2.3 โปรเซสซิงโครไนเซชั่น (Process Synchronization)

share data , share resource

2.3.1 ผู้ผลิต และ ผู้บริโภค (Producer and Consumer)

```
var   buffer   : array [0..n-1] of item ;
      in , out  : 0 .. n-1 ;
      counter  : 0 .. n ;
```

Producer process	Consumer process
repeat …. Producer an item in nextp …. **while** counter = n **do** no-op; buffer[in] := nextp ; in := (in +1) **mod** n ; counter := counter +1 ; **until** false;	**repeat** **while** counter = 0 **do** no-op; nextc := buffer[out] ; out := (out +1) **mod** n ; counter := counter –1 ; …. Consume the item in nextc …. **until** false;

รูปที่ 2.3.A โปรเซสผู้ผลิต และ โปรเซสผู้บริโภค

- "buffer" ซึ่งมีขนาด n ตำแหน่ง คือ 0 ถึง n-1 และเป็นตัวแปรแบบใช้ร่วมกันได้ทุกฟังก์ชั่น (share variable)
- "in" เป็นตัวแปรที่ใช้งานเฉพาะในโปรเซสผู้ผลิตเท่านั้น(local variable) ซึ่งมีค่าได้ตั้งแต่ 0 ถึง n-1
- "out" เป็นตัวแปรที่ใช้งานเฉพาะในโปรเซสผู้บริโภคเท่านั้น(local variable)ซึ่งมีค่าได้ตั้งแต่ 0 ถึง n-1
- "count" ตัวแปรแบบใช้ร่วมกันได้ทุกฟังก์ชั่น ซึ่งมีค่าได้ตั้งแต่ 0 ถึง n

โปรเซสผู้ผลิต : generate data (circular buffer)

โปรเซสผู้บริโภค : use data (circular buffer)

สิ่งที่เป็นปัญหาคือ

คำสั่ง “counter := counter +1”

และ “counter := counter -1”

เมื่อมีการทำงานไปพร้อมๆกันของทั้งสองโปรเซส ดังนี้

ถ้า “counter” ขณะนั้นมีค่าเท่ากับ 5 เมื่อมีการทำงานของคำสั่งดังกล่าว อาจจะทำให้ “counter” มีค่าผลลัพธ์เป็นได้ทั้ง 4 หรือ 5 หรือ 6 ก็ได้ หากพิจารณาคำสั่งดังกล่าวในเชิงภาษาเครื่อง(Machine Language) อาจจะเป็นดังในรูปที่ 2.3.B

counter := counter +1 ;	**counter := counter -1 ;**
$register_1$:= counter ; $register_1$:= $register_1$ +1 ; counter := $register_1$	$register_2$:= counter ; $register_2$:= $register_2$ -1 ; counter := $register_2$

รูปที่ 2.3.B แสดงลักษณะการทำงานในระดับล่างของแต่ละคำสั่งทั้งสองเทอม

ซึ่ง “$register_1$, $register_1$” นั้นเป็นรีจีสเตอร์ภายในซีพียู(local CPU register) โดยในทางกายภาพอาจจะใช้เป็นรีจีสเตอร์ตัวเดียวกันก็ได้ เช่น เป็น “accumulator” แล้วใช้การจัดการอินเทอรัพท์(interrupt handle) เพื่อบันทึกหรือนำข้อมูลออกมา เป็นต้น ซึ่งถ้าหากการทำงานตามเวลาของภาษาเครื่องเป็นลักษณะหนึ่งดังในรูปที่ 2.3.C ก็จะแสดงให้เห็นว่า ค่าผลลัพธ์ของตัวแปร ”counter” มีค่าได้ไม่แน่นอนดังที่กล่าวมา

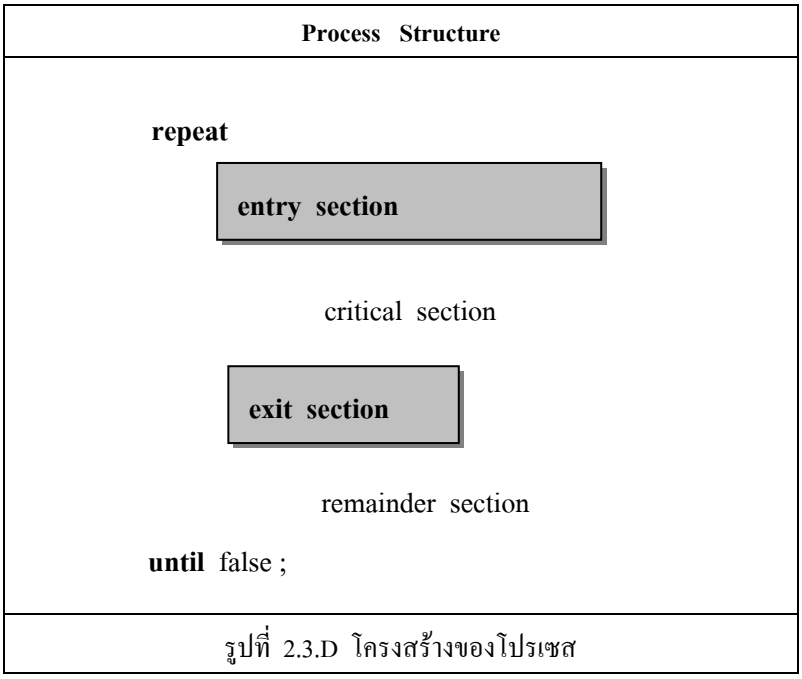
T_0 :	producer	**execute**	$register_1$:= counter	{ $register_1$ = 5 }
T_1 :	producer	**execute**	$register_1$:= $register_1$ + 1	{ $register_1$ = 6 }
T_2 :	consumer	**execute**	$register_2$:= counter	{ $register_2$ = 5 }
T_3 :	consumer	**execute**	$register_2$:= $register_2$ - 1	{ $register_2$ = 4 }
T_4 :	producer	**execute**	counter := $register_1$	{ counter = 6 }
T_5 :	consumer	**execute**	counter := $register_2$	{ counter = 4 }

รูปที่ 2.3.C แสดงตัวอย่างการทำงานในระดับล่างตามเวลา ของคำสั่งทั้งสองเทอม

ดังนั้น จะต้องทำการแก้ไขเพื่อทำการซิงโครไนเซชั่นให้การทำงานเป็นจังหวะกัน ลักษณะปัญหาดังกล่าวนี้เรียกว่า คริติคอลเซกชั่นพร็อบเบล็ม

2.3.2 คริติคอลเซกชั่นพร็อบเบล็ม (Critical Section Problem)

การแก้ปัญหาในลักษณะนี้ ได้มีการกำหนดให้โปรเซสต่างๆ มีโครงสร้างของโปรเซสเป็นดังนี้



รูปที่ 2.3.D โครงสร้างของโปรเซส

ลักษณะปัญหาคือ

โปรเซสจะมีการใช้ข้อมูลร่วมกัน(share data , common data) ดังนั้นเมื่อโปรเซสผู้ผลิตกำลังนำข้อมูลไปใส่ใน"buffer" พร้อมกับเพิ่มค่า"counter" นั้น ถือว่าเป็นคริติคอลเซกชั่นของโปรเซสผู้ผลิต จึงไม่ควรให้โปรเซสอื่นมากระทำกับ"buffer" จนกว่าจะออกจากคริติคอลเซกชั่นของโปรเซสผู้ผลิต ทำนองเดียวกันเมื่อโปรเซสผู้บริโภคกำลังนำเอาข้อมูลออกจาก"buffer" พร้อมกับลดค่า"counter" นั้น ถือว่าเป็นคริติคอลเซกชั่นของโปรเซสผู้บริโภค จึงไม่ควรให้โปรเซสอื่นมากระทำกับ"buffer" จนกว่าจะออกจากคริติคอลเซกชั่นของโปรเซสผู้บริโภค เช่นกัน

เซกชั่นต่างๆในโครงสร้างของโปรเซส มีความหมายดังนี้

- "entry section" เป็นชุดคำสั่งที่ทำหน้าที่ ในการประกาศห้ามไม่ให้โปรเซสใดเข้าสู่คริติคอลเซกชั่นของตัวมันเอง หากไม่ผ่านการตัดสินใจของชุดคำสั่งดังกล่าว ดังนั้นเซกชั่นนี้จึงจะเป็นตัวตัดสินใจในขณะนั้นถึงการให้โปรเซสเข้าหรือไม่ให้เข้านั่นเอง
- "critical section" เป็นชุดคำสั่งที่โปรเซสกระทำกับข้อมูลที่ใช้ร่วมกัน ซึ่งขณะนั้นจะต้องไม่มีโปรเซสอื่นเข้ามากระทำ ด้วย
- "exit section" เป็นชุดคำสั่งที่เป็นเงื่อนไขของการออกจากคริติคอลเซกชั่นของโปรเซสนั้นๆ
- "remainder section" เป็นชุดคำสั่งอื่นๆที่เหลือของโปรเซสนั้นๆ

การแก้ปัญหา คริติคอลเซกชั่นพร็อบเบล็มนี้ได้ จะต้องเป็นไปตามเงื่อนไข(satisfy conditions) ทั้ง 3 ข้อ (ขาดข้อใดข้อหนึ่งไม่ได้) ดังต่อไปนี้

1. " Mutual Exclusion " หมายถึงในขณะใดขณะหนึ่ง จะมีเพียงโปรเซสเดียวเท่านั้นที่อยู่ในคริติคอลเซกชั่น
2. " Progress " หมายถึงว่า ถ้าขณะนั้นไม่มีโปรเซสใดอยู่ในคริติคอลเซกชั่น แล้วมีโปรเซสหนึ่งมีความต้องการที่จะเข้าสู่คริติคอลเซกชั่นของตัวมันเอง โปรเซสนี้จะสามารถเข้าไปได้โดยไม่ต้องรอ
3. " Bounded waiting " หมายถึงว่า ถ้าขณะนั้นมีหลายๆโปรเซสต้องการเข้าสู่คริติคอลเซกชั่นพร้อมกัน แล้วโปรเซสต่างๆจะต้องมีการรอ การรอของแต่ละโปรเซสนั้นจะต้องมีขอบเขต(bound) หรือมีลิมิต(limit) ที่แน่นอน

อัลกอริทึ่มสำหรับกรณีที่มี เพียง 2 โปรเซส มีดังนี้

1. อัลกอริทึ่มที่ 1 (Algorithm 1)

กำหนดให้ โปรเซส $\mathbf{P_i}$ และ $\mathbf{P_j}$ ต้องการเข้าสู่คริติคอลเซกชั่น และมีกำหนดตัวแปรดังนี้

var turn : 0 .. 1 ;

และ ให้ j = i – 1 ซึ่งหมายถึง ถ้า i = 0 แล้ว j = 1 ดังนั้น $\mathbf{P_i = P_0}$, $\mathbf{P_j = P_1}$

ซึ่งทั้งสองโปรเซส มีลักษณะโปรแกรมเป็นดังรูปที่ 2.3.E

$\mathbf{P_i}$	$\mathbf{P_j}$
repeat	**repeat**
while turn ≠ i **do** no-op;	**while** turn ≠ j **do** no-op;
critical section	critical section
turn := j ;	turn := i ;
remainder section	remainder section
until false;	**until** false;

รูปที่ 2.3.E ลักษณะโปรแกรมอัลกอริทึ่มที่ 1 ของทั้งสองโปรเซส

หลักการทำงาน : ??

ซึ่งต้องพิจารณาการสอดคล้องกับเงื่อนไขการแก้ปัญหา คริติคอลเซกชั่นพร็อบเบล็ม ดังนี้

1. " Mutual Exclusion " : "satisfy" or "not satisfy"
2. " Progress " : "satisfy" or "not satisfy"
3. " Bounded waiting " : "satisfy" or "not satisfy"

2. อัลกอริทึ่มที่ 2 (Algorithm 2)

กำหนดให้ โปรเซส $\mathbf{P_i}$ และ $\mathbf{P_j}$ ต้องการเข้าสู่คริติคอลเซกชั่น และมีกำหนดตัวแปรดังนี้

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

และ ให้ i = 0 , j = 1 ดังนั้น $\mathbf{P_i = P_0}$, $\mathbf{P_j = P_1}$

ซึ่งทั้งสองโปรเซส มีลักษณะโปรแกรมเป็นดังรูปที่ 2.3.F

หลักการทำงาน : ??

$\mathbf{P_i}$	$\mathbf{P_j}$

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

รูปที่ 2.3.F ลักษณะโปรแกรมอัลกอริทึ่มที่ 2 ของทั้งสองโปรเซส

ซึ่งต้องพิจารณาการสอดคล้องกับเงื่อนไขการแก้ปัญหา คริติคอลเซกชั่นพร็อบเบล็ม ดังนี้

1. " Mutual Exclusion " : "satisfy" or "not satisfy"
2. " Progress " : "satisfy" or "not satisfy"
3. " Bounded waiting " : "satisfy" or "not satisfy"

3. อัลกอริทึ่มที่ 3 (Algorithm 3)

กำหนดให้ โปรเซส $\mathbf{P_i}$ และ $\mathbf{P_j}$ ต้องการเข้าสู่คริติคอลเซกชั่น และมีกำหนดตัวแปรดังนี้

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

และ ให้ i = 0 , j = 1 ดังนั้น $\mathbf{P_i = P_0}$, $\mathbf{P_j = P_1}$

รวมทั้งกำหนดให้ค่าเริ่มต้น(initial) flag[0] = flag[1] = false และค่า turn เริ่มต้นมีค่า เป็น 0 หรือ 1 ก็ได้ ซึ่งทั้งสองโปรเซส มีลักษณะโปรแกรมเป็นดังรูปที่ 2.3.G

$\mathbf{P_i}$	$\mathbf{P_j}$

Repeat
flag[i] := true;
turn := j ;
while flag[j] **and** turn = j **do** no-op ;
critical section
flag[i] := false;
remainder section
until false;
	repeat
	flag[j] := true;
	turn := i ;
	while flag[i] **and** turn = i **do** no-op ;
	critical section
	flag[j] := false;
	remainder section
	until false;

รูปที่ 2.3.G ลักษณะโปรแกรมอัลกอริทึ่มที่ 3 ของทั้งสองโปรเซส

หลักการทำงาน : ??

ซึ่งต้องพิจารณาการสอดคล้องกับเงื่อนไขการแก้ปัญหา คริติคอลเซกชั่นพร็อบเบล็ม ดังนี้

1. " Mutual Exclusion " : "satisfy" or "not satisfy"
2. " Progress " : "satisfy" or "not satisfy"
3. " Bounded waiting " : "satisfy" or "not satisfy"

2.3.3 ซิงโครไนเซชั่นฮาร์ดแวร์ (Synchronization Hardware)

เป็นการแก้ปัญหาคริติคอลเซกชั่นพร็อบเบล็มโดยสร้างชุดคำสั่งขนาดเล็กขึ้นมาใช้ในฮาร์ดแวร์ (simple hardware instructions) เพื่อทำหน้าที่ในลักษณะดังต่อไปนี้

1. “Test-and-Set instruction”

```
function  Test-and-Set ( var  target :boolean ) : boolean ;
    begin
        Test-and-Set :=  target  ;
        Target  :=  true ;
    end;
```

รูปที่ 2.3.J แสดงชุดคำสั่ง “Test-and-Set instruction”

นำไปใช้งานในโปรเซสต่างๆ เช่น $\mathbf{P_i}$ เป็นดังรูปที่ 2.3.K โดยกำหนดให้ตัวแปร lock เริ่มต้นกำหนดค่าเป็น false

P_i

```
repeat
    while Test-and-Set (lock)  do  no-op ;

        Critical  section

    lock  := false ;

        Remainder  section

until  false;
```

รูปที่ 2.3.K การใช้งานชุดคำสั่ง “Test-and-Set instruction” ในโปรเซส

หลักการทำงาน : ??

ซึ่งต้องพิจารณาการสอดคล้องกับเงื่อนไขการแก้ปัญหา คริติคอลเซกชั่นพร็อบเบล็ม

2. “Swap instruction”

```
procedure Swap ( var a , b : boolean );
    var temp : boolean;
    begin
        temp := a ;
        a := b ;
        b :=temp ;
    end;
```

รูปที่ 2.3.L แสดงชุดคำสั่ง “Swap instruction”

นำไปใช้งานในโปรเซสต่างๆ เช่น $\mathbf{P_i}$ เป็นดังรูปที่ 2.3.L โดยกำหนดให้ตัวแปร lock เริ่มต้นกำหนดค่าเป็น false

$\mathbf{P_i}$

```
repeat
    key := true ;
    repeat
        Swap( lock , key ) ;
    until key = false ;

        Critical section

    lock := false ;

        Remainder section

until false;
```

รูปที่ 2.3.M การใช้งานชุดคำสั่ง “Swap instruction” ในโปรเซส

หลักการทำงาน : ??

ซึ่งต้องพิจารณาการสอดคล้องกับเงื่อนไขการแก้ปัญหา คริติคอลเซกชั่นพร็อบเบล็ม

2.3.4 เซมาฟอร์ (Semaphores)

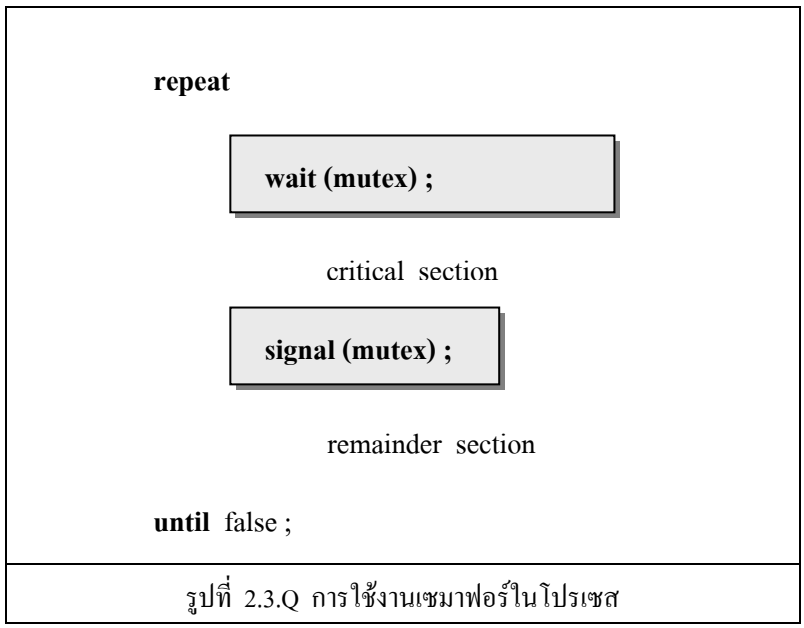
เครื่องมือสำหรับการทำซิงโครไนเซชั่น อย่างหนึ่งคือ เซมาฟอร์ ซึ่งเป็นการใช้ตัวแปร จำนวนเต็ม (integer) ที่ถูกแปรเปลี่ยนค่าได้โดยผ่านคำสั่งพื้นฐานสำหรับการทำซิงโครไนเซชั่นคือ

P (for wait) และ V (for signal)

ซึ่งมีลักษณะการทำงานเป็นดังรูปที่ 2.3.P

wait (S) : **while** S $\leq$ 0 **do** no-op ; S := S – 1 ;
signal (S) : S := S + 1 ;
รูปที่ 2.3.P แสดงลักษณะการทำงาน ของชุดคำสั่ง P และ V

ซึ่งเมื่อนำไปใช้งานในโปรเซส จะมีลักษณะการใช้งานเป็นดังรูปที่ 2.3.Q

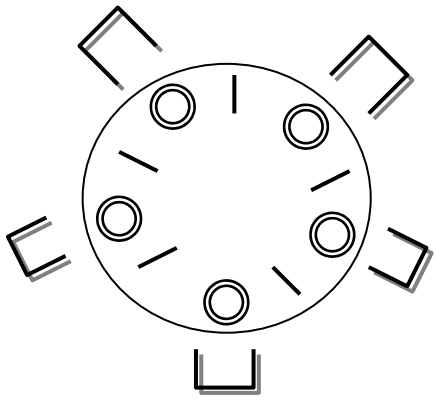


รูปที่ 2.3.Q การใช้งานเซมาฟอร์ในโปรเซส

ซึ่งต้องพิจารณาการสอดคล้องกับเงื่อนไขการแก้ปัญหา คริติคอลเซกชั่นพร็อบเบล็ม

ตัวอย่างปัญหา การบริโภคของนักปราชญ์ (the dining-philosophers problem)

สมมุติว่า มีนักปราชญ์ อยู่ 5 คน นั่งรอบโต๊ะกลมที่มีตะเกียบวางอยู่ 5 อันดังรูปที่ 2.3.S ซึ่งนักปราชญ์จะต้องใช้ตะเกียบ 2 อันที่อยู่ทางซ้ายมือและขวามือเท่านั้นในการบริโภคอาหารของแต่ละรอบ และเมื่อนักปราชญ์คนใดที่ได้ตะเกียบ 2 อันดังกล่าวพร้อมกับบริโภคอาหารหนึ่งครั้ง(1 รอบ)แล้วจะต้องวางตะเกียบ 2 อันดังกล่าวลงที่เดิมบนโต๊ะ เพื่อที่จะเริ่มรอบใหม่



รูปที่ 2.3.S การบริโภคอาหารของนักปราชญ์

Philosopher $_i$

```
repeat

    wait ( chopstick[i] ) ;
    wait ( chopstick[ ( i+1 ) mod 5 ] ) ;

        … eat …

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

        … think …

until false;
```

รูปที่ 2.3.T ลักษณะโครงสร้างโปรแกรมของโปรเซสของนักปราชญ์คนที่ i

ปัญหาที่เกิดขึ้นมีได้ดังนี้

1. เกิดการอับจน (deadlock) ดังนี้ ถ้านักปราชญ์ทุกคนหยิบตะเกียบทางด้านซ้ายมือขึ้นมา 1 อัน พร้อมๆ กัน จะทำให้นักปราชญ์ทุกคนจะต้องรอตะเกียบทางด้านขวามือไปตลอด ทำให้ไม่สามารถบริโภคอาหารได้

2. เกิดการอดตาย (starvation) ดังนี้ ถ้านักปราชญ์คนกลางใดๆ ถูกนักปราชญ์ที่อยู่ติดกันทางด้านซ้ายมือและขวามือ แย่งตะเกียบไปได้ทุกรอบ จะทำให้นักปราชญ์คนกลางไม่ได้บริโภคอาหารเลย

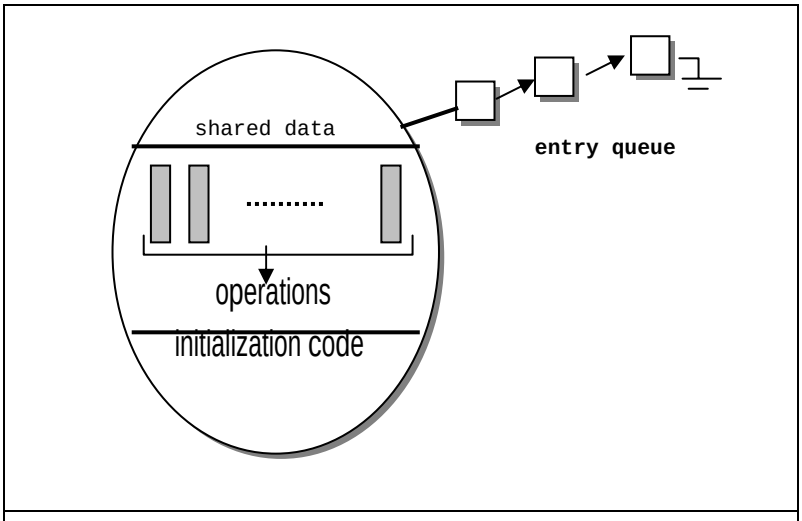
ซึ่งลักษณะโครงสร้างโปรแกรมของโปรเซสของนักปราชญ์คนที่ i เป็นดังรูปที่ 2.3.T

ตัวอย่างสองแนวทางในการแก้ปัญหามีดังนี้

1. ให้นักปราชญ์ทั้งหมดที่จะมานั่งบนโต๊ะได้เพียง 4 คน จากทั้งหมด 5 ที่นั่งรอบโต๊ะ
2. ให้นักปราชญ์แต่ละคนสามารถที่จะหยิบตะเกียบที่ละ 2 อันพร้อมกันได้ (ไม่ต้องหยิบที่ละอัน)

มอนิเตอร์(Monitor)

ในโครงสร้างของการซิงโครไนเซชั่นขั้นสูงขึ้น จะมีตัวควบคุมการทำงานร่วมกันของหลายๆโปรเซส อีกชนิดหนึ่งเรียกว่า มอนิเตอร์ ซึ่งตัวมอนิเตอร์นี้จะประกอบด้วยข้อมูลที่ใช้ร่วมกัน และส่วนของโปรแกรมต่างๆ ที่เกี่ยวข้องกับข้อมูลดังกล่าว ภาพรวมของมอนิเตอร์ และ ลักษณะโครงสร้างโปรแกรมของมอนิเตอร์ เป็นดังรูปที่ 2.3.U และ 2.3.V



รูปที่ 2.3.U ภาพรวมของมอนิเตอร์

Monitor

type monitor-name = **monitor**

Variable declarations

Procedure entry P1 (…) ;
begin ……. end ;

Procedure entry P2 (…) ;
begin ……. end ;

:

:

Procedure entry Pn (…) ;
begin ……. end ;

begin

Initialization code

end;

รูปที่ 2.3.V ลักษณะโครงสร้างโปรแกรมของมอนิเตอร์