แผนบริหารการสอนประจำบทที่ 6

การประสานเวลา (Synchronization)

วัตถุประสงค์

1. เพื่อเข้าใจหลักการการทำงานพร้อมกันหลายโปรแกรม
2. เพื่อสามารถอธิบายลักษณะทรัพยากรส่วนวิกฤต และหลักการควบคุมการใช้ส่วนวิกฤต
3. เพื่อสามารถอธิบายวิธีการจัดการประสานเวลากันของหลายโปรแกรมในรูปแบบการจัดการด้วยซอฟต์แวร์ และจัดการด้วยฮาร์ดแวร์
4. เพื่อสามารถอธิบายวิธีการเซมาฟอร์ (Semaphores) และ วิธีการตรวจสอบ (Monitors)

เนื้อหา

1. หลักการพื้นฐานของสภาวะการทำงานพร้อมกันหลายโปรแกรม
2. ทรัพยากรในส่วนวิกฤต และหลักควบคุมการใช้งาน
3. การจัดการประสานเวลาด้วยซอฟต์แวร์
4. การจัดการประสานเวลาด้วยฮาร์ดแวร์
5. วิธีการเซมาฟอร์ (Semaphores)
6. วิธีการตรวจสอบ (Monitors)

กิจกรรมการเรียนการสอน

1. บรรยาย
2. ฝึกปฏิบัติการระบบลีนุกซ์
3. วิดีโออนิเมชั่นการทำงานคอมพิวเตอร์
4. ค้นคว้าด้วยตัวเอง
5. รายงานหน้าชั้นเรียน
6. ฝึกปฏิบัติ

สื่อการเรียนการสอน

1. เอกสารประกอบการสอน
2. สไลด์การสอน
3. โปรแกรม google class room
4. โปรแกรม facebook
5. โปรแกรม VMware
6. โปรแกรมระบบปฏิบัติการลีนุกซ์ Ubantu

การวัดผลและการประเมินผล

1. แบบฝึกหัดท้ายบท
2. การถามตอบในชั้นเรียน
3. ใบงาน
4. สอบเก็บคะแนน

บทที่ 6

การประสานเวลา (Synchronization)

6.1 หลักการพื้นฐานการทำงานโปรแกรมพร้อมกัน (Principles of Concurrency)

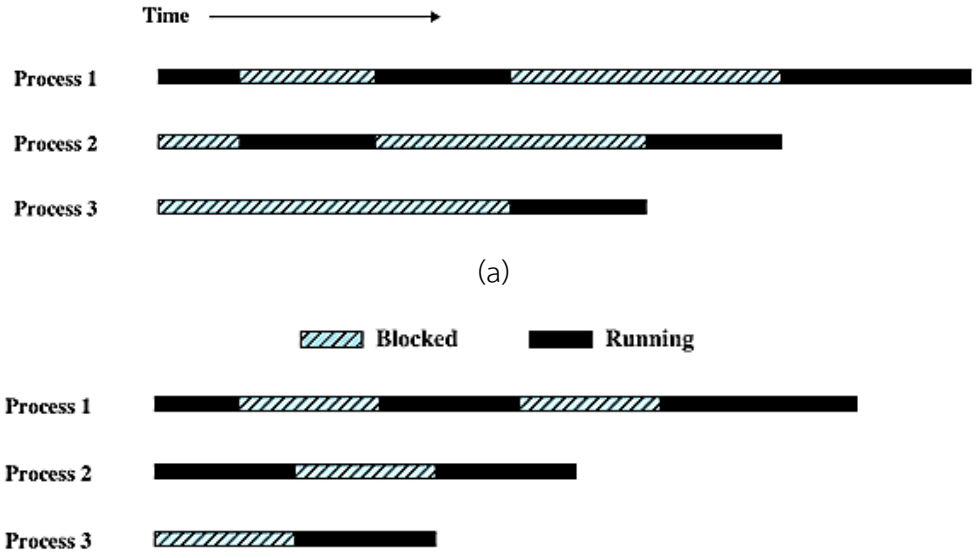
เนื้อหาส่วนใหญ่ในการออกแบบระบบปฏิบัติการอยู่ที่การบริหารจัดการการทำงานของคอมพิวเตอร์ให้สามารถประมวลผลได้หลาย ๆ งาน การจัดการงานทั้งในรูปแบบหลายโปรเซส (multi-processes) และหลายธเร็ด (multi-threads) จึงเป็นเป้าหมายสำคัญของระบบปฏิบัติการ นอกจากนี้ลักษณะระบบคอมพิวเตอร์ทางด้านฮาร์ดแวร์ก็ส่งผลต่อการออกแบบด้วยเช่นกัน เนื่องจากปัญหาและสภาพแวดล้อมที่แตกต่าง ๆ กัน ดังนั้นการพัฒนาโปรแกรมจึงจำเป็นต้องความเข้าใจหลักการต่อไปนี้

1) Multiprogramming: การจัดการหลาย ๆ งานภายใต้ระบบคอมพิวเตอร์ซึ่งมีหน่วยประมวลผล (processor) ชุดเดียว

2) Multiprocessing: การจัดการหลาย ๆ งานภายใต้ระบบคอมพิวเตอร์ซึ่งมีหน่วยประมวลมากกว่า 1 ชุด

3) Distributed processing: การจัดการหลาย ๆ งานภายใต้ระบบที่มีคอมพิวเตอร์หลายชุด

รูปที่ 6-1 แสดงให้เห็นความแตกต่างของลักษณะทั่วไปของการทำงานหลายงานบนระบบคอมพิวเตอร์ที่มีหน่วยประผลเดี่ยว และระบบหลายหน่วยประมวลผล ภายใต้ระบบคอมพิวเตอร์แบบหน่วยประมวลผลเดี่ยว (แสดงในรูปที่ 6-1(a)) ระบบปฏิบัติการมีภาระกิจในการบริหารโปรเซสให้ได้รับการประมวลผลสลับกันไปในช่วงเวลาหนึ่ง ๆ มีเพียงหนึ่งโปรเซสเท่านั้นที่ทำงานได้ ดังนั้นภาระส่วนหนึ่งของระบบปฏิบัติการที่ต้องแบกรับคือการสลับงานบ่อยครั้ง ในระบบคอมพิวเตอร์ที่มีหน่วยประมวลหลายชุดนั้น โปรเซสหลายชุดสามารถประมวลผลได้ในเวลาเดียวกันตามจำนวนของหน่วยประมวล ดังนั้นภาระของระบบนี้นอกจากจะต้องสลับลำดับงานตามความสามารถของหน่วยประมวลผลแล้ว ยังจำเป็นต้องจัดการเรื่องการทำงานทับซ้อนให้สอดคล้องกันระหว่างกลุ่มโปรเซส



รูปที่ 6-1: การทำงานหลายโปรแกรม (a) Multiprogramming (b) Mutiprocessing

ที่มา: (Stallings, 2017)

ประเด็นปัญหาที่ต้องคำนึงถึงในระบบทั้งสองประเภท (หน่วยประมวลผลเดี่ยวและหน่วยประมวลหลายชุด) มีความคล้ายคลึงกันคือ

1) The sharing of global resources: ปัญหาด้านการใช้ทรัพยากรร่วมกันระหว่างโปรเซสเช่น มีโปรเซส 2 ชุดเรียกใช้ตัวแปรประเภท global variable ตัวเดียวกัน การเขียนและอ่านตัวแปรดังกล่าวจึงอาจมีโอกาสขัดแย้งกันสูง

2) The allocation of resources: ปัญหาด้านการจัดสรรทรัพยากรให้กับโปรเซสต่าง ๆ ที่ต้องการใช้ เช่น ระบบทำการจองช่องทางสื่อสารกับอุปกรณ์ I/O ไว้ให้โปรเซส P1 แต่มีการขัดจังหวะจากโปรเซสอื่น P1 จึงเข้าสู่สภาวะการรอคอย (waiting state) ดังนั้นช่องทางสื่อสาร I/O ที่ถูกจองไว้ให้ P1 จึงไม่สามารถถูกเรียกใช้โดยโปรเซสอื่น ๆ ได้อีก

การใช้พื้นที่หน่วยความจำร่วมกันมีข้อดีในการใช้งานทรัพยากรให้เกิดประโยชน์สูงสุด อย่างไรก็ตามอาจเกิดปัญหาขัดแย้งกันทางข้อมูล ดังตัวอย่างด้านล่าง เป็นการใช้งานฟังก์ชัน *echo()* ซึ่งเป็นฟังก์ชันสาธารณะซึ่งมีโปรแกรมหลายชุดเข้าใช้งานได้ ฟังก์ชันดังกล่าวเมื่อถูกเรียกใช้ ก็จะรับค่าอักษรหนึ่งตัวจากผู้ใช้ผ่านแป้นพิมพ์ โดยเก็บไว้ในตัวแปร *chin* จากนั้นจึงถ่ายค่ามาให้ตัวแปร *chout* และแสดงค่าออกหน้าจอของผู้ใช้

หากพิจารณาสถานการณ์ที่มีความเป็นไปได้ดังรูปที่ 6-2 เช่น ผู้ใช้กำลังทำงานอยู่บนระบบคอมพิวเตอร์ที่มีหน่วยประมวลผลเดี่ยว ซึ่งมีชุดแป้นพิมพ์และจอภาพอย่างละชุด ผู้ใช้ต้องการทำงานหลายอย่างในเวลเดียวกัน โดยเปิดใช้งานหลายโปรแกรมไปพร้อม ๆ กัน โปรแกรมหนึ่ง (P1) ที่ผู้ใช้กำลังใช้งาน มีการ

```
#include <stdio.h>

void echo()
{
     chin = getchar();
     chout = chin;
     putchar(chout);
}
```

รูปที่ 6-2: การใช้ทรัพยกรร่วมกันบนระบบคอมพิวเตอร์แบบหน่วยประมวลผลเดี่ยว

เรียกฟังก์ชัน *echo()* และผู้ใช้พิมพ์ค่า ‘x’ ผ่านแป้นพิมพ์ ดังนั้นฟังก์ชัน *getchar()* ทำการเก็บค่าไว้ในตัวแปร *chin* อย่างไรก็ตามโปรเซส P1 ถูกขัดจังหวะโดยโปรเซส P2 ซึ่งเป็นอีกโปรแกรมหนึ่งที่ผู้ใช้กำลังใช้งาน โปรเซส P2 ในช่วงหนึ่งของการประมวลผล มีการเรียกใช้ฟังก์ชัน *echo()* เช่นกัน ในครั้งนี้ผู้ใช้ใส่ข้อมูล ‘y’ ดังนั้นค่าตัวแปร *chin* เปลี่ยนค่าจาก ‘x’ มาเป็น ‘y’ และถูกพิมพ์ออกหน้าจอ เมื่อโปรเซส P2 ทำงานเสร้จสิ้นจึงเป็นโอกาสให้โปรเซส P1 กลับมาทำงานที่เหลือ คือ ถ่ายค่าในตัวแปร *chin* สู่ตัวแปร *chout* ซึ่งขณะนี้มีค่า ‘y’ และพิมพ์ค่าออกหน้าจอเป็นค่า ‘y’ อีกครั้ง จะเห็นว่าข้อมูลมีความขัดแย้งกันระหว่างการทำงานของโปรเซส P1 และ P2 จากการเรียกใช้ฟังก์ชันและตัวแปรเดียวกัน

ในระบบคอมพิวเตอร์ที่มีโครงสร้างระบบเป็นแบบหน่วยประมวลผลหลายชุดก็พบปัญหาเช่นเดียวกัน พิจารณากรณีตัวอย่างในรูปที่ 6-3 โปรเซส P1 และ P2 ถูกผู้ใช้เรียกใช้งานในเวลาพร้อม ๆ กัน ทั้งนี้โปรเซส P1 และ P2 ประมวลผลคู่ขนานโดยหน่วยประมวลผลคนละชุด ทั้งสองโปรเซสมีการเรียกใช้ฟังก์ชัน *echo()* เหมือนกัน ซึ่งเหตุการณ์มีความเป็นไปได้ที่โปรเซส P1 สั่งงานฟังก์ชัน *getchar()* เพื่อรับค่าจากผู้ใช้มาเก็บไว้ในตัวแปร *chin* ล่วงหน้าไปเล็กน้อย ก่อน P2 จะสั่งงานฟังก์ชัน *getchar()* ตามหลัง ซึ่งพบว่าค่าในตัวแปร *chin* จากโปรเซส P1 ถูกเขียนทับด้วยเวลาอันรวดเร็ว จึงทำให้ข้อมูลเกิดความขัดแย้ง และการพิมพ์ค่าสู่หน้าจอผิดพลาดในที่สุด

```
Process P1                          Process P2
void echo()                         void echo()
{                                   {
    .                                   .
    chin = getchar();                   .
    .                                   chin = getchar();
    chout = chin;                       chout = chin;
    putchar(chout);                     .
    .                                   putchar(chout);
    .                                   .
}                                   }
```

รูปที่ 6-3: การเรียกใช้ทรัพยากรร่วมกันบนระบบคอมพิวเตอร์แบบหน่วยประมวลผลหลายชุด

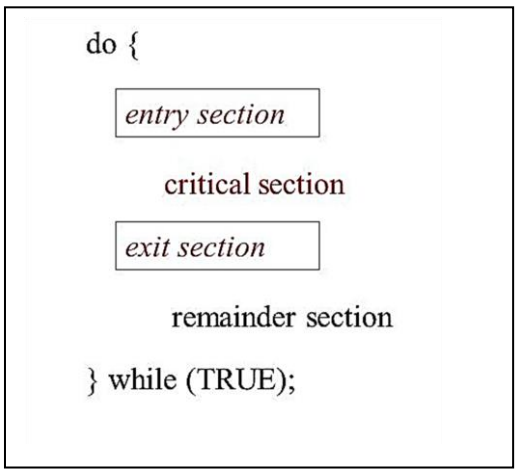
วิธีการป้องกันปัญหาดังกล่าวคือ การควบคุมการใช้ทรัพยากรที่ใช้งานร่วมกัน (control access to the shared resources) เช่น ตัวแปร ฟังก์ชัน และอุปกรณ์ I/O เป็นต้น ในกรณีตัวอย่างข้างต้น โปรเซส P2 จะไม่สามารถเข้าใช้ฟังก์ชัน *echo()* ได้ หากโปรเซส P1 ยังคงใช้งานอยู่ ไม่ว่าโปรเซส P1 จะอยู่ในสถานะรอคอยจากการขัดจังหวะหรือทำงานปกติ โปรเซส P2 จะถูกกันออกไปให้อยู่ในส่วน blocked queue จนกระทั่งโปรเซส P1 ทำงานเสร็จสิ้นจากฟังก์ชัน *echo()* แล้ว โปรเซส P2 จึงจะสามารถเข้าใช้ฟังก์ชัน *echo()* ได้

6.2 ส่วนวิกฤตและการควบคุมส่วนวิกฤต (Critical Sections and Mutual Exclusion)

6.2.1 ส่วนวิกฤตและการควบคุม

การควบคุมการใช้ทรัพยากรที่ใช้งานร่วมกันจำเป็นต้องทำความเข้าใจสองประการคือ ส่วนวิกฤต (critical section) และหลักควบคุมส่วนวิกฤต (mutual exclusion) ทรัพยากรต่าง ๆ เช่น ตัวแปร ฟังก์ชัน และอุปกรณ์ประเภท I/O หากมีการใช้งานร่วมกัน จะถือเป็นทรัพยากรที่วิกฤต (critical resources) ซึ่งสามารถก่อให้เกิดปัญหาความผิดพลาดในการใช้งานได้ จึงจำเป็นที่ต้องควบคุมการเข้าใช้ พิจารณาระบบปฏิบัติการหนึ่งที่มีจำนวนโปรเซส n โปรเซสคือ {P0, P1, ..., Pn-1) โดยที่โปรเซสเหล่านี้มีส่วนหนึ่งของเนื้อหาโปรแกรม (ชุดคำสั่ง) ที่ต้องเข้าใช้อุปรกณ์ I/O เช่นเครื่องพิมพ์ ซึ่งเป็นอุปกรณ์ประเภทวิกฤต (critical device) ดังนั้นส่วนของโปรแกรมดังกล่าวถือเป็นส่วนวิกฤต (critical section) ที่ต้องถูกควบคุมการใช้งานให้สอดคล้องร่วมกับโปรเซสอื่น ๆ

หลักควบคุมส่วนวิกฤต (mutual exclusion) ในอีกประการหนึ่ง คือเมื่อใดที่มีโปรเซสหนึ่งเข้าสู่การประมวลผลในส่วนวิกฤต ซึ่งเข้าใช้งานทรัพยากรที่จัดสรรไว้ใช้งานร่วมกันแล้ว ระบบปฏิบัติการจะไม่อนุญาตให้ โปรเซสอื่นสามารถใช้งานทรัพยากรดังกล่าวได้อีก จนกระทั่งโปรเซสที่ใช้งานทรัพยากรวิกฤตดังกล่าว เสร็จสิ้นงานเสียก่อน ดังรูปที่ 6-4



รูปที่ 6-4: การเขียนโปรแกรมด้วยคุณสมบัติควบคุมส่วนวิกฤต

6.2.2 ความปฏิสัมพันธ์ระหว่างโปรเซส (process interaction)

ความปฏิสัมพันธ์ระหว่างโปรเซสเป็นเรื่องสำคัญที่ระบบปฏิบัติการต้องคำนึงถึง เราสามารถแบ่งความปฏิสัมพันธ์ของโปรเซสได้ดังนี้

1) โปรเซสไม่มีปฏิสัมพันธ์กับโปรเซสอื่น (Processes unaware of each other): คือ โปรเซสไม่มีปฏิสัมพันธ์ระหว่างกัน โปรเซสต่างเป็นอิสระจากกัน ดังนั้นลักษณะการเข้าใช้ทรัพยากรจึงมีลักษณะแข่งขัน (competition) ตัวอย่างเช่น มี 2 โปรเซสอิสระจากกัน (P1 และ P2) ต้องการใช้งานเครื่องพิมพ์ในเวลาเดียวกัน จึงเกิดการแข่งขันกันใช้เครื่องพิมพ์ ซึ่งใช้ได้คราวละหนึ่งโปรเซส ดังนั้นระบบปฏิบัติการจึงต้องมีวิธีการจัดการที่มีประสิทธิภาพด้วยการควบคุมส่วนวิกฤต (multual exclusion) เพื่อป้องกันการทำงานผิดพลาด จากตัวอย่างนี้ ระบบปฏิบัติการจะยอมให้เพียง โปรเซสเดียวสามารถเข้าใช้เครื่องพิมพ์จนกระทั่งเสร็จงาน เช่น P1 ได้สิทธิ์การใช้เครื่องพิมพ์ P2 จำเป็นต้องรอจนกว่า P1 ทำงานเสร็จ

2) โปรเซสมีปฏิสัมพันธ์ระหว่างกันโดยทางอ้อม (Processes indirectly aware of each other): คือ การมีปฏิสัมพันธ์ระหว่างโปรเซสในทางอ้อม โดยโปรเซสต่าง ๆ ไม่มีการสื่อสารหรือแลกเปลี่ยนข้อมูลกันโดยตรง แต่มีการใช้ทรัพยากรบางอย่างร่วมกัน ดังนั้นการปฏิสัมพันธ์ของ โปรเซสเหล่านี้ เป็นแบบการทำงานร่วมกันในลักษณะร่วมกันใช้ (cooperation by sharing) เช่น ตัวแปร ไฟล์ หรือฐานข้อมูล ดังนั้น

การควบคุมข้อมูลที่ใช้งานร่วมกันให้ถูกต้องจึงมีความสำคัญ โดยปกติการเข้าถึงข้อมูลเป็นการอ่าน (read) หรือการเขียน (write) โดยการอ่านไม่มีผลกระทบใด ๆ ต่อความถูกต้องของข้อมูล ซึ่งตรงข้ามกับการเขียนที่สามารถส่งผลกระทบต่อข้อมูลที่โปรเซสอื่น ๆ จำเป็นต้องใช้งานร่วมด้วย ดังนั้นจึงจำเป็นต้องมีการควบคุมส่วนวิกฤตนี้ (mutual exclusion)

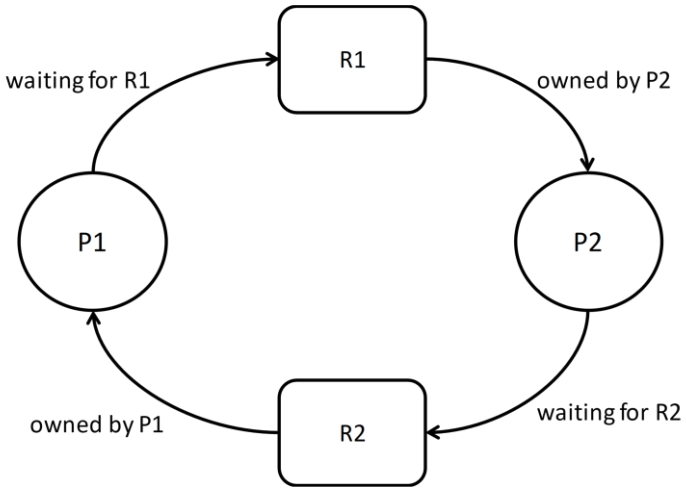
3) โปรเซสมีปฏิสัมพันธ์กันโดยตรง (Processes directly aware of each other): คือ โปรเซสต่าง ๆ ในระบบมีปฏิสัมพันธ์ระหว่างกันในทางตรง ด้วยวิธีการสื่อสารระหว่างโปรเซสโดยตรง โดยการระบุเลขประจำตัว (id) ของโปรเซสปลายทาง ดังนั้นโปรเซสเหล่านี้จึงมีการทำงานร่วมกัน (cooperation by communication) การสื่อสารระหว่างโปรเซสจะทำให้เกิดการทำงานที่สอดประสานเวลา (synchronise) ในงานต่าง ๆ อย่างไรก็ตามปัญหาการติดตาย (deadlock) และ การอดตาย (starvation) ของโปรเซสยังคงมีโอกาสเกิดขึ้นได้ เช่น ปัญหาติดตายเกิดขึ้นเมื่อ 2 โปรเซสต่างรอการติดต่อจากอีกฝั่ง ทำให้ไม่มีโปรเซสใดสามารถดำเนินการต่อไปได้ หรือปัญหาอดตาย เกิดขึ้นเมื่อ P1 ต้องการติดต่อ P2 หรือ P3 และในทางกลับกัน P2 และ P3 ก็ต้องการสื่อสารกับ P1 อย่างไรก็ตาม P1 กับ P2 สื่อสารกันอย่างต่อเนื่อง จนกระทั่ง P3 ไม่มีโอกาสติดต่อกับ P1 เลย จึงไม่สามารถดำเนินการต่อไปได้

6.2.3 ปัญหาการควบคุมส่วนวิกฤตเมื่อโปรเซสไม่มีปฏิสัมพันธ์ระหว่างกัน

เมื่อมีโปรเซสจำนวนหนึ่งทำงานในระบบ ไม่มีปฏิสัมพันธ์ระหว่างโปรเซสด้วยกัน จึงจำเป็นที่ต้องบริหารจัดการทรัพยากรด้วยการควบคุมส่วนวิกฤต เพื่อให้การใช้ทรัพยากรของระบบมีประสิทธิภาพ อย่างไรก็ตามพบว่าอาจมีปัญหาที่เกิดขึ้นได้ใน 2 ลักษณะดังต่อไปนี้

1) ปัญหาติดตาย (deadlock): ถือเป็นปัญหาที่ทำให้โปรเซสไม่สามารถประมวลผลต่อไปได้อย่างถาวร เนื่องจากการแข่งขันกันใช้ทรัพยากรของระบบคอมพิวเตอร์ ปัญหาติดตายเกิดขึ้นเมื่อ โปรเซสมากกว่า 2 โปรเซสมีความต้องการใช้ทรัพยากรของระบบจำนวนหนึ่ง จึงเกิดการแข่งขันกันเข้าครอบครองทรัพยากรเหล่านั้น แต่ปรากฏว่าโปรเซสได้รับทรัพยากรจากระบบปฏิบัติการไปคนละส่วน ดังนั้นโปรเซสจึงต่างต้องรอคอยให้ทรัพยากรที่ต้องการว่างลง กลายเป็นการคอยที่ไม่รู้จบ (circular waiting) ตัวอย่างเช่น P1 ต้องการใช้ทรัพยากรคือ R1 และ R2 สมมุติเป็นดิสก์และเครื่องพิมพ์ตามลำดับ เพื่ออ่านข้อมูลจากไฟล์ในดิกส์และพิมพ์ข้อมูลดังกล่าวออกทางเครื่องพิมพ์ ในขณะเดียวกัน P2 ก็ต้องการใช้ R1 และ R2 เช่นเดียวกัน ทั้ง P1 และ P2 ทำกระบวนการร้องขอใช้ทรัพยากรไปยังระบบปฏิบัติการ ด้วยกับจังหวะในการร้องขอจากทั้งสองโปรเซส ระบบปฏิบัติการจัดสรร เครื่องพิมพ์ (R2) ให้แก่โปรเซส P1 และ จัดสรรดิกส์ (R1) ให้แก่โปรเซส P2

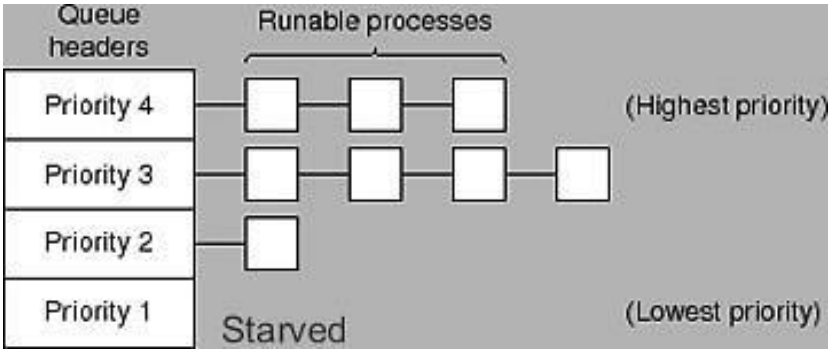
จึงทำให้ทั้งโปรเซส P1 และ P2 ไม่สามารถทำงานต่อไปได้เนื่องจากต่างคนต่างต้องรอทรัพยากรที่ต้องการใช้โดยไม่สิ้นสุด จึงเกิดปัญหาติดตายระหว่างโปรเซส P1 และ P2 (รูปที่ 6-5)



รูปที่ 6-5: ปัญหาติดตาย (Deadlock)

ที่มา: (B.P.Miller, 2013)

2) ปัญหาอดตาย (starvation): เป็นปัญหาทำให้โปรเซสไม่ได้รับโอกาสใช้ทรัพยากรที่ต้องการ โดยเกิดจากโปรเซสที่มีระดับความสำคัญมากกว่าครอบครองทรัพยากรอย่างต่อเนื่อง และระบบปฏิบัติการไม่จัดสรรทรัพยากรดังกล่าวให้แก่โปรเซสที่มีระดับความสำคัญต่ำมาก ๆ ได้ใช้งานเลย ทำให้เกิดปัญหาโปรเซสรอคอยทรัพยากรอย่างไม่รู้จบ ปัญหาดังกล่าวมักจะเกิดขึ้นกับวิธีการสลับงานด้วยการพิจารณาระดับความสำคัญของโปรเซส ตัวอย่างในรูปที่ 6-6 การเลือกโปรเซสในคิวเพื่อทำการสลับงานให้แก่ ซีพียู จัดแบ่งไปตามระดับความสำคัญไปตามประเภทของโปรเซส เฉพาะโปรเซสที่อยู่ในระดับความสำคัญสูงจะได้รับการจัดสรรทรัพยากรอย่างต่อไป จนกระทั่งโปรเซสในกลุ่มระดับความต่ำสุดไม่มีโอกาสได้ใช้ทรัยพยากรเหล่านั้นเลย



รูปที่ 6-6: ปัญหาอดตาย (Starvation)

ที่มา: (Difference between Deadlock and Starvation, 2017)

6.2.4 ปัญหาการใช้ทรัพยากรร่วมกันในปฏิสัมพันธ์ระหว่างโปรเซสทางอ้อม

การจัดสรรทรัพยากรและอนุญาตให้กลุ่มของโปรเซสได้เข้าใช้ทรัพยากรร่วมกัน การควบคุมส่วนวิกฤต (mutual exclusion) ยังมีความจำเป็นเพื่อควบคุมการเข้าถึงทรัพยากรให้มีความสอดประสาน อย่างไรก็ตามปัญหาความถูกต้องของข้อมูลอาจเกิดขึ้นได้ พิจารณาจากตัวอย่างต่อไปนี้ มีโปรแกรมหนึ่งที่มีข้อมูลตัวแปร 2 ตัวที่ต้องเท่ากันคือ a = b และมีโปรเซส 2 โปรเซสคือ P1 และ P2 เข้าใช้ตัวแปรทั้งสอง

```
P1:   a = a + 1;
      b = b + 1;
P2:   b = 2 * b;
      a = 2 * a;
```

กรณีนี้โปรเซส P1 และ P2 เข้าใช้งานตัวแปรทั้งสองภายใต้การควบคุมส่วนวิกฤตโดยควบคุมให้ทำงานทีละโปรเซส และได้ผลการทำงานที่ถูกต้อง

พิจารณาตัวอย่างต่อนี้ ทั้งโปรเซส P1 และ P2 ยังคงทำตามกฎเกณฑ์การควบคุมวิกฤต กล่าวคืออนุญาตเพียงหนึ่งโปรเซสเข้าใช้ทรัพยากรในเวลาหนึ่ง ๆ โปรเซสอื่นจะต้องรอจนกว่าโปรเซสที่กำลังใช้ทรัพยากร ทำงานจนเสร็จสิ้นก่อน

```
P1:   a = a + 1;
P2:   b = 2 * b;
P1:   b = b + 1;
P2:   a = 2 * a;
```

กรณีนี้จะเห็นว่า โปรเซส P1 และ P2 เข้าใช้งานตัวแปรทั้งสองภายใต้การควบคุมส่วนวิกฤตเช่นเดียวกัน แต่เป็นการทำงานสลับกันคนละคำสั่ง จะเห็นว่าผลลัพธ์ไม่สามารถรักษาคุณสมบัติ a = b ไว้ได้อีกต่อไป โดยที่หากเริ่มต้นตัวแปร a = b = 1 แล้ว ผลลัพธ์ที่ได้คือ a = 4, b = 3 ซึ่งปัญหาเกิดจากลำดับของการคำนวน ดังนั้นส่วนวิกฤตของกรณีนี้ (critical section) คือ ลำดับคำสั่ง

6.3 การควบคุมส่วนวิกฤตด้วยวิธีทางซอฟต์แวร์ (Mutual Exclusion: Software Approach)

การควบคุมส่วนวิกฤตเพื่อให้การใช้ทรัพยากรสอดประสานกันระหว่างโปรเซสสามารถทำได้ด้วยวิธีทางซอฟต์แวร์ วิธีการนี้สามารถใช้งานได้ทั้งระบบคอมพิวเตอร์ที่เป็นหน่วยประมวลผลเดี่ยว และระบบ

คอมพิวเตอร์ที่มีหน่วยประมวลผลหลายชุดแต่ใช้หน่วยความจำหลักร่วมกัน การควบคุมส่วนวิกฤตดังกล่าวกระทำการโดยควบคุมการการเข้าถึงชุดคำสั่งซึ่งอยู่ในระดับของหน่วยความจำหลัก (Lamport, 1991) วิธีการมีดังนี้

6.3.1 ขั้นตอนวิธีของเด๊กเกอร์ (Dekker's algorithm)

Dijkstra (Dijkstra, 1996) เป็นขั้นตอนวิธีสำหรับควบคุมส่วนวิกฤต จากการใช้ทรัพยากรของสองโปรเซส ขั้นตอนวิธีดังกล่าวได้รับการออกแบบโดยนักคณิตศาสตร์ชาวดัชชื่อ Dekker การอธิบายการออกแบบและกลไกการทำงานของขั้นตอนวิธีแบบ Dijkstra จะกระทำเป็นลำดับของความพยายาม ซึ่งจะสามารถชี้ให้เห็นภาพของปัญหาและการแก้ไขปัญหาเหล่านั้นเป็นขั้นเป็นตอน

<u>พยายามครั้งที่ 1 (First Attempt):</u>

ในช่วงเวลาใดเวลาหนึ่งโปรเซส P0 และ P1 ต้องการประมวลผลในส่วนวิกฤต (critical section) ในช่วงเวลาเดียวกัน ดังนั้นโปรเซสจะทำการตรวจสอบค่าในตัวแปร turn ซึ่งเป็นตัวแปรสาธารณะ (global variable) ที่โปรเซสต่าง ๆ สามารถเข้าถึงได้ หากตัวแปร turn มีค่าตัวเลขตรงกับรหัสประจำตัว (id) ของตัวมันเอง (เป็นรอบของฉัน) ก็จะประมวลผลส่วนวิกฤตได้ แต่หากไม่ตรงต้องคอยจนกว่าจะถึงรอบ เรียกกระบวนการนี้ว่า busy waiting

รูปที่ 6-7 แสดงการควบคุมส่วนวิกฤตด้วยโปรแกรมดังนี้ พิจารณาส่วนโปรแกรมของโปรเซส P0 เมื่อโปรเซสต้องการใช้งานส่วนวิกฤต ซึ่งเขียนแทนด้วย (/* critical section */) โปรเซส P0 จะทำการตรวจสอบตัวแปร turn หากค่ายังไม่ใช่ 0 (id ของ P0) จะทำการรอไปเรื่อย ๆ จนกระทั่งตัวแปร turn = 0 สังเกตุได้ว่าค่าตัวแปร turn กลายเป็น 0 ก็ต่อเมื่อโปรเซส P1 ใช้งานส่วนวิกฤตเสร็จสิ้น โปรเซส P0 จะได้สิทธิในการเข้าใช้งานส่วนวิกฤต และทำการคืนสิทธินี้กลับไปให้แก่โปรเซส P1 โดยการกำหนดค่า turn = 1

```
/* P0 */
While (turn != 0)
        { /* do noting */}
/* critical section */
turn = 1;
```

```
/* P1 */
While (turn != 1)
        { /* do noting */}
/* critical section */
turn = 0;
```

รูปที่ 6-7: ความพยายามครั้งที่ 1 ในการควบคุมส่วนวิกฤติ

ที่มา: (Stallings, 2017)

อย่างไรก็ตามการพยายามครั้งแรกพบปัญหาดังนี้ คือ 1) การสลับการประมวลผลส่วนวิกฤตกลับไปกลับมาระหว่าง 2 โปรเซส ทำให้มีปัญหากลไกไม่ตอบสนองต่อความต้องการของโปรเซสที่ต้องการประมวลผลในส่วนนี้บ่อยกว่า เช่น โปรเซส P0 ต้องการใช้ส่วนวิกฤตเพียงชั่วโมงละครั้ง ในขณะที่โปรเซส P1 ต้องการใช้ทุก ๆ นาที จะเห็นว่าโปรเซส P1 จะต้องรอตามจังหวะของโปรเซส P0 และ 2) ปัญหาการถูกกีดขวาง (block) เนื่องจากโปรเซสหนึ่งเกิดข้อผิดพลาด ทำให้ตัวแปร turn ไม่ถูกเปลี่ยนค่า จนทำให้โปรเซสอื่นไม่ได้รับสิทธิในการประมวลผลส่วนวิกฤตเลย

<u>พยายามครั้งที่ 2 (Second Attempt):</u>

ในความพยายามครั้งที่สอง Dekker ใช้ตัวแปรอาร์เรย์ flag[ ] เป็นค่า Boolean เปรียบเสมือนธงบอกถึงสถานะการใช้งานส่วนวิกฤตของโปรเซส เช่นหาก flag[0] = true เสมือนโปรเซส P0 ชักธงขึ้นเพื่อแสดงถึงสถานะโปรเซส P0 กำลังใช้งานส่วนวิกฤต ขั้นตอนวิธีมีกลไกดังนี้คือ โปรเซสที่ต้องการใช้งานส่วนวิกฤตต้องทำการตรวจสอบ flag[ฝั่งตรงข้าม] ว่าอยู่ในสถานะใด โปรเซสจะสามารถเข้าทำงานส่วนวิกฤตได้ก็ต่อเมื่อ flag[ฝั่งตรงข้าม} เป็นเท็จ ‘false’ นอกจากนี้ก่อนจะทำงานส่วนวิกฤตจะต้องเปลี่ยนสถานะ flagp[ตัวเอง] ให้เป็นจริง ‘true’ เพื่อประกาศสถานะการใช้งานส่วนวิกฤตให้โปรเซสอื่นได้ทราบ และเมื่อโปรเซสใช้งานส่วนวิกฤตเสร็จสิ้นแล้วก็จะให้ค่า flag[ตัวเอง] เป็นเท็จ ‘false’ (รูปที่ 6-8)

```
/* P0 */
While (flag[1])
        { /* do noting */}
Flag[0] = true;
/* critical section */
Flag[0] = false;
```

```
/* P1 */
While (flag[0])
        { /* do noting */}
Flag[1] = true;
/* critical section */
Flag[1] = false;
```

รูปที่ 6-8: ความพยายามครั้งที่ 2 ในการควบคุมส่วนวิกฤต

ที่มา: (Stallings, 2017)

ความพยายามครั้งที่สองนี้ แก้ไขปัญหาที่เกิดขึ้นกับครั้งก่อนได้บางส่วน กล่าวคือ หากเกิดกรณีโปรเซส P0 เกิดข้อผิดพลาดในขณะที่ทำงานนอกส่วนวิกฤต (flag[0] = ‘false’) แล้วจะไม่เกิดส่งผลกระทบต่อการทำงานของโปรเซส P1 แต่อย่างใด โปรเซส P1 ยังสามารถเข้าใช้ส่วนวิกฤตได้ทุกเมื่อ

อย่างไรก็ตามปัญหากีดขวางอาจเกิดขึ้น เช่น 1) หากโปรเซส P0 เกิดทำงานผิดพลาดในขณะที่อยู่ในช่วงประมวลผลส่วนวิกฤตซึ่งมีค่า flag[0] = ‘true’ และทำให้ flag[0] ไม่สามารถเปลี่ยนแปลงได้

จึงเกิดปัญหากีดขวาง ทำให้โปรเซส P1 ไม่สามารถเข้าใช้ส่วนวิกฤตได้ นอกจากนั้น 2) วิธีการนี้ยังไม่สามารถประกันการควบคุมส่วนวิกฤตได้ เช่น ในกรณีโปรเซส P0 และ P1 ตรวจสอบ flag[ ] ของกันและกันในเวลาพร้อม ๆ กัน และพบว่า flag[ฝั่งตรงข้าม] = ‘false’ ซึ่งหมายถึงสามารถเข้าใช้ส่วนวิกฤตได้ ส่งผลให้โปรเซสทั้งคู่เข้าไปประมวลผลในส่วนวิกฤตดังกล่าวในเวลาเดียวกัน

พยายามครั้งที่ 3 (Third Attempt):

ในครั้งที่ 3 เป็นความพยายามแก้ไขปัญหาไม่สามารถประกันการควบคุมการเข้าทำงานส่วนวิกฤตได้จากกลไกครั้งก่อน ปัญหาถูกแก้ไขโดยการให้มีการกำหนด flag[ตัวเอง] ให้เป็นจริงไว้ก่อนเมื่อต้องการใช้ส่วนวิกฤต พิจารณาที่ P0 กำหนด flag[0] = ‘true’ เมื่อต้องการเข้าใช้งานส่วนวิกฤต จากนั้นตรวจสอบ flag[ฝั่งตรงข้าม] (flag[1] = ?) หากพบว่า flag[1] = ‘true’ แล้ว P0 ต้องรอ จนกว่า P1 ทำงานในส่วนวิกฤตเสร็จสิ้นและเปลี่ยน flag[1] เป็น ‘false’ จากนั้น P0 จึงสามารถเข้าใช้งานส่วนวิกฤตได้ต่อไป (รูปที่ 6-9) จะเห็นได้ว่าไม่มีโอกาสที่โปรเซสทั้ง 2 ตัวเข้าทำงานส่วนวิกฤตพร้อมกัน ดังนั้นการควบคุมส่วนวิกฤตได้รับการประกัน

```
/* P0 */
Flag[0] = true;
While (flag[1])
        { /* do noting */}
/* critical section */
Flag[0] = false;
```

```
/* P1 */
Flag[1] = true;
While (flag[0])
        { /* do noting */}
/* critical section */
Flag[1] = false;
```

รูปที่ 6-9: ความพยายามครั้งที่ 3 ในการควบคุมส่วนวิกฤต

ที่มา: (Stallings, 2017)

อย่างไรก็ตามยังคงพบปัญหาที่เกิดขึ้นกับกลไกข้างต้น คือ 1) ไม่สามารถแก้ไขปัญหากีดขวาง (block) ได้ เมื่อโปรเซสอาจเกิดความผิดพลาดในการทำงานภายในส่วนวิกฤต จึงส่งผลให้โปรเซสไม่สามารถเปลี่ยนสถานะ flag[ตัวเอง] ให้เป็น ‘false’ ได้ จึงเกิดการกีดขวางโปรเซสอื่นจนไม่สามารถเข้าประมวลผลส่วนวิกฤตได้ 2) นอกจากนี้อาจเกิดปัญหาติดตาย (deadlock) เมื่อ flag ของโปรเซสทั้งคู่อยู่ในสถานะเป็น ‘true’ ซึ่งเกิดขึ้นในตอนต้นเมื่อโปรเซสต้องการใช้งานส่วนวิกฤต และกำหนดค่า flag[ตัวเอง] ให้เป็น ‘true’ พร้อมกัน จึงเป็นสาเหตุทำให้ไม่มีโปรเซสใดสามารถเข้าสู่ส่วนวิกฤตได้

พยายามครั้งที่ 4 (Fouth Attempt):

ในครั้งที่ 4 มีความพยายามแก้ไขปัญหาติดตายโดยการใช้ค่าหน่วงเวลา (delay) แบบสุ่ม เพื่อสร้างความแตกต่างของช่วงเวลาให้เกิดขึ้น ในกรณีที่มีสองโปรเซสต้องการใช้ส่วนวิกฤตและเปลี่ยนแปลง flag[ตัวเอง] เป็น ‘true’ พร้อมกัน ตัวอย่างเช่น P0 และ P1 ต้องการประมวลผลในส่วนวิกฤต จากนั้นต่างคนต่างเปลี่ยนค่า flag[0] และ flag[1] = ‘true’ และตรวจสอบสถานะฝั่งตรงข้าม ซึ่งพบว่า flag[ฝั่งตรงข้าม] เป็น ‘true’ จึงทำให้ต่างคนต่างเปลี่ยน flag[ตัวเอง] กลับมาเป็น ‘false’ อีกครั้ง พร้อมกับรอคอยช่วงระยะเวลาหนึ่งซึ่งได้จากการสุ่ม ในช่วงจังหวะนี้โปรเซสทั้งสอง ที่เคยทำงานในขั้นตอนพร้อมกัน เริ่มมีความแตกต่างเนื่องจากค่าหน่วงเวลาที่ไม่เท่ากัน ทำให้โปรเซสที่เวลารอคอยสั้นกว่าเริ่มเปลี่ยน flag[ตัวเอง] เป็น ‘true’ ก่อน และย้อนกลับไปตรวจสอบ flag[ฝั่งตรงข้าม] อีกครั้ง ซึ่งจะพบว่าเป็น flag[ฝั่งตรงข้าม] เป็น ‘false’ โปรเซสดังกล่าวจึงสามารถเข้าประมวลผลส่วนวิกฤตได้ในที่สุด ดังรูปที่ 6-10

```
/* P0 */
Flag[0] = true;
While (flag[1])
    { flag[0] = false;
      /* delay */
      flag[0] = true;  }
/* critical section */
Flag[0] = false;
```

```
/* P1 */
Flag[1] = true;
While (flag[0])
    { flag[1] = false;
      /* delay */
      flag[1] = true;  }
/* critical section */
Flag[1] = false;
```

รูปที่ 6-10: ความพยายามครั้งที่ 4 ในการควบคุมส่วนวิกฤต

ที่มา: (Stallings, 2017)

อย่างไรก็ตามยังคงพบปัญหา เมื่อโปรซสทั้งสองมี flag[ตัวเอง] เป็น ‘true’ เหมือนกันทั้งคู่ และใช้การหน่วงเวลาแบบสุ่ม เพื่อให้จังหวะเวลาการทำงานมีความแตกต่างกัน ตามกลไกที่กล่าวมาข้างต้น แต่พบว่าอาจมีการวนรอบอยู่ระยะหนึ่ง จนกระทั่งจังหวะเวลาการทำงานมีความแตกต่างกันเพียงพอ และทำให้ค่า flag แตกต่างกันได้ เรียกปัญหาลักษณะนี้ว่า ติดช่วงหนึ่ง (livelock)

ขั้นตอนวิธีที่ถูกต้อง (Correct Version):

ขั้นตอนวิธีของเด็กเกอร์ที่ถูกต้องที่สุดแก้ไขปัญหาที่เกิดขึ้นได้ ด้วยการใช้ตัวแปร ‘turn’ เป็นค่าที่บอกให้ทราบว่าเป็นรอบของโปรเซสใด แทนการใช้ค่าหน่วงเวลา ซึ่งสามารถแก้ไขได้ทั้งปัญหาติดตาย (deadlock), ติดช่วงหนึ่ง (livelock) และ ประกันคุณลักษณะควบคุมส่วนวิกฤต (mutual exclusion) ได้อีกด้วย กลไกของขั้นตอนวิธีคือใช้ตัวแปรสาธารณะ ‘turn’ เป็นตัวบ่งชี้ถึงรอบของโปรเซส และใช้ flag[ ] เปรียบเสมือนธง ซึ่งบ่งชี้ถึงสถานะการทำงานของโปรเซสในส่วนวิกฤต ตัวอย่างเช่น P0 ต้องการประมวลผลส่วนวิกฤต จึงเปลี่ยน flag[0] = ‘true’ และตรวจสอบ flag[ฝั่งตรงข้าม] หากพบว่า flag[1] == 1 จะทำการตรวจสอบรอบการทำงานเป็นของใคร โดยพิจารณาตัวแปร ‘turn’ หากมีค่าเป็น ‘1’ แสดงว่าเป็นรอบของ P1 จึงรอคอยจนกว่าจะถึงรอบตัวเอง (turn = 0) หากเมื่อใดที่ P0 พบว่าตัวแปร turn = 0 ก็จะสามารถเข้าทำงานส่วนวิกฤตได้ พร้อมเปลี่ยน flag[0] = true เพื่อประกาศให้โปรเซสอื่นทราบ และเมื่อทำงานส่วนวิกฤตเสร็จสิ้นก็จะทำการคืนสิทธิให้ P1 (turn = 1) และเปลี่ยนชักธงลง (flag[0] = false) ดังรูปที่ 6-11

```
                /* P0 */
Flag[0] = true;
While (flag[1])
   {  if (turn == 1)
         {  flag[0] = false;
            while (turn ==1)
                  /* do noting */
            flag[0] = true;
         }
   }
/* critical section */
turn = 1;
flag[0] = false;
```

```
                /* P1 */
Flag[1] = true;
While (flag[0])
   {  if (turn == 0)
         {  flag[1] = false;
            while (turn ==0)
                  /* do noting */
            flag[1] = true;
         }
    }
/* critical section */
turn = 0;
flag[1] = false;
```

รูปที่ 6-11: ขั้นตอนวิธีสมบูรณ์ในการควบคุมส่วนวิกฤต

ที่มา: (Stallings, 2017)

6.3.2 ขั้นตอนวิธีของปีเตอร์ซัน (Peterson's algorithm)

ขั้นตอนวิธีของ Peterson เป็นวิธีการที่เรียบง่ายกว่าขั้นตอนวิธีของ Dekker และสามารถแก้ไขปัญหาติดตาย (deadlock), อดตาย (starvation), และติดช่วงหนึ่ง (livelock) ได้ ตัวอย่างเช่นโปรเซส P0 และ P1 ต้องการประมวลผลในส่วนวิกฤต โดยเริ่มจากการเปลี่ยน flag[ตัวเอง] ให้มีค่า 'true' ซึ่งอาจจะทำในเวลาเดียวกันทั้งสองโปรเซส อย่างไรก็ตามโปรเซส P0 และ P1 ต้องกำหนดตัวแปร 'turn' ให้เป็นรอบของโปรเซสฝั่งตรงข้าม เนื่องจากตัวแปร 'turn' เป็นตัวแปรสาธารณะ ดังนั้นตัวแปร 'trun' จะได้รับค่าล่าสุดที่โปรเซสกำหนดให้ สมมุติ P0 กำหนดค่า turn = 1 หลังจากนั้นอีกเสี้ยววินาทีโปรเซส P1 กำหนดค่า turn = 0 ดังนั้นสิทธิถูกกำหนดให้เป็นรอบการทำงานของโปรเซส P0 จากนั้นโปรเซส P0 และ P1 ตรวจสอบ flag[ฝั่งตรงข้าม] และตัวแปร 'turn' หากพบว่าเป็นฝั่งตรงข้ามกำลังทำงานก็จะคอยจนกระทั่งได้สิทธิการประมวลผลส่วนวิกฤต เมื่อทำงานเสร็จสิ้นก็จะเอาธงลง ดังรูปที่ 6-12 (Hofri, 1990)

```
/* P0 */
While (true)
{
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn ==1)
       /* do noting */
   /* critical section */
   flag[0] = false;
}
```

```
/* P1 */
While (true)
{
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn ==0)
       /* do noting */
   /* critical section */
   flag[1] = false;
}
```

รูปที่ 6-12: การควบคุมส่วนวิกฤตด้วยขั้นตอนวิธีของ Peterson

ที่มา: (Stallings, 2017)

6.4 การจัดการประสานเวลาด้วยฮาร์ดแวร์ (Mutual Exclusion: Hardware Approach)

ในการควบคุมส่วนวิกฤต (mutual exclusion) โดยเฉพาะการใช้พื้นที่หน่วยความจำร่วมกัน สามารถกระทำได้ในระดับของคำสั่งในภาษาเครื่อง (machine instruction) ซึ่งเป็นคำสั่งพิเศษ คำสั่งพิเศษนี้ถูกออกแบบให้มีการทำงานรวดเร็ว โดยใช้เวลาการประมวลผลคำสั่งเพียง 1 วงรอบเครื่อง (1 machine cycle) เรียกคำสั่งประเภทนี้ว่า atomic instruction

นอกจากนั้น ถึงแม้ระบบคอมพิวเตอร์ที่เป็นแบบหน่วยประมวลผลหลายชุด (multi-processor system) ในเวลาหนึ่ง ๆ จะมีเพียงคำสั่งพิเศษนี้ถูกประมวลผลเพียงคำสั่งเดียว ซึ่งป้องกันไม่ให้เกิดกรณีโปรเซสได้รับสิทธิเข้าประมวลผลในส่วนวิกฤตพร้อมกัน ตัวอย่างคำสั่งควบคุมส่วนวิกฤตมี 2 ประเภทคือ ประเภทคำสั่งตรวจสอบ-กำหนดค่า (Test-Set instruction) และ คำสั่งประเภทแลกเปลี่ยนค่า (Exchange instruction) (Raynal, 1986) (Stone, 1993)

6.4.1 คำสั่งตรวจสอบ-กำหนดค่า (Test-Set instruction)

ฟังก์ชัน testset() เป็น atomic instruction ซึ่งมีการตรวจสอบค่าตัวแปร bolt ที่รับเข้ามา ซึ่งเป็นตัวแปรสาธารณะที่โปรเซสต่าง ๆ ใช้ร่วมกัน หากพบว่า bolt มีค่าเป็น 0 จึงกำหนดค่าเป็น 1 และส่งค่ากลับเป็น ‘true’ หากพบว่า bolt มีค่าเป็น 1 จะไม่ทำอะไร และส่งค่ากลับเป็น ‘false’

เมื่อโปรเซสใดต้องการใช้งานส่วนวิกฤต ก็จะเริ่มตรวจสอบค่าตัวแปร bolt ด้วยฟังค์ชั่น testset(bolt) ในฟังค์ชั่น testset(bolt) หากพบว่า bolt = 0 แล้ว โปรเซสนั้นจะได้เข้าทำงานในส่วนวิกฤต และตัวแปร bolt ถูกกำหนดให้เป็น 1 เพื่อป้องกันไม่ให้โปรเซสเข้ามาทำงานส่วนวิกฤตอีก เหตุการณ์ถูกขัดจังหวะในการทำฟังก์ชัน testset() จะไม่เกิดขึ้นเนื่องจากเป็นฟังกชั่นที่เป็นประเภท atomic instruction เมื่อโปรเซสทำงานส่วนวิกฤตเสร็จสิ้น ก็จะกำหนดค่าตัวแปร bolt เป็น 0 เพื่อให้โปรเซสอื่นมีโอกาสเข้าประมวลผลส่วนวิกฤตบ้าง ทั้งนี้ในทางตรงข้ามเมื่อโปรเซสตรวจสอบพบว่าตัวแปร bolt มีค่าเป็น 1 โปรเซสก็จะรอคอยจนกว่าตัวแปร bolt เปลี่ยนสถานะเป็น 0 ดังรูปที่ 6-13

```
/* program mutual exclusion */
const int n = /*number of processes*/;
int bolt = 0;
void P(int i)
{   while (true)
      {  while (!testset(bolt))
            /* do noting */ ;
         /* critical section */;
         bolt = 0;    }
}
```

```
    /*  Atomic test-set instruction */
Boolean  testset (bolt)
{
   if (bolt == 0)
    {bolt = 1;
       return true;
    }
    else
    {  return = false; }
}
```

รูปที่ 6-13: ควบคุมส่วนวิกฤตด้วยคำสั่งตรวจสอบ – กำหนดค่า (test – set instruction)

ที่มา: (Stallings, 2017)

6.4.2 คำสั่งประเภทแลกเปลี่ยนค่า (Exchange instruction)

ฟังค์ชั่น exchange (reg , mem) เป็น atomic instruction ซึ่งทำการแลกเปลี่ยนค่ากันระหว่างสองตัวแปร reg และ mem ซึ่งรับค่ามาจาก keyi และ bolt โดยที่ keyi เป็นตัวแปรภายในของโปรเซสนั้น ๆ ซึ่งมีค่าเป็น 1 สำหรับตัวแปร bolt เป็นตัวแปรสาธารณะที่สามารถเข้าถึงได้ทุกโปรเซส โดยโปรเซสใดที่ต้องการใช้งานส่วนวิกฤตจะทำการเรียกฟังก์ชัน exchange(keyi, bolt) ซึ่งจะสลับค่าตัวแปร bolt และ keyi โปรเซสใดที่พบว่า bolt = 0 ก่อนที่จะสลับค่าให้ตัวแปร keyi ก็จะออกจากการวนลูป และสามารถเข้าทำงานในส่วนวิกฤตได้ เมื่อเสร็จสิ้นงานก็จะสลับค่าตัวแปร bolt และ keyi เพื่อให้ตัวแปร bolt มีค่าเป็น ‘0’ อีกครั้ง ดังรูปที่ 6-14

```
/* program mutual exclusion */
const int n = /*number of processes*/
int  bolt = 0;
void P(int i)
{  int keyi;
   while (true)
     {  keyi = 1;
        while (keyi != 0)
             exchange(keyi  ,  bolt);
        /* critical section */;
        exchange(keyi  ,  bolt);
      }
}
```

```
    /*  Atomic exchange instruction */
void exchange (int reg , int mem)
 {
    int temp;
    temp = mem;
    mem = reg;
    reg = temp;
 }
```

รูปที่ 6-14: ควบคุมส่วนวิกฤตด้วยคำสั่งแลกเปลี่ยนค่า (Exchange instruction)

ที่มา: (Stallings, 2017, p. 215)

6.5 วิธีการเซมาฟอร์ (Semaphores)

วิธีการเซมาฟอร์ (Semaphores) เป็นเครื่องมือสำหรับการควบคุมการใช้งานส่วนวิกฤตของโปรเซสต่าง ๆ ให้มีความสอดประสานกัน นอกจากจะสามารถใช้ในการควบคุมส่วนวิกฤตในรูปแบบทีละ 1 โปรเซสแล้ว เซมาฟอร์ยังสามารถนำมาควบคุมการทำงานส่วนวิกฤตสำหรับโปรเซสหลาย ๆ ตัว ขึ้นอยู่กับความจุของส่วนวิกฤต

6.5.1 การทำงานของเซมาฟอร์

เซมาฟอร์ประกอบด้วย 2 ฟังก์ชัน ซึ่งเป็น atomic function คือ wait(s) และ signal(s) โดยที่ตัวแปร s เป็นตัวแปรสาธารณะเลขจำนวนเต็ม ซึ่งสะท้อนปริมาณความจุคงเหลือในส่วนวิกฤต ฟังก์ชันทั้งสองมีหน้าที่ดังนี้ :

1) ฟังก์ชัน wait() : ฟังก์ชัน wait() จะถูกเรียกใช้ เมื่อโปรเซสใด ๆ ต้องการเข้าใช้ส่วนวิกฤต โดยเซมาฟอร์จะทำการลดค่าตัวแปร s ลงหนึ่งค่า และตรวจสอบค่า s ตามเงื่อนไข หาก s >= 0 เซมาฟอร์อนุญาตให้โปรเซสเข้าประมวลผลส่วนวิกฤต แต่หาก s < 0 เซมาฟอร์นำโปรเซสบรรจุไว้ในคิวรอคอย (waiting queue)

2) ฟังก์ชัน signal() : ฟังก์ชัน signal() จะถูกเรียกใช้โดยโปรเซสที่ทำงานในส่วนวิกฤตเสร็จสิ้น และต้องการออกจากส่วนวิกฤต โดยเซมาฟอร์จะทำการเพิ่มค่าตัวแปร s ขึ้นหนึ่งค่า และตรวจสอบค่า s ตามเงื่อนไข หาก s <= 0 เซมาฟอร์ดึงโปรเซสจากคิวรอคอยที่มีความสัมพันธ์กับลำดับคิว เข้าทำงานส่วนวิกฤต

เงื่อนไขการพิจารณาค่าตัวแปร s สามารถสรุปความหมายได้ดังนี้

(1) s > 0 หมายถึง ส่วนวิกฤตยังมีที่ว่างสำหรับโปรเซสที่ร้องขอ

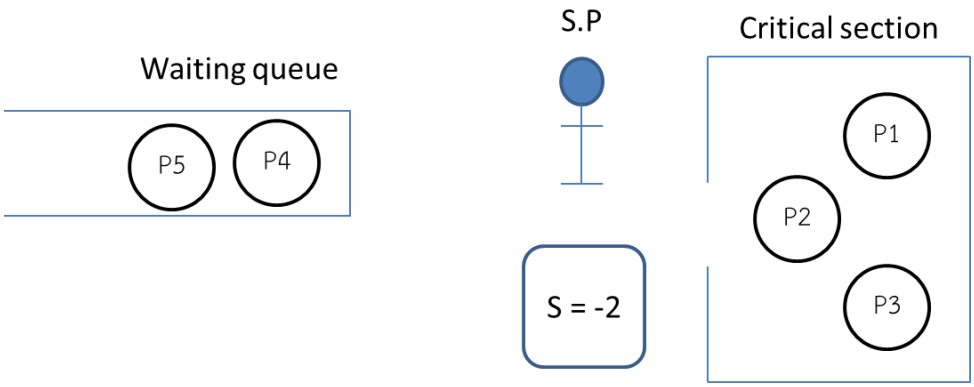
(2) s = 0 หมายถึง ส่วนวิกฤตมีโปรเซสทำงานเต็มความจุ โดยไม่มีโปรเซสใดรอคอยอยู่ในคิว

(3) s < 0 หมายถึง ส่วนวิกฤตมีโปรเซสทำงานเต็มความจุ และมีโปรเซสรอคอยตามจำนวนที่ติดลบ

พิจารณาตัวอย่างจากรูปที่ 6-15 เซมาฟอร์ (S.P) เปรียบเสมือนคนเจ้าหน้าที่ควบคุมการเข้าออกของผู้ใช้บริการ โดยสมมุติให้พื้นที่ส่วนวิกฤตคือร้านบริการตัดผม ซึ่งมีความจุตัดพร้อมกันได้ 3 คน เซมาฟอร์ควบคุมผู้มาขอใช้บริการตัดผมอยู่ยุติธรรม โดยให้แผ่นป้าย s เพื่อบ่งบอกว่าที่นั่งสำหรับตัดผมเหลือจำนวนเท่าใดในช่วงเวลาหนึ่ง ๆ เนื่องจากร้านตัดผมมีความจุ 3 ที่นั่ง ดังนั้นค่าเริ่มต้น s = 3 ผู้บริการคนแรก (P1) ร้องขอเพื่อเข้าใช้บริการด้วยการเรียกฟังก์ชัน wait() ดังนั้น S.P ลดค่าตัวแปร s ลงหนึ่งค่า (s = 2) ตรวจสอบค่า s >= 0 จึงอนุญาตให้ P1 เข้าใช้บริการ เซมาฟอร์ปฏิบัติต่อผู้ใช้บริการรายถัดมาเช่นเดียวกัน

จนกระทั่งถึงผู้ใช้บริการรายที่ 4 (P4) ซึ่งขณะนี้ s = 0 หมายถึงไม่มีที่ว่างเหลือให้บริการ เมื่อ P4 ร้องขอโดยเรียกฟังก์ชัน wait() เซมาฟอร์ทำการลดค่า s = -1 ตรวจสอบค่า s < 0 จึงนำ P4 เข้าสู่คิวรอคอย (waiting queue) หากมีรายอื่นร้องขอใช้บริการเพิ่มเติม เซมาฟอร์ก็กระทำเช่นเดิม เช่น P5 เรียกฟังก์ชัน wait() เซมาฟอร์จึงลดค่า s = -2 และนำ P5 มาต่อแถวในคิว

เมื่อมีผู้ใช้บริการคนหนึ่งคนใด (P1, P2, หรือ P3) เสร็จจากการตัดผม จะทำการแจ้งเซมาฟอร์ สมมุติให้ P1 ตัดผมเสร็จเป็นรายแรกจึงเรียกฟังก์ชัน signal() เซมาฟอร์ทำการเพิ่มค่า s ขึ้นหนึ่งค่า นำผู้ขอใช้บริการที่รออยู่ในคิวรายแรก P4 เข้าสู่ร้านตัดผม (ส่วนวิกฤต) เพื่อรับบริการต่อไป และตรวจสอบค่า s = -1 พบว่าเหลือผู้คอยใช้บริการหนึ่งราย การควบคุมจะดำเนินไปเช่นนี้ต่อเนื่องเรื่อยไป



รูปที่ 6-15: ตัวอย่างเซมาฟอร์ควบคุมส่วนวิกฤตที่มีความจุ 3 โปรเซส

6.5.2 ปัญหาของเซมาฟอร์ที่อาจเกิดขึ้น

เนื่องจากการใช้ตัวแปรแบบ counter ดังแสดงในรูปที่ 6-16 เพื่อนับจำนวนโปรเซสที่เข้าและออกจากส่วนวิกฤต ทำให้เซมาฟอร์มีจุดอ่อนจากความไม่สัมพันธ์กันระหว่างการนับเพิ่มและลบ ทั้งนี้อาจเกิดจากความผิดพลาดของการทำงานโปรเซส หรือแม้แต่โปรแกรมเมอร์เองที่ทำการเรียกฟังค์ชั่น wait() และ signal() ไม่ถูกต้อง เพียงการทำงานผิดพลาดของโปรเซสเดียวที่เกิดขึ้นในระบบ ก็จะทำให้การเข้าถึงส่วนวิกฤตผิดพลาดทั้งหมด

ตัวอย่างเช่น ลำดับการเรียกฟังก์ชันของโปรเซสหนึ่ง เรียก wait() → ประมวลผลส่วนวิกฤต → wait() ก็จะเกิดปัญหาติดตาย (deadlock) คือโปรเซสดังกล่าวไม่สามารถออกจากส่วนวิกฤตได้ สืบเนื่องจากการเรียกฟังก์ชันผิดพลาดของโปรแกรมเมอร์ หรืออีกหนึ่งตัวอย่าง โปรแกรมเมอร์สลับการเรียกฟังค์ชั่นเป็น signal() → ประมวลผลส่วนวิกฤต → wait() ก็จะทำให้เกิดปัญหาโปรเซสเข้าทำงานเกินความจุของส่วนวิกฤต เนื่องจากโปรเซสเรียกฟังก์ชัน signal() ทำให้เซมาฟอร์เข้าผิดคิดว่ามีโปรเซสเสร็จสิ้นภารกิจและออกจากส่วนวิกฤต จึงอนุญาตให้โปรเซสดังกล่าวเข้าใช้งานส่วนวิกฤต ทั้งที่ยังมีโปรเซสอื่นทำงานอยู่ในขณะนั้น เป็นต้น

```
struct semaphore {
    int  count;
    queueType  queue;
}
void  wait(semaphore   s)
{   s.count -- ;
    if (s.count <0)
    {   place this process in s.queue;
        block this process;
    }
}
void  signal(semaphore  s)
{   s.count ++ ;
    if (s.count <= 0)
    {  remove a process P from s.queue;
       place process P on ready list;
    }
}
```

```
/* program mutualexclusion */
const  int  n = /*number of processes*/
Semaphore  s = 1;
void P(int  i)
{   while (true)
    {   wait(s);
        /* critical section */;
        signal(s);
        /* remainder */;
     }
}
void main()
{
    parbegin(P(1), P(2), ..., P(n));
}
```

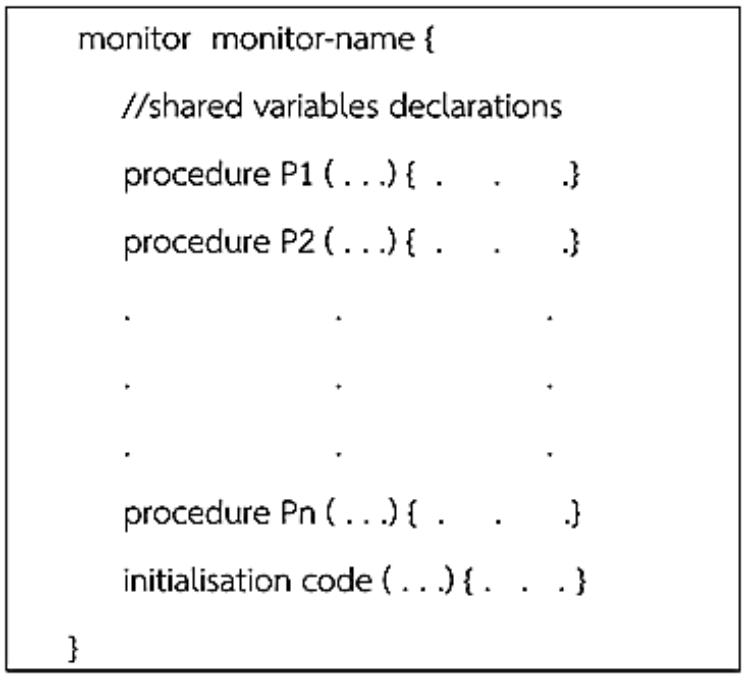
รูปที่ 6-16: ควบคุมส่วนวิกฤตด้วยเซมาฟอร์ (Semaphore)

(Stallings, 2017)

6.6 วิธีการตรวจสอบ (Monitors)

ตัวตรวจสอบ (Monitors) เป็นเครื่องมือในการจัดการปัญหาที่เกิดขึ้นกับเซมาฟอร์ โดยผู้พัฒนาออกแบบให้มีโครงสร้างการสอดประสานเวลาในระดับสูง (high-level synchronization construct) โดยใช้ภาษาคอมพิวเตอร์ชั้นสูง เช่น c++ หรือ java หลักการของการทำงานของตัวตรวจสอบ คือ ส่วนวิกฤต เช่น ส่วนพื้นที่การแบ่งปันข้อมูล (shared data) นั้นจะเข้าใช้งานได้ ต้องผ่านตัวดำเนินการแทน (procedure()) ดังนั้น ข้อมูลที่ใช้งานร่วมกันจึงมีลักษณะเป็นส่วนตัว (private) ซึ่งจะเข้าถึงได้โดยผ่านตัวดำเนินการเท่านั้น ตัวดำเนินการถูกประกาศเป็นสาธารณะ (public) เพื่อให้โปรเซสใด ๆ สามารถเรียกใช้ได้ ดังนั้นความถูกต้อง และความสอดคล้องกันของข้อมูลจะถูกควบคุมโดยตัวดำเนินการ

นอกจากนี้ตัวตรวจสอบ มีคุณสมบัติควบคุมการเข้าถึงส่วนวิกฤต (mutual exclusion) โดยการทำงานของตัวตรวจสอบ จะยินยอมให้มีเพียงโปรเซสเดียวที่สามารถเรียกใช้ตัวดำเนินการภายในตัวตรวจสอบในเวลาหนึ่ง ๆ ได้ ทำให้ข้อมูลที่ใช้ร่วมกันถูกใช้งานเพียงโปรเซสเดียวในแต่ละครั้ง



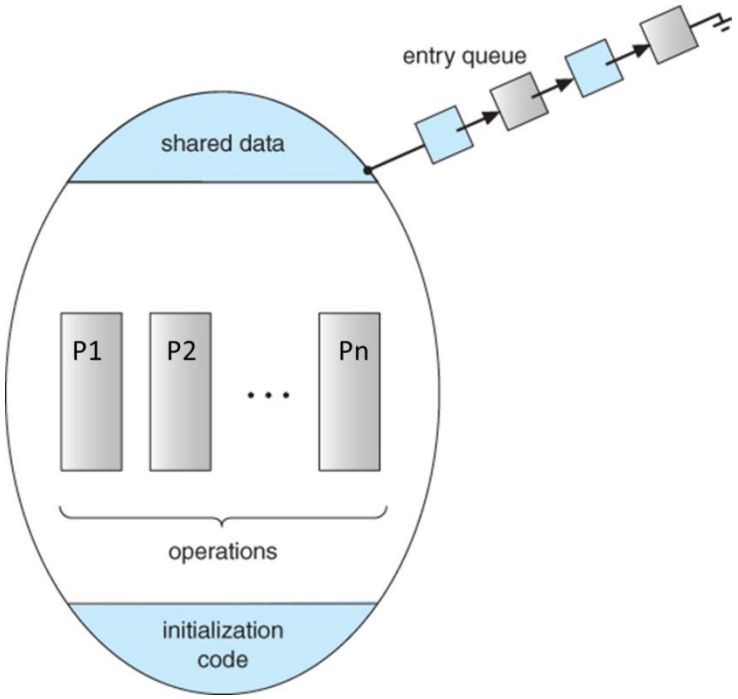
รูปที่ 6-17: โครงสร้างตัวตรวจสอบ (Monitors)

ที่มา: (Silberschatz, Galvin, & Gagne, 2010)

6.6.1 โครงสร้างตัวตรวจสอบ (structure of a monitor)

รูปที่ 6-17 แสดงโครงสร้างของตัวตรวจสอบซึ่งมีลักษณะโครงสร้างเชิงวัตถุในภาษาคอมพิวเตอร์ระดับสูง ประกอบด้วย ประเภทคลาสชนิดตัวตรวจสอบ (monitor) ชื่อตัวตรวจสอบ (monitor-name) และรายละเอียดของคลาส ภายในคลาส ส่วนวิกฤตคือข้อมูลที่ใช้ร่วมกัน (shared variables) ซึ่งถูกประกาศให้เป็นประเภทส่วนตัว (private) ภายในคลาสตัวตรวจสอบสามารถมีหลายตัวดำเนินการ ซึ่งมีได้ตั้งแต่ procedure P1() ถึง Pn() ในแต่ละตัวดำเนินการก็จะมีการปฏิบัติต่อข้อมูลแตกต่างกันไป เช่น เพิ่มค่า หรือลดค่า เป็นต้น ในส่วนท้ายของคลาสจะเป็นค่าเริ่มต้นตัวแปรต่าง ๆ ที่ต้องใช้ในคลาสตัวตรวจสอบ

เมื่อนำโครงสร้างมาเขียนภาพแผนผังตัวตรวจสอบ สามารถแสดงแผนผังได้ดังรูปที่ 6-18 โดยแบ่งออกเป็น 3 ส่วนคือ ส่วนวิกฤต (shared data) ส่วนตัวดำเนินการ (procedure or operation) และส่วนกำหนดค่าตัวแปรเริ่มต้น นอกจากนี้ยังมีคิว (entry queue) สำหรับโปรเซสใดที่เข้ามาทำงานภายในตัวตรวจสอบ อาจจะยังไม่เหมาะสมในการเข้าถึงข้อมูลในส่วนวิกฤต จึงจำเป็นต้องรอคอยเวลาภายในคิวเสียก่อน ซึ่งเป็นอีกหนึ่งคุณสมบัติของวิธีการตรวจสอบในการควบคุมความสอดคล้องข้อมูล



รูปที่ 6-18: แผนผังของตัวตรวจสอบ

ที่มา: (Silberschatz, Galvin, & Gagne, 2010)

6.6.2 การควบคุมการสอดประสานของข้อมูล

นอกจากการควบคุมการเข้าใช้ส่วนวิกฤตแล้ว วิธีการตรวจสอบนี้สามารถควบคุมข้อมูลให้มีความสอดคล้องกันระหว่างการใช้งานจากหลากหลายโปรเซส กลไกที่ใช้คือตัวแปรเงื่อนไข (condition variables) ซึ่งอาจมีได้หลายตัว เช่น

condition x, y;

โดย x และ y เป็นตัวแปรเงื่อนไข ตัวแปรเหล่านี้เป็นตัวแทนเงื่อนไขใด ๆ ที่อาจมีการเปลี่ยนแปลงได้ตลอดเวลา เช่น ตัวแปรเงื่อนไข ‘notfull’ อ้างถึงเงื่อนไขคิวที่ไม่เต็ม หรือตัวแปรเงื่อนไข ‘notempty’ ซึ่งอ้างถึงเงื่อนไขคิวไม่ว่างเปล่า เป็นต้น

การใช้งานตัวแปรเงื่อนไข สามารถกระทำผ่านตัวกระทำเพียง 2 ประเภทเท่านั้นคือ wait() และ signal() ดังตัวอย่างด้านล่าง

x.wait ();

หมายถึงโปรเซสที่ทำงานอยู่ในตัวตรวจสอบ (monitor) จำเป็นต้องรอคอยอยู่ในคิว เนื่องจากอยู่ภายใต้เงื่อนไข x ที่เกิดขึ้น และอีกหนึ่งตัวอย่าง

x.signal ();

หมายถึงโปรเซสที่รอคอยอยู่ในคิวภายใต้เงื่อนไข x ขณะนี้สามารถดำเนินการต่อได้ หรือหากไม่มีโปรเซสใด ๆ ในคิวจะไม่มีอะไรเกิดขึ้น

6.6.3 ตัวอย่างวิธีการตรวจสอบสำหรับปัญหาผู้ผลิตกับผู้บริโภค (monitor in producer-consumer problem)

มูลฐานของปัญหาผู้ผลิตกับผู้บริโภคคือความสอดคล้องกันของข้อมูล ทั้งฝ่ายผู้ผลิตและฝ่ายผู้บริโภคใช้พื้นที่ข้อมูลร่วมกัน สมมุติให้เป็นพื้นที่อาร์เรย์ชื่อ SHELL เปรียบเสมือนแผงวางขายสินค้า ปัญหาที่อาจเกิดขึ้นมีดังนี้:

1) ฝ่ายผู้ผลิต: ฝ่ายผู้ผลิตจะผลิตสินค้ามาวางขายในแผงสินค้า SHELL ซึ่งจำเป็นต้องคำนึงถึงพื้นที่ว่างในแผงสินค้า หากผลิตสินค้าโดยที่แผงสินค้าไม่ว่างเพราะมีสินค้าเต็มอยู่ก่อนแล้ว จะมีปัญหาไม่มีที่สำหรับวางสินค้าใหม่ ดังแสดงในรูปที่ 6-19 (b) ดังนั้นฝ่ายผลิตจำเป็นต้องรอให้แผงสินค้ามีพื้นที่ว่างสำหรับสินค้าใหม่

2) ฝ่ายผู้บริโภค: ฝ่ายผู้บริโภคจะหยิบสินค้าออกจากแผงวางขายสินค้า SHELL ซึ่งจะต้องตรวจสอบก่อนว่ามีสินค้าใดเหลืออยู่บนแผงสินค้าหรือไม่ หากไม่พบว่ามีสินค้าใดอยู่บนแผงสินค้าเลยดังแสดงในรูปที่ 6-19 (a) ก็จำเป็นต้องรอให้ฝ่ายผลิตนำสินค้ามาวางเสียก่อน

เมื่อพิจารณาปัญหาสินค้าบนแผงวางขาย เปรียบเสมือนโปรเซสที่ใช้งานพื้นที่ข้อมูลร่วมกันโดยมีฝ่ายหนึ่งผลิตข้อมูล และอีกฝ่ายนำข้อมูลไปใช้ ปัญหาการความสอดคล้องของข้อมูลดังกล่าว สามารถจัดการได้ด้วยวิธีตรวจสอบ (monitor) จากคุณสมบัติของวิธีการตรวจสอบมี 2 ประการสำคัญคือ 1) สามารถควบคุมโปรเซสให้ทำงานส่วนวิกฤตได้ (mutual exclusion) และ 2) สามารถควบคุมการสอดประสานเวลาการใช้ข้อมูลร่วมกันผ่านตัวแปรเงื่อนไข

SHELL

(a)

SHELL

A	B	C	D	E	F	G	H	I	J

(b)

รูปที่ 6-19: ปัญหาความสอดคล้องข้อมูลระหว่างผู้ผลิตและผู้บริโภค
(a) ไม่มีข้อมูลใน SHELL (b) ข้อมูลเต็ม SHELL

คลาสแสดงในรูปที่ 6-20 เป็นคลาสของตัวตรวจสอบชื่อ boundedBuffer ซึ่งมีตัวแปรอาร์เรย์ buffer[ ] เป็นส่วนวิกฤตที่ใช้งานร่วมกันระหว่างโปรเซสใด ๆ ตัวแปรนี้เป็นพื้นที่รองรับข้อมูล Item ที่ผลิตและนำไปใช้โดยโปรเซสที่เกี่ยวข้อง (บรรทัดที่ 2) นอกจากนี้คลาสตัวตรวจสอบมีการใช้งานตัวแปรเงื่อนไข 2 ตัวคือ ‘notfull’ และ ‘notempty’ โดยที่

1) ‘notfull’ ใช้กับเงื่อนไขพื้นที่เก็บข้อมูล buffer[ ] ไม่เต็ม และ

2) ‘notempty’ ใช้กับเงื่อนไขพื้นที่เก็บข้อมูล buffer[ ] ไม่ว่างเปล่า

คลาสตัวตรวจสอบมี 2 ตัวดำเนินการ (procedure) คือตัวดำเนินการ Append() (บรรทัดที่ 6 - 13) และ Take() (บรรทัดที่ 15 – 23) รองรับการใช้งานให้แก่โปรเซส โดยที่

1) Append() เป็นตัวดำเนินการสำหรับโปรเซสฝ่ายผลิตที่ต้องการนำข้อมูล Item มาวางไว้ที่อาร์เรย์ buffer[ ]

2) Take() เป็นตัวดำเนินการสำหรับโปรเซสฝ่ายผู้บริโภคที่ต้องการนำข้อมูล Item ในอาร์เรย์ buffer[ ] ออกไปใช้งาน

ส่วนสุดท้ายของคลาสเป็นการกำหนดค่าเริ่มต้นให้กับตัวแปร (Initialisation_code) (บรรทัด 25 – 29) ซึ่งตัวแปรที่ใช้ภายในคลาสคือ nextin, nextout และ count โดยที่

1) nextin เป็นตัวชี้ลำดับการนำเข้าข้อมูลแก่อาร์เรย์ buffer[ ]

2) nextout เป็นตัวชี้ลำดับการนำออกข้อมูลจากอาร์เรย์ buffer[ ] และ

3) count เป็นตัวนับจำนวนข้อมูลที่อยู่ในอาร์เรย์ buffer[ ]

```
1:  Monitor     boundedBuffer {
2:                  Item   buffer[k];
3:                  interger   nextin, nextout, count;
4:                  condition   notfull, notempty;
5:
6:                  Append (Item v) {
7:                        if (count == k)
8:                              notfull.wait();
9:                        buffer[nextin] = v;
10:                       nextin = (nextin + 1) mod k;
11:                       count ++;
12:                       notempty.signal();
13:                 }
14:
15:                 Item   Take () {
16:                       if (count == 0)
17:                             notempty.wait();
18:                       v = buffer[nextout];
19:                       nextout = (nextout + 1) mod k;
20:                       count --;
21:                       notfull.signal();
22:                       return v;
23:                  }
24:
25:                  Initialisation_code () {
26:                       nextin = 0;
27:                       nextout = 0;
28:                       count = 0;
29:                   }
30:  }
```

รูปที่ 6-20: คลาสตัวตรวจสอบสำหรับปัญหาผู้ผลิตและผู้บริโภค

ในการใช้งานตัวแปรอาร์เรย์ซึ่งเป็นส่วนวิกฤตต้องกระทำผ่านตัวตรวจสอบ (monitor) ชื่อ boundedBuffer ตัวตรวจสอบจะควบคุมให้เพียงหนึ่งโปรเซสทำงานภายใต้คลาสได้ ตัวอย่างเช่น เมื่อโปรเซส P0 มีข้อมูลต้องการเก็บบันทึกไว้ในตัวแปร buffer[ ] ก็จะทำการเรียกตัวดำเนินการ Append () โดยตัวดำเนินการจะทำการตรวจสอบก่อนว่าตัวแปร buffer[ ] มีข้อมูลเต็มแล้วหรือไม่ หากเต็มแล้วโปรเซส P0 ต้องรอคอยในคิวภายใต้เงื่อนไขตัวแปร buffer[ ] เต็ม (บรรทัด 7 – 8) เพื่อรอให้ตัวแปร buffer[ ] มีพื้นที่ว่างสำหรับข้อมูลใหม่ จึงจะทำการบันทึกข้อมูลได้ เมื่อ P0 สามารถบันทึกข้อมูลในตัวแปร buffer[ ] แล้ว ค่าของตัวแปรลำดับนำเข้าข้อมูล (nextin) เพิ่มขึ้นมาชี้ลำดับถัดไป พร้อมกับเพิ่มค่าตัวแปรนับจำนวน (count) (บรรทัด 10 – 11) จากนั้นส่งสัญญาณบอกให้โปรเซสซึ่งอาจรอคอยอยู่ในคิวภายใต้เงื่อนไขตัวแปร buffer[ ] ว่างเปล่า ให้ทราบว่าขณะนี้มีข้อมูลใหม่เข้ามาแล้ว และสามารถนำออกไปใช้ได้ (บรรทัด 12)

ในกรณีโปรเซสต้องการนำข้อมูลออก ตัวอย่างเช่น โปรเซส P1 ต้องการนำข้อมูลจากตัวแปร buffer[ ] ออกมาใช้งาน ต้องกระทำผ่านคลาสตัวตรวจสอบ โดยเรียกใช้ตัวดำเนินการ Take() ซึ่งในตอนเริ่มต้นตัวดำเนินการ Take() จะทำการตรวจสอบความพร้อมของข้อมูลว่าตัวแปร buffer[ ] มีข้อมูลอยู่หรือไม่ ถ้าหากไม่พบว่ามีข้อมูลอยู่เลย โปรเซส P1 จำเป็นต้องรอคอยอยู่ในคิวภายใต้เงื่อนไข buffer[ ] ว่างเปล่า (บรรทัด 16 – 17) หากโปรเซส P1 พบว่า buffer[ ] มีข้อมูล จะนำข้อมูลออก จากนั้นเพิ่มค่าตัวแปรชี้ลำดับการนำข้อมูลออก (nextout) และลดค่าตัวแปรนับจำนวนข้อมูลลง นอกจากนี้ตัวดำเนินการ Take() ส่งสัญญาณบอกให้โปรเซสที่รอคอยในคิวภายใต้เงื่อนไข buffer[ ] เต็ม ให้ทราบว่าขณะนี้มีที่ว่างในตัวแปร buffer[ ] แล้ว สามารถนำข้อมูลใหม่บันทึกลงในตัวแปร buffer[ ] ได้ (บรรทัด 21)

บทสรุป

ในบทนี้ได้อธิบายหน้าที่ระบบปฏิบัติการ ในการจัดการหลาย ๆ โปรเซสทำงานพร้อม ๆ กันให้มีความสอดประสานเวลา เมื่อมีโปรเซสทำงานอยู่ในระบบ การจัดสรรทรัพยากรซึ่งมีอยู่อย่างจำกัด ให้สามารถใช้งานร่วมกันได้จึงเป็นสิ่งสำคัญสำหรับระบบปฏิบัติการ ทรัพยากรที่ใช้ร่วมกันถือเป็นส่วนวิกฤตที่ต้องถูกควบคุมจากการใช้งานของบรรดาโปรเซส การควบคุมดังกล่าวมีวัตถุประสงค์เพื่อป้องกันปัญหาความขัดแย้งกันของข้อมูล ความไม่ถูกต้องข้อมูล ปัญหาติดตาย ปัญหาอดตาย และปัญหาติดช่วงระยะหนึ่ง การควบคุมส่วนวิกฤต (mutual exclusion) สามารถกระทำได้ใน 2 ระดับคือ ซอฟต์แวร์ และฮาร์ดแวร์

การควบคุมส่วนวิกฤตในระดับซอฟต์แวร์นำเสนอ 2 วิธีคือขั้นตอนวิธีของ Dekker และขั้นตอนวิธีของ Peterson ทั้งสองวิธีมุ่งควบคุมสองโปรเซสที่แข่งขันกันเพื่อเข้าใช้ส่วนวิกฤต สามารถเข้าใช้ได้เพียงครั้งละหนึ่งโปรเซส โดยทั้งสองวิธีใช้ตัวแปร flag[ ] เพื่อบอกสถานะการครอบครองการใช้งานส่วนวิกฤตให้แก่โปรเซสอีก

ฝั่งได้รับทราบ นอกจากนั้นตัวแปร turn ซึ่งเป็นประเภทสาธารณะที่โปรเซสทั้งสองมองเห็นได้ มีไว้สำหรับบ่งบอกถึงรอบการเข้าใช้ส่วนวิกฤตเป็นของโปรเซสใด จากกลไกแต่ละแบบทำให้สามารถควบคุมให้โปรเซสเพียงโปรเซสเดียวเข้าใช้งานส่วนวิกฤต ส่วนอีกโปรเซสต้องอยู่ในสถานะรอคอยให้โปรเซสก่อนหน้าทำงานเสร็จเสียก่อนจึงจะเข้าใช้งานได้

การควบคุมส่วนวิกฤตในระดับฮาร์ดแวร์ สามารถทำได้โดยระบบปฏิบัติการมีคำสั่งพิเศษซึ่งอยู่ในระดับภาษาเครื่องให้ใช้งาน คำสั่งพิเศษเป็นประเภท atomic instruction ซึ่งสามารถประมวลคำสั่งเสร็จได้ภายในหนึ่งรอบเครื่อง (1 cycle machine) ประกอบด้วย 2 เทคนิคคือ แบบตรวจสอบ-กำหนดค่า (Test-Set instruction) และแบบแลกเปลี่ยนค่า (Exchange instruction) การควบคุมในระดับฮาร์ดแวร์สามารถรองรับการใช้งานของโปรเซสหลาย ๆ โปรเซสที่ประสงค์เข้าใช้ส่วนวิกฤต การให้สิทธิแก่โปรเซสใดโปรเซสหนึ่งทำโดยผ่านกลไกคำสั่งพิเศษดังกล่าว ทำให้สิทธิที่ให้ไปนั้นไม่มีโอกาสซ้ำซ้อนกันถึงแม้จะเป็นระบบคอมพิวเตอร์แบบหลายหน่วยประมวลผลก็ตาม

นอกจากนี้การควบคุมส่วนวิกฤตยังสามารถกระทำผ่านเครื่องมือพิเศษคือ เซมาฟอร์ (semaphore) และตัวตรวจสอบ (monitor) เซมาฟอร์สามารถควบคุมจำนวนโปรเซสให้ทำงานส่วนวิกฤตเป็นไปตามความจุของส่วนวิกฤตได้ โดยการควบคุมการเข้าออกผ่านฟังก์ชัน wait() และ signal() เซมาฟอร์ยังสามารถจัดการคิวให้แก่โปรเซสได้เข้าใช้ส่วนวิกฤตได้อีกด้วย ในด้านของตัวตรวจสอบ (monitor) แก้ไขปัญหาที่อาจเกิดขึ้นกับเซมาฟอร์จากความผิดพลาดของโปรแกรมเมอร์ โดยตัวตรวจสอบมีคุณสมบัติ 2 ประการสำคัญคือควบคุมการเข้าส่วนวิกฤตได้โดยคอมไพรเลอร์ และควบคุมความสอดคล้องกันของข้อมูลโดยใช้ตัวแปรเงื่อนไขมากำกับ

แบบฝึกหัดท้ายบท

6.1. จงให้ความหมายสิ่งต่อไปนี้ mutiprograming, multiprocessing และ distributed processing

6.2. จงอธิบายปัญหาที่เกิดขึ้นกับการใช้ทรัพยากรร่วมกันหลายโปรเซสที่เกิดขึ้นทั้งในระบบคอมพิวเตอร์หน่วยประมวลผลเดี่ยว และหน่วยประมวลผลหลายชุด

6.3. จงอธิบายทรัพยากรที่เป็นส่วนวิกฤต สาเหตุแห่งการวิกฤต พร้อมยกตัวอย่าง

6.4. จงอธิบายหลักการของการควบคุมส่วนวิกฤต (mutual exclusion) มาให้เข้าใจ

6.5. ปฏิสัมพันธ์ระหว่างโปรเซสมีอะไรบ้าง อธิบายในแต่ละแบบ

6.6. อธิบายปัญหาที่เกิดขึ้นของปฏิสัมพันธ์ของโปรเซสในแต่ละแบบ

6.7. จงอธิบายปัญหาที่เกิดขึ้นกับขั้นตอนวิธีควบคุมส่วนวิกฤตของ Dekker ในความพยายามครั้งที่ 3 และวิธีแก้ไขในความพยายามครั้งที่ 4

6.8. จงอธิบายหน้าที่ของตัวแปร flag[ ] และ turn ในขั้นตอนวิธีของ Dekker

6.9. จงเปรียบเทียบความแตกต่างของวิธีการของ Perterson และ Dekker (สมบูรณ์)

6.10.จงเขียนขั้นตอนวิธีในแบบรหัสเทียบ (psedo code) การควบคุมการใช้ฟังก์ชัน echo() ด้วยวิธีของ Peterson โดยที่

```
#include <stdio.h>

void echo()
{
     chin = getchar();
     chout = chin;
     putchar(chout);
}
```

6.11.จงอธิบายสาเหตุที่การควบคุมส่วนวิกฤตด้วยฮาร์ดแวร์จึงสามารถใช้ได้กับระบบคอมพิวเตอร์ที่มีหน่วยประมวลผลหลายชุด

6.12.จงอธิบายกลไกของการควบคุมส่วนวิกฤตด้วยคำสั่งตรวจสอบ-กำหนดค่า (Test-Set instruction)

6.13.จงอธิบายฟังก์ชัน wait() และ signal() ของเซมาฟอร์ (semaphore)

6.14.จงอธิบายการทำงานของเซมาฟอร์ สำหรับส่วนวิกฤตที่มีความจุ 1 โปรเซส ในขณะที่มีโปรเซสต้องใช้งาน 3 โปรเซส (อธิบายเป็นขั้นตอน)

6.15.จงอธิบายวิธีการตรวจสอบ (monitor) แก้ไขปัญหาสิ่งใดที่เกิดขึ้นกับเซมาฟอร์

6.16.จงอธิบายฟังก์ชัน wait() และ signal() ในวิธีการตรวจสอบ

6.17.จงอธิบายการป้องกันไม่ให้โปรเซสผู้ผลิต ใส่ข้อมูลเกินเนื้อที่เก็บ

6.18.จงอธิบายการควบคุมโปรเซสผู้บริโภค ดึงข้อมูลในขณะที่พื้นที่เก็บข้อมูลว่างเปล่า

บรรณานุกรม

B.P.Miller. (2013). *Deadlock*. Retrieved 10 08, 2017, from Lecture Note of Operating Concepts: http://pages.cs.wisc.edu/~bart/537/lecturenotes/s12.html

Difference between Deadlock and Starvation. (2017). Retrieved 10 08, 2017, from Difference between, Descriptive Analysis and Comparison: http://www.differencebetween.info/difference-between-deadlock-and-starvation

Dijkstra, E. (1996). Cooperating Sequential Processes. *IEEE Press (Reprinted in Greate Papers in Computer Science)*.

Hofri, M. (1990). Proof of a Mutual Exclusion Algorithm. *Operating System Revies*.

Lamport, L. (1991). The Mutual Exclusion Problem Has Been Solved. *Communications of the ACM*.

Raynal, M. (1986). *Algorithms for Mutual Exclusion.* Cambridge: MIT Press.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2010). *Operating System Concepts* (8th ed.). Asia: John Wiley & Sons.

Stallings, W. (2017). *Operating Systems : Internals and Design Principles* (Fourth Edition ed.). Pearson.

Stone, H. (1993). *High-Performance Computer Architecture.* Reading: Addison-Wesley.