที่ได้จากโปรแกรมไปพิมพ์ที่เครื่องพิมพ์

- การแปลโปรแกรมภาษาซีให้เป็นภาษาเครื่อง

กระบวนการของผู้สร้างคือตัวแก้ไข (Editor) มีหน้าที่สร้างตัวโปรแกรม (Source Program) ของภาษาซีขึ้นมา

กระบวนการของผู้ใช้คือและผู้สร้าง คือคอมไพเลอร์ (Compiler) ทำหน้าที่เป็นผู้ใช้โดยเอาตัวโปรแกรมของภาษาซี ไปแปลงให้เป็นโปรแกรมภาษาเครื่อง (Object Program) จึงเป็นผู้สร้างโปรแกรมภาษาเครื่องซึ่งจะถูกใช้โดยตัวโหลด (Loader)

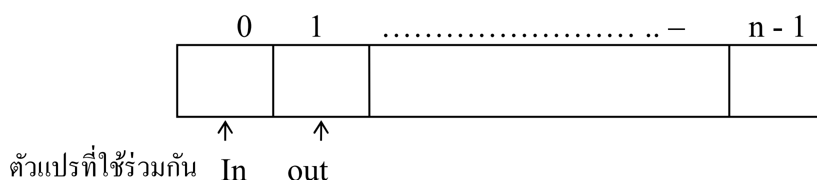
กระบวนการของผู้ใช้คือ ตัวโหลด (Loader) มีหน้าที่เอาโปรแกรมภาษาเครื่องไปรวมกับโปรกแรมส่วนอื่น ๆ ที่จำเป็น แล้วสร้างเป็นโปรแกรมสำหรับประมวลผล (Execute Program)

การที่จะยอมให้กระบวนการของผู้สร้างและ กระบวนการของผู้ใช้ทำงานไปพร้อม ๆ กันได้ เราจำเป็นต้องสร้างบัฟเฟอร์ของรายการข้อมูล (Buffer of Items) ที่จะให้กระบวนการใส่รายการข้อมูล (Items) ลงไปและสามารถนำรายการข้อมูลในนั้นมาใช้ได้ โดยที่กระบวนการของผู้สร้างสร้างรายการข้อมูลใส่ลงบัฟเฟอร์ ขณะเดียวกันกระบวนการของผู้ใช้เอารายการข้อมูลออกจากบัฟเฟอร์นั้นไปใช้ ในขณะที่กระบวนการของผู้สร้างสร้างรายการข้อมูลหนึ่ง ในขณะที่กระบวนการของผู้ใช้ใช้รายการข้อมูลอื่น ๆ ทั้งที่กระบวนการของผู้สร้างและกระบวนการของผู้ใช้จะต้องประสาน (Synchronize) การทำงานกัน ดังนั้น ถ้ากระบวนการของผู้ใช้พยายามที่จะใช้รายการข้อมูลที่กระบวนการของผู้สร้างยังผลิตไม่เสร็จ กระบวนการของผู้ใช้จะต้องรอจนรายการข้อมูลนั้นถูกผลิตจนเสร็จ และข้อมูลที่ผู้สร้างสร้างขึ้นมาต้องจัดเก็บลงในบัฟเฟอร์ ซึ่งบัฟเฟอร์ที่ใช้มี 2 แบบคือ

1. บัฟเฟอร์ แบบไม่จำกัด (Unbounded – Buffer) มีคุณสมบัติคือ
   - ขนาดของบัฟเฟอร์ไม่มีขอบเขตจำกัด
   - กระบวนการของผู้ใช้ อาจจะรอรายการข้อมูลใหม่จากบัฟเฟอร์
   - แต่กระบวนการของผู้สร้างจะผลิตรายการข้อมูลใหม่ไว้เรื่อย ๆ ใส่ในบัฟเฟอร์
   - อาจจะเกิดบัฟเฟอร์ว่างเสมอ ๆ
2. บัฟเฟอร์ แบบจำกัด (Bounded – Buffer) มีคุณสมบัติคือ
   - ขนาดของบัฟเฟอร์มีขอบเขตจำกัด
   - กระบวนการของผู้ใช้จะต้องรอถ้าบัฟเฟอร์ว่าง
   - กระบวนการของผู้สร้างจะต้องรอถ้าบัฟเฟอร์เต็ม

ปัญหาของบัฟเฟอร์แบบจำกัด

เนื่องจากบัฟเฟอร์แบบจำกัดสามารถเก็บรายการข้อมูลได้จำนวนจำกัดด้วยซึ่งบางครั้ง บัฟเฟอร์อาจจะว่างหรือบางครั้งบัฟเฟอร์อาจจะเต็ม ขึ้นกับกระบวนการผู้สร้างและระบวนการผู้ใช้ สามารถแสดงได้ดังรูป 5.1 แสดงบัฟเฟอร์แบบจำกัด



รูปที่ 5.1 บัฟเฟอร์แบบจำกัด

จากรูปที่ 5.1 มีอัลกอริธึมในการทำงานสสำหรับบัฟเฟอร์แบบจำกัด ดังนี้

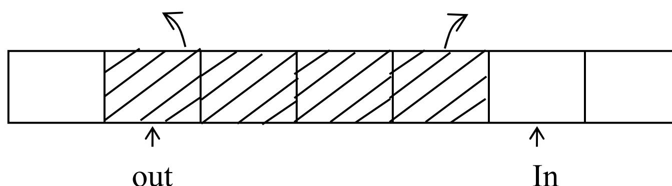
var in, out, n;

type item = ………… ;

var buffer : array [0,….. n - 1] of item ;

in, out : 0….. n – 1;

ให้ตัวแปร in และ out มีค่าเริ่มต้นเป็น 0 ทั้งกระบวนการของผู้สร้างและกระบวนการของผู้ใช้จะใช้บัฟเฟอร์ร่วมกัน โดยมีตัวชี้บัฟเฟอร์ คือ in กับ out โดยที่ in ชี้ที่ว่างถัดไปในบัฟเฟอร์เพื่อที่ใส่รายการข้อมูลใหม่ลงไป ส่วน out ชี้ที่ตำแหน่งแรกของบัฟเฟอร์ที่มีรายการข้อมูลเพื่อที่จะอ่านขึ้นมาดังรูปที่ 5.2 แสดงการใช้บัฟเฟอร์แบบจำกัด และบัฟเฟอร์จะว่างเมื่อ out = in และบัฟเฟอร์จะเต็มเมื่อ (in + 1) mod n = out



รูปที่ 5.2 การใช้บัฟเฟอร์แบบจำกัด

อัลกอริธึม ของกระบวนการของผู้สร้าง

```
repreat
    ⋮
    produce an item in next p.
    ⋮
    while (in + 1) mod n = out do no – op ;
    ⋮
    buffer [in] : = next p ;
    in : = (in + 1) mod n ;
until false ;
```

อัลกอริธึมของกระบวนการของผู้ใช้

```
Repeat
    While in = out do no – op;
    nextc : = buffer [out] ;
    out : = (out + 1) mod n ;
    ⋮
    Consume the item in nextc
    ⋮
Until false ;
```

ในการทำงานของอัลกอริธึมทั้งสองอัลกอริธึม ถ้าเริ่มต้นยอมให้รายการข้อมูล n – 1 รายการ อยู่ใน บัฟเฟอร์ ในเวลาเดียวกัน แล้ว ปรับแก้อัลกอริธึมให้เพิ่มตัวแปร counter เพื่อนับจำนวนรายการข้อมูล โดยการบวกค่าในตัวแปร counter

ให้ค่าเริ่มต้น counter = 0 ในการทำงานจะปรับเปลี่ยนค่า counter ดังนี้

- counter จะถูกเพิ่มทุกครั้งที่มีการเพิ่ม รายการข้อมูลใหม่เข้าใน บัฟเฟอร์
- counter จะถูกลดทุกครั้งที่ รายการข้อมูลถูกนำออกจาก บัฟเฟอร์

ถ้าทั้งสองกระบวนการการทำงานพร้อม ๆ กันดังนี้

produce process	consumer process
repeat	repeat
produce an item in next p.	while counter = 0 do no – op;
⋮	⋮
while counter = n do no- op;	nextc : = buffer [out];
⋮	out : = out + 1 mod n ;
buffer [in] : = next p;	counter : = counter – 1 ;
in : = in + 1 mod n ;	⋮
counter : = counter +1 ;	consume the item in nextc
⋮	⋮
until false ;	until false ;

ถึงแม้ว่าทั้งสองเวลาทำงาน (Runtimes) นี้จะแยกกันอย่างชัดเจน แต่มันอาจจะทำหน้าที่ไม่ถูกต้อง เมื่อมันทำการประมวลผลพร้อม ๆ กัน

สมมุติว่าเริ่มต้นตัวแปร counter = 5

เมื่อกระบวนการผู้สร้างและกระบวนการผู้ใช้ดำเนินการประมวลผล

สำหรับคำสั่ง "counter : = counter + 1" และ "counter : = counter – 1" ถูกทำพร้อมกัน ผลที่ได้จากการประมวลผลของ 2 คำสั่งนี้ ค่าของตัวแปร counter อาจเป็น 4 หรือ 5 หรือ 6 แต่ผลลัพธ์ที่ถูกต้องคือ 5

การพิจารณาการประสานความสอดคล้องของกระบวนการจากคำสั่ง

counter : = counter + 1 อาจจะแปลงเป็น ภาษาเครื่อง มีการทำงานดังนี้

```
register 1  : = counter ;
register 1  : = register 1 + 1 ;
counter     : = register 1
```

เช่นเดียวกัน statement "counter : = counter – 1"

```
register 2  : = counter ;
register 2  : = register 2 – 1  ;
counter     : = register 2
```

เมื่อคำสั่ง counter : = counter + 1 และ คำสั่ง counter : = counter – 1 ประมวลผลพร้อมกัน มีลำดับการทำงานตามคาบเวลาซึ่งกำหนดให้คำสั่งละ 2 คาบเวลาดังนี้

T 0 : ทำคำสั่งกระบวนการผู้สร้าง register 1 : = counter (จะได้ register 1 มีค่า 5)

T 1 : ทำคำสั่งกระบวนการผู้สร้าง register 1 : = register 1 + 1 (จะได้ register 1 มีค่า 6)

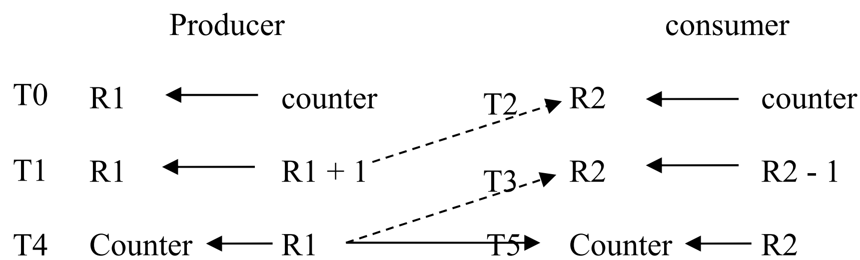
T 2 : ทำคำสั่งกระบวนการผู้ใช้ register 2 : = counter (จะได้ register 2 มีค่า 5)

T 3 : ทำคำสั่งกระบวนการผู้ใช้ register 2 : = register 2 – 1 (จะได้ register 2 มีค่า 4)

T 4 : ทำคำสั่งกระบวนการผู้สร้าง counter : = register 1 (จะได้ counter มีค่า 6)

T 5 : ทำคำสั่งกระบวนการผู้ใช้ counter : = register 2 (จะได้ counter มีค่า 4)

ความสัมพันธ์ในการทำงานของทั้งสองกระบวนการ แสดงได้ดังนี้



ผลลัพธ์ได้ counter = 4 ซึ่งไม่ถูกต้อง

ถ้าสลับลำดับ T4 กับ T5 ผลลัพธ์ จะได้ counter = 6 ไม่ถูกต้องอีกเช่นกัน

สาเหตุที่ counter ไม่ถูกต้อง เพราะเรายอมให้ทั้งสองกระบวนการใช้ตัวแปร counter พร้อมกัน การจะแก้ปัญหานี้จะต้องแน่ใจว่า ณ เวลาใดเวลาหนึ่งจะมี เพียงกระบวนการเดียวเท่านั้นที่เรียกใช้ตัวแปร counter ดังนั้นจึงต้องมีการประสานการทำงานของกระบวนการจะได้กล่าวถึงต่อไป

ปัญหาเขตวิกฤต

การทำงานของระบบที่ประกอบด้วยกระบวนการ n กระบวนการคือ $P_0, P_1 \ldots\ldots P_{n-1}$ แต่ละกระบวนการมีบริเวณของรหัสคำสั่ง (Code) ที่เรียกว่าเขตวิกฤต (Critical Section)

เขตวิกฤตหรือเรียกสั้น ๆ ว่า CS เป็นบริเวณที่กระบวนการใช้ในการเปลี่ยนแปลงค่าตัวแปร ปรับปรุงตารางบันทึกไฟล์ข้อมูล และอื่น ๆ ข้อกำหนดที่สำคัญของระบบคือ เมื่อกระบวนการหนึ่งกำลังทำการประมวลผลในเขตวิกฤต จะไม่ยอมให้กระบวนการอื่นเข้าไปทำการประมวลผลในเขตวิกฤตนั้น ดังนั้น การประมวลผลของกระบวนการในเขตวิกฤตจะมีเพียงกระบวนการเดียวเท่านั้น หรือเรียกว่าการไม่ใช่ร่วมกัน (Mutually Exclusion)

ปัญหาเขตวิกฤต จะต้องออกแบบโปรโตคอล ให้กระบวนการสามารถใช้ร่วมกันได้โดยที่แต่ละกระบวการต้องมีการร้องขอเพื่อเข้าในเขตวิกฤตจะต้องมีการดำเนินงาน 3 ขั้นตอนคือ

ขั้นตอนแรก การขอเข้าเขตวิกฤต เรียกว่าส่วนการเข้า (Entry Section) เป็นส่วนของรหัสคำสั่งที่จะจัดการให้กระบวนการที่ขอเข้าเขตวิกฤต ถูกคัดเลือกเข้าสู่เขตวิกฤตได้

ขั้นตอนที่สอง การออกจากเขตวิกฤต เรียกว่าส่วนการออก (Exit Section) เมื่อกระบวนการได้เข้าทำในเขตวิกฤตแล้วจะต้องออกจากเขตวิกฤตเพื่อให้กระบวนการอื่นได้เข้าทำเขตวิกฤตบ้าง ส่วนนี้จะจัดการให้กระบวนการที่อยู่เขตวิกฤตออกจากเขตวิกฤต

ขั้นตอนที่สาม ส่วนที่ไม่เกี่ยวข้องกับเขตวิกฤต เรียกว่าส่วนที่เหลือ (Remainder Section) คือ รหัสคำสั่งที่ทำงานอื่น ๆ ไม่เกี่ยวกับการเข้าทำในเขตวิกฤต

ข้อกำหนดการแก้ปัญหาของเขตวิกฤต

ในการแก้ปัญหาเขตวิกฤตของกระบวนการที่ดำเนินการร่วมกัน อัลกอริธึมที่ใช้จัดการจะต้องมีคุณสมบัติครบ ทั้ง 3 ข้อต่อไปนี้

1. การไม่ใช่ร่วมกัน (Mutual Exclusion) ในการดำเนินการให้มีได้เพียงกระบวนการเดียว เช่นถ้ากระบวนการ P_0 กำลังดำเนินการอยู่ในเขตวิกฤต ดังนั้นจะต้องไม่มีกระบวนการอื่นเข้าไปดำเนินการในเขตในเขตวิกฤต

2. ความก้าวหน้า (Progress) ถ้าไม่มีกระบวนการที่กำลังดำเนินการอยู่ในเขตวิกฤตและยังมีบางกระบวนการที่ต้องการเข้าเขตวิกฤต ดังนั้นกระบวนการเหล่านั้นจะต้องมีส่วนร่วมในการถูกเลือกว่ากระบวนการไหนจะเข้าไปในเขตวิกฤตต่อไปและการเลือกกระบวนการจะต้องไม่มีการเลื่อนไปโดยไม่มีกำหนด

3. ขอบเขตการรอ (Bounded Waiting) แต่ละกระบวนการต้องมีขอบเขตของการรอเข้าเขตวิกฤต สำหรับกระบวนการที่ร้องขอเข้าเขตวิกฤติจะต้องได้มีโอกาสได้เข้าไปในเขตวิกฤต คือแต่ละกระบวนการมีโอกาสได้เข้า เขตวิกฤตไม่ใช่รอไปเรื่อย ๆ อย่างไม่มีกำหนด

สำหรับอัลกอริธึมในการกำหนดความสอดคล้อง (Synchronization) ของกระบวนการและมีการกำหนดแปรให้ P_i แทนกระบวนการ มีรูปแบบโครงสร้างโดยทั่วไปดังนี้

```
Repeat
        [ ส่วนการเข้า ]
          เขตวิกฤต
        [ ส่วนการออก ]
          ส่วนที่เหลือ
until false ;
```

ส่วนการเข้าและส่วนการออก เป็นส่วนที่สำคัญที่จะทำให้กระบวนการต่าง ๆ ได้เข้าทำในเขตวิกฤตของตัวเอง สำหรับอัลกอริธึมต้องมีคุณสมบัติครบตามข้อกำหนดการแก้ปัญหาของเขตวิกฤต

การแก้ปัญหาของ 2 กระบวนการ

อัลกอริธึมสำหรับจัดการกระบวนการสองกระบวนการในช่างเวลาที่กำหนด คือ $\mathbf{P_0}$ และ $\mathbf{P_1}$ ซึ่งมีการพัฒนาอัลกอริธึมหลายวิธีแต่ละวิธีจะมีประสิทธิภาพต่างกันตามคุณมบัติการแก้ปัญหาของเขตวิกฤตดังนี้

<u>อัลกอริธึม 1</u>

อัลกอริธึมมีการทำงานดังนี้

ให้ทั้งสองกระบวนการ $\mathbf{P_0}$ และ $\mathbf{P_1}$ ใช้ตัวแปร turn ซึ่งเป็นประเภทจำนวนเต็มร่วมกัน

ให้ turn มีค่าเริ่มต้นเป็น 0 (หรือ 1)

ถ้า turn = i ดังนั้น P_i ถูกอนุญาตให้เข้าไปดำเนินการในเขตวิกฤต

```
Repeat
        While turn ≠ i do no – op :
        critical section
        turn  = j ;
        remainder section
until false ;
```

- วิธีนี้มั่นใจได้ว่ามีเพียงกระบวนการเดียวเท่านั้น ณ เวลาหนึ่ง ที่สามารถเข้าไปในเขตวิกฤต

- แต่ไม่ผ่านคุณสมบัติความก้าวหน้า (Progress) เพราะกระบวนการทั้งสองต้องสลับกันเข้าเขตวิกฤตคนละครั้ง

ตัวอย่างเช่น ถ้า turn = 0 และ P_1 พร้อมที่จะเข้าไปในเขตวิกฤต แต่ไม่สามารถเข้าเขตวิกฤตได้ถึงแม้ว่า P_0 จะไม่อยู่ในเขตวิกฤตอาจจะอยู่ในส่วนที่เหลือ

- และไม่ผ่านขอบเขตการรอ ถ้าเขตวิกฤตว่างกระบวนการใด ๆ จะเข้าไม่ได้ ต้องรอ P_j เข้าก่อน

<u>อัลกอริธึม 2</u>

ปัญหาของอัลกอริธึม 1 คือไม่สามารถเก็บข้อมูลที่เกี่ยวกับสถานะของแต่ละกระบวนการได้ มันจะจำได้เพียงว่ากระบวนการไหนถูกอนุญาตให้เข้าเขตวิกฤตเท่านั้น ในการแก้ไขปัญหาโดยการให้ตัวแปร array ของ flag เป็ชนิดบูลีน แทนตัวแปร turn ดังนี้

```
var flag : array [0….1] of boolean ;
```

ทุกสมาชิกของ array ถูกกำหนดค่าเริ่มต้นให้เป็น false

ถ้า flag [i] เป็น true หมายความว่า P_i พร้อมที่จะเข้าใน เขตวิกฤต

```
repeat
        flag [i] : = true ;
        while flag [j] do no – op ;
        critical section
        flag [ i] : = false ;
        remainder section
until false ;
```

สำหรับอัลกอริธึม นี้กระบวนการ P_i ครั้งแรก flag [i] ถูกกำหนดให้มีค่าให้เป็น true หมายความว่า P_i พร้อมที่จะเข้าไปในเขตวิกฤต

- แล้วตรวจสอบว่า กระบวนการ P_j ไม่ได้อยู่ในเขตวิกฤต แต่ถ้า P_j อยู่ในเขตวิกฤต P_i ต้องคอยจนกว่า P_j จะออกจากเขตวิกฤต โดยการให้ flag [j] เป็น false P_i จึงจะเข้าเขตวิกฤตได้
- เมื่อ P_i ออกจาก เขตวิกฤตจะกำหนดค่า flag ให้เป็น false ทำให้กระบวนการอื่นที่กำลังรออยู่สามารถเข้าเขตวิกฤตได้

เพราะฉะนั้นวิธีนี้ ผ่านคุณสมบัติให้มีเพียงกระบวนการเดียวแต่ไม่ผ่านคุณสมบัติความก้าวหน้าของข้อกำหนดกระบวนการเพราะ ณ เวลา

T_0 : P_0 ให้ flag [0] = true

T_1 : P_1 ให้ flag [1] = true

ทั้ง P_0 และ P_1 จะอยู่ในวงรอบของ while ไม่รู้จบ

ถ้าสลับลำดับคำสั่ง สำหรับการกำหนด flag [i] กับการทดสอบค่าของ flag [j] ดังนี้

```
Repeat
        While flag [j] : do no – op ;
        flag [i] : = true ;
            critical section
        flag [ i] : = false ;
            remainder section
    until false ;
```

ยังไม่สามารถแก้ปัญหาได้เพราะ เป็นไปได้ที่ทั้งสองกระบวนการจะเข้าไปในเขตวิกฤต พร้อมกัน ซึ่งขัดกับคุณสมบัติข้อแรก ให้มีเพียงกระบวนการเดียว

อัลกอริธึม 3

รวมแนวความคิดของ อัลกอริธึม 1 และ อัลกอริธึม 2 เข้าด้วยกันจะได้คุณสมบัติครบทั้งสามข้อโดยให้ใช้ตัวแปรร่วมกันดังนี้

```
var flag : array [0,….1]  of boolean  ;      กำหนดตัวแปรร่วม
 Turn := 0..1 ;                               turn มีค่าเป็น 0 หรือ 1
flag [0] = flag [1] = false                  flag [0] และ flag [1] มีค่า false
repeat                                       วงรอบของกระบวนการ เริ่มที่ p_i
    flag [i] : = true ;        / กระบวนการแรกได้เข้า เขตวิกฤต
    turn : = j ;              /ให้ตัวชี้ของอีกกระบวนการหนึ่งเตรียมเข้า
    While (flag [j] and turn = j) do no – op ;  /ถ้ามีกระบวนการ
              อยู่ในเขตวิกฤตกระบวนการอื่นต้องรอก่อน
    critical section
    flag [ i] : = false ; /กระบวการแรกออกจึงให้ มีค่าเป็น false
    remainder section
 until false ;
```

ในการเข้าเขตวิกฤต กระบวนการ P_i ครั้งแรก กำหนดให้ flag [i] เป็น true และ กำหนดให้ turn เป็นของ กระบวนการอื่น (turn = j) ถ้าทั้งสองกระบวนการคือ i และ j พยายามเข้าเขตวิกฤตพร้อมกัน จะมีเพียงกระบวนการเดียวเท่านั้นที่ได้เข้า คือกระบวนการที่ถูกกำหนดค่าให้ turn ทีหลัง เราต้องพิสูจน์ว่า อัลกอริธึม 3 นี้ผ่านคุณสมบัติทั้ง 3 ข้อคือ

1. ให้มีเพียงกระบวนการเดียว

- แต่ละ P_i จะเข้าเขตวิกฤตได้ถ้า flag [j] = false หรือ turn = i
- ถ้าทั้งสองกระบวนการสามารถ ดำเนินการ ใน เขตวิกฤต ในเวลาเดียวกันแสดงว่า

  flag [0] = flag [1] = true

- ที่คำสั่ง while ค่าของ turn เป็นไปได้อย่างใดอย่างหนึ่ง คือ 0 หรือ 1
- P_j จะเข้า เขตวิกฤต ก็ต่อเมื่อ flag [j] = true และ turn = i

  ถ้า P_i เข้าก่อน flag [i] = true และ turn = i P_j ยังเข้าไม่ได้

  ต้องรอ P_i ออกจากเขตวิกฤตแล้วกำหนดให้ flag [i] : = false ;

  เพราะฉะนั้น มีเพียงกระบวนการเดียวได้เข้าเขตวิกฤต

2. ความก้าวหน้า

- P_i สามารถถูกป้องกันไม่ให้เข้าเขตวิกฤตถ้าวงรอบ while มี เงื่อนไข

  flag [j] = true และ turn = j ;

- ถ้า P_j ไม่พร้อมที่จะเข้าเขตวิกฤตflag [j] = false และ P_i สามารถเข้า เขตวิกฤต.
- ถ้า P_j ให้ flag [j] = true ในวงรอบ while ถ้า

  turn = i แล้ว P_i จะเข้าเขตวิกฤต

  turn = j แล้ว P_j จะเข้า เขตวิกฤต.

- ถ้า P_j ออกจากเขตวิกฤตจะกำหนดให้ flag [j] เป็น false ทำให้ P_i เข้าเขตวิกฤตได้
- ถ้า P_j ต้องการเข้าอีกให้ flag [j] เป็น true ให้ turn = i P_i สามารถเข้าเขตวิกฤต

  เขตวิกฤตว่างกระบวนการที่ต้องการเข้าได้เข้า ฉะนั้นผ่านคุณสมบัติก้าวหน้า

และ 3. มีขอบเขตการรอ

- P_i รออยู่ จะเข้าได้โดย P_j รออย่างมากไม่เกิน 1 รอบ (มีขอบเขตการรอ)

$\therefore$ อัลกอริธึม สามารถ แก้ปัญหาวิกฤตได้ สำหรับ 2 กระบวนการ

การแก้ปัญหาระบบที่มีหลายกระบวนการ

จากอัลกอริธึม 3 ใช้สำหรับการแก้ปัญหาเขตวิกฤตของระบบที่มี 2 กระบวนการเท่านั้น สำหรับระบบที่มี n กระบวนการจะใช้อัลกอริธึมที่เรียกว่า เบเกอรีอัลกอริธึม (Bakery Algorithm)

โดยได้แนว ความคิดจากการจัดลำดับลูกค้าที่เข้ามาใช้บริการในร้านขายขนมปัง ไอศกรีม และร้านอื่น ๆ

วิธีการคือเมื่อมีลูกค้าเข้ามารับบริการในร้านจะได้รับบัตรคิวที่เรียงลำดับตัวเลขจากน้อยไปมาก ลูกค้าที่ได้บัตรคิวเลขน้อยจะได้รับบริการก่อน ถ้าลูกค้าได้รับหมายเลขเดียวกันคือถึงพร้อมกันให้เรียงลำดับด้วยชื่อของลูกค้า

ถ้า P_i และ P_j ได้เบอร์คิว และ ถ้า i < j P_i ได้รับบริการก่อน วิธีนี้สามารถทำได้เป็นระบบเพราะแต่ละกระบวนการมีหมายเลขประจำหรือชื่อที่ไม่ซ้ำกันและเรียงกันอยู่แล้ว

โครงสร้างข้อมูลทั่วไปของ ดังนี้

var choosing : array [0…..n-1] of boolean ; {ture, false}

number : array [0…..n-1] of integer ;

ค่าเริ่มต้น ของ choosing เป็น false

ค่าเริ่มต้น ของ number เป็น 0

กำหนดให้

(a, b) < (c, d) ถ้า a < c หรือถ้า a = c และ b < d

$\max(a,......,a_{n-1})$ คือ ฟังก์ชันหาค่ามากที่สุด จาก $a_0 - a_{n-1}$

a_k ซึ่ง $k \geq a_i$ for i = 0,....., n-1

<u>อัลกอริธึมเบเกอรี่</u>

โครงสร้างการทำงานของอัลกอริธึมเบเกอรี่สำหรับจัดการแก้ปัญหาวิกฤตหลายกระบวนการ ต้องสอดต้องผ่านคุณสมบัติข้อกำหนดการแก้ปัญหาของเขตวิกฤตทั้ง **3** ข้อ

```
Repeat
        choosing [i]    : =   true ;  /กระบวนการแรกเข้าเขตวิกฤต
         number  [i]    : =   max (number [0] , number [1] , ... , number [n-1] ) + 1;  /ให้บัตรคิว
                                              คนต่อไป
        choosing [i]    : =   false ;           /รอ
        for     j  : = 0    to n-1              / สำหรับ n  กระบวนการ
                do begin
                        while choosing [j] do no – op ; /เช็คว่าถ้ามีใครอยู่ในเขตวิกฤต จะต้องรอ
                          while number  [j] ≠ 0        /กระบวนการที่ไม่เป็น0 จะได้เข้าเขตวิกฤต
                            and (number [j] , j) < (number [j] , i) do no – op ;
                                              /เชคลำดับและชื่อ ผู้ที่มีสิทธิ์สูงจะได้เข้าเขตวิกฤต
```

```
            end ;
            critical section          / ดำเนินการในส่วนของเขตวิกฤต
            number [i] : = 0 ;        / ออกจากเขตวิกฤต
            remainder section         / ดำเนินการในส่วนที่เหลือ
      until false ;
```

พิสูจน์คุณสมบัติของ อัลกอริธึมเบเกอรี่

1. ให้มีเพียงกระบวนการเดียว

ถ้า P_i อยู่ในเขตวิกฤตและ P_k พยายามจะเข้าเขตวิกฤต

P_k ในคำสั่ง while ที่สอง ให้ j = i

number [i] $\neq$ 0 และ

(number[i], j) < (number[k], k)

ดังนั้นจึงมี การวนรอบใน while ต่อไปจนกระทั่ง P_i ออกจากเขตวิกฤต

2. ความก้าวหน้าและขอบเขตการรอ

กระบวนการเข้าเขตวิกฤตแบบมาก่อนได้ก่อน และแต่ละกระบวนการรอไม่เกิน n-1 รอบ

การประสานทางฮาร์ดแวร์

ในการแก้ปัญหาวิกฤตบางระบบสามารถใช้ฮาร์ดแวร์ ช่วยในการแก้ปัญหาได้ ในที่นี้จะกล่าวถึง คำสั่งง่าย ๆ เกี่ยวกับฮาร์ดแวร์ในการแก้ปัญหาเขตวิกฤต

แก้ปัญหาเขตวิกฤตสามารถแก้ปัญหาได้อย่างง่าย ๆ ถ้าเราไม่อนุญาตให้เกิดอินเทอรัพในขณะที่มีการกำหนดตัวแปรร่วมซึ่งการทำแบบนี้ทำไม่ได้เสมอไป มีหลายเครื่องที่มีคำสั่งพิเศษสำหรับฮาร์ดแวร์โดยทำการ ตรวจสอบและแก้ไขค่าของคำ หรือสลับค่าของคำ 2 คำโดยอัตโนมัติ คำสั่งพิเศษเหล่านี้สามารถใช้แก้ปัญหาเขตวิกฤตได้ ซึ่งมีหลายวิธีดังนี้

วิธีการใช้คำสั่ง Test – and – set

```
Function Test – and – set (var target : boolean) : boolean ;
      begin
            Test – and – set : = target ;
            target : = true ;
      end ;
```

ถ้าเครื่องเอื้อต่อการใช้คำสั่งทางฮาร์ดแวร์ " Test – and – set " เราสามารถแสดงคุณสมบัติข้อแรกคือ ให้มีเพียงกระบวนการเดียว โดยการกำหนดให้ lock เป็นตัวแปรแบบบูลีน และมีค่าเริ่มต้นเป็น false

โครงสร้างของกระบวนการ P_i

```
Repeat
          While Test – and – set (lock) do no – op ;

          critical section

          lock : = false ;

          remainder section

until false ;
```

จะแสดงว่ามีคุณสมบัติให้มีเพียงกระบวนการเดียว

- P_0 เข้าเขตวิกฤตได้ เพราะ lock เป็น false
- ขณะที่ P_0 อยู่ในเขตวิกฤต lock เป็น true P_1 จะเข้าเขตวิกฤตเข้าไม่ได้
- เมื่อ P_0 ออกจากเขตวิกฤต lock เป็น false P_1 จึงเข้าเขตวิกฤตได้

ดังนั้นมีกระบวนการอยู่ในเขตวิกฤตกระบวนการเดียว

การสลับคำสั่ง คือ สลับค่าในคำ 2 คำดังนี้

```
procedure swap (var a, b : boolean) ;
          var temp : Boolean ;
          begin
                    temp : = a ;
                    a : = b ;
                    b : = temp ;
          end ;
repeat
     waiting [i] : = true ;
     key : = true ;
     while waiting [i] and key
        do key : =  Test – and – set (lock) ;
     waiting [i] : = false ;
```

critical section

```
j  : = i + 1 mod n ;
while (j ≠ i) ; and (not waiting [j] do j : = j + 1 mod n) ;
if  i  = j then lock : = false ;
        else waiting [j] : = false ;
```

remainder section

until false ;

พิสูจน์ข้อกำหนด ให้มีเพียงกระบวนการเดียว

- P_i สามารถเข้าเขตวิกฤตได้ถ้า waiting [i] = false หรือ key = false

  key จะเป็น false เมื่อดำเนินการคำสั่ง Test – and – set

  กระบวนการแรก ดำเนินการ Test – and – set ได้ key = false

  กระบวนการอื่น ๆ ต้องรอ

- ตัวแปร waiting [i] จะเป็น false เมื่อกระบวนการอื่น ออกจากเขตวิกฤต

  มีเพียง waiting [i] เดียว ที่ถูกกำหนด เป็น false

  เพราะฉะนั้น มีคุณสมบัติเป็นมีเพียงกระบวนการเดียว

พิสูจน์ข้อกำหนด ก้าวหน้า

- กระบวนการที่อยู่นอกเขตวิกฤตจะได้เข้าก็ต่อเมื่อ

  กำหนด lock ให้เป็น false เริ่มแรกไม่มีกระบวนการในเขตวิกฤต

  กำหนด waiting [i] ให้เป็น false กระบวนการอื่นออกจากเขตวิกฤต เพราะฉะนั้นถ้าไม่มีกระบวนการดำเนินการอยู่ในเขตวิกฤตกระบวนการที่รออยู่ข้างนอกสามารถเข้าได้

พิสูจน์ข้อกำหนดขอบเขตการรอ

- เมื่อกระบวนการออกจากเขตวิกฤตจะรับค่า จากอะเรย์ของ waiting แบบวงรอบตามลำดับ (i + 1,i + 2,...,n –1, 0, ..., i – 1) และตัดสินให้กระบวนการแรกในลำดับที่มีค่า

  ส่วนการเข้า waiting [i] = true เป็นกระบวนการที่จะเข้าเขตวิกฤตต่อไป

เพราะฉะนั้นกระบวนการใดๆ ที่ รอเข้าเขตวิกฤตจะได้เข้าภายในช่วง n – 1

วิธีการเซมาฟอร์

วิธีการแก้ปัญหาเขตวิกฤต ที่กล่าวมาใช้แก้ปัญหาที่ซับซ้อนได้ไม่ง่ายนัก เราจึงใช้ วิธีใหม่ ที่เรียกว่า เซมาฟอร์ (Semaphore)

เซมาฟอร์ S คือ ตัวแปรแบบ integer ที่ใช้แก้ปัญหาเขตวิกฤต ที่กำหนดค่าเริ่มต้น และใช้งานโดย ผ่าน 2 คำสั่งที่เป็นมาตรฐานคือ wait และ signal

อัลกอริธึมของ wait and signal

```
wait (S) :      while S ≤ 0  do no-op;
                S  :=  S-1;
signal (S)  :   S  :=  S+1;
```

เป็นการเปลี่ยนแปลงค่าของจำนวนเต็มของเซมาฟอร์ ในการกระทำของ wait และ Signal เมื่อกระบวนการหนึ่ง แก้ไขค่าเซมาฟอร์ จะไม่มีกระบวนการใดสามารถแก้ไขค่าเซมาฟอร์นั้น ในกรณีของ wait (S) จะทดสอบค่าของ ($S \leq 0$) และ แก้ไข (S:=S-1) และต้องถูก ดำเนินการ โดยปราศจากการอินเทอรัพ จะแสดงให้เห็นว่า การกระทำ เหล่านี้สามารถถูกสร้างได้

การใช้งานเซมาฟอร์

- เซมาฟอร์ สามารถถูกใช้กับปัญหาเขตวิกฤตของหลายกระบวนการ (n- process) โดยใช้ตัวแปรร่วมของเซมาฟอร์ คือ mutex มีค่าเริ่มต้นเป็น 1 แต่ละกระบวนการ Pi มีโครงสร้างดังนี้

```
Repeat
        wait (mutex) ;
        critical section
        signal (mutex) ;
remainder section
until false ;
```

เซมาฟอร์ สามารถใช้แก้ปัญหา การประสานกระบวนการที่เปลี่ยนไป ตัวอย่างเช่น มี 2 กระบวนการกำลังรันพร้อมกัน โดยที่ P1 ทำคำสั่ง S1 และ P2 ทำคำสั่ง S2 สมมุติว่า เราต้องการให้ S2 ถูก ดำเนินการหลังจาก S1 ทำเสร็จแล้ว เราสามารถทำได้โดยให้ P1 และ P2 ใช้ตัวแปรประสานของเซมาฟอร์ร่วมกัน คือ VAR SYNCH : semaphore = 0

กำหนดค่าเริ่มต้นให้ SYNCH Semaphore เป็น 0 และ เพิ่มคำสั่ง

→ S1 ;
signal (SYNCH)
} ในกระบวนการP1

→ wait (SYNCH)
S2 ;
} ในกระบวนการP2

เนื่องจาก SYNCH ถูกกำหนดให้เป็น 0

P2 จะ ดำเนินการ S2 หลังจาก P1 ได้ ดำเนินการ signal (SYNCH) ซึ่งทำหลังจาก S1 เสร็จ

การสร้างเซมาฟอร์ (Semaphore Implementation)

ข้อเสียของการปัญหาเขตวิกฤตที่กล่าวมาคือการรอคอย ในขณะที่กระบวนการหนึ่งอยู่ในเขตวิกฤตกระบวนการอื่น ๆ พยายามที่จะเข้าเขตวิกฤตจะต้องเข้าวงรอบต่อไปในส่วนคำสั่งทางข้าเขตวิกฤต จึงเป็นปัญหากับระบบมลัติโปรแกรมมิ่งที่มีซีพียูเดียวและถูกใช้ร่ามกันระหว่างหลายกระบวนการ สามารถดำเนินการได้ดังนี้

- เราสามารถแก้ไขข้อกำหนดของการกระทำของ wait และ signal เมื่อกระบวนการดำเนินการ คำสั่ง wait และพบว่าค่า semaphore ไม่เป็นค่าบวก มันต้องรอ busy waiting กระบวนการสามารถ block ตัวเองได้

- การทำ block ตัวเอง จะนำกระบวนการใส่ลงไปใน waiting queue ที่เกี่ยวเนื่องกับเซมาฟอร์ และสถานะของกระบวนการเปลี่ยนไปเป็นสถานการณ์รอ (Waiting State)

- ต่อจากนั้นการควบคุมถูกส่งไปยังตัวจัดตารางเวลาของซีพียู (CPU Scheduler) เลือกกระบวนการนั้น ดำเนินการ

- กระบวนการที่ถูก blocked ที่กำลังรอเซมาฟอร์ S จะถูกให้เริ่มทำใหม่จากคำสั่ง signal ของกระบวนการอื่น

- กระบวนการที่ถูกให้เริ่มทำโดยคำสั่ง wakeup จะเปลี่ยนสถานะของกระบวนการจากการรอ (Waiting) เป็นพร้อม (Ready) และถูกใส่ลงในคิวพร้อม

- ในการสร้างเซมาฟอร์ภายใต้นิยามนี้ เรากำหนดเรคคอร์ดของเซมาฟอร์ ดังนี้ type semaphore = record

```
value : integer ;
L : list of process ;
end ;
```

- แต่ละ semaphore มีตัวแปรชนิด integer และมีรายการ (List) ของกระบวนการ เมื่อกระบวนการ wait บน semaphore มันจะถูกเพิ่มเข้าไปในรายการของกระบวนการ

- การกระทำ signal ย้ายกระบวนการออกจากรายการของ waiting กระบวนการหนึ่ง กระบวนการสามารถนิยามการทำเซมาฟอร์ ได้ดังนี้

```
wait (S) ;    S. value : = S. value – 1 ;
              if S. value < 0
                then begin
                     add this process to S.L ;
                     block ;
                     end ;
Signal (S) : S.value : = S. value + 1 ;
              if S. value ≤ 0
                then begin
                     remove a process P from S.L ;
                     wakeup (P) ;
                     end ;
```

- คำสั่ง block จะหยุดกระบวนการชั่วคราว
- wakeup (P) จะส่งค่าการดำเนินการของกระบวนการPที่ถูก block ให้ทำต่อ
- ทั้งสองการดำเนินการถูกกำหนดโดยระบบปฏิบัติการเช่นเดียวกับคำสั่งเรียกระบบ
- จากนิยามของเซมาฟอร์กับการรอแบบไม่ว่าง (Busy Waiting) ค่าของเซมาฟอร์ไม่เคยมีค่าเป็นลบ (Negative) แต่การสร้างนี้ ค่าเซมาฟอร์อาจมีค่าเป็นลบได้ จำนวนของกระบวนการที่รอใน เซมาฟอร์มีมาก จะมีการทำคำสั่ง wait มาก
- ลิสต์ (List) ของกระบวนการสามารถสร้างโดยลิงค์กับฟิลด์ในแต่ละบล็อกควบคุมกระบวนการแต่ละเซมาฟอร์ มีค่าเป็น integer และชี้ไปยังลิสต์ของบล็อกควบคุม มีทางเดียวที่จะเพิ่มหรือลดกระบวนการจากลิสต์ ซึ่งแน่ใจได้ว่ามีขอบเขตการรอจะเป็นแบบเข้าก่อนออกก่อน (FIFO)
- เซมาฟอร์ มีตัวชี้ไปยังต้นลิสต์และท้ายลิสต์ของคิว อาจใช้การจัดการคิวแบบอื่น ๆ เช่น เข้าก่อนออกก่อน หรือเข้าทีหลังออกก่อน หรือแบบตามลำดับสำคัญ หรือแบบอื่น ๆ
- การใช้เซมาฟอร์ที่ถูกต้องไม่ขึ้นกับลักษณะของการจัดการคิวสำหรับลิสต์เซมาฟอร์

สิ่งสำคัญของเซมาฟอร์คือไม่ให้มีสองกระบวนการจะไปดำเนินการใน wait และ signal ในเซมาฟอร์เดียวกัน ณ เวลาเดียวกัน กรณีเนี้ปืนเป็นปัญหาเขตวิกฤตแบบหนึ่ง ซึ่งมีวิธีจัดการ 2 แบบ คือ

- แบบแรก ระบบที่มีซีพียูเดียว เราสามารถขัดขวางการอินเทอรัพ ในช่วงเวลาที่กำลังทำคำสั่ง wait และ signal นี้ กระบวนการอื่นจะไม่สามารถขัดขวางการทำอินเทอรัพได้
- แบบที่สอง ระบบที่มีซีพียูหลายตัว การขัดขวางการอินเทอรัพไม่สามารถทำได้ เนื่องจาก อาจมีการประมวลผลคำสั่งอื่นบนซีพียูอื่น ถ้าไม่มีคำสั่งพิเศษสำหรับฮาร์ดแวร์จัดการเราสามารถใช้ซอฟต์แวร์จัดการสำหรับปัญหาเขตวิกฤตซึ่ง ประกอบด้วย wait และ signal

ในการทำ wait และ signal เราไม่สามารถกำจัดการรอแบบไม่ว่าออกได้อย่างสมบูรณ์ จะต้องกำจัดการรอแบบไม่ว่าง จากส่วนการเข้าของเขตวิกฤตของโปรแกรมประยุกต์ จากนั้นเราก็มันให้อยู่ภายในเขตวิกฤตของการกระทำ wait และ signal ซึ่งเป็นช่วงเวลาสั้น ๆ (ประมาณไม่เกิน 10 คำสั่ง) ดังนั้นเขตวิกฤตอาจจะว่างตลอดเวลา เสมือนไม่มีการรอแบบไม่ว่างเกิดขึ้น หรืออาจจะเกิดเป็นเวลาสั้นมาก ๆ เมื่อเทียบกับเขตวิกฤตของโปรแกรมอื่นทั่วไป ถ้าการคอบครองเขตวิกฤตกินเวลานานมากอาจเป็นชั่วโมง จึงไม่ควรใช้การป้องกันแบบมีการรอแบบไม่ว่าง

วงจรอับและการรอแบบไม่สิ้นสุด (Deadlocks and Starvation)

ในการสร้างเซมาฟอร์ด้วยการจัดลำดับการรอ (Waiting Queue) อาจเกิดเหตุการณ์สองกระบวนการหรือมากกว่าสองกระบวนการเกิดการรอกันแบบไม่สิ้นสุด (Starvation) มีสาเหตุเนื่องมาจากมีเพียง 1 กระบวนการที่กำลังรอทำการ signal เราเรียกเหตุการณ์อย่างนี้ว่าวงจรอับหรือการติดตาย (Deadlocked)

พิจารณาระบบที่ประกอบด้วยสองกระบวนการคือ P0 และP1 ต่างเข้าทำสองเซมาฟอร์ S และ Q ดังนี้

P0	P1
wait (S) ;	wait (Q) ;
wait (Q) ;	wait (S) ;
⋮	⋮
Signal (S) ;	Signal (Q) ;
Signal (Q) ;	Signal (S) ;

สมมุติว่า P_0 ทำคำสั่ง wait (S) และ P_1 ทำคำสั่ง wait (Q)
เมื่อ P_0 ทำคำสั่ง wait (Q) จะต้องรอจนกระทั่ง P_1 ทำคำสั่ง signal (Q)
เมื่อ P_1 ทำคำสั่ง wait (S) จะต้องรอจนกระทั่ง P_0ทำคำสั่ง signal (S)

ทั้งสองการกระทำ signal P_0 และ P_1 ไม่สามารถทำคำสั่ง ได้เรียกว่า เกิดวงจรอับคือเมื่อทุกกระบวนการอยู่ในกลุ่มต่างอยู่ในสถานะรอ เหตุการณ์ที่เป็นสาเหตุการรอโดยกระบวนการอื่นในกลุ่มนั้น เหตุการณ์เช่นนี้เราจะเกี่ยวข้องกับการถือครอง และการปล่อยทรัพยากร

ปัญหาอื่น ๆ ที่เกี่ยวข้องกับวงจรอับ (Deadlocks) คือการบล็อกที่ไม่มีกำหนดหรือเรียกว่าการแช่แข็ง (Indefinite Blocking) หรือการรอที่ไม่มีที่สิ้นสุด อาจจะเกิดขึ้นถ้าเราเพิ่มหรือลบกระบวนการออกจากลิสต์แบบลำดับที่เข้าทีหลังออกก่อน

ต้นแบบปัญหาของการประสานงาน

ปัญหาในการประสานงาน ใช้เป็นตัวแทนปัญหาต่าง ๆ ในระบบการทำงานแบบขนาน ต้นแบบปัญหาของการประสานงาน(Classical Problems of Synchronization)สามารถใช้ทดสอบขั้นตอนวิธีใหม่ ๆ ที่สร้างขึ้นมาเพื่อจัดการปัญหาการประสานงานได้ ดังต่อไปนี้

ปัญหาบัฟเฟอร์แบบจำกัด (The Bounded – Buffer Problem)

การมีที่พักข้อมูล (Spool) ประกอบด้วย n ตำแหน่งแต่ละตำแหน่ง จุข้อมูล 1 ตัว ให้ตัวแปร mutex ของเซมาฟอร์ที่จัดการเกี่ยวกับการให้มีโปรแกรมเดียว (Mutual Exclusion) สำหรับการเข้าใช้ข้อมูลจากที่พักข้อมูลแต่ละตำแหน่ง และกำหนด ค่าเริ่มต้นของตัวแปรที่ใช้ร่วมกันคือ mutex = 1 empty = n และ full = 0

คำสั่งของกระบวนการผู้ผลิต

```
repeat
    produce an item in next p
    wait (empty) ;
    wait (mutex) ;
        :
    add next p to buffer
    signal (mutex) ;
    signal (full) ;
until false ;
```

คำสั่งของกระบวนการผู้ใช้

```
repeat
    wait (full) ;
    wait (mutex) ;
        :
    remove an item from buffer to next c
        :
    signal (mutex) ;
    signal (empty) ;
        :
    consume the item in next c
        :
until false ;
```

ข้อสังเกตจะเห็นว่าคำสั่งของกระบวนการผู้ผลิต และคำสั่งของกระบวนการผู้ใช้จะสมมาตรกัน สามารถใช้เป็นโปรแกรมร่วมกัน จะได้ว่ากระบวนการผู้ผลิตเมื่อทำการผลิตแล้วส่งค่า full ให้กระบวนการผู้ใช้ หรือกระบวนการผู้ใช้ เมื่อใช้แล้วจะส่งค่า empty ให้กระบวนการผู้ผลิต

ปัญหาผู้อ่านและผู้เขียน (Readers and Writers Problem)

ข้อมูลอาจอยู่ในรูปแบบของไฟล์ หรือเรคคอร์ดถูกใช้ร่วมกันระหว่างหลายกระบวนการที่ทำงานพร้อมกัน บางกระบวนการต้องการอ่านข้อมูลมาใช้งาน บางกระบวนการต้องการ แก้ไขข้อมูล เราแบ่งกลุ่มกระบวนการนี้ออกเป็น สองแบบคือ กระบวนการที่อ่านอย่างเดียว เรียกว่าผู้อ่าน (Reader) กระบวนการที่ต้องการแก้ไขเรียกว่าผู้เขียน (Writer) ผู้อ่าน อ่านข้อมูลร่วมมาใช้พร้อมกันจะไม่มีปัญหา แต่ ถ้ากระบวนการใด กระบวนการหนึ่งที่เป็นผู้เขียน กระทำข้อมูลร่วมพร้อมกัน อาจจะเกิดปัญหา ดังนั้นเราต้องการให้ผู้เขียนมีการป้องกันการประสานการเข้าใช้ข้อมูลร่วม เรียกว่า ปัญหาผู้อ่าน- ผู้เขียน

ข้อกำหนดปัญหาผู้อ่าน- ผู้เขียน

1. ถ้าข้อมูลร่วมมีผู้อ่านอื่นกำลังอ่านอยู่ ผู้ที่เข้ามาใหม่สามารถใช้ข้อมูลร่วมได้ ถึงแม้จะมีผู้เขียนรอใช้อยู่

2. ถ้าผู้เขียนพร้อม ผู้เขียนสามารถใช้ข้อมูลร่วมได้ให้เร็วที่สุด แต่ผู้อ่านจะเข้าไปอ่านไม่ได้ต้องรอจนกว่าผู้เขียนจะเขียนเสร็จ

การทำทั้งสองแบบอาจเกิดการการรอแบบไม่จบ คือ ผู้เขียนอาจจะรอไม่รู้จบ หรือผู้อ่านอาจจะรอไม่รู้จบ

<u>การแก้ปัญหา ผู้อ่าน - ผู้เขียน</u>

ต้อง กำหนดตัวแปร ต่อไปนี้

```
var mutex , wrt : semaphore ;
readcount  :  integer ;
```

ตัวแปร mutex and wrt ของ semaphore ถูก กำหนดค่าเริ่มต้น เป็น 1 และตัวแปร readcount ให้ เป็น 0 โดยที่ mutex ใช้ป้องกันตัวแปร readcount และ wrt ใช้ป้องกันผู้เขียน ผู้อ่านแรกและสุดท้ายในการเข้าออกเขตวิกฤต

โครงสร้างทั่วไปของกระบวนการผู้เขียน

```
wait (wrt) ;
        . . .
        writing is performed
```

```
        . . .
    signal (wrt) ;
```

∴ ฟังก์ชัน wait (wrt) เป็น เซมาฟอร์ที่ให้มีเพียงการบวนการเดียว สำหรับการ write

โครงสร้างทั่วไปของกระบวนการผู้อ่าน

```
    wait (mutex) ;
        readcount := readcount + 1 ;
        if readcount = 1 then wait (wrt) ;  /wrt ที่เข้า เขตวิกฤต ถูกใช้โดย first reader
    signal (mutex) ;
        . . .
        reading is performed
        . . .
    wait (mutex) ;                        /ใช้ป้องกันการ update read count
      readcount := readcount – 1 ;
      if readcount = 0 then signal (wrt) ; / wrt ที่ออกจาก เขตวิกฤต ถูกใช้โดย last reader
      signal (mutex) ;
```

semaphore wrt เป็นทั้งของการบวนการอ่านและเขียน

semaphore mutex ใช้ให้แน่ใจว่า มีเพียงกระบวนการเดียว เมื่อตัวแปร readcount ถูกแก้ไข

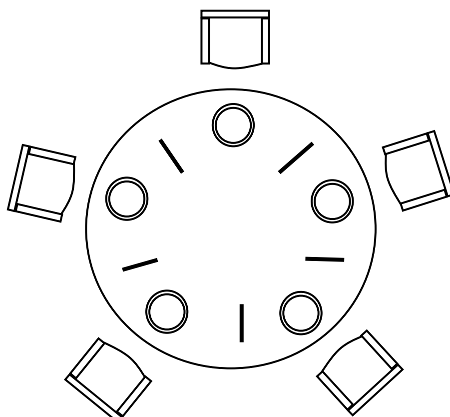
readcount เก็บข้อมูลว่ามีกี่กระบวนการที่ทำการอ่านพร้อมกัน

<u>หมายเหตุ</u>

- ถ้ามีค่า 0 แสดงว่าผู้เขียนอยู่ในเขตวิกฤตและมี n ผู้อ่าน กำลังรอผู้อ่าน 1 คนรออยู่ ที่ semaphore wrt (เพราะ wrt = 0) และ มีผู้อ่าน n – 1 คนรอที่ semaphore mutex (เพราะ mutex = 0) และ เมื่อ ผู้เขียน ทำคำสั่ง signal (wrt) จะทำการ ปลุกผู้อ่านที่กำลังรอทั้งหมดหรือผู้เขียน 1 คน ที่รออยู่ขึ้นกับตัวจัดตารางแถวคอย

ปัญหาการรับประทานอาหารเย็นของนักปราชญ์ (The Dining – Philosophers Problem)

มีนักปราชญ์ 5 คนนั่งรอบโต๊ะกลมรับประทานอาหารเย็นโดยมีหน้าที่คิดและกิน แต่ละคน มีถ้วยเป็นของตัวเองแต่มี มีตะเกียบ 5 ข้างวางอยู่ระหว่างชาม เมื่อหิวนักปราชญ์แต่ละคนพยายามที่ จะหยิบตะเกียบ 2 อันที่อยู่ใกล้ตัวด้วยมือซ้ายและขวา อาจมีนักปราชญ์ที่ได้ตะเกียบข้างเดียว บาง คนไม่ได้เลย ถ้าตะเกียบไม่ว่างเขาต้องรอ ถ้าใครอิ่มจะวางงตะเกียบและนั่งคิดต่อไป ดังรูปที่ 5.3



รูปที่ 5.3 สถานการณ์โต๊ะอาหารเย็นของนักปราชญ์

ปัญหาการรับประทานอาหารเย็นของนักปราชญ์เป็นปัญหาแบบประสาน (Synchronization Problem) วิธีหนึ่งที่จะแก้ปัญหาอย่างง่าย ๆ คือใช้เซมาฟอร์ นักปราชญ์ พยายามที่จะแย่งตะเกียบโดยการทำเซมาฟอร์ wait และเขาก็วางตะเกียบโดยการทำ signal กำหนดตัวแปรร่วมดังนี้

var chopstick : array [0,4] of semaphore ;

ทุกสมาชิกของ chopstick ถูก กำหนดให้มีค่าให้เริ่มต้นเป็น 1

โครงสร้างของนักปราชญ์คนที่ i

```
Repeat
        wait (chopstick [ i ]) ;
        wait (chopstick [ i + 1 mod 5] ) ;
        ...
        eat
        ...
        signal (chopstick [ i ]) ;
        signal (chopstick [ i + 1 mod 5] ) ;
        ...
        think
        ...
until false ;
```

ถึงแม้ว่าการแก้ปัญหานี้จะรับรองว่าไม่มีสองคนติดกัน กินข้าวพร้อมกัน แต่อาจไม่ได้กินทั้งหมด เพราะว่าเกิด วงจรอับ (Dead lock) สมมุติว่า ทั้ง 5 คนหิวพร้อมกัน และคว้าตะเกียบทางซ้ายทุกสมาชิกของ chopstick จะมีค่าเป็น 0 เมื่อนักปราชญ์พยายามที่จะแย่งตะเกียบทางขวาเขาจะรอตลอดกาล

อัลกอริธึม ที่แน่ใจได้ว่าจะไม่เกิดวงจรอับ

1. ยอมให้ นักปราชญ์นั่งได้ 4 คน
2. ยอมให้ นักปราชญ์หนึ่งคนหยิบตะเกียบก็ต่อเมื่อตะเกียบทั้งสองข้างว่าง
3. ให้การแก้ปัญหาที่เหมือนกัน (Symmetric Solution) คือการให้นักปราชญ์หมายเลขคี่หยิบตะเกียบทางซ้ายก่อน แล้วจึงหยิบตะเกียบทางขวา และให้นักปราชญ์หมายเลขคู่หยิบตะเกียบทางขวาก่อน แล้วจึงหยิบทางซ้าย

ปฏิบัติการที่ 5

1. ให้นักศึกษาแต่ละกลุ่มเขียนโปรแกรมสำหรับจัดการกระบวนการสองกระบวนการที่มีการใช้ตัวแปรร่วมกันในช่างเวลาที่กำหนดและให้เป็นไปตามข้อกำหนดการแก้ปัญหาของเขตวิกฤต โดยใช้แนวคิดจากอัลกอริธึมที่ได้ศึกษามา
2. แต่ละกลุ่มเสนอแนวคิด และแลกเปลี่ยนเรียนรู้กัน

คำถามท้ายบท

1. อธิบายลักษณะการทำงานร่วมกันของกระบวนการของผู้สร้าง และกระบวนการของผู้ใช้ และพบว่ามีปัญหาอะไรบ้าง
2. เขตวิกฤตของกระบวนการที่ร่วมกันทำงานคืออะไร และมีข้อกำหนดการแก้ปัญหาของเขตวิกฤต อย่างไรบ้าง
3. จงอธิบายวิธีการจัดการ บัฟเฟอร์แบบไม่จำกัด และ แบบจำกัด
4. จงอธิบายปัญหาของบัฟเฟอร์แบบจำกัด
5. การใช้ฮาร์ดแวร์ ช่วยในการแก้ปัญหาเขตวิกฤติมีวิธีการหลายวิธี ให้อธิบายมา 1 วิธี
6. จงอธิบายหลักการหรือวิธีการของการจัดการปัญหาการประสานงานปัญหาบัฟเฟอร์แบบจำกัด
7. จงอธิบายวิธีการแก้ปัญหาเขตวิกฤตด้วยวิธีเซมาฟอร์ (Semaphore)
8. จงอธิบายวิธีการแก้ปัญหาเขตวิกฤตด้วยวิธีวิธีการใช้คำสั่ง Test – and – set
9. จงอธิบายวิธีการแก้ปัญหาเขตวิกฤตด้วยวิธีปัญหาผู้อ่านและผู้เขียน (Readers and Writers Problem)
10. จงอธิบายวิธีการแก้ปัญหาเขตวิกฤตด้วยวิธีปัญหาการรับประทานอาหารเย็นของนักปราชญ์ (The Dining – Philosophers Problem)