บทที่ 6 การประสานงานระหว่างโพรเซส (2)

วัตถุประสงค์ของเนื้อหา

- การสร้างส่วนวิกฤติด้วย semaphore
- ศึกษากรณีประเด็นปัญหาตัวอย่างที่นำเซมาฟอร์ไปใช้งาน
- ศึกษาถึงประเด็นอื่นๆ ที่สัมพันธ์กับการสร้างส่วนวิกฤติ

สิ่งที่คาดหวังจากการเรียนในบทนี้

- นักศึกษาเข้าใจถึงเหตุผลของการใช้งานส่วนวิกฤติ และเข้าใจถึงกลไกการทำงานของส่วนวิกฤติ
- นักศึกษาเข้าใจถึงหลักการพื้นฐานของการทำงานของฮาร์ดแวร์คอมพิวเตอร์ และระบบปฏิบัติการ ในส่วนที่เกี่ยวเนื่องกับส่วนวิกฤติ
- นักศึกษาเข้าใจถึงการประยุกต์กลไกต่างๆ ของฮาร์ดแวร์ และระบบปฏิบัติการ เพื่อสนับสนุนทฤษฎีของส่วนวิกฤติ

วัตถุประสงค์ของปฏิบัติการท้ายบท

- นักศึกษานำเอาขั้นตอนวิธีต่างๆ ที่ได้เรียนรู้ ไปใช้สร้างส่วนวิกฤติ
- นักศึกษาประยุกต์กลไกของระบบปฏิบัติการที่มี ในการจัดการส่วนวิกฤติ

สิ่งที่คาดหวังจากปฏิบัติการท้ายบท

- นักศึกษาเข้าใจและเห็นภาพถึงสถานะแข่งขัน และเข้าใจถึงกลไกการทำงานของส่วนวิกฤติ
- นักศึกษาสามารถใช้งานกลไกของระบบปฏิบัติการ ไปประยุกต์สร้างกลไกส่วนวิกฤติ เพื่อแก้ไขปัญหาในทางปฏิบัติได้จริง
- นักศึกษาได้เห็นภาพเบื้องต้นของสภาวะติดตาย (deadlock) ซึ่งจะนำไปสู่ความเข้าใจที่ลึกซึ้งขึ้นไปในบทถัดไป

เวลาที่ใช้ในการเรียนการสอน

- ทฤษฎี 2 ชั่วโมง
    - การใช้งานเซมาฟอร์ 0.5 ชั่วโมง
    - กรณีศึกษาที่น่าสนใจ 1 ชั่วโมง
    - เนื้อหาอื่นๆ ที่สัมพันธ์กับส่วนวิกฤติ 0.5 ชั่วโมง
- ปฏิบัติ 2 ชั่วโมง
    - ศึกษาตัวอย่างเซมาฟอร์ และกรณีประเด็นปัญหาต่างๆ 1 ชั่วโมง
    - การทำความเข้าใจและศึกษาเบื้องต้นถึงกรณีสภาวะติดตาย 1 ชั่วโมง

บทที่ 6 การประสานงานระหว่างโพรเซส (2)

6.1 เซมาฟอร์ (Semaphores)

เซมาฟอร์เป็นอีกกลไกที่สามารถนำมาใช้เพื่อการจัดการการประสานงานระหว่างโพรเซสและสภาวะวิกฤติได้ โดย

- กำหนดตัวแปร เซมาฟอร์ S เป็นข้อมูลจำนวนเต็มที่มีค่าเริ่มต้นค่าหนึ่ง
- ตัวแปร S นี้จะถูกจัดการด้วยการดำเนินการแบบ atomic สองตัวคือ
    - wait(S) (แต่เดิมตัวนี้ใช้ศัพท์ว่า P มาจากภาษาดัทช์ proberen แปลว่า to test)
    - signal(S) (แต่เดิมตัวนี้ใช้ศัพท์ว่า V มาจาก verhogen แปลว่า to increment)

```
wait(S){
      while (S<=0) ;
      S--;
}
```

```
signal(S){
      S++;
}
```

- ด้วยความเป็น atomic operation ของ wait() และ signal() หมายความว่าจะมีเพียงโพรเซส(หรือเธรด)เดียวเท่านั้นที่จะเข้าถึงตัวแปร S นี้เพียงตัวเดียวในเวลาใดเวลาหนึ่ง และโพรเซสจะสามารถผ่าน wait() ไปได้หากค่าใน S มีค่ามากกว่าศูนย์ แต่ถ้า S มีค่าเท่ากับศูนย์ในจังหวะที่โพรเซสเรียกใช้ wait() โพรเซสก็จะหยุดรอจนกว่าค่าใน S จะเพิ่มมากกว่าศูนย์ จึงจะเดินทางต่อไปได้
- สังเกตว่ากลไกของเซมาฟอร์นั้น มีความคล้ายคลึงกับกลไกของ mutex เพียงแต่มีความซับซ้อนมากขึ้นที่ตัว lock หรือ S ในเซมาฟอร์ไม่ได้มีค่าแค่เป็น จริง/เท็จ เหมือนใน mutex แต่สามารถมีค่าเป็นตัวเลขจำนวนเต็มบวก
- เซมาฟอร์ที่ค่าใน S มีค่าได้มากกว่าหนึ่ง เรียกว่า**เซมาฟอร์นับ (**counting semaphore**)**
- เซมาฟอร์ที่ค่าใน S มีค่าเท่ากับศูนย์หรือหนึ่งเท่านั้น เรียกว่า**เซมาฟอร์ไบนารี (**binary semaphore**)** ทำให้ผลการทำงานจะมีค่าเทียบเท่ากับการทำ mutex lock ตามปกติ
- เราสามารถใช้ counting semaphone เพื่อกำหนดจำนวนโพรเซส(หรือเธรด) ที่จะผ่านเข้าสู่ส่วนวิกฤติได้ในเวลาเดียวกัน ทำให้เราสามารถควบคุมการใช้ทรัพยากรที่มีให้ใช้มากกว่าหนึ่งหน่วยต่อประเภทได้ (โดยควบคุมผ่านค่าเริ่มต้นของ S)

การใช้เซมาฟอร์ไบนารี

ในกรณีนี้ เรากำหนดค่าเริ่มต้น S ให้เท่ากับ 1 โครงของส่วนโปรแกรมจะเป็นดังนี้

```
while(1){
      //  ส่วนก่อนเข้าส่วนวิกฤติ
      ....
      waiting(S);
      // ส่วนวิกฤติ
      ....
      signal(S);
      //ส่วนหลังส่วนวิกฤติ
}
```

การใช้เซมาฟอร์เพื่อประสานงานการทำงานระหว่างสองโพรเซส

เราใช้ binary semaphore และกำหนดค่าเริ่มต้นใน S เป็นศูนย์ โดยมีตัวอย่างการใช้งานดังโครงด้านล่าง

โพรเซสแรก
```
while(1){
      ....
      signal(S);
      ....
}
```

โพรเซสที่สอง
```
while(1){
      ....
      wait(S);
      ....
}
```

จากตัวอย่างขั้นตอนวิธีข้างต้น เมื่อโพรเซสที่สองทำงานไปจนถึง wait() ก็จะหยุดรอจนกระทั่งโพรเซสแรกทำงานไปจนถึง signal() ซึ่งส่งผลทำให้ค่า S เป็น 1 และทำให้โพรเซสที่สองสามารถทำงานต่อจาก wait() ที่ค้างอยู่ได้โดยในจังหวะดังกล่าวโพรเซสที่สองก็จะลดค่า S เป็น 0 อีกครั้ง ทำให้เมื่อวนกลับมาทำงานถึงโค้ด ณ ตำแหน่ง wait() ก็จะหยุดรอจนกว่าโพรเซสแรกทำงานไปจนถึง signal() ในรอบถัดไป ในลักษณะดังกล่าว การทำงานของโพรเซสแรกและโพรเซสที่สองแม้ว่าจะใช้เวลาในการทำงานในส่วนอื่นๆ ที่แตกต่างกัน แต่ก็จะสามารถกลับมาเข้าจังหวะพร้อมกันได้ที่ตำแหน่ง signal() และ wait() ดังกล่าว

การปรับปรุงเซมาฟอร์และกลไกที่คล้ายคลึงกันเพื่อนำไปใช้งานจริง

เซมาฟอร์ และกลไกอื่นที่ทำงานคล้ายคลึงกันในโครงสร้างของการวนลูปรอเพื่อเข้าสู่ส่วนวิกฤตินั้น มีข้อเสียคือส่วนคำสั่งวนรอบ while ที่ทุกโพรเซส(หรือเธรด) จะต้องหยุดรอ การวนรอด้วยการทำคำสั่ง NOP (หรือที่เรานำเสนอด้วย null statement ในภาษาซี) นั้น แม้ว่าไม่ได้เกิดผลลัพธ์ใดๆ เกิดขึ้น แต่ซีพียูยังคงจำเป็นต้องทำคำสั่งและเสียเวลาในการทำงานอยู่ดี ลักษณะดังกล่าวเรียกว่า busy waiting ซึ่งแม้ว่าจะเขียนโค้ดได้ง่าย และการเสียเวลารอจะมีน้อยลงถ้าโพรเซสต่างๆ เข้าใช้ส่วนวิกฤติไม่บ่อย แต่ถ้ามีโพรเซสหยุดรอส่วนวิกฤติเป็นจำนวนมาก ก็จะส่งผลทำให้เกิดการคำนวณที่ไม่จำเป็น (การวนรอบหยุดรอ) อยู่มาก เป็นการเสียเปล่าโดยใช่เหตุ

หลักการแก้ไขคือ แทนที่โพรเซสจะเสียเวลาวนรอที่จะเข้าส่วนวิกฤติ ก็จะเพิ่มกลไกการโยกโพรเซสดังกล่าวเข้าไปรอใน waiting queue และเมื่อโพรเซสที่ผ่านพ้นส่วนวิกฤติมาแล้ว ก็จะไปโยกโพรเซสที่อยู่ใน waiting queue กลับเข้ามาสู่ ready queue เพื่อรอให้ถูกเรียกมาประมวลต่อไป

โดยมีโอเปอเรชันเพิ่มขึ้นอีกสองตัว

- ฟังก์ชัน block() เรียกโดยโพรเซสที่จะเข้าสู่การรอ เพื่อส่งตัวเองให้เข้าไปอยู่ใน waiting queue
- ฟังก์ชัน wakeup() เรียกโดยโพรเซสที่จะพ้นจากส่วนวิกฤติ เพื่อดึงโพรเซสที่รอใน waiting queue กลับมารอประมวลต่อ

waiting list อาจนำเสนอได้ด้วยลิงค์ลิสต์โดยแต่ละหน่วยอาจเก็บ context ของโพรเซสไว้ ดังเช่น

```
typedef struct{
     int value;
     struct process *list;
}semaphore;
```

ส่วนการดำเนินการ wait() และ signal() ก็จะแก้ไขมาเป็นดังนี้

```
wait(semaphore *S){
      S->value--;
      if(S->value < 0){
         S->insert(S->list,P);
         block();
      }
}
```

```
signal(semaphore *S){
      S->value++;
      if(S->value <=0){
         P=S->remove(S->list);
         wakeup(P);
      }
}
```

ในกรณีนี้จะเห็นว่า เมื่อมีโพรเซสใหม่ที่พยายามจะเข้าสู่ส่วนวิกฤติเมื่อไม่มีทรัพยากรเหลือแล้ว ค่า S จะมีค่าน้อยกว่าศูนย์ จะเพิ่ม process context ปัจจุบันเข้าลิงค์ลิสต์ แล้วจึงบล็อกโพรเซสดังกล่าว และเมื่อจบส่วนวิกฤติ โพรเซสที่จบส่วนวิกฤติจะไปเปิดลิงค์ลิสต์เพื่อดึงเอา process context ใน waiting list ดังกล่าวเพื่อสั่งให้กลับมาอยู่ใน ready queue เพื่อให้พร้อมเข้าสู่ส่วนวิกฤติต่อไป

สังเกตว่า การดำเนินการ wait() และ signal() มีความซับซ้อนมากขึ้น ด้วยกลไกบังคับที่ว่า จะมีเพียงโพรเซสเดียว ณ เวลาใดเวลาหนึ่งเท่านั้นที่สามารถเรียก wait() และ signal() ได้ (atomic) ลักษณะของการทำงานจึงไม่สามารถทำงานในลักษณะ single instruction ได้เหมือนในกรณีหัวข้อก่อนหน้า

ในทางปฏิบัตินั้น กรณีของคอมพิวเตอร์ที่มีซีพียูเดียว มักจะใช้วิธีการยกเลิกการอินเทอร์รัพต์เมื่อเข้าสู่ฟังก์ชัน และคืนการอินเทอร์รัพต์เมื่อจะจบฟังก์ชัน สำหรับกรณีคอมพิวเตอร์หลายซีพียู การยกเลิกอินเทอร์รัพต์จะต้องกระทำกับซีพียูทุกหน่วย ซึ่งสร้างความซับซ้อนและเสียเวลาในการทำงานโดยรวมมากขึ้น ในทางปฏิบัติระบบปฏิบัติการจึงต้องมีทางเลือกอื่นนอกเหนือจากการใช้ semaphore แบบซับซ้อนนี้ไว้ด้วย

ประเด็นที่น่าสนใจอีกอย่างเกี่ยวกับการพยายามแก้ไขปัญหาการรอของโพรเซสที่จะเข้าสู่ส่วนวิกฤตินั้น แม้ว่าเราจะได้แก้ไขหรือลดการคำนวณที่ไม่จำเป็นลง แต่นั่นก็ไม่ได้หมายความว่าเราแก้ไขปัญหาเรื่องสภาวะวิกฤติของโพรเซสลง โพรเซสยังคงต้องรอเข้าเพื่อประมวลสภาวะวิกฤติเหมือนเดิม และในกรณีที่การทำงานในโค้ดส่วนสภาวะวิกฤตินั้น หากมีความยาวไม่มากนัก (เช่นโดยทั่วไปมักแค่เป็นการเขียนหรืออ่านหน่วยความจำหนึ่งค่า) โค้ดส่วนการจัดการ wait() และ signal() ก็จะมีความยาวมากกว่าโค้ดส่วนวิกฤติ (เมื่อเทียบว่าโค้ดส่วน wait() และ signal() อาจจะใช้สัก 10 ชุดคำสั่งภาษาเครื่อง ในขณะที่ส่วนวิกฤติอาจจะกินเพียงสองสามคำสั่งของภาษาเครื่อง) ทำให้ส่วนวิกฤตินั้นต้องรอทำงานจากการเสียเวลาคำนวณในส่วนการจัดการ ดังนั้นสำหรับกรณีที่มีส่วนวิกฤติทำงานไม่มากนัก หรือวนรอบของโพรเซสมีความยาวไม่มากนัก เราก็จะพบว่าการจัดการในลักษณะดังกล่าวนี้ไม่ค่อยมีประสิทธิภาพเท่าใดนัก แต่สำหรับโพรเซสที่เข้าใช้ส่วนวิกฤติไม่บ่อย โพรเซสลักษณะดังกล่าวก็จะสามารถใช้งานกลไกนี้ได้อย่างคุ้มค่า

6.2 คุณลักษณะการมีชีวิต (Liveness)

คุณลักษณะการมีชีวิต (Liveness) ของโพรเซส หมายถึงคุณลักษณะต่างๆ ที่เป็นอยู่ของระบบที่รับประกันว่าโพรเซสต่างๆ ในระบบจะสามารถทำงานคืบหน้าต่อไปได้

- โพรเซสต่างๆ ที่ทำงานภายใต้ลักษณะเช่นนี้ แม้ว่าอาจจะต้องหยุดรอที่ทางเข้าส่วนวิกฤติ แต่ก็จะต้องสามารถเข้าส่วนวิกฤติและทำงานต่อไปได้ในที่สุด
- การรอคอยแบบไม่มีกำหนด (ไม่ได้ตามข้อกำหนดของ bounded waiting) ทำให้โพรเซสนั้นๆ ไม่สามารถ(รับประกันว่า)ทำงานต่อไปได้
    - นั่นหมายถึง Liveness failure ความมีชีวิตล้มเหลว

<u>สภาวะติดตาย Deadlock</u> คือสภาวะที่ ภายใต้การทำงานของโพรเซสหลายตัวนั้น พบว่ามี***โพรเซสบางตัว หรือทุกตัวไม่สามารถทำงานต่อไปได้ เนื่องจากต้องหยุดรอเพื่อเข้าใช้ส่วนวิกฤติอย่างไม่มีเวลากำหนด***

ลองดูกรณีตัวอย่างต่อไปนี้คือ มีโพรเซสสองตัว ที่ใช้ทรัพยากรร่วมกันสองหน่วย โดยกำหนดเซมาฟอร์เป็นไบนารีและส่วนของเซมาฟอร์ทั้งสองคือ S และ Q และลำดับการทำงานของโพรเซสทั้งสองในการจัดการเป็นดังนี้

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

สมมติว่า ในเวลาใดเวลาหนึ่ง P_0 เกิดทำงานมาถึง wait (S) พร้อมๆ กันกับที่ P_1 ทำงานมาถึง wait (Q) พอดี ในสภาพนี้เราจะพบว่าทั้งโพรเซส P_0 และ P_1 สามารถทำงานส่วนดังกล่าวได้ เพราะเดิมมีค่าใน S และ Q เป็น 1 แต่เมื่อทำงานจนไปถึง wait() ที่สอง เนื่องจากค่าในเซมาฟอร์ S และ Q ในคราวนี้เหลือศูนย์แล้ว จะพบว่าโพรเซสทั้งสองติดอยู่ที่ wait() ตัวที่สองอย่างไม่มีโอกาสได้เข้าสู่ส่วนวิกฤติและไม่มีโอกาสจะประมวล signal() นั่นคือทั้งสองโพรเซสจะไม่สามารถทำงานต่อไปได้ เกิดสภาวะติดตาย (deadlock) นั่นเอง

ในทางทั่วไป เรามักจะพบว่า สภาวะติดตายนี้ มักเกิดจากการที่โพรเซสใดๆ ในกลุ่ม ยังคงรอคอย (wait) เพื่อจะใช้ทรัพยากรร่วมกัน ที่ถูกจองไปแล้วโดยโพรเซสอื่นๆ ในกลุ่ม (ที่ใช้ส่วนวิกฤติร่วมกัน) ดังนั้นเราอาจเรียกปัญหานี้ว่าเป็นปัญหาของ ***การร้องขอและการปล่อยทรัพยากร (resource aquisition and release)***

ในอีกลักษณะหนึ่งที่คล้ายกับการติดตายของโพรเซส นั่นคือโพรเซสที่วนรอเพื่อจะทำงานในส่วนวิกฤติ แต่ไม่มีโอกาสได้ทำงานในส่วนวิกฤติเพราะต้องหยุดรอที่ wait อย่างไม่มีวันสิ้นสุด ในลักษณะเช่นนี้โพรเซสอื่นอาจจะทำงานไปได้ แต่โพรเซสดังกล่าวนี้เท่ากับว่าเกิดการค้าง ไม่สามารถทำงานต่อไปได้ เราเรียกสภาวะนี้ว่า starvation หรือ indefinite blocking (ทางแก้ไขก็คือต้องใช้เซมาฟอร์แบบที่มีคิว เพื่อให้ทุกโพรเซสที่เข้าสู่การรอ ได้เข้าไปต่อท้ายคิว และโพรเซสที่จะทำงานต่อจะถูกดึงจากหัวคิว ทำให้ทุกโพรเซสได้มีโอกาสทำงานโดยไม่มีโพรเซสใดแย่งชิงได้

ในสภาวะที่มีการจัดลำดับความสำคัญของโพรเซส (priotiry) เรามักจะเปิดโอกาสให้โพรเซสที่มีลำดับความสำคัญสูงกว่าได้มีโอกาสที่จะใช้เวลาของซีพียูได้มากกว่า หรือมีสิทธิแซงคิวเพื่อไปทำงานได้ก่อน ในลักษณะเช่นนี้ อาจเกิดกรณีที่โพรเซสที่มีความสำคัญต่ำกว่า ร้องขอทรัพยากรได้ก่อนโพรเซสที่มีลำดับความสำคัญสูงกว่า ทำให้โพรเซสที่มีลำดับความสำคัญสูง ต้องหยุดรอโพรเซสที่มีลำดับความสำคัญต่ำกว่า เราเรียกกรณีนี้ว่า priority inversion

6.3 ตัวอย่างปัญหาการประสานงานระหว่างโพรเซส

ปัญหาการจัดการการส่งต่อข้อมูลผ่านพื้นที่จัดเก็บที่มีจำกัด (Bounded-buffer problem)

ปัญหาการจัดการ bounded-buffer นั้นเราได้เห็นกันมาแล้วในเนื้อหาก่อนหน้านี้ แต่ในเนื้อหาบทนี้เรานำเซมาฟอร์เข้ามาจัดการ

โครงสร้างและข้อกำหนดของการประยุกต์เป็นดังนี้

- สมมติว่ามีบัฟเฟอร์ทั้งหมด N หน่วย แต่ละหน่วยเก็บข้อมูลได้ 1 ตัว (เช่นเป็นคิววงกลมที่เก็บข้อมูลได้ N หน่วย)
- กำหนดเซมาฟอร์ mutex เพื่อใช้ในการเข้าส่วนวิกฤติ ค่าเริ่มต้นเท่ากับ 1
- กำหนดเซมาฟอร์ empty เพื่อใช้นับจำนวนหน่วยของบัฟเฟอร์ที่ว่าง ค่าเริ่มต้นเท่ากับ N
- กำหนดเซมาฟอร์ full เพื่อใช้นับจำนวนหน่วยบัฟเฟอร์ที่เก็บข้อมูลอยู่ ค่าเริ่มต้นเท่ากับ 0

จากกรณี producer-consumer เราจะสามารถเขียนขั้นตอนวิธีเพื่อนำเสนอโดยใช้เซมาฟอร์ได้ดังนี้

```
Producer
do  {
      //ผู้ผลิตจัดหาข้อมูลใหม่
      wait (empty);
      wait (mutex);
      //ใส่ข้อมูลลงในบัฟเฟอร์
      signal (mutex);
      signal (full);
} while (TRUE);
```

```
Consumer
do  {
      wait (full);
      wait (mutex);
      //อ่านข้อมูลจากในบัฟเฟอร์
      signal (mutex);
      signal (empty);
      //นำข้อมูลไปใช้งาน
} while (TRUE);
```

จะเห็นว่า ในลักษณะเช่นนี้ เราไม่ต้องเขียนโค้ดในการตรวจสอบว่าบัฟเฟอร์เต็มหรือบัฟเฟอร์นั้นว่างหรือไม่ โดยเราผลักภาระการตรวจสอบดังกล่าวไปให้เซมาฟอร์จัดการแทน

ปัญหาการจัดการการจัดสรรทรัพยากรระหว่างผู้เขียนและผู้อ่าน (Reader-writer problem)

มีกรณีศึกษาอีกอย่างหนึ่งที่พบได้บ่อยในการเขียนโปรแกรมที่ต้องการรับส่งข้อมูลในระหว่างหลายโพรเซส นั่นก็คือกรณีศึกษาของผู้เขียนและผู้อ่าน (reader-writer problem)

ในลักษณะนี้ จะมีโพรเซสแบ่งออกเป็นสองกลุ่มคือ

- ผู้อ่าน (reader) คือโพรเซสที่จะอ่านข้อมูลจากบัฟเฟอร์เพียงอย่างเดียวโดยไม่มีการแก้ไขข้อมูลแต่ประการใด
- ผู้เขียน (writer) คือโพรเซสที่นำข้อมูลไปใส่ในบัฟเฟอร์ ซึ่งผู้เขียนก็จะสามารถอ่านข้อมูลที่เขียนได้เช่นกัน

กรณีศึกษานี้มีสิ่งน่าสังเกตที่สำคัญก็คือ ผู้อ่านนั้น ***จะไม่มีการเปลี่ยนแปลงข้อมูลส่วนกลาง*** และในการประยุกต์ใช้งานจริง เรามักพบว่าผู้อ่านอาจจะต้องการข้อมูลร่วมกัน (ชิ้นเดียวกัน) หรืออย่างน้อยก็เป็นข้อมูลคนละส่วน แต่ถูกเตรียมไว้โดยผู้เขียนโพรเซสเดียวหรือมีจำนวนโพรเซสของผู้เขียนน้อยกว่าจำนวนโพรเซสของผู้อ่านเป็นอย่างมาก ถ้าเราใช้การจัดการสภาวะวิกฤติแบบปกติที่จะอนุญาตให้โพรเซสเดียวเท่านั้นที่จะเข้าสู่ส่วนวิกฤติ เราก็จะพบว่าจะมีโพรเซสจำนวนมากที่จะแย่งชิงเข้าสู่ส่วนวิกฤติ แต่เมื่อพิจารณาว่า โพรเซสผู้อ่านจะไม่มีการเปลี่ยนแปลงข้อมูลส่วนกลาง ดังนั้นเราจึงสามารถยอม***ให้โพรเซสผู้อ่านสามารถเข้าสู่ส่วนวิกฤติพร้อมกันได้***โดยไม่มีผลเสียหายใดๆ ต่อตัวข้อมูล และทำให้จำนวนโพรเซสที่รอส่วนวิกฤตินั้นลดลง *เพิ่มประสิทธิภาพการทำงานโดยรวม* ดังนั้น ในกรณีนี้ เราสรุปได้ว่า

- โพรเซสผู้อ่าน (reader) สามารถเข้าส่วนวิกฤติได้พร้อมๆกันโดยไม่ต้องรอ (หากมีโพรเซสผู้อ่านตัวอื่นกำลังอยู่ในส่วนวิกฤติ โพรเซสผู้อ่านที่จะเข้าสู่ส่วนวิกฤติ สามารถเข้าได้ทันทีโดยไม่ต้องรอให้โพรเซสอื่นที่อยู่ในส่วนวิกฤติออกมาก่อน
- โพรเซสผู้เขียน (writer) สามารถเข้าส่วนวิกฤติได้เพียงหนึ่งตัวในเวลาใดเวลาหนึ่ง และโพรเซสผู้เขียนไม่สามารถเข้าส่วนวิกฤติในขณะที่มีโพรเซสผู้อ่านอยู่ในส่วนวิกฤติได้ และโพรเซสผู้อ่านเองก็ไม่สามารถเข้าส่วนวิกฤติในขณะที่มีโพรเซสผู้เขียนอยู่ในส่วนวิกฤติเช่นกัน

การประยุกต์กลไกการจัดการส่วนวิกฤติแบบผู้เขียน-ผู้อ่าน มีทางเลือกได้ในสองแนวทางคือ

1. reader สามารถแซงคิว writer เข้าส่วนวิกฤติได้ทันที หากพบว่าในส่วนวิกฤติมีผู้อ่านใช้งานอยู่
2. เมื่อ writer ทำงานมาถึงส่วนวิกฤติ reader ตัวอื่นที่ยังไม่เข้าส่วนวิกฤติทุกตัวต้องรอจนกว่า writer จะผ่านส่วนวิกฤติไปแล้ว

การประยุกต์ทั้งสองรูปแม้ว่าจะดูง่าย แต่ก็อาจส่งผลทำให้เกิดการขาดโอกาสทำงานของโพรเซส (starvation) ได้ อย่างในกรณี 1) หากมี reader จำนวนมาก เราจะพบว่า writer จะไม่มีโอกาสได้เข้าส่วนวิกฤติเลยเพราะถูก reader แซงคิวอยู่ตลอดเวลา (และส่งผลทำให้บัฟเฟอร์ว่าง และโพรเซสทั้งหมดอาจทำงานต่อไปไม่ได้เลย) ส่วนในกรณีที่ 2) หาก writer ออกจากส่วนวิกฤติไปรับข้อมูลใหม่และกลับเข้ามาส่วนวิกฤติได้รวดเร็ว เราอาจจะพบว่า reader บางตัวอาจจะไม่มีโอกาสได้เข้าส่วนวิกฤติ เพราะ reader ตัวที่กลับเข้ามารอเข้าส่วนวิกฤติช้ากว่า writer จะถูก writer แซงคิว

สำหรับแนวทางประยุกต์แบบแรก ทำได้ดังนี้

- กำหนดเซมาฟอร์ mutex และ wrt สำหรับใช้ควบคุมส่วนวิกฤติ โดยกำหนดค่าเริ่มต้นของทั้งสองตัวเป็น 1
- กำหนดตัวแปร readcount ใช้นับจำนวนผู้อ่าน โดยกำหนดค่าเริ่มต้นเป็น 0

```
Writer
do  {
      //ผู้เขียนจัดหาข้อมูลใหม่
      wait (wrt);
      //ใส่ข้อมูลลงในบัฟเฟอร์
      signal (wrt);
} while (TRUE);
```

```
Reader
do  {
      wait (mutex);
      readcount++;
      if(readcount == 1)
            wait(wrt);
      signal(mutex)
      //อ่านข้อมูลจากในบัฟเฟอร์
      wait (mutex);
      readcount--;
      if(readcount == 0)
            signal(wrt);
      signal (mutex);
} while (TRUE);
```

จากขั้นตอนวิธีข้างต้น มีข้อด้อยอยู่ที่ หากผู้อ่านมีจำนวนมากและทยอยเข้าออกส่วนวิกฤติตลอดเวลา จะส่งผลทำให้ผู้เขียนไม่สามารถเข้าส่วนวิกฤติได้ (เพราะต้องรอให้ผู้อ่านออกหมดเสียก่อน) ส่งผลทำให้สถานะของระบบไม่มีโอกาสอัปเดต เกิดสถานะ starvation ของโพรเซสผู้เขียน

ในลักษณะเช่นนี้ เราจึงปรับปรุงขั้นตอนวิธี เพิ่มกลไกให้ในกรณีที่ผู้เขียนเข้ามารอที่ทางเข้าส่วนวิกฤติ ผู้อ่านที่ตามมาทีหลังจะต้องหยุดรอให้ผู้เขียนดังกล่าวเข้าส่วนวิกฤติไปเปลี่ยนสถานะของระบบเสียก่อน (เป็นไปตามแนวทางประยุกต์แบบที่สองที่ได้กล่าวมาก่อนหน้านี้)

- กำหนดเซมาฟอร์ rmutex wmutex readTry และ resource สำหรับใช้ควบคุมส่วนวิกฤติ
- กำหนดตัวแปร readcount และ writecount

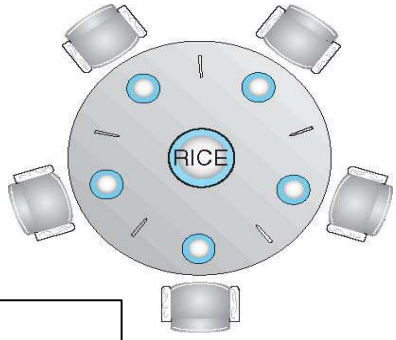
```
Writer
do  {
      //ผู้เขียนจัดหาข้อมูลใหม่
      wait (wmutex);
      writecount++;
      if(writecount==1)
            wait(readTry);
      signal(wmutex);
      //กำหนดส่วนวิกฤติสำหรับผู้เขียนตัวนี้เท่านั้น
      wait(resource);
      //ใส่ข้อมูลลงในบัฟเฟอร์
      //ผู้เขียนออกจากส่วนวิกฤติ
      signal(resource);

      wait(wmutex);
      writecount--;
      if(writecount==0)
            signal(readTry);
      signal (wmutex);
} while (TRUE);
```

```
Reader
do  {
      wait (readTry);
      wait (rmutex);
      readcount++;
      if(readcount == 1)
            wait(resource);
      signal(rmutex);
      signal(readTry);
      //อ่านข้อมูลจากในบัฟเฟอร์
      wait (rmutex);
      readcount--;
      if(readcount == 0)
            signal(resource);
      signal (rmutex);
} while (TRUE);
```

เราพบว่า ในบางระบบปฏิบัติการ (เช่น ลินุกซ์) จะมีกลไกการจัดการ reader-writer problem เพิ่มเติมมาให้เพื่อความสะดวกในการพัฒนาโปรแกรม และเป็นการผลักภาระการควบคุมการทำงานและการแก้ไขปัญหา ไปให้ระบบปฏิบัติการดูแลแทน

ปัญหาการจัดสรรทรัพยากรแบบโต๊ะอาหารของนักปรัชญา (Dining-Philosophers problem)



สมมติว่ามีนักปรัชญาจำนวนหนึ่ง กำลังนั่งรับประทานอาหารด้วยกันบนโต๊ะจีน อาหารถูกจัดวางอยู่กึ่งกลางโต๊ะ และมีตะเกียบหนึ่งข้างวางคั่นระหว่างนักปรัชญาแต่ละคน นักปรัชญาแต่ละคนจะใช้เวลาส่วนหนึ่งคิด และอีกส่วนหนึ่งคว้าตะเกียบที่อยู่ด้านซ้ายและด้านขวาของตน เพื่อมาคีบอาหารวางบนจานของตนรับประทาน จากนั้นจึงวางตะเกียบลงที่เดิม

เราสามารถจำลองกลไกแบบโต๊ะอาหารของนักปรัชญาด้วยเซมาฟอร์ดังนี้

```
do  {
      wait (chopstick[i]);
      wait (chopstick[(i+1) % 5]);

      //ส่วนวิกฤติ

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

} while (TRUE);
```

ลักษณะปัญหาเช่นนี้เกิดขึ้นได้บ่อยในกลไกของระบบปฏิบัติการในปัจจุบัน เนื่องจากมีโพรเซสจำนวนมากทำงานอยู่ และโพรเซสหลายตัวได้แบ่งใช้ทรัพยากรร่วมกันไปมาเป็นโครงข่ายซับซ้อน โพรเซสแต่ละตัวอาจจองทรัพยากรไปจำนวนหนึ่งแต่ยังไม่ครบที่จะทำงานต่อ แล้วค่อยจองเพิ่มในภายหลัง แต่ปรากฏว่าไม่สามารถจองทรัพยากรเพิ่มเพื่อให้ทำงานต่อไปได้ ทั้งนี้เพราะทรัพยากรที่ขาดนั้น อีกโพรเซสได้จองไปแล้ว และกำลังรอทรัพยากรของอีกโพรเซสหนึ่ง วนเวียนซึ่งกันและกัน ทำให้โพรเซสทุก

ตัวไม่สามารถทำงานต่อไปได้ เกิดสภาวะ**ติดตาย (deadlock)** เปรียบได้กับนักปรัชญาแต่ละคน สามารถคว้าตะเกียบขึ้นมาได้คนละข้าง ในกรณีเช่นนี้จะพบว่าไม่มีใครสามารถทานอาหารต่อไปได้เลย

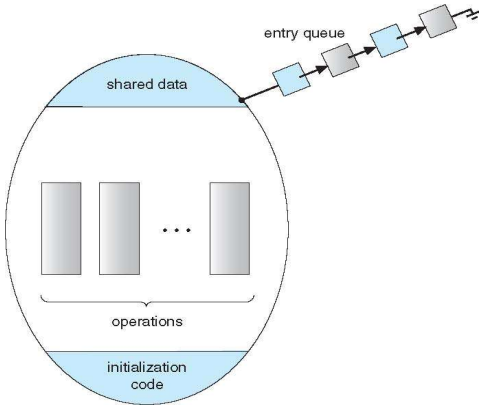
เพื่อไม่ให้เกิดสภาวะติดตาย ในเบื้องต้น เราอาจจะกำหนดกฎเกณฑ์ขึ้นมาจัดการได้ดังเช่น

- **ต้องมีเก้าอี้ว่างอย่างน้อยหนึ่งตัว** นั่นคือ นักปรัญชาที่นั่งข้างเก้าอี้ว่าง จะสามารถคว้าตะเกียบได้หนึ่งข้างในทันที และรอเพียงอีกหนึ่งข้างเพื่อรับประทาน ในสภาวะที่เลวร้ายที่สุด นักปรัญชาที่นั่งข้างเก้าอี้ว่างคนหนึ่งจะไม่สามารถคว้าตะเกียบได้ทัน รวมทั้งคนอื่นๆ (ถือกันคนละข้าง) แต่คนที่นั่งข้างเก้าอี้ว่างอีกฝั่งจะสามารถคว้าคะเกียบได้สองข้างรับประทาน แล้วปล่อยตะเกียบให้คนที่นั่งติดกันได้ใช้ต่อไป
- **ต้องอนุญาตให้นักปรัญชาหยิบตะเกียบทั้งสองข้างขึ้นพร้อมกันเท่านั้น** (สร้างส่วนวิกฤติขึ้นมาคร่อมกลไกการหยิบตะเกียบทั้งสองข้าง)
- **ใช้ขั้นตอนวิธีแบบอสมมาตร (asymmetric algorithm)** หรืออีกนัยหนึ่ง ใช้ขั้นตอนวิธีที่ไม่เหมือนกันระหว่างนักปรัญชาแต่ละคน เพื่อไม่ให้เกิดแนวทางปฏิบัติที่เหมือนกันหรือเป็นไปในทำนองเดียวกันจนเกิดสภาวะติดตาย เช่น กำหนดให้คนหนึ่งหยิบตะเกียบข้างซ้ายก่อน อีกคนที่นั่งติดกัน หยิบตะเกียบข้างขวาก่อน เป็นต้น

6.4 โครงสร้างข้อมูลสำหรับเฝ้าระวังส่วนวิกฤติ (Monitors)

ในทางปฏิบัติ การจัดการส่วนวิกฤติ ต้องอาศัยความร่วมมือจากผู้พัฒนาซอฟต์แวร์ หากผู้พัฒนาซอฟต์แวร์เพียงคนใดคนหนึ่งเกิดเผลอเรอเขียนกลไกการจัดการส่วนวิกฤติผิด เช่น ใช้ signal() เพื่อเข้าส่วนวิกฤติ และใช้ wait() เมื่อออกจากส่วนวิกฤติ หรือลืมเรียกกลไกการจัดการส่วนวิกฤติเลย ความผิดพลาดเพียงโพรเซสเดียว อาจส่งผลทำให้เกิดสภาวะติดตาย หรือเกิดความผิดพลาดร้ายแรง เช่นมีสองโพรเซสเข้าสู่ส่วนวิกฤติพร้อมกัน เป็นต้น

การลดความผิดพลาดที่เกิดขึ้นจากผู้พัฒนา จึงอาศัยกลไกเพิ่มเติมโดยภาษาคอมพิวเตอร์และคอมไพเลอร์ในยุคใหม่ด้วยคลาส monitor โดยมีตัวแปรสมาชิกและฟังก์ชันสมาชิกอยู่ภายใน โดยฟังก์ชันสมาชิกแต่ละตัวจะทำงานได้เพียงตัวเดียวในเวลาใดเวลาหนึ่งเท่านั้น (กลไกการเพิ่มเติมการจัดการส่วนวิกฤติ ยกภาระให้เป็นหน้าที่ของคอมไพเลอร์ที่จะแทรกโค้ดเข้าไปยังจุดต่างๆ แทนที่จะเขียนโดยผู้พัฒนาโปรแกรมเอง)



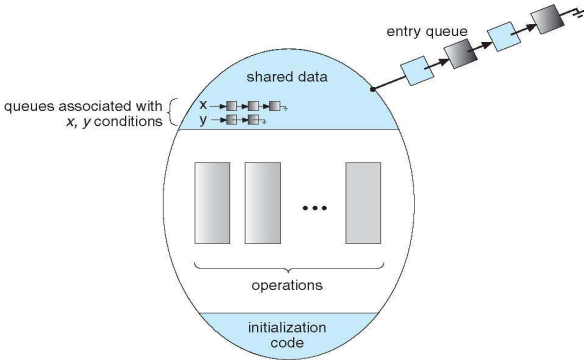

```
monitor monitor-name
{
      // shared variable declarations
      procedure P1 (…) { …. }

      procedure Pn (…) {……}

    Initialization code (…) { … }
      }
}
```

ดังนั้น เวลาที่ผู้พัฒนาโปรแกรมในทีมที่จะต้องรับผิดชอบในการออกแบบโพรเซสหรือเธรดต่างๆ ที่ต้องใช้ส่วนวิกฤติร่วมกัน ก็จะต้องส่งโค้ดที่เป็นส่วนวิกฤติมาในลักษณะของฟังก์ชันสมาชิก การควบคุมการพัฒนาโค้ดโปรแกรมจึงทำได้อย่างเป็นระบบเพราะการนิยามในรูปของคลาสนี้ จะทำให้ง่ายต่อการตรวจสอบและควบคุมไม่ให้บุคลากรในทีมเผลอเรอในการจัดการส่วนวิกฤติได้

นอกจากนี้ อาจจะเพิ่มเติมโครงสร้างข้อมูลพิเศษอีกตัวหนึ่งที่มีฟังก์ชันสมาชิกคือ wait() และ signal() เพื่อช่วยอำนวยความสะดวกในกรณีที่ต้องการการจัดการที่ซับซ้อนขึ้นไป โดยอาจจะกำหนดตัวแปรสมาชิกให้มีชนิดเป็นโครงสร้างข้อมูลพิเศษนี้ เพื่อให้ฟังก์ชันสมาชิกของโพรเซสต่างๆ ได้เรียกใช้งาน (สมมติให้โครงสร้างข้อมูลพิเศษดังกล่าวมีชื่อว่า condition) ดังเช่น



condition x,y;

โดยมีกลไกเพิ่มเติมดังนี้

- หากโพรเซสใดเรียกใช้ฟังก์ชันสมาชิกของตัวแปรชนิด condition เช่น x.wait() โพรเซสนั้นๆ จะต้องรอให้โพรเซสอื่นใดเรียกฟังก์ชันสมาชิกของตัวแปรเดียวกันนี้ x.signal()
- เมื่อโพรเซสหนึ่งๆ เรียก x.signal() ก็จะส่งผลให้โพรเซสที่กำลังหยุดรอด้วย x.wait() หนึ่งโพรเซส ได้ทำงานต่อ (ส่วนโพรเซสอื่นๆ ที่หยุดรอด้วย x.wait() ด้วยกันนั้น ก็จะต้องหยุดรอต่อไปจนกว่าจะได้ x.signal() ต่อไป)
- ในกรณีที่ไม่มีโพรเซสใดๆหยุดรออยู่เลย การใช้ x.signal() จะไม่ส่งผลใดๆ เกิดขึ้น (จะเห็นว่าในกรณีนี้ กลไกของ signal() นั้นมีความซับซ้อนขึ้นกว่าเซมาฟอร์ปกติ การกระทำเช่นนี้เป็นการปิดกั้นไม่ให้การเขียนโปรแกรมผิดพลาดของผู้พัฒนาส่งผลต่อส่วนวิกฤติ

เนื่องจากเรานิยามตัวแปร conditionนี้ภายใต้ monitor ดังนั้นฟังก์ชันสมาชิกเพียงตัวเดียวเท่านั้นที่จะทำงานได้ ณ เวลาใดเวลาหนึ่ง สมมติว่า โพรเซส Q กำลังหยุดรอด้วย x.wait() เมื่อโพรเซส P เรียกใช้ x.signal() จะเห็นว่า Q นั้นควรที่จะสามารถทำงานต่อไปได้ แต่เนื่องจากกฎการนิยาม monitor ที่ว่าฟังก์ชันสมาชิกใน monitor ไม่สามารถทำงานพร้อมกันได้ กลไกทางภาษาอาจจะยอมให้เกิดในกรณีใดกรณีหนึ่งดังนี้

- **signal and wait** โพรเซสที่เรียกใช้ signal() จะหยุดการทำงานและยอมให้โพรเซสที่รออยู่ได้ทำงานต่อไปจนกระทั่งจบการทำงานในฟังก์ชันสมาชิก (พ้นจาก monitor) โพรเซสที่เรียก signal() จึงจะทำงานต่อไปได้
- **signal and continue** โพรเซสที่เรียกใช้ signal() จะยังคงทำงานต่อไปจนกว่าจะจบการทำงานในฟังก์ชันสมาชิก หลังจากนั้นแล้ว โพรเซสที่หยุดรอ จึงจะสามารถทำงานต่อไปได้

ภาษาโปรแกรมเชิงวัตถุในยุคใหม่ที่เน้นการออกแบบตัวภาษาให้ลดโอกาสการเขียนโค้ดผิดพลาดให้น้อยที่สุด ได้เริ่มนำเสนอแนวคิดของ monitor นี้ในรูปคลาสแล้ว ดังเช่นภาษา java และ C# เป็นต้น

จากกรณีศึกษาเรื่องโต๊ะอาหารของนักปรัชญา เราสามารถนำเอา monitor นี้มาประยุกต์ใช้ได้ดังตัวอย่าง

```
monitor DiningPhilosophers{
       enum (THINKING, HUNGRY, EATING) state[5] ;
       condition self[5];

       void pickup (int i){
           state[i] = HUNGRY;
           test(i);
           if (state[i] != EATING) self[i].wait();
       }

      void putdown (int i) {
            state[i] = THINKING;
           // test left and right neighbors
            test((i + 4) % 5);
            test((i + 1) % 5);
      }

       void test (int i) {
            if((state[(i + 4) % 5] != EATING) &&
               (state[i] == HUNGRY) &&
               (state[(i + 1) % 5] != EATING) ) {
                     state[i] = EATING ;
                      self[i].signal() ;
            }
       }

      initialization_code() {
            for (int i = 0; i < 5; i++)
                state[i] = THINKING;
       }
};
```

ในแต่ละโพรเซส ตรงส่วนที่จะเข้าส่วนวิกฤติ ก็เรียกใช้ monitor ดังกล่าวเช่น

```
DiningPhilosophers.pickup (i);
      // ส่วนวิกฤติ
DiningPhilosophers.putdown (i);
```

<u>การประยุกต์ monitor ด้วยเซมาฟอร์</u>

สำหรับภาษาโปรแกรมที่ไม่รองรับ monitor หรือผู้พัฒนาโปรแกรมปรารถนาจะประยุกต์ใช้เซมาฟอร์โดยตรง เราสามารถนำเสนอ monitor ในรูปแบบของเซมาฟอร์ได้ดังนี้

- กำหนดตัวแปรเซมาฟอร์และตัวแปรเพื่อใช้งาน

```
semaphore mutex=1;          //เซมาฟอร์สำหรับการเข้าส่วนวิกฤติ
semaphore next=1; //เซมาฟอร์สำหรับตัวแปรคอนดิชัน
int next_count = 0; //จำนวนโพรเซสที่รออยู่ด้วยการใช้ตัวแปรคอนดิชัน
```

ส่วนวิกฤติสำหรับใช้ครอบการทำงานที่ปรากฏเป็นฟังก์ชันใน monitor

```
wait(mutex);
// ส่วนการทำงานเดิมที่นิยามเป็นฟังก์ชันสมาชิกใน monitor
if (next_count > 0)
      signal(next)
else
      signal(mutex);
```

สำหรับคอนดิชัน เราอาจจะกำหนดเป็นโครงสร้างข้อมูลดังนี้

```
struct condition{
      semaphore x=0;
      int count=0;

      void wait(){
            count++;
            if(next_count > 0)
                  signal(next);
            else
                  signal(mutex);
            wait(x);
            count--;
      }

      void signal(){
            if(count>0{
                  next_count++;
                  signal(x);
                  wait(next);
                  next_count--;
            }
      }
};
```

ตัวแปรเข้มเหล่านี้ นิยามไว้เป็น global

การเลือกโพรเซสที่จะดำเนินการต่อภายใต้มอนิเตอร์เมื่อได้รับ signal()

เมื่อมีโพรเซสหนึ่งเรียกใช้ x.signal() และมีโพรเซสหลายตัวได้หยุดรอด้วย x.wait() เราอาจจะใช้วิธีการง่ายๆ คือใครเข้ามารอก่อนได้ก่อน (First-come-first-served FCFS) แต่ในการใช้งานจริงบางครั้งโพรเซสที่เข้ามาทีหลังอาจจะมีความจำเป็นมากกว่า ดังนั้น เราอาจจะใช้หลักการกำหนดเลขกำหนดลำดับความสำคัญ (priority number) โดยในจังหวะที่เรียก x.wait() ก็ให้ผ่านค่าลำดับความสำคัญไว้ด้วยเช่น x.wait(c) และเมื่อโพรเซสหนึ่งเรียก x.signal() โพรเซสที่เรียก x.wait() ที่มีค่าลำดับความสำคัญสูงสุดก็จะได้รับสิทธิให้ทำงานต่อ

ปัญหาทิ้งท้ายสำหรับการขอใช้ทรัพยากรของโพรเซส

การจัดสรรทรัพยากรของโพรเซสที่เราได้ศึกษามาในหัวข้อนี้ ยังมีได้แก้ไขปัญหาที่สำคัญดังต่อไปนี้

- โพรเซสอาจจะเข้าใช้ทรัพยากรโดยพลการ โดยไม่ได้ใส่กลไกการเข้าส่วนวิกฤติเอาไว้
- โพรเซสอาจจะไม่คืนทรัพยากรให้ระบบปฏิบัติการ เมื่อได้เข้าสู่ส่วนวิกฤติ และออกจากส่วนวิกฤติแล้ว
- โพรเซสอาจจะพยายามคืนทรัพยากรที่ตนเองมิได้ร้องขอ (เช่น โพรเซสอาจจะไปได้ไอดีของทรัพยากรของโพรเซสอื่นที่ได้ร้องขอ) ทำให้เกิดความผิดพลาดขึ้นที่โพรเซสที่เป็นเจ้าของจริง
- โพรเซสพยายามร้องขอทรัพยากรเดิมซ้ำๆ (โดยไม่ยอมคืนทรัพยากรเดิมก่อน) ทำให้ทรัพยากรดังกล่าวที่ระบบปฏิบัติการมีลดลงเรื่อยๆ จนหมด (เช่น เปิดไฟล์โดยไม่ปิดไฟล์ จนไฟล์แฮนเดิลที่ระบบมีถูกใช้จนหมด)

ปฏิบัติการ

6.1 การใช้งานไบนารีเซมาฟอร์ในลินุกซ์

ตัวอย่างต่อไปนี้เป็นการใช้งานเซมาฟอร์แบบ unnamed semaphore ซึ่งใช้ระหว่างเธรด แต่ถ้าต้องการใช้งานเซมาฟอร์ในระดับระหว่างโพรเซส สามารถกระทำได้โดยการใช้งาน named semaphore ซึ่งตัวเซมาฟอร์จะต้องเริ่มการทำงานด้วยฟังก์ชัน sem_open ซึ่งมีโปรโตไทป์ดังนี้

```
sem_t *sem_open(const char *name, int oflag,..., int initvalue);
```

และต้องปิดเซมาฟอร์(จบการทำงาน) ด้วยฟังก์ชัน sem_close ซึ่งมีโปรโตไทป์ดังนี้

```
int sem_close(sem_t *sem);
```

การใช้งาน named semaphore มีลักษณะคล้ายคลึงกับการจัดการกับ namepipe ในลินุกซ์ ซึ่งมองตัวเซมาฟอร์เปรียบเสมือนไฟล์หนึ่งในไดเรกตอรีนั่นเอง...

ตัวอย่างการนิยามตัวแปรเซมาฟอร์สำหรับ named semaphore และการใช้งาน

```
sem_t *sem;
        ....
sem = sem_open("/tmp/mysem",O_CREAT,0666,1);
        ....
        sem_wait(sem);
        ....
        sem_post(sem);

sem_close(sem);
```

ตัวอย่างข้างล่างนี้เป็นการจัดการ unnamed semaphore ซึ่งใช้ระหว่างเธรด นักศึกษาอาจลองเปลี่ยนมาใช้เป็นการจัดการด้วย named semaphore ดูว่าโค้ดจะเปลี่ยนไปอย่างไร

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <semaphore.h>

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
        {"rice with ","chicken curry, and ","fish sauce."},
        {"wiskey with ","lemonade, and ","soda."},
        {"bread with ","cheese, and ","ketchup."},
        {"icecream with ","banana, and ","chocolate."}};
int  isFull;
int  isReady;

sem_t sem;

void randomDelay(void);
void serve(char *dest,char *src);
void *chefWork(void *who);
void *customer(void *who);

int main(void){
        int i;
        int param[5]={0,1,2,3,4};
        pthread_t tid[5];      // Thread ID
```

```
        pthread_attr_t attr[5];       // Thread attributes

        isFull=false;          // Customer tell all chefs to stop making food
        isReady=false;         // Producer tell customer that food is ready
        // Locking mechanism,  the lock starts with false

        sem_init(&sem, // Semaphore
                0,      // using only by threads of the process (nonzero value for other processes)
                1);     // Initial value

        for(i=0;i<5;i++)
                pthread_attr_init(&attr[i]);  // Get default attributes

        // Create 4 threads for producers
        for(i=0;i<4;i++)
                pthread_create(&tid[i],&attr[i],chefWork,(void *)&param[i]);
        // Create 1 threads for consumer
        pthread_create(&tid[4],&attr[4],customer,(void *)&param[i]);

       // Wait until all threads finish
        for(i=0;i<5;i++)
                pthread_join(tid[i],NULL);

        sem_destroy(&sem);

        return 0;
}

void randomDelay(void){
    int stime = ((rand()%100)+1)*1000;
    usleep(stime);
}

void serve(char *dest,char *src){
   int i;
   srand(time(NULL));

   for(i=0;(i<63)&&(src[i]!=0);i++){
        dest[i]=src[i];
        randomDelay();
   }
   dest[i]=0;

}

void *chefWork(void *who){
    int plateNo,chefNo,i,j;

    chefNo = (int)*((int *)who);

    for(i=0;i<(chefNo+2);i++) printf("        ");
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);

    while(!isFull){

        if(isReady) {
            randomDelay(); // Wait for food to be taken
        }

        sem_wait(&sem);

        if((!isFull)&&(!isReady)){
        // Critical Section
                for(i=0;i<(chefNo+2);i++) printf("      ");
                printf("Chef NO %d is serving the food\n",chefNo+1);
                fflush(stdout);

                for(j=0;j<3;j++)
                    serve(plate[j],chef[chefNo][j]);
                isReady=true; //Tel customer that the food is ready

                for(i=0;i<(chefNo+2);i++) printf("      ");
                printf("Chef NO %d has serve the food\n",chefNo+1);
                fflush(stdout);
        }
        sem_post(&sem);

        if(isFull) break;
```

```
        // Remaining Section
        randomDelay();
    }
    pthread_exit(0);
}

void *customer(void *who){
    int i,j;
    char dinner[256]={0};

    for(i=0;i<10;i++){
        while(!isReady) randomDelay();

        sem_wait(&sem);
                // Critical Section
        printf("Choochok starts grab a set of dinner\n");
        fflush(stdout);

        serve(dinner,plate[0]); dinner[63]=0;
        serve(dinner + strlen(dinner),plate[1]); dinner[127]=0;
        serve(dinner + strlen(dinner),plate[2]); dinner[191]=0;

        printf("Plate NO:%d Choochok is eating %s\n",i+1,dinner);
        fflush(stdout);

         isReady=false; // Food taken

        sem_post(&sem);
    }
    isFull = true; // Tell other producer threads to stop;
    pthread_exit(0);
}
```

6.2 การใช้งานไบนารีเซมาฟอร์ในวินโดวส์

สำหรับระบบปฏิบัติการวินโดวส์นั้น มีเซมาฟอร์อ็อปเจ็คต์ให้เลือกใช้ (windows.h) นอกเหนือจากคลาส semaphore ใน MFC

เซมาฟอร์ในวินโดวส์จะมองเป็นวัตถุตัวหนึ่ง และอ้างโดยใช้แฮนเดิล การหยุดรอวัตถุจะใช้ WaitForSingleObject() โดยในแต่ละครั้งที่เกิดการเรียกใช้ฟังก์ชันดังกล่าว ค่าในเซมาฟอร์จะลดลงหนึ่ง และจะทำงานต่อไปได้ แต่ถ้าค่าในเซมาฟอร์มีค่าเป็น 0 อยู่แล้ว โพรเซสก็จะรอ ณ จุดนี้จนกว่าค่าในเซมาฟอร์จะมีค่ามากกว่าศูนย์ แล้วจึงทำงานต่อ ส่วนการเปลี่ยนค่าในเซมาฟอร์เราใช้ ReleaseSemaphore() ซึ่งฟังก์ชันนี้สามารถเพิ่มค่าในเซมาฟอร์ได้ตามต้องการ (ในที่นี้เรากำหนดค่าเพิ่มขึ้น 1 ค่าในแต่ละครั้งที่เรียกใช้ฟังก์ชันดังกล่าว

```
#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>


char plate[3][64]; // Consumer table
char chef[4][3][64] = {
        {"rice with ","chicken curry, and ","fish sauce."},
        {"wiskey with ","lemonade, and ","soda."},
        {"bread with ","cheese, and ","ketchup."},
        {"icecream with ","banana, and ","chocolate."}};
int  isFull;
int  isReady;

HANDLE sem;

DWORD sem_wait(HANDLE sem);
DWORD sem_signal(HANDLE sem);

void randomDelay(void);
void serve(char *dest,char *src);
DWORD WINAPI chefWork(LPVOID who);
DWORD WINAPI customer(LPVOID who);
```

```
int main(void){
        int i;
        DWORD tid[5];                           // Thread ID
        HANDLE th[5];                           // Thread Handle
        int param[5]={0,1,2,3,4};

        sem = CreateSemaphore(
                NULL,   // default security attributes
                1,   // initial count
                1,   // maximum count
                NULL);  // unnamed semaphore

        // Create 5 threads
        for(i=0;i<4;i++)
        th[i] = CreateThread(
                NULL,                           // Default security attributes
                0,                              // Default stack size
                chefWork,                       // Thread function
                (void *)&param[i],              // Thread function parameter
                0,                              // Default creation flag
                &tid[i]);                       // Thread ID returned.

        th[4] = CreateThread(
                NULL,                           // Default security attributes
                0,                              // Default stack size
                customer,                       // Thread function
                (void *)&param[4],              // Thread function parameter
                0,                              // Default creation flag
                &tid[4]);                       // Thread ID returned.

       // Wait until all threads finish
        for(i=0;i<5;i++)
                if(th[i]!=NULL)
                        WaitForSingleObject(th[i],INFINITE);
        CloseHandle(sem);
        return 0;
}

void randomDelay(void){
    int stime = ((rand()%100)+1);
    Sleep(stime);
}

void serve(char *dest,char *src){
   int i;
   srand(time(NULL));

   for(i=0;(i<63)&&(src[i]!=0);i++){
        dest[i]=src[i];
        randomDelay();
   }
   dest[i]=0;
}


DWORD WINAPI chefWork(LPVOID who){
    int chefNo,i,j;

    chefNo = (int)*((int *)who);

    for(i=0;i<(chefNo+2);i++) printf("        ");
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);

    while(!isFull){

        if(isReady) {
            randomDelay(); // Wait for food to be taken
        }

        sem_wait(sem);
        // Critical Section
                if((!isFull)&&(!isReady)){
                for(i=0;i<(chefNo+2);i++) printf("      ");
                        printf("Chef NO %d is serving the food\n",chefNo+1);
                        fflush(stdout);

                        for(j=0;j<3;j++)
                        serve(plate[j],chef[chefNo][j]);
```

```
                        isReady=true; //Tel customer that the food is ready

                for(i=0;i<(chefNo+2);i++) printf("      ");
                        printf("Chef NO %d has serve the food\n",chefNo+1);
                        fflush(stdout);
                }
        sem_signal(sem);
        // Remaining Section

        if(isFull) break;
        randomDelay();  // This will cause some chef to waiting indefinitly;
    }
    return 0;
}

DWORD WINAPI customer(LPVOID who){
    int i;
    char dinner[256]={0};

    for(i=0;i<10;i++){
//      printf("Choochok waits for food to be served\n");
//      fflush(stdout);

        while(!isReady) randomDelay();

        sem_wait(sem);
        // Critical Section

                printf("Choochok starts grab a set of dinner\n");
                fflush(stdout);

                serve(dinner,plate[0]); dinner[63]=0;
                serve(dinner + strlen(dinner),plate[1]); dinner[127]=0;
                serve(dinner + strlen(dinner),plate[2]); dinner[191]=0;

                printf("Plate NO:%d Choochok is eating %s\n",i+1,dinner);
                fflush(stdout);

                isReady=false; // Food taken

        sem_signal(sem);
        // Remaining Section
        }
        isFull = true; // Tell other producer threads to stop;
    return 0;
}


DWORD sem_wait(HANDLE sem){
    DWORD result = WaitForSingleObject(sem,INFINITE);

        switch(result){
                case WAIT_OBJECT_0:return 1;
                case WAIT_TIMEOUT: return 0;
        }
        return 0;
}

DWORD sem_signal(HANDLE sem){
    return ReleaseSemaphore(sem,1,NULL); // Increase by one
}
```

6.3 การใช้งานไบนารีเซมาฟอร์ลินุกซ์ร่วมกับตัวแปรคอนดิชัน

ลินุกซ์มีการนิยามตัวแปรคอนดิชัน ในลักษณะเดียวกันกับตัวแปรคอนดิชันที่มีใช้ในมอนิเตอร์ ในที่นี้เราจะหันมาบล็อคเธรดโดยการใช้ตัวแปรคอนดิชันแทนการใช้เซมาฟอร์

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
```

```
// This example uses mutex to simulate a semaphore

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
        {"rice with ","chicken curry, and ","fish sauce."},
        {"wiskey with ","lemonade, and ","soda."},
        {"bread with ","cheese, and ","ketchup."},
        {"icecream with ","banana, and ","chocolate."}};
int  isFull;

//=======================================================
#define SEM_MAX 256

typedef struct{
   int value;
   pthread_mutex_t mutex;
   pthread_cond_t condition;
}semaphore;

void sem_init(semaphore *sem,int value){
   sem->value = value;
   pthread_mutex_init(&(sem->mutex),NULL);
   pthread_cond_init(&(sem->condition),NULL);
}

void sem_destroy(semaphore *sem){
    pthread_cond_destroy(&(sem->condition));
    pthread_mutex_destroy(&(sem->mutex));
}

void sem_wait(semaphore *s){
        pthread_t pid;
        pthread_mutex_lock(&(s->mutex));
        s->value--;
        printf("value=%d\n",s->value);fflush(stdout);
        if(s->value < 0){
                printf("put a thread to wait\n");
                pthread_cond_wait(&(s->condition),&(s->mutex));
                pthread_mutex_unlock(&(s->mutex));

        }else{
                pthread_mutex_unlock(&(s->mutex));
        }
}

void sem_signal(semaphore *s){

        pthread_mutex_lock(&(s->mutex));
        s->value++;
        if(s->value <=0){
                pthread_cond_signal(&(s->condition));
                pthread_mutex_unlock(&(s->mutex));
                printf("continue a thread\n");
        } else
           pthread_mutex_unlock(&(s->mutex));
}

//=======================================================

semaphore sem;

void randomDelay(void);
void serve(char *dest,char *src);
void *chefWork(void *who);

int main(void){
        int i;
        int param[5]={0,1,2,3,4};
        pthread_t tid[5];      // Thread ID
        pthread_attr_t attr[5];       // Thread attributes

        isFull=10;             // Customer tell all chefs to stop making food
        // Locking mechanism,  the lock starts with false

        sem_init(&sem,1);

        for(i=0;i<4;i++)
                pthread_attr_init(&attr[i]);  // Get default attributes
```

```
        // Create 4 threads for producers
        for(i=0;i<4;i++)
                pthread_create(&tid[i],&attr[i],chefWork,(void *)&param[i]);

       // Wait until all threads finish
        for(i=0;i<4;i++)
                pthread_join(tid[i],NULL);

        sem_destroy(&sem);

        return 0;
}

void randomDelay(void){
    int stime = ((rand()%100)+1)*1000;
    usleep(stime);
}

void serve(char *dest,char *src){
   int i;
   srand(time(NULL));

   for(i=0;(i<63)&&(src[i]!=0);i++){
        dest[i]=src[i];
        randomDelay();
   }
   dest[i]=0;

}

void *chefWork(void *who){
    int plateNo,chefNo,i,j;

    chefNo = (int)*((int *)who);

    for(i=0;i<(chefNo+2);i++) printf("        ");
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);

    while(isFull>0){

        sem_wait(&sem);

        // Critical Section
                for(i=0;i<(chefNo+2);i++) printf("      ");
                printf("Chef NO %d is serving the food\n",chefNo+1);
                fflush(stdout);

                for(j=0;j<3;j++)
                    serve(plate[j],chef[chefNo][j]);

                for(i=0;i<(chefNo+2);i++) printf("      ");
                printf("Chef NO %d has serve the food\n",chefNo+1);
                fflush(stdout);
        sem_signal(&sem);
        isFull--;
        randomDelay();
    }
    pthread_exit(0);
}
```

6.4 การประยุกต์ใช้เซมาฟอร์กับกรณีศึกษา producer-consumer สำหรับลินุกซ์

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h> // for usleep()
#include <stdlib.h> // for exit() and random generator
#include <wait.h>   // for wait()
#include <time.h>
#include <semaphore.h>
#include <pthread.h>

#define BUF_SIZE 256
```

```
#define MAX_SIZE 255

sem_t empty;
sem_t full;
sem_t mutex;

int rp;
int wp;
char data[BUF_SIZE];


void randomDelay(void);
void *consumer(void *who);
void *producer(void *who);

int main(){
        pthread_t  tid[2];   // Child Process ID
        pthread_attr_t attr[2];
        int i;

// Initialize semaphores
        sem_init(&empty,0,MAX_SIZE);
        sem_init(&full,0,0);
        sem_init(&mutex,0,1);

        rp = wp = 0;

        for(i=0;i<2;i++)
                pthread_attr_init(&attr[i]);

        pthread_create(&tid[0],&attr[0],producer,NULL);
        pthread_create(&tid[1],&attr[1],consumer,NULL);

        pthread_join(tid[0],NULL);
        pthread_join(tid[1],NULL);

        sem_destroy(&empty);
        sem_destroy(&full);
        sem_destroy(&mutex);
}

void *consumer(void *who){
    int i=0;
    char d='a';
    int se,sf;

    while(1){
        if(!sem_getvalue(&full,&sf))
            printf("        (Consumer) full=%d\n",sf);fflush(stdout);
        sem_wait(&full);
        sem_wait(&mutex);
                if(!sem_getvalue(&full,&sf))
                     printf("        (Consumer) full=%d\n",sf);fflush(stdout);
                d = data[rp];
                printf("        (Consumer) Data number:%d = %c\n",i++,data[rp]);fflush(stdout);
                rp = (rp+1)%BUF_SIZE;
                rp++;
        sem_post(&mutex);
        sem_post(&empty);

// Remaining Section
      printf("       (Consumer) After exiting critical section\n");fflush(stdout);
      if(d=='0') break;
      randomDelay();
        randomDelay();
    }
    pthread_exit(0);
}

void *producer(void *who){
    int i=0;
    int se,sf;
    char temp[40];

    while(1){
        printf("(Producer) Please enter a character :");fflush(stdout);
        fgets(temp,40,stdin);
        if(!sem_getvalue(&empty,&se))
            printf("(Producer) empty=%d\n",se);fflush(stdout);
```

```
        sem_wait(&empty);
        sem_wait(&mutex);
                if(!sem_getvalue(&empty,&se))
                    printf("(Producer) empty=%d\n",se);fflush(stdout);
                printf("(Producer) Add %c into buffer\n",temp[0]);fflush(stdout);
                data[wp]=temp[0];
                // move the write pointer so that the consumer know when to read.
                wp = (wp+1)%BUF_SIZE;
                wp++;
        sem_post(&mutex);
        sem_post(&full);

// Remaining Section
        printf("(Producer) After exiting critical section\n");fflush(stdout);
        if(temp[0]=='0')break;
        randomDelay();
    }
    pthread_exit(0);
}

void randomDelay(void){
// This function provides a delay which slows the process down so we can see what happens
   srand(time(NULL));
   int stime = ((rand()%1000)+100)*1000;
   usleep(stime);
}
```

6.5 การประยุกต์ใช้เซมาฟอร์กับกรณีศึกษา producer-consumer สำหรับวินโดวส์

```
#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>

#define BUF_SIZE 256
#define MAX_SIZE 255

HANDLE empty,full,mutex;

DWORD sem_wait(HANDLE sem);
DWORD sem_signal(HANDLE sem);

int rp;
int wp;
char data[BUF_SIZE];

void randomDelay(void);
DWORD WINAPI consumer(LPVOID who);
DWORD WINAPI producer(LPVOID who);

int main(void){
        int i;
        DWORD tid[2];                           // Thread ID
        HANDLE th[2];                           // Thread Handle
        int param[]={0,1};

        rp = wp =0;

        empty = CreateSemaphore(
                NULL,           // default security attributes
                MAX_SIZE,       // initial count
                MAX_SIZE,       // maximum count
                NULL);          // unnamed semaphore
        full = CreateSemaphore(
                NULL,           // default security attributes
                0,              // initial count
                MAX_SIZE,       // maximum count
                NULL);          // unnamed semaphore

        mutex = CreateSemaphore(
                NULL,           // default security attributes
                1,              // initial count
                1,              // maximum count
                NULL);          // unnamed semaphore

        // Create 5 threads
```

```
        th[0] = CreateThread(
                NULL,                           // Default security attributes
                0,                              // Default stack size
                producer,                       // Thread function
                (void *)&param[0],              // Thread function parameter
                0,                              // Default creation flag
                &tid[0]);                       // Thread ID returned.

        th[1] = CreateThread(
                NULL,                           // Default security attributes
                0,                              // Default stack size
                consumer,                       // Thread function
                (void *)&param[1],              // Thread function parameter
                0,                              // Default creation flag
                &tid[1]);                       // Thread ID returned.

       // Wait until all threads finish
       for(i=0;i<2;i++)
                if(th[i]!=NULL)
                        WaitForSingleObject(th[i],INFINITE);
       CloseHandle(empty);
       CloseHandle(full);
       CloseHandle(mutex);
       return 0;
}

void randomDelay(void){
    int stime = ((rand()%1000)+100);
    Sleep(stime);
}


DWORD WINAPI producer(LPVOID who){
        char temp[40];

        while(1){
                printf("(Producer) Please enter a character :");fflush(stdout);
                fgets(temp,40,stdin);

        sem_wait(empty);
        sem_wait(mutex);
        // Critical Section
         printf("(Producer) Add %c into buffer\n",temp[0]);fflush(stdout);
         data[wp]=temp[0];
        // move the write pointer so that the consumer know when to read.
         wp = (wp+1)%BUF_SIZE;
                wp++;

        sem_signal(mutex);
        sem_signal(full);
        // Remaining Section

        printf("(Producer) After exiting critical section\n");fflush(stdout);
        if(temp[0]=='0')break;
//      randomDelay();
    }
    return 0;
}

DWORD WINAPI consumer(LPVOID who){
    int i=0;
        char d='a';

    while(1){
        sem_wait(full);
        sem_wait(mutex);
        // Critical Section

                d = data[rp];
        printf("        (Consumer) Data number:%d = %c\n",i++,data[rp]);fflush(stdout);
        rp = (rp+1)%BUF_SIZE;
                rp++;

        sem_signal(mutex);
        sem_signal(empty);

        // Remaining Section
        printf("       (Consumer) After exiting critical section\n");fflush(stdout);
    if(d=='0') break;
```

```
    randomDelay();
        randomDelay();
        }
    return 0;
}


DWORD sem_wait(HANDLE sem){
    DWORD result = WaitForSingleObject(sem,INFINITE);

        switch(result){
                case WAIT_OBJECT_0:return 1;
                case WAIT_TIMEOUT: return 0;
        }
        return 0;
}

DWORD sem_signal(HANDLE sem){
    return ReleaseSemaphore(sem,1,NULL); // Increase by one
}
```

6.6 การประยุกต์ใช้เซมาฟอร์กับกรณีศึกษา reader-writer สำหรับลินุกซ์

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h> // for usleep()
#include <stdlib.h> // for exit() and random generator
#include <wait.h>   // for wait()
#include <time.h>
#include <semaphore.h>
#include <pthread.h>

sem_t wrt;
sem_t mutex;
int readcount;

char data[1024];


void randomDelay(void);
void *writer(void *who);
void *reader(void *who);

int main(){
        pthread_t  tid[5];   // Child Process ID
        pthread_attr_t attr[5];
        int param[5]={0,1,2,3,4};
        int i;

// Initialize semaphores
        sem_init(&wrt,0,1);
        sem_init(&mutex,0,1);

        for(i=0;i<5;i++)
                pthread_attr_init(&attr[i]);

        for(i=0;i<4;i++)
                pthread_create(&tid[i],&attr[i],reader,(void *)&param[i]);
        pthread_create(&tid[4],&attr[4],writer,NULL);

        for(i=0;i<5;i++)
                pthread_join(tid[i],NULL);

        sem_destroy(&wrt);
        sem_destroy(&mutex);
        return 0;
}

void *reader(void *who){
    int readerNo=(int)*((int *)who);
    int i=0;

    while(1){
        sem_wait(&mutex);
```

```
        readcount++;
        if(readcount==1)
                sem_wait(&wrt);
        sem_post(&mutex);

                randomDelay();
                printf("        (Consumer %d) Data is:%s\n",readerNo,data);fflush(stdout);
                randomDelay();

        sem_wait(&mutex);
        readcount--;
        if(readcount==0)
                sem_post(&wrt);
        sem_post(&mutex);

// Remaining Section
      printf("       (Consumer %d) After exiting critical section\n",readerNo);fflush(stdout);
      if(data[0]=='0') break;
      randomDelay();
    }
    pthread_exit(0);
}

void *writer(void *who){
    int i=0;
    char temp[1024];

    while(1){
        printf("(Producer) Please enter a text (starting with 0 to stop) :");fflush(stdout);
        fgets(temp,1000,stdin);

        sem_wait(&wrt);

                printf("(Producer) Add %s into buffer\n",temp);fflush(stdout);
                for(i=0;i<1023;i++){
                        if(temp[i]==0)break;
                        data[i]=temp[i];
                }
                data[i]=0;
        sem_post(&wrt);

// Remaining Section
        printf("(Producer) After exiting critical section\n");fflush(stdout);
        if(temp[0]=='0')break;
    }
    pthread_exit(0);
}

void randomDelay(void){
// This function provides a delay which slows the process down so we can see what happens
   srand(time(NULL));
   int stime = ((rand()%2000)+100)*1000;
   usleep(stime);
}
```

จากบริเวณที่ตีกรอบไว้ในตัวอย่างโปรแกรมข้างบน ให้นักศึกษาทดลองแก้ไขโค้ดให้เป็นดังนี้

```
for(int ii=0;ii<5;ii++)
        randomDelay();
```

เมื่อนักศึกษาลองรันโปรแกรมนี้ดูอีกครั้ง จะพบว่าคราวนี้โพรเซสผู้อ่านมีการหน่วงเวลาเพื่ออยู่ในส่วนวิกฤตินานขึ้น และเริ่มพบปัญหาที่ผู้อ่านแต่ละตัวผลัดกันเข้าออกส่วนวิกฤติ จนทำให้ผู้เขียนไม่สามารถเข้าส่วนวิกฤติได้เลย (นักศึกษาจะไม่สามารถแก้ไขข้อความได้อีก)

ตัวอย่างต่อไปนี้ เป็นการปรับแก้ขั้นตอนวิธีการจัดการผู้เขียนผู้อ่าน ให้อยู่ในลักษณะดังที่ได้ศึกษามาแล้วในบทเรียนข้างต้น ส่งผลทำให้โพรเซสผู้อ่านที่เข้ามาในภายหลังจากที่ผู้เขียนเข้ามาแล้ว จะแซงเข้าส่วนวิกฤติไปไม่ได้ ต้องรอให้ผู้เขียนทั้งหมดที่รอเข้าส่วนวิกฤติ ต่างเข้าส่วนวิกฤติไปให้หมดก่อน ผู้อ่านที่รออยู่จึงจะตามเข้าส่วนวิกฤติไปได้

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h> // for usleep()
#include <stdlib.h> // for exit() and random generator
#include <wait.h>   // for wait()
#include <time.h>
#include <semaphore.h>
#include <pthread.h>

sem_t rmutex,wmutex,readTry,resource;
int readcount,writecount;

char data[1024];


void randomDelay(void);
void *writer(void *who);
void *reader(void *who);

int main(){
        pthread_t  tid[5];   // Child Process ID
       pthread_attr_t attr[5];
       int param[5]={0,1,2,3,4};
       int i;
       readcount=writecount=0;
// Initialize semaphores
       sem_init(&rmutex,0,1);
       sem_init(&wmutex,0,1);
       sem_init(&readTry,0,1);
       sem_init(&resource,0,1);

       for(i=0;i<5;i++)
              pthread_attr_init(&attr[i]);

       for(i=0;i<4;i++)
              pthread_create(&tid[i],&attr[i],reader,(void *)&param[i]);
       pthread_create(&tid[4],&attr[4],writer,NULL);

       for(i=0;i<5;i++)
              pthread_join(tid[i],NULL);

       sem_destroy(&rmutex);
       sem_destroy(&wmutex);
       sem_destroy(&readTry);
       sem_destroy(&resource);
       return 0;
}

void *reader(void *who){
    int readerNo=(int)*((int *)who);
    int i=0;

    for(int ii=0;ii<readerNo;ii++) randomDelay();
    while(1){
    sem_wait(&readTry);
       sem_wait(&rmutex);
       readcount++;
       if(readcount==1)
              sem_wait(&resource);
       sem_post(&rmutex);
       sem_post(&readTry);

              randomDelay();
              printf("        (Consumer %d) Data is:%s\n",readerNo,data);fflush(stdout);
              for(int ii=0;ii<5;ii++)
           randomDelay();

       sem_wait(&rmutex);
       readcount--;
       if(readcount==0)
              sem_post(&resource);
       sem_post(&rmutex);

// Remaining Section
      printf("       (Consumer %d) After exiting critical section\n",readerNo);fflush(stdout);
```

```
      if(data[0]=='0') break;
      randomDelay();
    }
    pthread_exit(0);
}

void *writer(void *who){
    int i=0;
    char temp[1024];

    while(1){
        printf("(Producer) Please enter a text (starting with 0 to stop) :");fflush(stdout);
        fgets(temp,1000,stdin);

        sem_wait(&wmutex);
        writecount++;
        if(writecount==1)
         sem_wait(&readTry);
    sem_post(&wmutex);
    sem_wait(&resource);

                printf("(Producer) Add %s into buffer\n",temp);fflush(stdout);
                for(i=0;i<1023;i++){
                        if(temp[i]==0)break;
                        data[i]=temp[i];
                }
                data[i]=0;
        sem_post(&resource);
        sem_wait(&wmutex);
        writecount--;
        if(writecount==0)
         sem_post(&readTry);
    sem_post(&wmutex);
// Remaining Section
        printf("(Producer) After exiting critical section\n");fflush(stdout);
        if(temp[0]=='0')break;
    }
    pthread_exit(0);
}

void randomDelay(void){
// This function provides a delay which slows the process down so we can see what happens
   srand(time(NULL));
   int stime = ((rand()%2000)+100)*1000;
   usleep(stime);
}
```

6.7 การประยุกต์ใช้เซมาฟอร์กับกรณีศึกษา reader-writer สำหรับวินโดวส์

```
#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>


HANDLE wrt,mutex;
int readcount;
char data[1024];


DWORD sem_wait(HANDLE sem);
DWORD sem_signal(HANDLE sem);

void randomDelay(void);
DWORD WINAPI writer(LPVOID who);
DWORD WINAPI reader(LPVOID who);

int main(void){
        int i;
        DWORD tid[5];                           // Thread ID
        HANDLE th[5];                           // Thread Handle
        int param[5]={0,1,2,3,4};

        wrt = CreateSemaphore(
                NULL,   // default security attributes
                1,   // initial count
                1,   // maximum count
```

```
                NULL);  // unnamed semaphore
        mutex = CreateSemaphore(
                NULL,   // default security attributes
                1,   // initial count
                1,   // maximum count
                NULL);  // unnamed semaphore

        // Create 5 threads
        for(i=0;i<4;i++)
        th[i] = CreateThread(
                NULL,                           // Default security attributes
                0,                                      // Default stack size
                reader,                 // Thread function
                (void *)&param[i],      // Thread function parameter
                0,                                      // Default creation flag
                &tid[i]);                       // Thread ID returned.

        th[4] = CreateThread(
                NULL,                           // Default security attributes
                0,                                      // Default stack size
                writer,                 // Thread function
                (void *)&param[4],      // Thread function parameter
                0,                                      // Default creation flag
                &tid[4]);                               // Thread ID returned.

       // Wait until all threads finish
        for(i=0;i<5;i++)
                if(th[i]!=NULL)
                        WaitForSingleObject(th[i],INFINITE);
        CloseHandle(wrt);
        CloseHandle(mutex);
        return 0;
}

void randomDelay(void){
    int stime = ((rand()%2000)+100);
    Sleep(stime);
}


DWORD WINAPI reader(LPVOID who){
    int readerNo,i=0;

    readerNo = (int)*((int *)who);

        while(1){
                sem_wait(mutex);
                readcount++;
                if(readcount==1)
                        sem_wait(wrt);
                sem_signal(mutex);

                        randomDelay();
                printf("        (Consumer %d) Data is:%s\n",readerNo,data);fflush(stdout);
                    randomDelay();

                sem_wait(mutex);
                readcount--;
                if(readcount==0)
                        sem_signal(wrt);
                sem_signal(mutex);

// Remaining Section
                printf("       (Consumer %d) After exiting critical
section\n",readerNo);fflush(stdout);
                if(data[0]=='0') break;
                randomDelay();
        }
    return 0;
}

DWORD WINAPI writer(LPVOID who){
    int i=0;
    char temp[1024];

while(1){
        printf("(Producer) Please enter a text (starting with 0 to stop) :");fflush(stdout);
        fgets(temp,1000,stdin);
```

```
        sem_wait(wrt);

                printf("(Producer) Add %s into buffer\n",temp);fflush(stdout);
                for(i=0;i<1023;i++){
                        if(temp[i]==0)break;
                        data[i]=temp[i];
                }
                data[i]=0;
        sem_signal(wrt);

// Remaining Section
        printf("(Producer) After exiting critical section\n");fflush(stdout);
        if(temp[0]=='0')break;
    }
    return 0;
}


DWORD sem_wait(HANDLE sem){
    DWORD result = WaitForSingleObject(sem,INFINITE);

        switch(result){
                case WAIT_OBJECT_0:return 1;
                case WAIT_TIMEOUT: return 0;
        }
        return 0;
}

DWORD sem_signal(HANDLE sem){
    return ReleaseSemaphore(sem,1,NULL); // Increase by one
}
```

6.8 การประยุกต์ใช้เซมาฟอร์กับกรณีศึกษา dining philosophers สำหรับลินุกซ์

ตัวอย่างนี้ต้องการไฟล์ conio.h ที่ได้เตรียมไว้ให้เป็นพิเศษ สำหรับยูนิกซ์หรือลินุกซ์ที่รันผ่านเทอร์มินัลตามมาตรฐาน VT-100 หรือสูงกว่า ให้นักศึกษาดาวน์โหลดไฟล์จากเว็ปช่วยสอนมาใช้งานประกอบตัวอย่างด้วย

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h> // for usleep()
#include <stdlib.h> // for exit() and random generator
#include <wait.h>   // for wait()
#include <time.h>
#include <semaphore.h>
#include <pthread.h>
#include "../../conio.h"


#define CHAIRNUM 5

sem_t chopstick[CHAIRNUM];
int timeUsed[CHAIRNUM];

void randomDelay(void);
void *philosopher(void *who);

int main(){
        pthread_t  tid[CHAIRNUM];   // Child Process ID
        pthread_attr_t attr[CHAIRNUM];
        int param[CHAIRNUM];
        int i;

        clrscr();
// Initialize semaphores
        for(i=0;i<CHAIRNUM;i++){
                param[i]=i;
                timeUsed[i]=0;
                sem_init(&chopstick[i],0,1);
        }
```

```
        for(i=0;i<CHAIRNUM;i++)
                pthread_attr_init(&attr[i]);

        for(i=0;i<CHAIRNUM;i++)
                pthread_create(&tid[i],&attr[i],philosopher,(void *)&param[i]);

        for(i=0;i<CHAIRNUM;i++)
                pthread_join(tid[i],NULL);

        for(i=0;i<CHAIRNUM;i++)
                sem_destroy(&chopstick[i]);

        return 0;
}

void *philosopher(void *who){
    int no=(int)*((int *)who);
    int i=0,j;

    for(i=0;i<10;i++){
       gotoxy(1,no*4+1);
       printf("Mr. %c is thinking...
\n",'A'+no);fflush(stdout);
        randomDelay();

        sem_wait(&chopstick[no]);
                gotoxy(1,no*4+1);
                printf("Mr. %c is taking a chopstick on the left side...
\n",'A'+no);fflush(stdout);
        sem_wait(&chopstick[(no+1)%CHAIRNUM]);
                gotoxy(1,no*4+2);
                printf("Mr. %c is taking a chopstick on the right
side...\n",'A'+no);fflush(stdout);

// Critical Section
        randomDelay();
        timeUsed[no]++;
        timeUsed[(no+1)%CHAIRNUM]++;
        gotoxy(no*10+1,CHAIRNUM*4+1);
                printf("CH[%d]=%d ",no,timeUsed[no]);
        gotoxy(((no+1)%CHAIRNUM)*10+1,CHAIRNUM*4+1);
                printf("CH[%d]=%d ",(no+1)%CHAIRNUM,timeUsed[(no+1)%CHAIRNUM]);
        gotoxy(no*10+1,CHAIRNUM*4+2);
                printf("  [%c]=%d ",'A'+no,i);

        gotoxy(1,no*4+1);
        printf("                                                                    \n");
        gotoxy(1,no*4+2);
        printf("                                                                    \n");
        randomDelay();

        sem_post(&chopstick[no]);
                gotoxy(1,no*4+1);
                printf("Mr. %c drops a chopstick on the left side...
\n",'A'+no);fflush(stdout);
        sem_post(&chopstick[(no+1)%CHAIRNUM]);
                gotoxy(1,no*4+2);
                printf("Mr. %c drops a chopstick on the right side...
\n",'A'+no);fflush(stdout);
// Remaining Section
        randomDelay();
        gotoxy(1,no*4+2);
        printf("                                                                    \n");
        gotoxy(1,no*4+1);
        printf("Mr. %c is chewing food...                              \n",'A'+no);fflush(stdout);
        randomDelay();
    }
    gotoxy(1,no*4+1);
    printf("Mr. %c is full...                                       \n",'A'+no);fflush(stdout);
    pthread_exit(0);
}

void randomDelay(void){
// This function provides a delay which slows the process down so we can see what happens
   srand(time(NULL));
   int stime = ((rand()%2000)+100)*1000;
   usleep(stime);
}
```

6.9 การประยุกต์ใช้เซมาฟอร์กับกรณีศึกษา dining philosiphers สำหรับวินโดวส์

ตัวอย่างนี้ต้องการไฟล์ winconio.h ที่ได้เตรียมไว้ให้เป็นพิเศษ สำหรับวินโดวส์เพื่อให้ทำงานได้อย่างถูกต้องใน text console ให้นักศึกษาดาวน์โหลดไฟล์จากเว็ปช่วยสอนมาใช้งานประกอบตัวอย่างด้วย

```
#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>
#include "winconio.h"

#define CHAIRNUM 5

HANDLE chopstick[CHAIRNUM];
int timeUsed[CHAIRNUM];

void randomDelay(void);
DWORD WINAPI philosopher(LPVOID who);
DWORD sem_wait(HANDLE sem);
DWORD sem_signal(HANDLE sem);

int main(void){
        int i;
        DWORD tid[CHAIRNUM];                        // Thread ID
        HANDLE th[CHAIRNUM];                        // Thread Handle
        int param[CHAIRNUM];

        clrscr();
        for(i=0;i<CHAIRNUM;i++){
                chopstick[i] = CreateSemaphore(
                                                    NULL,   // default security attributes
                                                    1,      // initial count
                                                    1,      // maximum count
                                                    NULL);  // unnamed semaphore
                param[i]=i;
                timeUsed[i]=0;
        }

        // Create n threads
        for(i=0;i<CHAIRNUM;i++)
        th[i] = CreateThread(
                NULL,                           // Default security attributes
                0,                              // Default stack size
                philosopher,                    // Thread function
                (void *)&param[i],              // Thread function parameter
                0,                              // Default creation flag
                &tid[i]);                       // Thread ID returned.

       // Wait until all threads finish
        for(i=0;i<CHAIRNUM;i++)
                if(th[i]!=NULL)
                        WaitForSingleObject(th[i],INFINITE);

        for(i=0;i<CHAIRNUM;i++)
                CloseHandle(chopstick[i]);
        return 0;
}

void randomDelay(void){
    int stime = ((rand()%2000)+100);
    Sleep(stime);
}


DWORD WINAPI philosopher(LPVOID who){
    int no,i=0;

    no = (int)*((int *)who);

        for(i=0;i<10;i++){
                gotoxy(1,no*4+1);
                printf("Mr. %c is thinking...                          \n",'A'+no);fflush(stdout);
                randomDelay();
```

```
                sem_wait(chopstick[no]);
                        gotoxy(1,no*4+1);
                        printf("Mr. %c is taking a chopstick on the left side...
\n",'A'+no);fflush(stdout);
                sem_wait(chopstick[(no+1)%CHAIRNUM]);
                        gotoxy(1,no*4+2);
                        printf("Mr. %c is taking a chopstick on the right
side...\n",'A'+no);fflush(stdout);

                        // Critical Section
        if(!no)clrscr();  // To reduced garbage from writing in the wrong location
                randomDelay();
                timeUsed[no]++;
                timeUsed[(no+1)%CHAIRNUM]++;
                gotoxy(no*10+1,CHAIRNUM*4);
                        printf("CH[%d]=%d ",no,timeUsed[no]);
                gotoxy(((no+1)%CHAIRNUM)*10+1,CHAIRNUM*4);
                        printf("CH[%d]=%d ",(no+1)%CHAIRNUM,timeUsed[(no+1)%CHAIRNUM]);
                gotoxy(no*10+1,CHAIRNUM*4+1);
                        printf("  [%c]=%d ",'A'+no,i);fflush(stdout);

                gotoxy(1,no*4+1);
                printf("                                                              \n");
                gotoxy(1,no*4+2);
                printf("                                                              \n");fflush(stdout);
                randomDelay();

                sem_signal(chopstick[no]);
                        gotoxy(1,no*4+1);
                        printf("Mr. %c drops a chopstick on the left side...    \n",'A'+no);
fflush(stdout);
                sem_signal(chopstick[(no+1)%CHAIRNUM]);
                        gotoxy(1,no*4+2);
                        printf("Mr. %c drops a chopstick on the right side...     \n",'A'+no);
fflush(stdout);
        // Remaining Section
                randomDelay();
                gotoxy(1,no*4+2);
                printf("                                                               \n");
                gotoxy(1,no*4+1);
                printf("Mr. %c is chewing food...                                     \n",'A'+no);
fflush(stdout);
                randomDelay();
    }
    gotoxy(1,no*4+1);
    printf("Mr. %c is full...                                          \n",'A'+no);fflush(stdout);
    return 0;
}

DWORD sem_wait(HANDLE sem){
    DWORD result = WaitForSingleObject(sem,INFINITE);

        switch(result){
                case WAIT_OBJECT_0:return 1;
                case WAIT_TIMEOUT: return 0;
        }
        return 0;
}

DWORD sem_signal(HANDLE sem){
    return ReleaseSemaphore(sem,1,NULL); // Increase by one
}
```