บทที่ 7 สภาวะติดตาย

วัตถุประสงค์ของเนื้อหา

- ศึกษาถึงคุณลักษณะของสภาวะติดตาย และการตรวจจับสภาวะติดตาย
- ศึกษาถึงวิธีการจัดการต่อสภาวะติดตาย การหลีกเลี่ยงสภาวะติดตาย การป้องกันสภาวะติดตาย และการกู้คืนระบบเมื่อเกิดสภาวะติดตาย
- ศึกษาถึงขั้นตอนวิธีต่างๆ ที่นำมาใช้ตรวจจับสภาวะติดตาย การหาสภานะปลอดภัยของระบบ (safe state) และการกู้คินจากสภาวะติดตายในรูปแบบต่างๆ

สิ่งที่คาดหวังจากการเรียนในบทนี้

- นักศึกษาเข้าใจถึงสภาวะติดตายว่ามีลักษณะเช่นใด
- นักศึกษาสามารถปรับปรุงซอฟต์แวร์ของนักศึกษา ให้สามารถลดโอกาสเสี่ยงต่อสภาวะติดตาย และสามารถพัฒนาซอฟต์แวร์ให้มีกลไกที่จะกู้คืนจากสภาวะติดตายได้

วัตถุประสงค์ของปฏิบัติการท้ายบท

- นักศึกษาได้ทดลองตัวอย่างโปรแกรมที่แสดงให้เห็นถึงสภาวะติดตาย
- นักศึกษาได้เห็นตัวอย่างต่างๆ ของสภาวะติดตายที่เกิดขึ้นในระบบ
- นักศึกษาได้ทดลองพัฒนาโปรแกรมเพื่อจัดการต่อสภาวะติดตาย

สิ่งที่คาดหวังจากปฏิบัติการท้ายบท

- นักศึกษาสามารถประยุกต์แนวคิดการจัดการต่อสภาวะติดตาย และการกู้คืนต่อสภาวะติดตาย กับโปรแกรมตัวอย่างได้
- นักศึกษาสามารถนำแนวคิดการจัดการต่อสภาวะติดตาย ไปใช้งานในอนาคตได้

เวลาที่ใช้ในการเรียนการสอน

- ทฤษฎี 2 ชั่วโมง
    - ศึกษาถึงคุณลักษณะของสภาวะติดตาย และการจัดการต่อสภาวะติดตาย 1 ชั่วโมง
    - ศึกษาถึงขั้นตอนวิธีการหาสถานะปลอดภัยของระบบ 1 ชั่วโมง
- ปฏิบัติ 2 ชั่วโมง
    - ศึกษาลักษณะของสภาวะติดตายจากโปรแกรมตัวอย่าง 1 ชั่วโมง
    - ปรับปรุงแก้ไขโปรแกรมตัวอย่างที่อยู่ในสภาวะติดตาย ให้สามารถกู้คืนจากสภาวะติดตายได้ 1 ชั่วโมง

บทที่ 7 สภาวะติดตาย (Deadlock)

7.1 คำจำกัดความของสภาวะติดตาย

สภาวะติดตาย (Deadlock) เป็นสภาวะที่กลุ่มของโพรเซส อันประกอบไปด้วยโพรเซสหลายตัวไม่สามารถทำงานต่อไปได้อีก อันเนื่องมาจากต้องรอคอยทรัพยากรที่โพรเซสอื่นในกลุ่มได้รับไปแล้ว ในลักษณะการวนร้องขอกันเป็นลูกโซ่ ทำให้ไม่มีโพรเซสใดภายในกลุ่มสามารถเดินหน้าต่อไปได้ (รอคอยทรัพยากรซึ่งกันและกัน)

จากที่ได้ศึกษามาในบทก่อนๆ หน้านั้น การจัดสรรทรัพยากร (resource) กระทำโดยระบบปฏิบัติการ โดยเราพิจารณาทุกอย่างที่โพรเซสหนึ่งๆ จำเป็นต้องใช้งาน ถือเป็นทรัพยากรทั้งสิ้น อาทิเช่น เวลาในการครอบครองซีพียู พื้นที่หน่วยความจำ อุปกรณ์เอาต์พุตอินพุต ไฟล์ในหน่วยความจำสำรอง เป็นต้น โพรเซสหนึ่งๆ เมื่อจะใช้งานทรัพยากรที่มีจำกัดในซีพียู จำเป็นต้องมีกระบวนการตามลำดับคือ เริ่มจากการ**ร้องขอ (request)** ทรัพยากรจากระบบปฏิบัติการ เมื่อระบบปฏิบัติการได้มอบทรัพยากรนั้นๆ ไปแล้ว โพรเซสอื่นๆ หากจะใช้งานทรัพยากรนั้นก็ต้องรอ ในขณะที่โพรเซสที่ได้รับทรัพยากรไป ก็จะ**ใช้งาน (use)** ทรัพยากรนั้นจนเสร็จงาน จากนั้นโพรเซสก็จะ**คืน(release)** ทรัพยากรให้กับระบบปฏิบัติการเพื่อระบบปฏิบัติการจะได้มอบให้กับโพรเซสอื่นที่ร้องขอต่อไป

สมมติว่ามีโพรเซสสองตัว มีกลไกการทำงานดังนี้

```
void *pOne(void *param){
      pthread_mutex_lock(&mutex1);
      pthread_mutex_lock(&mutex2);

      // Critical Section

      pthread_mutex_unlock(&mutex2);
      pthread_mutex_unlock(&mutex1);
      pthread_exit(0);
}
```

```
void *pTwo(void *param){
      pthread_mutex_lock(&mutex2);
      pthread_mutex_lock(&mutex1);

      // Critical Section

      pthread_mutex_unlock(&mutex1);
      pthread_mutex_unlock(&mutex2);
      pthread_exit(0);
}
```

สมมติว่าโพรเซสทั้งสองเริ่มต้นทำคำสั่ง pthread_mutex_lock() ตัวแรกพร้อมกัน จะเห็นว่าทั้งสองโพรเซสจะสารถทำคำสั่งกล่าวได้อย่างไม่น่าจะมีปัญหา แต่เราจะพบว่า โพรเซสทั้งสองจะหยุดอยู่ที่คำสั่ง pthread_mutex_lock() ตัวที่สอง โดยแต่ละโพรเซสจะรอคอยให้อีกโพรเซสหนึ่งปล่อยการครอบครองทรัพยากรดังกล่าวเสียก่อน ซึ่งถ้าแต่ละโพรเซสไม่ยอมที่จะปล่อยล็อคที่ตนครอบครองตัวแรกไว้ ทั้งสองโพรเซสก็จะไม่สามารถทำงานต่อไปได้ เข้าสู่สภาวะติดตาย (deadlock) ในที่สุด

7.2 การจำแนกสภาวะติดตาย

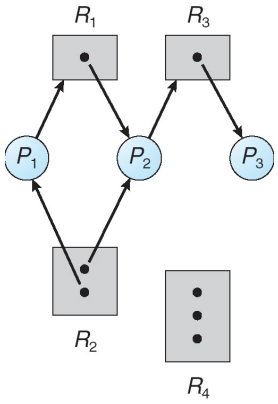
สภาวะติดตายสามารถเกิดขึ้นได้ หากมีปัจจัยดังกล่าวนี้เกิดขึ้นพร้อมกัน

- มีการกำหนดการทำงานที่โพรเซสหลายตัวจะต้องใช้ทรัพยากรร่วมกันในลักษณะที่ทรัพยากรดังกล่าวจะถูกครอบครองโดยโพรเซส**เพียงโพรเซสเดียวในเวลาใดเวลาหนึ่ง(mutual exclusion)**
- โพรเซสหนึ่งได้ครอบครองทรัพยากรที่จำเป็นต่อการทำงานไว้บางส่วนแล้ว แต่ไม่สามารถทำงานต่อได้ เพราะ**กำลังรอคอยให้โพรเซสอื่นปล่อยการครอบครองทรัพยากร**อีกส่วนหนึ่งที่จำเป็นต่อการทำงาน **(hold and wait)**
- เมื่อโพรเซสได้รับทรัพยากรไปแล้ว **ไม่มีกลไกอื่นใดมาบังคับให้โพรเซสนั้นๆ ปล่อยการครอบครองทรัพยากร**ดังกล่าวนอกจากความสมัครใจของโพรเซสนั้นๆ **(no preemption)**
- โพรเซสหนึ่งๆ กำลังรอคอยทรัพยากรจากอีกโพรเซส ในขณะที่โพรเซสดังกล่าวก็กำลัง**รอคอยทรัพยากรจากอีกโพรเซสวนกันไปเป็นวงกลม (circular wait)** เช่นมีโพรเซสอยู่ n โพรเซส P_0 รอทรัพยากรจาก P_1 ในขณะที่ P_1 รอทรัพยากรจาก P_2 ไปจนถึง P_{n-1} กำลังรอทรัพยากรจาก P_0 เป็นต้น

<u>กราฟแสดงการครอบครองทรัพยากร (Resource-allocation graph)</u>

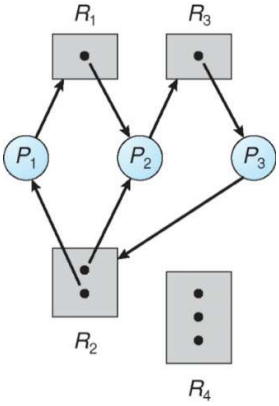
สำหรับการวิเคราะห์โพรเซสต่างๆ ภายในระบบว่าจะสามารถก่อให้เกิดสภาวะติดตายได้หรือไม่นั้น ทำได้โดยการนำเสนอด้วยกราฟแสดงการครอบครองทรัพยากร (resource-allocation graph)

- กำหนดกราฟให้มีโหนด (vertice, V) แบ่งออกเป็นสองประเภทคือ
  - $P=\{P_1,P_2,P_3,...,P_n\}$ คือเซ็ตของโพรเซสที่รันอยู่ในระบบจำนวน n โพรเซส
  - $R=\{R_1,R_2,R_3,...,R_m\}$ คือเซ็ตของทรัพยากรที่มีอยู่ในระบบจำนวน m ทรัพยากร
- กราฟมีเส้นทาง (edge, E) เป็นเส้นทางแบบทางเดียว (one-way)โดยสองฝั่งของเส้นทางนั้น ฝั่งหนึ่งเชื่อมกับโหนด P และอีกฝั่งเชื่อมกับโหนด R โดยที่
  - หากจุดเริ่มต้นเป็นโพรเซสและปลายทางคือทรัพยากร เราจะเรียกว่าเป็น**<u>เส้นทางการร้องขอ request edge</u>** ($P_i \rightarrow R_i$) หมายถึงโพรเซสดังกล่าวกำลังร้องขอทรัพยากร Ri อยู่(แต่ยังมิได้รับมา)
  - หากจุดเริ่มต้นเป็นทรัพยากรและปลายทางคือโพรเซส เราจะเรียกว่าเป็น**<u>เส้นทางการให้ทรัพยากร assignment edge</u>** ($R_i \rightarrow P_i$) หมายถึงโพรเซสดังกล่าวได้ทรัพยากร Ri มาแล้วแต่ยังมิได้ส่งคืนระบบ



จากตัวอย่างด้านซ้าย จะเห็นว่ามีโพรเซสอยู่ในระบบ 3 โพรเซสด้วยกัน และมีทรัพยากรในระบบ 4 ชนิด โดยแบ่งเป็น R_1 ซึ่งมีอยู่เพียงหนึ่งหน่วย ดังนั้นโพรเซสหนึ่งๆ จะใช้งาน R_1 ได้เพียงหนึ่งตัวในเวลาใดเวลาหนึ่ง R_2 มีอยู่สองหน่วย ซึ่งสามารถแบ่งให้โพรเซสใช้งานได้พร้อมกัน 2 โพรเซสในเวลาใดเวลาหนึ่ง (โดยมีเงื่อนไขว่า โพรเซสหนึ่งๆ ใช้เพียงหนึ่งหน่วยในเวลาใดเวลาหนึ่ง) R_3 มีหนึ่งหน่วย และ R_4 มีสามหน่วย

การนำเสนอในกราฟ เป็นสภาพ ณ เวลาหนึ่งที่ P_1 ได้ทรัพยากรจากR_2 มาแล้วหนึ่งหน่วย และกำลังร้องขอทรัพยากร R_1 ซึ่งในขณะนั้น P_2 กำลังถือครองอยู่ P_2 เองในขณะนั้นก็ยังถือครองทรัพยากรอีกหนึ่งหน่วยจาก R_2 และกำลังร้องขอทรัพยากรจาก R_3 ในขณะที่ P_3 ได้ถือครอง R_3 ที่มีเพียงหนึ่งหน่วยนั้นไว้



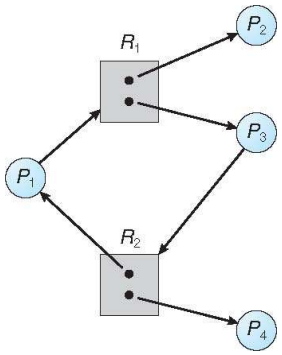
คราวนี้ สมมติต่อไปว่า ในช่วงเวลาดังกล่าว P_3 กำลังร้องขอทรัพยากรจาก R_2 ด้วย ซึ่งในขณะนั้น ทรัพยากรที่มีอยู่ 2 หน่วยใน R_2 ได้มอบให้ P_1 และ P_2 ไปแล้ว จากกราฟ จะเห็นได้ชัดว่าเกิด**<u>เส้นทางที่เป็นวงกลม (circular path)</u>** ขึ้นคือ

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

และ

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

ในลักษณะเช่นนี้ เราจะเห็นว่า หากไม่มีโพรเซสหนึ่งโพรเซสใดยอมปล่อยทรัพยากรที่ตนถือครองอยู่ ทั้งสามโพรเซสจะดำเนินต่อไปไม่ได้ และเกิดสภาวะติดตายเกิดขึ้น



การที่เห็นเส้นทางวนเป็นวงกลมในกราฟ **<u>ไม่จำเป็นต้องหมายความว่ากำลังเกิดสภาวะติดตาย</u>** ดังเช่นรูปด้านซ้ายนี้ แม้ว่าจะมีเส้นทางเป็นวงกลมผ่าน P_1 และ P_2 อยู่ก็ตาม แต่สังเกตว่าทรัพยากรที่R_1 และ R_2 ซึ่งมีอยู่อย่างละ 2 หน่วยนั้น ถูกครอบครองโดย P_2 และ P_4 ตามลำดับ ซึ่ง P_2 และ P_4 นั้นไม่ได้ต้องการทรัพยากรอื่นเพิ่มเติมอีกเพื่อจะทำงานให้เสร็จ ดังนั้นเมื่อ P_2 และ P_4 ทำงานในส่วนที่ต้องใช้ทรัพยากรดังกล่าวเสร็จสิ้นแล้ว ก็จะปล่อยการครอบครองทรัพยากร และทำให้มีหน่วยทรัพยากรเหลือใน R_1 และ R_2 เพื่อให้ P_1 และ P_3 ทำงานต่อไปได้

กล่าวโดยสรุป

- หากไม่พบเส้นทางเป็นวงกลมในกราฟ หมายถึงจะไม่มีสภาวะติดตายเกิดขึ้น
- หากพบเส้นทางวงกลมในกราฟ
    - ถ้าทรัพยากรที่เป็นส่วนหนึ่งของเส้นทางวงกลม มีเพียง**หนึ่งหน่วยต่อชนิดทรัพยากร** นั่นคือกำลังเกิดสภาวะติดตาย
    - ถ้าทรพยากรที่เป็นส่วนหนึ่งของเส้นทางวงกลมมีอยู่**มากกว่าหนึ่งหน่วยต่อชนิดทรัพยากร** หมายความว่ามีความเป็นไปได้ที่อาจจะเกิดสภาวะติดตาย

7.3 วิธีการจัดการต่อสภาวะติดตาย

สภาวะติดตายสามารถจัดการได้ในหลายลักษณะ ดังเช่น

- **การป้องกันจากสภาวะติดตาย** Deadlock prevention ออกแบบโพรเซสที่ทำงานเกี่ยวข้องกันในระบบและตัวระบบปฏิบัติการเอง ไม่ให้มีโอกาสเกิดสภาวะติดตายขึ้นอย่างเด็ดขาด
- **การหลีกเลี่ยงจากสภาวะติดตาย** Deadlock avoidance ออกแบบโพรเซสและตัวระบบปฏิบัติการ ให้มีกลไกที่จะตรวจสอบความน่าจะเป็นว่าอาจจะเกิดสภาวะติดตาย และมีกลไกที่จะเลี่ยงไม่ให้เข้าสู่สภาวะติดตายที่อาจจะเกิดขึ้นได้
- มี**ระบบการตรวจสอบสภาวะติดตาย** โดยเมื่อเกิดสภาวะติดตายเกิดขึ้น ก็ให้ระบบเข้าไปจัดการแก้ไขกู้คืนจากสภาวะติดตายด้วยตนเอง
- **ปล่อยให้สภาวะติดตายเกิดขึ้นตามธรรมชาติ** โดยปล่อยหน้าที่ของการกู้สภาวะติดตายให้เป็นหน้าที่ของนักพัฒนาโปรแกรมจัดการหรือโดยตัวผู้ใช้เอง (แนวทางนี้เป็นแนวทางที่พบได้ในระบบปฏิบัติการทั่วไป ทั้งในยูนิกซ์ ลินุกซ์และวินโดวส์)

ในทางปฏิบัติ เราอาจจะใช้การจัดการข้างต้นมากกว่าหนึ่งหนทางผสมผสานกัน

7.3.1 การป้องกันจากสภาวะติดตาย

จากที่ได้ทราบถึงปัจจัยต่างๆ ที่จะส่งผลต่อให้เกิดสภาวะติดตายได้ การป้องกันจากสภาวะติดตาย จึงเป็นการออกแบบระบบและตัวซอฟต์แวร์ผู้ใช้ ไม่ให้เข้าข่ายตามปัจจัยดังกล่าวนั้น

แต่กระนั้น ในสภาพความเป็นจริงของการพัฒนาซอฟต์แวร์ภายใต้ระบบปฏิบัติการปัจจุบัน การไม่ให้ซอฟต์แวร์เข้าข่ายตามปัจจัยที่กล่าวมานั้นเป็นไปไม่ได้ แต่กระนั้น การศึกษาและพัฒนาซอฟต์แวร์โดยคำนึงถึงปัจจัยดังกล่าวยังเป็นเรื่องสำคัญ ดังนี้

- ลดการจัดการในลักษณะที่ต้องสงวนทรัพยากรให้กับโพรเซสเดียว ***(mutual exclusion)*** ลงให้น้อยที่สุด
    - ในกรณีทรัพยากรที่ไม่สามารถใช้ร่วมกันได้ ในลักษณะนี้จำเป็นต้องสงวนทรัพยากรไว้ให้กับโพรเซสที่ต้องการก็ควรจะให้โพรเซสเข้าส่วนวิกฤติและออกจากส่วนวิกฤติให้เร็วที่สุด
    - ในกรณีที่ทรัพยากรที่สามารถใช้ร่วมกันได้ โดยเฉพาะอย่างยิ่งทรัพยากรที่โพรเซสรับข้อมูลจากทรัพยากรนั้นๆ เพียงอย่างเดียว (โพรเซสผู้อ่าน) เช่นจัดการกับไฟล์ที่อ่านได้อย่างเดียว (read-only) ซึ่งสามารถยอมให้โพรเซสมากกว่าหนึ่งตัวเข้าถึงได้พร้อมๆ กัน ทั้งนี้เพราะการเขียนอ่านไม่มีผลต่อการเปลี่ยนแปลงข้อมูลในไฟล์ ในกรณีเช่นนี้ ก็ไม่จำเป็นต้องพัฒนาโปรแกรมให้มี mutual exclusion หรือหากมีโพรเซสผู้เขียนปะปนอยู่ร่วมด้วย ให้หันมาใช้การจัดการแบบ reader-writer แทนการจัดการแบบปกติ เป็นต้น

- ลดโอกาสหรือช่วงเวลาที่โพรเซสหนึ่งกำลังครอบครองทรัพยากรบางส่วน และร้องขอทรัพยากรส่วนที่เหลือ ***(hold and wait)*** ลงให้น้อยที่สุด
    - ปรับปรุงการเขียนโปรแกรม ในกรณีที่ต้องทยอยร้องขอทรัพยากรมาเป็นส่วนๆ แล้วค่อยทำงานในขั้นตอนสุดท้าย โดยพยายามหน่วงการร้องขอทรัพยากรเป็นส่วนๆ ให้ล่าช้าที่สุด (ไม่รีบขอแล้วทิ้งไว้ก่อนเป็นเวลานาน)
    - ถ้าไม่สามารถเลื่อนการทยอยร้องขอทรัพยากรเป็นส่วนๆ ให้เวลาในการร้องขอเหล่านั้นใช้เวลาให้สั้นที่สุดได้ ก็ให้หันมาเป็นลักษณะของการร้องขอทรัพยากรที่ต้องการทั้งหมดเสียก่อนในคราวเดียว แล้วจึงค่อยดำเนินการอื่นๆ ต่อไป
    - กำหนดให้โพรเซสที่จะร้องขอทรัพยากรใดๆ จะสามารถขอทรัพยากรทั้งหมดที่ต้องการได้ในคราวเดียวเท่านั้น โดยถ้ากำลังถือครองทรัพยากรบางส่วนไว้แล้วก่อนหน้านี้ จะต้องปล่อยทรัพยากรที่ถือครองทั้งหมดเสียก่อน แล้วให้ร้องขอทรัพยากรทั้งหมดใหม่อีกครั้ง

  แต่ทั้งนี้ วิธีการต่างๆ ที่กล่าวมา จะส่งผลเสียต่อระบบโดยรวมในสองประการคือ
    - ประสิทธิภาพการทำงานโดยรวมของระบบลดลง เพราะขั้นตอนการร้องขอและการปล่อยทรัพยากรโดยปกติแล้วจะมีกลไกที่เสียเวลาค่อนข้างมาก (เมื่อเทียบกับการทำงานอื่นตามปกติ) การที่โพรเซสต้องร้องขอทรัพยากรจำนวนมากในคราวเดียว หรือต้องปล่อยแล้วร้องขอใหม่ ทำให้ระบบต้องเสียเวลาไปกับขั้นตอนดังกล่าวมาก
    - การที่มีโพรเซสจำนวนมากในระบบ ร้องขอทรัพยากรจำนวนมากๆ ไปครอบครองไว้ จะส่งผลทำให้ทรัพยากรในระบบลดลงหรือหมดลงได้โดยง่าย และจะเกิดโอกาสที่โพรเซสอื่นๆไม่สามารถทำงานต่อไปได้ (starvation) เพราะทรัพยากรในระบบไม่เพียงพอ
- เพิ่มช่องทางการคืนทรัพยากรก่อนเวลากำหนด เพื่อแก้ไขสภาวะการที่โพรเซสไม่ยอมคืนทรัพยากรให้กับระบบ***(no preemtion)***
    - วิธีแก้ไขวิธีแรก เมื่อโพรเซสหนึ่งๆ เริ่มต้นร้องขอทรัพยากรไปแล้วบางส่วนและทำงานไปบ้าง เมื่อพยายามจะร้องขอทรัพยากรเพิ่มอีก แต่ไม่ได้รับ ระบบปฏิบัติการจะดึงคืนทรัพยากรทั้งหมดกลับมาก่อน โดยบันทึกรายการทรัพยากรที่เคยได้ไปแล้วไว้พร้อมกับรายการทรัพยากรที่จะต้องขอเพิ่ม พร้อมกับหยุดการทำงานของโพรเซสนั้นไว้ชั่วคราว โพรเซสดังกล่าวจะเริ่มทำงานต่อไปก็ต่อเมื่อ ระบบปฏิบัติการสามารถมอบทรัพยากรทั้งในส่วนที่โพรเซสได้เคยครอบครองไว้แล้ว พร้อมกับทรัพยากรที่โพรเซสต้องการ
    - วิธีแก้ไขเพิ่มเติมต่อไปอาจทำได้โดยการตรวจสอบก่อนว่า ระบบปฏิบัติการจะมีทรัพยากรพอมอบให้โพรเซสที่กำลังร้องขออยู่ ได้หรือไม่ หากมีไม่ครบ ให้ตรวจสอบโพรเซสอื่นๆ ที่กำลังหยุดรอว่า โพรเซสเหล่านั้นมีทรัพยากรที่โพรเซสนี้ต้องการหรือไม่ ถ้ามีก็ให้ปลดการครอบครองจากโพรเซสที่หยุดรอนั้น มาให้โพรเซสที่กำลังพิจารณา แต่ถ้ายังมีไม่พอ ก็ให้โพรเซสนี้หยุดรอไปด้วย และในทำนองเดียวกัน หากมีโพรเซสอื่นจะใช้ทรัพยากร ก็อาจจะมาขอดึงทรัพยากรจากโพรเซสนี้ไปด้วยได้ โพรเซสแต่ละตัวจะทำงานต่อไป ก็ต่อเมื่อได้รับทรัพยากรที่กำลังจะร้องขอเพิ่มเติม พร้อมกับทรัพยากรที่ที่เคยถือครองแต่ถูกดึงไปใช้ก่อนแล้วเท่านั้น

  สังเกตว่าทรัพยากรที่มีลักษณะการเรียกคืนก่อนกำหนดเหล่านี้ มักจะเป็นทรัพยากรที่เราสามารถเก็บสถานะไว้ที่อื่นแล้วคืนสถานะกลับเมื่อได้ทรัพยากรคืนแล้วเท่านั้น ดังเช่นเรจิสเตอร์ในซีพียู พื้นที่หน่วยความจำหลัก เป็นต้น ทรัพยากรบางอย่างไม่สามารถกระทำได้ในลักษณะเช่นนี้ เพราะจะส่งผลต่อการทำงาน เช่นเครื่องพิมพ์ (ไม่สามารถพิมพ์ไปครึ่งหน้า แล้วให้โพรเซสอื่นพิมพ์ แล้วกลับมาพิมพ์อีกครึ่งหน้าที่เหลือได้) เป็นต้น
- ป้องกันการร้องขอทรัพยากรในลักษณะที่ทำให้เกิดการ***รอเป็นวงกลม*** ***(circular wait)*** ซึ่งกระทำได้โดยการ**กำหนดค่าหมายเลขลำดับ (enumeration)** ของทรัพยากรทุกตัวไว้เป็นค่าเฉพาะแต่ละทรัพยากร และกำหนดให้ลำดับการเข้าร้องขอทรัพยากรของโพรเซสทุกตัวนั้น ต้องกระทำตามค่าลำดับเท่านั้น (เช่นต้องร้องขอจากทรัพยากรที่มีเลขค่าลำดับต่ำไป

หาสูงเท่านั้น) โดยการนี้ การถือครองทรัพยากรจะมีลำดับขั้นตอนที่จะไม่ส่งผลให้โพรเซสไปครอบครองทรัพยากรบางอย่างก่อนแล้วค่อยมาร้องขอที่เหลือในลักษณะวนสลับกัน ยกตัวอย่างในโค้ดโปรแกรมในหน้าแรก จะเห็นว่า ถ้าทั้งสองโพรเซสล็อก mutex1 ก่อน mutex2 ทั้งสองโพรเซสจะไม่เกิดสภาวะติดตายเกิดขึ้น เพราะจะมีโพรเซสหนึ่งได้ทรัพยากรที่ทั้งสองโพรเซสต้องการไปก่อน และอีกโพรเซสต้องรอการร้องขอในลักษณะตามลำดับที่กำหนดเท่านั้น

7.3.2 การหลีกเลี่ยงจากสภาวะติดตาย

จะเห็นได้ว่า เราไม่สามารถสร้างกลไกการป้องกันไม่ให้เกิดสภาวะติดตายได้อย่างสมบูรณ์ ดังนั้น เราอาจหันมาออกแบบระบบให้สามารถตรวจสอบสภาวะที่น่าจะปลอดภัยจากการเกิดสภาวะติดตาย และหากระบบไม่สามารถอยู่ในสภาวะดังกล่าวได้ เราก็จะสั่งการให้โพรเซสบางตัวหยุดรอไปก่อน เพื่อไม่ให้ระบบโดยรวมเกิดโอกาสที่จะเข้าสู่สภาวะติดตายได้

เราอาจจะออกแบบระบบให้สามารถหลีกเลี่ยงจากสภาวะที่จะนำไปสู่สภาวะติดตายได้ดังเช่น

- ออกแบบให้โพรเซสต้องแจ้งชนิดและจำนวนทรัพยากรสูงสุดของแต่ละชนิดที่จะต้องใช้ ก่อนที่จะร้องขอทรัพยากร เพื่อระบบปฏิบัติการจะได้พิจารณาเสียแต่เนิ่นๆ ว่าจะยอมให้โพรเซสดังกล่าวทำงานต่อไปหรือไม่
- ออกแบบขั้นตอนวิธีที่คอยตรวจสอบกลไกการร้องขอและมอบให้ทรัพยากร เพื่อป้องกันไม่ให้เกิดการรอทรัพยากรกันเป็นวงกลม (circular wait)

สถานะปลอดภัย (Safe state)

สถานะปลอดภัย (Safe state) คือสถานะที่หากในขณะนั้นมีโพรเซสอยู่กลุ่มหนึ่งที่ทำงานในระบบ (และอาจพบว่ามีบางโพรเซสหยุดรอ) ถ้าเราสามารถจัดการให้โพรเซสทั้งหมดนั้น ทยอยทำงานผ่านส่วนวิกฤติไปทีละตัวจนกระทั่งสุดท้ายแล้ว สามารถคืนทรัพยากรให้กับระบบได้ (นั่นหมายความว่าในขณะนั้น โพรเซสอื่นๆ อาจจะหยุดรอทั้งหมด) และเราสามารถทยอยทำโพรเซส/เธรดไปทีละตัว จนกระทั่งทุกตัวสามารถคืนทรัพยากรที่ถือครอง ณ ขณะนั้นลงได้ เราจะพิจารณาว่า ณ ขณะนั้น ระบบอยู่ในสถานะปลอดภัย

คำนิยาม

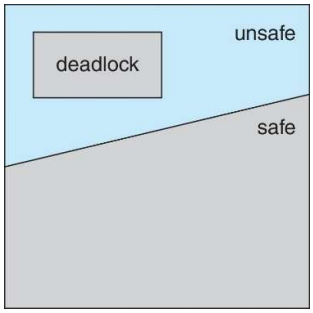
ถ้าเรามีโพรเซสในระบบทั้งหมดตามเซ็ตต่อไปนี้คือ $<P_1, P_2, P_3, \dots, P_N>$ และเราสามารถทดลองรันโพรเซสไปทีละตัว(จะมีกลไกคร่าวๆ ในการพิจารณาลำดับว่าใครทำก่อนหลัง) และเราได้ลำดับของโพรเซสที่ทำงานจนคืนทรัพยากรได้ตามลำดับเซ็ตอีกตัวหนึ่ง ซึ่งเซ็ตผลลัพธ์นี้จะมีรายชื่อโพรเซสทั้งหมดในเซ็ตแรก จะหมายความว่า ขณะนั้น ระบบอยู่ในสภานะปลอดภัย

ขั้นตอนวิธีการตรวจสอบสถานะปลอดภัย (safe state)

- ระบบปฏิบัติการพิจารณาโพรเซสที่กำลังทำงานไปตามลำดับ โดยดูว่าแต่ละโพรเซสจะต้องการทรัพยากรมากน้อยขนาดไหนที่จะยังคงให้ระบบอยู่ในสถานะปลอดภัยอยู่ตลอดเวลา
- ระบบจะอยู่ใน**สถานะปลอดภัย**ก็ต่อเมื่อ
    - สมมติมีโพรเซสทั้งหมด n ตัว ทำงานไล่เรียงกันไปตามลำดับของการร้องขอทรัพยากรดังนี้ $<P_1,P_2,\dots,P_n>$
    - พิจารณาที่โพรเซสแต่ละตัว Pi ว่าต้องการทรัพยากร**อีก**เท่าไหร่จึงจะทำงานต่อไปได้ โดยให้พิจารณาจากจำนวนทรัพยากรที่ระบบมีให้อยู่ ณ ขณะนั้น รวมกับทรัพยากรที่ถือครองโดยโพรเซสเองที่ได้ไว้ก่อนหน้า
    - หากเป็นสภาพปกติ Pi จะสามารถทำงานต่อไปได้จนกระทั่งถึงจุดที่คืนทรัพยากรที่ครอบครอง ณ ขณะนั้นกลับคืนให้แก่ระบบ

- แล้วเราหาโพรเซสถัดไป Pj ที่มีลักษณะเช่นเดียวกันกับ Pi (รวมทรัพยากรกลางของระบบที่มีอยู่ กับทรัพยากรที่โพรเซส Pj ถือครองอยู่) แล้วคัดโพรเซสนั้นขึ้นมาทำงานต่อ (หากมีหลายโพรเซสที่เข้าข่าย ก็เลือกตัวใดก็ได้ แต่ถ้าหนทางแรกไม่ผ่าน ก็อาจจะย้อนขั้นตอนวิธีกลับมาเลือกหนทางอื่นที่มี)
- ขั้นตอนวิธีจะจบลงเมื่อทุกโพรเซสได้โอกาสรับมอบทรัพยากรที่เหลือ ทำงานไปจนจบ แล้วคืนทรัพยากรทั้งหมดให้ระบบ ลักษณะเช่นนี้จะถือว่าระบบอยู่ในสภานะที่ปลอดภัย

- ระบบจะอยู่ใน**<u>สถานะไม่ปลอดภัย</u>**ก็ต่อเมื่อ
  - ไม่สามารถจัดลำดับการร้องขอทรัพยากรของโพรเซส $<P_1,P_2,...,P_n>$ ที่สามารถให้สถานะเป็นไปตามขั้นต้น (ไม่สามารถจัดเรียงโพรเซสให้ร้องขอทรัพยากรไปตามลำดับในลักษณะที่ประเมินได้ว่าปลอดภัย) กล่าวคือ ไม่สามารถหา Pj ที่มีความต้องการทรัพยากรน้อยกว่าที่ระบบส่วนกลางถือครองอยู่
  - ใน**<u>สถานะที่ไม่ปลอดภัย (Unsafe)</u>** หมายความว่าระบบอาจจะไปสู่สภาวะติดตายได้ การหลีกเลี่ยงสภาวะติดตาย จึงทำได้โดยให้แน่ใจว่าลำดับการร้องขอทรัพยากรและการทำงานของโพรเซสต่างๆ อยู่ในสถานะปลอดภัยตลอดเวลา
- ในทางปฏิบัติ ขั้นตอนเหล่านี้จะไม่ได้ดำเนินการจริงๆ แต่จะเป็นการทดลองการมอบและคืนทรัพยากร โดยพิจารณาจากคำสั่งที่ปรากฏอยู่ในแต่ละโพรเซส



หากระบบอยู่ในสภานะไม่ปลอดภัย ไม่ได้หมายความว่าระบบจะเกิดสภาวะติดตายเสมอไป แต่หากระบบอยู่ในสภาวะปลอดภัย จะเป็นการรับประกันว่าจะไม่เกิดสภาวะติดตายขึ้น

ตัวอย่างเช่น สมมติว่ามีพื้นที่หน่วยความจำที่โพรเซสต้องการทั้งหมด12 บล็อก มีโพรเซสกำลังทำงานอยู่สามโพรเซส ซึ่งมีรายการพื้นที่ที่ต้องการสูงสุด และพื้นที่ที่กำลังใช้งานอยู่ดังนี้

	จำนวนบล็อกสูงสุดที่ต้องการ	จำนวนที่มีอยู่
P1	10	5
P2	4	2
P3	9	2
เหลือพื้นที่ว่าง		3

หากเรากำหนดลำดับการร้องขอหน่วยความจำเป็น <P2,P1,P3> เราจะพบว่าโพรเซสสามารถทำงานต่อไปได้ เพราะ P2 จะได้ทรัพยากรไปอีก 2 หน่วย (เหลือในระบบอีก 1 หน่วย) และเมื่อ P2 ทำงานเสร็จ P1 ที่ต้องหยุดรอเพราะจำนวนทรัพยากรไม่พอ ณ ขณะนั้น จะได้ทรัพยากรทั้งหมดที่ P2 คืนมาคือ 4 หน่วย รวมกับที่ยังว่างอยู่ 1 หน่วย นำไปใช้งานต่อได้ และเมื่อ P1 ทำงานเสร็จ P3 ที่รอทรัพยากรอยู่ ก็จะมีทรัพยากรที่ระบบพร้อมจะให้ได้ 10 หน่วย โดย P3 จะนำไปใช้เพียง 7 หน่วย

หากสมมติว่า P3 เกิดร้องขอทรัพยากรไป 1 หน่วย กลายมาเป็นจำนวนที่มีอยู่ 3 หน่วย ในลักษณะเช่นนี้เมื่อเราพิจารณาตารางอีกครั้งหนึ่งดังนี้

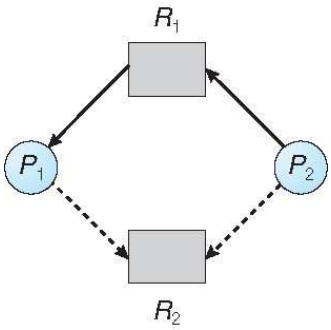
	จำนวนบล็อกสูงสุดที่ต้องการ	จำนวนที่มีอยู่
P1	10	5
P2	4	2
P3	9	3
เหลือพื้นที่ว่าง		2

เราจะพบว่า P2 นั้นสามารถทำงานต่อไปได้ เพราะยังมีทรัพยากรเหลือพอจะทำงาน แต่เมื่อ P2 ทำงานเสร็จ ทรัพยากรที่เหลือในระบบจะมีเพียง 4 หน่วย ซึ่งไม่พอต่อการทำงานต่อไปทั้ง P1 หรือ P3 ในสภาวะเช่นนี้เราจะพบได้ทันทีว่า เกิดสภาวะติดตายขึ้น

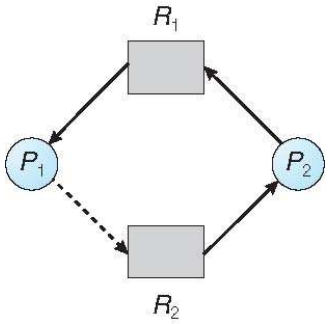
ดังนั้น ขั้นตอนวิธีที่จะนำมาใช้เพื่อการป้องกันสภาวะติดตาย จะต้องให้แน่ใจว่า ในการร้องขอทรัพยากรเพิ่มจากระบบแต่ละครั้ง ระบบจะให้ทรัพยากรเพิ่มได้ก็ต่อเมื่อจะไม่ทำให้ระบบโดยรวมเข้าสู่สภาวะติดตาย (เช่น การจะร้องขอทรัพยากรเพิ่มอีกหนึ่งหน่วยของ P3 ในลักษณะข้างต้นจะกระทำไม่ได้ P3 จะต้องหยุดรอไม่ได้รับทรัพยากรเพิ่มตามที่ขอดังกล่าว)

ขั้นตอนวิธีสำหรับทรัพยากรที่มีเพียงหนึ่งหน่วยต่อชนิด : การใช้กราฟนำเสนอการมอบทรัพยากร

- สร้างกราฟนำเสนอการมอบทรัพยากร (resource-allocation graph) เพื่อจำลองสถานะของโพรเซสทุกตัวในระบบ ณ ขณะนั้น
- เพิ่ม**เส้นทางการร้องขอในอนาคต** (**claim edge** -เส้นทางที่แสดงว่าจะมีการขอทรัพยากรนั้นๆ เพิ่มในอนาคต) ด้วยเส้นประโดยลากจากโพรเซสไปยังทรัพยากร P_i->R_j
- เส้นการร้องขอในอนาคต จะเปลี่ยนเป็นเส้นทึบเมื่อมีการร้องขอเกิดขึ้นจริง
- **เส้นการร้องขอ (request edge)** จะเปลี่ยนเป็น**เส้นการครอบครอง (assignment edge)** เมื่อทรัพยากรนั้นได้รับอนุญาตให้โพรเซสครอบครองได้
- เมื่อโพรเซสปล่อยการถือครองทรัพยากร เส้นการครอบครอง(assignment edge) จะเปลี่ยนกลับเป็นเส้นการทางร้องขอในอนาคต
- ก่อนที่โพรเซสต่างๆ จะเริ่มต้นทำงาน จะต้องมีขั้นตอนให้โพรเซสที่จะทำงานนั้น ต้องแจ้งการร้องขอทั้งหมดที่จะเกิดขึ้นในอนาคตให้กับระบบเสียก่อน (claim a priori) เพื่อที่ระบบจะได้สามารถสร้างกราฟเพื่อตรวจสอบเส้นทางหรือความต้องการของโพรเซสที่จะครอบครองทรัพยากรต่างๆ ในขณะที่โพรเซสกำลังทำงานได้
- หลักการตรวจสอบการครอบครองเพื่อที่จะไม่ทำให้เกิดสภาวะติดตายก็คือ การที่ระบบปฏิบัติการจะ**<u>จำลอง</u>**การมอบทรัพยากรให้โพรเซสใดๆ เข้าครอบครองจะต้องไม่ทำให้เกิดเส้นทางเป็นวงกลมเกิดขึ้นได้ (circular edge) หากการร้องขอนั้นจะทำให้เกิดเส้นทางวงกลม โพรเซสนั้นจะต้องรอไปก่อน)



จากรูป สมมติการเริ่มต้นสภาวะที่ P1 และ P2 จะร้องขอทรัพยากร R2 ในอนาคต โดยที่ในขณะนั้น P1 กำลังครอบครอง R1 อยู่และ P2 อยู่ในสถานะการรอการครอบครอง R1



จะเห็นว่า ในลักษณะเช่นนี้ P2 จะไม่สามารถร้องขอทรัพยากร R2 ได้ เพราะถ้า P2 ได้รับทรัพยากร R2 มา จะทำให้เกิดเส้นทางเป็นวงกลมเกิดขึ้น **ซึ่งในกรณีที่ระบบมีทรัพยากรเพียงหนึ่งหน่วยต่อชนิดนั้น จะเสี่ยงต่อการเกิดสภาวะติดตาย** เพราะต่อจากนี้เมื่อใดก็ตามที่ P1 การร้องขอทรัพยากร R2 (โดยที่ P1 ยังไม่คืนทรัพยากร R1 ให้กับระบบเสียก่อน) จะเกิดสถาวะติดตายขึ้นทันที ดังนั้นในกรณีเช่นนี้ P2 จะไม่ได้รับอนุมัติให้ครอบครอง R2 โดยจะสั่งหยุดการทำงาน P2 ไว้ก่อน เว้นเสียแต่เมื่อ P1 ปลดการครอบครอง R1 ไปก่อนแล้วเท่านั้น (ดังนั้น P2 จะต้องรอจนกว่า P1 จะร้องขอทรัพยากร R2 และทำงานจนเสร็จสิ้นแล้วปล่อยทั้ง R1 และ R2 ระบบก็จะปลุกการทำงานของ P2 ขึ้นแล้วจึงมอบ R2 ให้ต่อไป)

ขั้นตอนวิธีสำหรับทรัพยากรที่มีมากกว่าหนึ่งหน่วยต่อชนิด:ขั้นตอนวิธีของนักการธนาคาร(Banker's algorithm)

ขั้นตอนวิธีแบบใช้กราฟในข้างต้น แม้ว่าจะสามารถจัดการหลีกเลี่ยงสภาวะการณ์ที่จะนำไปสู่สภาวะติดตายได้ แต่ในทางปฏิบัติพบว่า ทรัพยากรแต่ละชนิดในคอมพิวเตอร์อาจมีได้มากกว่าหนึ่งหน่วย (เช่นบล็อกหน่วยความจำ เวลาที่ระบบให้โพรเซสใช้งานซีพียู เป็นต้น) ในลักษณะนี้เราจึงต้องหันมาใช้ขั้นตอนวิธีอื่น แม้ว่าจะมีประสิทธิภาพไม่เทียบเท่า อย่างขั้นตอนวิธีต่อไปนี้ ถูกตั้งชื่อว่า **ขั้นตอนวิธีของนักการธนาคาร(banker's algorithm)** ด้วยเหตุที่ขั้นตอนวิธีนี้สามารถนำไปใช้ในการจัดสรรเงินสดหมุนเวียนที่ธนาคารสาขาถือครองอยู่ ณ ขณะนั้นๆ ให้กับลูกค้าเมื่อลูกค้าต้องการถอนเงินสดที่ธนาคาร ซึ่งตามความเป็นจริงแล้วธนาคารสาขาจะไม่สำรองเงินสดหมุนเวียนไว้ครบตามจำนวนเงินรวมของทุกๆ บัญชีลูกค้า ดังนั้น ถ้ามีใครที่ขอเบิกเงินสดจำนวนมากกว่าที่สาขามีอยู่ ธนาคารจะให้ผลัดรอไปก่อน จนกว่าจะได้เงินสดเพิ่มจากผู้ที่ฝากเงินในวันนั้นๆ หรือไปเบิกเงินสดเพิ่มจากส่วนกลางมาให้ในวันหลัง

กลไกการทำงานเริ่มจากเมื่อโพรเซสเริ่มต้นการทำงาน โพรเซสจะต้องแจ้งแก่ระบบปฏิบัติการว่า ตนต้องการทรัพยากรชนิดใดบ้าง และแต่ละชนิด**ต้องการสูงสุด**กี่หน่วย โดยจำนวนที่แจ้งจะต้องไม่เกินไปกว่าที่ระบบจะให้ได้ (หากเกินกว่า โพรเซสนั้นจะไม่ได้รับอนุญาตให้เริ่มทำงาน) ต่อจากนั้น เมื่อโพรเซสเริ่มร้องขอทรัพยากรบางส่วนจากที่ต้องการทั้งหมด ระบบปฏิบัติการจะตรวจสอบก่อนว่า ถ้าเกิดให้ทรัพยากรไปแล้ว หลังจากนั้น ระบบจะยังคง**อยู่ในสถานะปลอดภัย (safe state)หรือไม่** ถ้าใช่ระบบปฏิบัติการจึงจะมอบทรัพยากรให้ตามที่ร้องขอไว้จริง แต่ถ้าไม่ ระบบปฏิบัติการจะสั่งหยุดการทำงานของโพรเซสนั้นไว้ชั่วคราว จนกว่าจะมีโพรเซสอื่นปล่อยทรัพยากรออกมา ระบบปฏิบัติการจึงจะกลับมาพิจารณาใหม่อีกครั้งว่าจะมอบทรัพยากรที่ต้องการแก่โพรเซสที่กำลังหยุดรอหรือไม่

ขั้นตอนวิธีจะจัดเก็บข้อมูลที่เกี่ยวข้องกับการจัดการดังนี้

- **Available**: คือค่าเว็กเตอร์(vector หรือชุดลำดับข้อมูล) ของจำนวนทรัพยากรที่มีต่อชนิดทรัพยากรนั้นๆ ที่มีอยู่พร้อมใช้ในระบบ เช่น avaliable[j]=k หมายความว่าทรัพยากร j (Rj) ณ ขณะนั้นมีจำนวนหน่วย k หน่วยที่ระบบจะสามารถมอบให้กับโพรเซสใดๆ ได้ ณ เวลานั้น
- **Max:** เป็นเมตริกซ์ขนาด n คูณ m โดย **n คือจำนวนโพรเซสที่รันอยู่ ณ เวลานั้นๆ และ m คือจำนวนของทรัพยากรนับตามชนิดของทรัพยากรที่มีอยู่ในระบบ** Max[i,j] = k เป็นสมาชิกในเมตริกซ์ที่โพรเซส i ต้องแจ้งล่วงหน้าว่าจะต้องการทรัพยากรชนิด j สูงสุดกี่หน่วย (เช่นเท่ากับ k หน่วย) ในช่วงเวลาการทำงานของโพรเซสนั้นๆ
- **Allocation**: เป็นเมตริกซ์ขนาด n คูณ m Allocation[i,j]=k เป็นค่าของสมาชิกในเมตริกซ์ที่บอกว่าโพรเซส i ณ ขณะนั้นกำลังครอบครองทรัพยากรชนิด j อยู่จำนวน k หน่วย
- **Need**: เป็นเมตริกซ์ขนาด n คูณ m Need[i,j]=k เป็นค่าของสมาชิกในเมตริก์ที่บอกว่า โพรเซส i จะต้องการทรัพยากรชนิด j อีกจำนวน k หน่วยจึงจะสามารถทำงานเสร็จสิ้นได้ ดังนั้น

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

ดังนั้น เมื่อพิจารณาเมตริกซ์ Max, Allocation, และ Need ในแนวนอน (row) เราจะเห็นถึงค่าความต้องการจำนวนทรัพยากรสูงสุด ค่าจำนวนทรัพยากรที่กำลังถือครองอยู่ และค่าจำนวนทรัพยากรที่จะต้องการเพิ่มเติม ของแต่ละโพรเซส เราอาจจะพิจารณาแยกเป็นรายแถวแนวนอนว่าเป็น $Need_i$ Max_i และ $Allocation_i$ คือค่าประจำของแต่ละโพรเซส โดยแต่ละสมาชิกของข้อมูลใน เป็น $Need_i$ Max_i และ $Allocation_i$ ก็จะนำเสนอตามลำดับทรัพยากร j นั่นเอง

เพื่อความสะดวกในการพิจารณาต่อไปของขั้นตอนวิธี เราจะนิยามเวกเตอร์ (ค่าลำดับข้อมูล) ของ X และ Y ว่า

$$Y \leq X$$

ถ้าสมาชิกในแต่ละลำดับของ Y ต่างมีค่าน้อยกว่าหรือเท่ากับค่าในสมาชิกแต่ละลำดับที่ตรงกันของ X

ตัวอย่างเช่น สมมติว่า **Y = (0,3,2,1)** และ **X=(1,7,3,2)** จะถือว่า **Y<X** (สมาชิกแต่ละลำดับต้องน้อยกว่าในลำดับเดียวกันของเว็กเตอร์อีกตัวเท่านั้น)

และ **Y=X** ก็ต่อเมื่อค่าในเวกเตอร์ทั้งสองมีค่าตรงกันทุกสมาชิก

และ **Y>X** เมื่อไม่เข้าข่ายในกรณีต่างๆ ข้างต้น

ขั้นตอนวิธีตรวจสอบว่าระบบจะอยู่ในสถานะปลอดภัยหรือไม่ (Safety algorithm)

1. กำหนดเว็กเตอร์ Work (ขนาด m ทรัพยากร) และ Finish (ขนาด n โพรเซส) เพื่อใช้ทดลองมอบทรัพยากรเป็นการชั่วคราว
   Work = Avaliable
   Finish[i] = false for i=0,1,...,n-1 (กำหนดค่าสมาชิกทุกตัวในเว็กเตอร์ Finish ให้เป็น false)
2. ค้นหาโพรเซสที่ i ซึ่ง
   a. **Finish[i] = false** และ
   b. $\mathbf{Need_i \leq Work}$ (รายการความต้องการทรัพยากรชนิดต่างๆ ของโพรเซส i น้อยกว่ารายการทรัพยากรที่ระบบจำลองว่ากำลังมีอยู่ ณ ขณะนั้น)

   ถ้าไม่พบโพรเซสใดๆ ที่เข้าข่ายทั้งสองกรณีย่อย ให้กระโดดไปข้อ 4
3. $\mathbf{Work = Work + Allocation_i}$ (สมมติให้โพรเซส i ได้รับอนุญาตให้ทำงานต่อไปโดยได้ทรัพยากรเพิ่มเติมตามที่ร้องขอ และเมื่อโพรเซสทำงานเสร็จสิ้นแล้ว ก็จะคืนทรัพยากรที่ได้ครอบครองทั้งหมดให้แก่ระบบ)
   Finish[i] = true (สมมติว่า โพรเซส i จบการทำงานหลังจากที่ได้คืนการครอบครองทรัพยากรแล้ว)
   กระโดดไปยังข้อ 2
4. ถ้า **Finish[i] == true** **ในทุกๆ i ที่มีอยู่** = ถ้าทุกโพรเซสสามารถจำลองการทำงานและจำลองว่าจบการทำงานได้สำเร็จ แสดงว่าระบบอยู่ใน**สภาวะปลอดภัย (safe state)**

ในการจำลองสถานการณ์เพื่อดูว่าระบบจะอยู่ในสภานะปลอดภัยหรือไม่นั้น อาจจะต้องทดลองสลับลำดับก่อนหลังการพิจารณาโพรเซส หากพบเส้นทางที่ตัน กล่าวคือ ไม่สามารถมอบทรัพยากรให้ต่อไปได้ ก็ให้ย้อนใน ณ จุดที่มีทางเลือกในการมอบทรัพยากรให้มากว่าหนึ่งหนทาง ดังนั้นการคำนวณอาจจะต้องใช้จำนวนรอบทดลองถึง $m \times n^2$ รอบ

ขั้นตอนวิธีการร้องขอทรัพยากรสำหรับโพรเซสต่างๆ

ขั้นตอนวิธีนี้ (ตามข้อ 2. ข้างต้น) ใช้ในการตรวจสอบว่าโพรเซสจะสามารถได้รับทรัพยากรไปตามที่ร้องขอหรือไม่ โดยสมมติให้ $\mathbf{Request_i}$ คือเวกเตอร์แสดงการร้องขอทรัพยากรของโพรเซส i โดยแต่ละหน่วยข้อมูลในเวกเตอร์ คือจำนวนทรัพยากรต่อชนิดทรัพยากรที่ต้องการร้องขอเพิ่มเติมในคราวหนึ่งๆ

1. ถ้า $\mathbf{Request_i \leq Need_i}$ กระโดดไปยังข้อ 2 มิฉะนั้นให้แจ้งความผิดพลาด เนื่องจากโพรเซสพยายามร้องขอทรัพยากรมากกว่าที่ได้เคยแจ้งไว้แล้วแต่แรก
2. ถ้า $\mathbf{Request_i \leq Available_i}$ ให้กระโดดไปยังข้อ 3 หมายความว่าโพรเซสนั้นๆ จะต้องรอไปก่อน เนื่องจากจำนวนทรัพยากรที่ระบบมีเหลืออยู่ไม่พอที่จะมอบให้แก่โพรเซสตามคำร้องขอ
3. สมมติกระบวนการมอบทรัพยากรให้กับโพรเซส โดยทดลองเปลี่ยนค่าต่างๆ ดังนี้

   $\mathbf{Available = Available - Request;}$ *(ลองมอบให้ตามที่ร้องขอ)*

   $\mathbf{Allocation_i = Allocation_i + Request_i;}$ *(ปรับทรัพยากรที่ระบบเหลืออยู่หลังการมอบไปแล้ว)*

   $\mathbf{Need_i = Need_i - Request_i;}$ *(เมื่อโพรเซสดังกล่าวทำงานเสร็จ แล้วคืนทรัพยากรที่ถือครองออกมาทั้งหมด เราจะถือว่าโพรเซสนี้ผ่าน และปรับค่าทรัพยากรที่ระบบได้เพิ่มเข้ามาจากที่คืนทั้งหมด)*

   - พิจารณาว่าในสภาพใหม่นี้ ระบบจะยังคงสถานะปลอดภัยหรือไม่ ถ้าใช่ ก็ให้มอบทรัพยากรตามที่ร้องขอต่อไป (หา i ตัวใหม่ แล้วทำตาม 1-3) แล้วเซ็ตแฟล็ก $\mathbf{finish[i] = true}$
   - ถ้าตรวจสอบแล้วพบว่า การมอบทรัพยากรให้โพรเซส ส่งผลอาจทำให้ระบบเข้าสู่สภาวะไม่ปลอดภัย ก็ให้โพรเซสหยุดรอไปก่อน และคืนสภาพก่อนการร้องขอเป็นดังเดิม (ยกเลิกที่ทดลองการสมมติการเปลี่ยนค่า) แล้วไปทดลองกับโพรเซสอื่นที่แฟล็ก $\mathbf{finish[i]}$ ยังเป็น $\mathbf{false}$ อยู่

ตัวอย่างการจัดสรรทรัพยากรตามขั้นตอนวิธีของนักการธนาคาร

สมมติว่ามีโพรเซสทั้งหมด 5 ตัว และมีทรัพยากรสามชนิด ดังต่อไปนี้

Snapshot at time *T*0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
*P*0	0 1 0	7 5 3	3 3 2
*P*1	2 0 0	3 2 2	
*P*2	3 0 2	9 0 2	
*P*3	2 1 1	2 2 2	
*P*4	0 0 2	4 3 3	

คำนวณหาเมตริกซ์ความต้องการได้ดังนี้

	Need
	A B C
*P*0	7 4 3
*P*1	1 2 2
*P*2	6 0 0
*P*3	0 1 1
*P*4	4 3 1

จากสภาวะดังกล่าว เราจะพบว่าระบบยังคงอยู่ในสภาวะปลอดภัย หากเรามอบทรัพยากรให้โพรเซสตามลำดับคือ

< *P*1, *P*3, *P*4, *P*2, *P*0>

สมมติต่อไปว่า คราวนี้ โพรเซส P1 ร้องขอทรัพยากรคือ (1,0,2) (ขอ A เพิ่มหนึ่งหน่วยและ C เพิ่ม 2 หน่วย)

1. ตรวจสอบทรัพยากรว่าร้องขอเกินไปกว่าที่โพรเซสแจ้งไว้แต่แรกหรือไม่ และตรวจสอบว่าทรัพยากรที่ร้องขอมีพอที่ระบบจะให้ได้ในคราวเดียวหรือไม่ คำตอบคือจาก Need พบว่าไม่มากกว่า และจาก Available พบว่ามีเหลือเพียงพอ ดังนั้นจึงทดลองการมอบทรัพยากรให้ P1 โครงสร้างของเมตริกซ์ต่างๆ จะเปลี่ยนไปเป็นดังนี้

	Allocation	Need	Available
	A B C	A B C	A B C
P0	0 1 0	7 4 3	2 3 0
P1	3 0 2	0 2 0	
P2	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4	0 0 2	4 3 1	

2. ตรวจสอบว่าระบบอยู่ในสภาวะปลอดภัยหรือไม่ ทดลองตามลำดับการมอบทรัพยากรให้โพรเซสในลำดับ < *P*1, *P*3, *P*4, *P*0, *P*2> พบว่า ปลอดภัย ดังนั้น ระบบสามารถมอบทรัพยากรให้ P1 ตามการร้องขอได้

ให้นักศึกษาทดลองทำต่อไปนี้

1) หากโพรเซส P4 ร้องขอทรัพยากร (3,3,0) จะให้ได้หรือไม่

2) หากโพรเซส P0 ร้องขอทรัพยากร (0,2,0) จะให้ได้หรือไม่

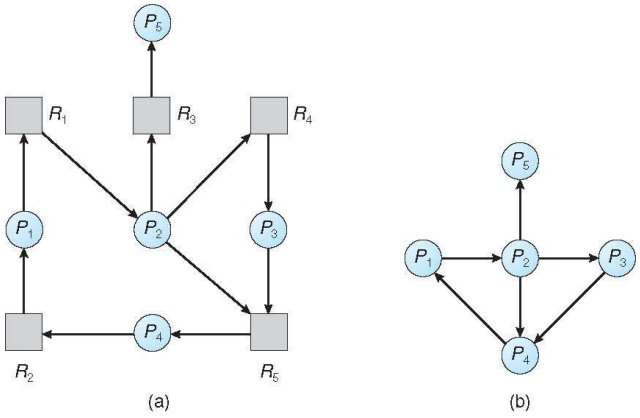
7.3.4 การตรวจสอบว่าระบบอยู่ในสภาวะติดตายหรือไม่

ในกรณีที่ระบบมิได้มีกลไกในการจัดการป้องกันหรือหลีกเลี่ยงจากสภาวะติดตาย สภาวะติดตายย่อมสามารถเกิดขึ้นได้ ดังนั้นระบบปฏิบัติการจะต้องมีกลไกเพิ่มเติมเพื่อจัดการต่อสภาวะติดตายดังต่อไปนี้

1. ตรวจสอบว่า ณ ปัจจุบัน ระบบอาจจะกำลังอยู่ในสภาวะติดตายหรือไม่
2. ดำเนินตามขั้นตอนวิธีกู้สภาพจากสภาวะติดตาย

การตรวจสอบสำหรับทรัพยากรที่มีเพียงหนึ่งเดียวต่อชนิด : การใช้กราฟหยุดรอ (wait-for graph)

ในลักษณะเช่นนี้ ระบบปฏิบัติการจะต้องมีโพรเซสพิเศษซึ่งจะตื่นขึ้นมาทำงานเป็นระยะๆ โดยจะสร้างกราฟการครอบครองทรัพยากรของโพรเซสต่างๆในระบบ (resource-allocation graph) จากนั้น จึงสร้าง**กราฟแสดงการหยุดรอ** (wait-for graph)ของโพรเซส โดยพิจารณาว่าโพรเซสใดกำลังหยุดรอทรัพยากรที่โพรเซสใดกำลังถือครองอยู่ ก็ให้สร้างเส้นทาง (edge) ที่เชื่อมจากโพรเซสที่หยุดรอ ไปทรัพยากร แล้วต่อไปยังโพรเซสที่กำลังถือครอง โดยเส้นทางใหม่นี้จะเป็นการลัดช่วงจากโพรเซสที่หยุดรอไปยังโพรเซสที่กำลังถือครองทรัพยากรนั้นๆ อยู่ ดังรูป (b)



เมื่อได้กราฟแสดงการหยุดรอแล้ว ก็ให้ดูว่าในกราฟดังกล่าวมีเส้นทางเป็นวงกลมหรือไม่ ถ้ามี แสดงว่า ขณะนั้นเกิดสภาวะติดตายเกิดขึ้น

การตรวจสอบสำหรับทรัพยากรที่มีได้มากกว่าหนึ่งหน่วยต่อชนิด : การใช้เมตริกซ์การครอบครองทรัพยากร

ในกรณีนี้ เราจะยืมกลไกต่างๆ มาจากขั้นตอนวิธีของนักการธนาคาร โดยมีเวกเตอร์ Avaiable เมตริกซ์ Allocation และ Request ที่ระบบปฏิบัติการต้องจัดเก็บไว้เพื่อจัดการในขั้นตอนวิธีของนักการธนาคารนั่นเอง โดยมีขั้นตอนวิธีตรวจสอบสภาวะติดตายดังนี้ (ซึ่งก็คือขั้นตอนวิธีการหาสภาวะปลอดภัย Safe state นั่นเอง)

1. กำหนดสภาพเริ่มต้นให้กับเว็กเตอร์ Work และ Finish ดังนี้

```
a. Work = Available
b. For i = 1,2,...,n
        if(Allocation_i != 0)
              Finish[i] = false
        else
              Finish[i] = true
```

2. ค้นหาโพรเซส i ที่มีลักษณะ

   a. `Finish[i] == false` และ

   b. $Request_i \leq Work$ ระบบจำลองพบว่า สามารถให้ทรัพยากรชนิดต่างและจำนวนตามที่โพรเซส i ร้องขอ

   ถ้าไม่พบโพรเซส i ที่ครบตามที่ต้องการแล้ว ให้กระโดดไปยัง 4

3. $Work = Work + Allocation_i$ สมมติว่าโพรเซสสามารถรับทรัพยากรมาแล้วทำงานไปจนสำเร็จ และคืนทรัพยากรให้ระบบได้

   `Finish[i] = true`

   กระโดดไปยังข้อ 2

4. **if Finish[i] == false for some i, $1 \leq i \leq n$** ถ้ายังพบว่ามีบางโพรเซสไม่สามารถทำงานให้จบได้ แสดงว่าระบบอาจตกอยู่ในสภาวะติดตาย โดยโพรเซสใดที่ค้างในสภาวะ false แสดงว่าโพรเซสเหล่านั้นอาจกำลังอยู่ในสภาวะติดตาย

การทดลองการทำงาน จำเป็นต้องทดลองสลับลำดับการร้องขอทรัพยากรของโพรเซสต่างๆ ในระบบไปมา ในกรณีที่มีทางเลือกโพรเซสมากกว่าหนึ่งตัวในแต่ละขั้นตอน หากเส้นทางแรกล้มเหลว ก็ให้ทดลองต่อในเส้นทางที่เหลือ ดังนั้นอาจจะต้องวนซ้ำในจำนวนมากถึง $m \times n^2$ ครั้ง

ตัวอย่างการทำงานของขั้นตอนวิธีทดสอบสภาวะติดตาย

สมมติว่ามีโพรเซสทั้งหมด 5 ตัว และมีทรัพยากรทั้งหมดในระบบคือ A 7 หน่วย B 2 หน่วย และ C 6 หน่วย และในขณะนั้นมีโพรเซส P1, P3 และ P4 กำลังร้องขอทรัพยากรเพิ่มพร้อมๆ กัน

Snapshot at time *T*0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
*P*0	0 1 0	0 0 0	0 0 0
*P*1	2 0 0	2 0 2	
*P*2	3 0 3	0 0 0	
*P*3	2 1 1	1 0 0	
*P*4	0 0 2	0 0 2	

หากทดลองลำดับการร้องขอคือ **<*P*0, *P*2, *P*3, *P*1, *P*4>** จะพบว่าสามารถจบการทำงานได้

คราวนี้ให้นักศึกษาทดลองเปลี่ยนแปลงการร้องขอทรัพยากรของโพรเซส P2 มาเป็นดังรูปบ้าง

	Request
	A B C
*P*0	0 0 0
*P*1	2 0 2
*P*2	**0 0 1**
*P*3	1 0 0
*P*4	0 0 2

คราวนี้ แม้เราจะเห็นว่าP0 จะสามารถทำงานต่อไปจนเสร็จได้ แต่โพรเซสอื่นๆ จะไม่สามารถทำงานต่อไปได้ ถือว่าเกิดสภาวะติดตายขึ้น

ในทางปฏิบัติ หนทางที่ดีที่สุดในการตรวจสอบสภาวะติดตายคือการเรียกใช้ขั้นตอนวิธีนี้ทุกครั้งเมื่อมีการร้องขอทรัพยากร แต่เนื่องจากการคำนวณใช้เวลา จึงทำให้ระบบอาจจะไม่มีประสิทธิภาพนักหากกระทำดังกล่าว แต่ถ้าเรียกช้าเกินไปจนกระทั่งเกิดสภาวะติดตายขึ้นอย่างซับซ้อนในระบบ การค้นหาสภาวะติดตายจะทำให้ไม่สามารถค้นได้ว่าโพรเซสใดเป็นต้นเหตุ และจะส่งผลให้เราต้องจัดการกับโพรเซสที่ไม่จำเป็น(ไม่ใช่ต้นเหตุ) ไปพร้อมกันด้วย เป็นการเสียเปล่าโดยใช่เหตุ อีกหนทางหนึ่งของการตรวจสอบคือ ให้โพรเซสตรวจสอบสภาวะติดตายลุกขึ้นมาทำงานเป็นระยะ เช่นทุกๆ ชั่วโมง และอาจตรวจสอบ CPU utilization ด้วยว่าหากซีพียูมีค่า utilization ต่ำกว่า 40 เปอร์เซนต์ โพรเซสตรวจสอบสภาวะติดตายจึงจะทำงานได้ (สภาวะติดตายจะส่งผลให้โพรเซสจำนวนหนึ่ง อยู่ในสภาวะหยุดรอ ในสภาวะดังกล่าว จึงทำให้ซีพียูมีเวลาประมวลเหลืออยู่มาก

7.4 การกู้คืนระบบจากสภาวะติดตาย

เมื่อเกิดสภาวะติดตายเกิดขึ้น เราสามารถกู้ระบบจากสภาวะติดตายได้ในสองลักษณะคือ

<u>การหยุดการทำงานของโพรเซสที่อยู่ในสภาวะติดตาย</u>

การหยุดโพรเซสที่อยู่ในสภาวะติดตาย จะส่งผลให้โพรเซสเหล่านั้นคืนทรัพยากรที่จองไว้แก่ระบบโดยอัตโนมัติ และระบบปฏิบัติการก็จะได้นำเอาทรัพยากรเหล่านั้นไปให้โพรเซสอื่นไปใช้งานได้ต่อไป การหยุดโพรเซสที่อยู่ในสภาวะติดตายมีหนทางทำได้ดังต่อไปนี้

- **<u>หยุดทุกโพรเซสที่อยู่ในสภาวะติดตาย</u>** แน่นอนว่าเป็นวิธีการที่ได้ผลเพราะทุกโพรเซสที่มีปัญหาได้จบการทำงาน แต่ก็ถือว่าเป็นการเสียเปล่าในการใช้ทรัพยากร เพราะบางโพรเซสอาจจะใช้เวลาคำนวณมานานมาก ใช้ข้อมูลไปมาก การหยุดโพรเซสเหล่านั้นจึงเป็นการเสียเปล่าทั้งเวลาครอบครองซีพียูและทรัพยากรอื่นๆ ที่ได้ใช้ไปแล้วอย่างมาก
- **<u>เลือกหยุดโพรเซสที่อยู่ในสภาวะติดตายไปทีละโพรเซส</u>** ในลักษณะเช่นนี้ แทนที่จะหยุดทุกโพรเซส เราก็เลือกหยุดเพียงหนึ่งโพรเซสจากโพรเซสทั้งหมดในสภาวะติดตาย โดยคาดหวังว่าเมื่อโพรเซสที่ถูกหยุดการทำงาน ปล่อยทรัพยากรที่ครอบครองออกมา อาจจะเพียงพอให้โพรเซสอื่นทำงานต่อไปได้ การเลือกว่าจะหยุดโพรเซสใดนั้นอาจมีกฎเพิ่มเติมดังเช่น
  - เลือกหยุดโพรเซสที่มี**<u>ค่าความสำคัญต่ำ</u>**ก่อน
  - เลือกหยุดโพรเซสที่เพิ่ง**<u>เริ่มทำงานมาไม่นาน</u>** หรือยัง**<u>เหลือเวลาที่ต้องคำนวณต่ออีกอยู่มาก</u>**
  - เลือกหยุดโพรเซสที่ถือ**<u>ครองทรัพยากรที่สามารถปลดทรัพยากรได้ง่าย</u>**กว่า (ในกรณีที่ทรัพยากรนั้นสามารถปลดการครอบครองชั่วคราว แล้วส่งคืนเพื่อให้ครอบครองใหม่อีกครั้งโดยไม่เสียหาย ในกรณีนี้ควรเลือกที่จะสั่งให้โพรเซสหยุดรอ แล้วปลดทรัพยากรไปให้โพรเซสอื่นใช้ชั่วคราวน่าจะดีกว่า เป็นต้น)
  - เลือกหยุดโพรเซสที่**<u>ต้องการทรัพยากรอีกจำนวนมาก</u>**
  - อาจจะเลือก**<u>หยุดโพรเซสหลายๆโพรเซส</u>**พร้อมกัน
  - โพรเซสอาจจะทำงานในลักษณะทำงานตามคำสั่งผู้ใช้มาทีละลำดับ หรือทำงานต่อเนื่องกันเป็น batch (มี batch file หรือ script file ที่เขียนเพื่อจัดการลำดับการทำงานไว้ล่วงหน้า) ในกรณีเช่นนี้ การเลือกหยุดโพรเซสที่ทำงานอย่างอัตโนมัติตามลำดับ จะส่งผลเสียมากกว่าโพรเซสที่รันตามคำสั่งผู้ใช้ไปคราวละหนึ่งงาน **<u>(เลี่ยงการหยุดโพรเซสในกลุ่มนี้)</u>**

<u>กลไกการปลดจากสภาวะติดตายอาจจะกระทำได้ในหลายลักษณะดังเช่น</u>

- การ**<u>หยุดการทำงานของโพรเซส</u>** วิธีนี้เป็นวิธีที่ง่ายที่สุด เพราะระบบปฏิบัติการไม่ต้องเก็บสถานะต่างๆ ของระบบไว้ตลอดเวลา แต่ก็มีผลเสียตรงที่โพรเซสที่หยุดทำงานไป ถือเป็นการเสียเปล่าทางทรัพยากรที่ได้ใช้ไปแล้ว
- การ**<u>ถอยสภาพกลับไปยังสถานะก่อนหน้า (rollback)</u>** ในลักษณะเช่นนี้ ระบบจะต้องเก็บสภาพการทำงาน และผลลัพธ์ในแต่ละขั้นตอนเอาไว้ เมื่อเกิดสภาวะติดตาย ก็จะถอยสภาพของระบบกลับไปยังสถานะก่อนหน้าสภาวะติดตายที่ได้บันทึกไว้ โดยคาดหวังว่ากลไกต่อจากนี้ไปของระบบจะไม่ได้เดินไปในเส้นทางที่จะก่อให้เกิดสภาวะติดตายซ้ำขึ้นมาอย่างเดิม
- การ**<u>ป้องกันมิให้บางโพรเซสต้องตกเป็นผู้เสียหายตลอดไป (starvation)</u>** ไม่ว่าจะเป็นการสั่งหยุดการทำงานของโพรเซสหรือสั่งให้โพรเซสต้องรอไปก่อนอันเป็นผลจากการถอยสภาพ ด้วยการกำหนดให้โพรเซสมี<u>ลำดับความสำคัญ (priority)</u> ต่างกัน เพื่อให้โพรเซสที่มีลำดับความสำคัญต่ำกว่า ต้องหยุดการทำงานแล้วเริ่มต้นใหม่ หรือต้องรอต่อไปนั้น อาจจะส่งผลให้โพรเซสเหล่านั้น<u>ไม่มีโอกาสได้ทำงานเสร็จเลย (เกิดสภาวะ starvaion)</u> ดังนั้นในทางปฏิบัติ จะต้องมีขั้นตอนวิธีเพิ่มเติมเพื่อช่วยเหลือโพรเซสเหล่านี้ได้มีโอกาสทำงานได้เสร็จสิ้นด้วย เช่น อาจบันทึกว่าโพรเซสที่ต้องหยุดการทำงานหรือต้องรอนั้น ได้กระทำการดังกล่าวซ้ำไปแล้วกี่ครั้ง หากเกินกว่าจำนวนครั้งที่กำหนด จะไม่สั่งให้โพรเซสดังกล่าวหยุดการทำงานหรือหยุดรออีก (แต่หันไปสั่งโพรเซสอื่นแทน) หรืออาจใช้วิธีการ aging กล่าวคือ ทุกๆ ครั้งที่ถูกสั่งให้หยุดทำงาน ก็จะเพิ่มค่า priority ของระบบขึ้นเล็กน้อย ดังนั้น หากโพรเซสนี้ถูกสั่งเริ่มใหม่หลายครั้ง ก็จะไปถึงจุดที่โพรเซสมีค่าลำดับความสำคัญสูงกว่าโพรเซสอื่น เพื่อเปิดโอกาสให้หยุดการทำงานโพรเซสอื่นบ้าง

ปฏิบัติการ

<u>หมายเหตุ</u> ปฏิบัติการต่อไปนี้ต้องการ conio.h เวอร์ชันพิเศษสำหรับ VT-100 เพื่อใช้ในลินุกซ์ และ winconio.h ที่เขียนขึ้นพิเศษสำหรับวินโดวส์ นักศึกษาสามารถดาวน์โหลดได้จากเว็ปช่วยสอน

7.1 การหลีกเลี่ยงสภาวะติดตายสำหรับกรณีการใช้ mutex ในลินุกซ์

ตัวอย่างต่อไปนี้ เป็นการแสดงถึงสภาวะติดตายอันเนื่องมากจากมีโพรเซสที่ครอบครองทรัพยากร (ในที่นี้คือ mutual exclusion lock) ส่วนหนึ่ง และพยายามครอบครองทรัพยากรอีกส่วนหนึ่ง แต่ปรากฏว่าอีกโพรเซสได้ครอบครองไว้แล้ว และพยายามจะครอบครองทรัพยากรที่โพรเซสตัวแรกได้ครองไว้แล้ว เกิดสภาวะติดตายขึ้น

จากตัวอย่าง เราสมมติว่าพื้นที่หน่วยความจำส่วนกลางที่โพรเซสครอบครองนั้น จะใช้ได้เพียงหนึ่งโพรเซสในเวลาใดเวลาหนึ่ง เราจึงได้สร้าง mutex lock ขึ้นมาเพื่อควบคุมส่วนวิกฤติ นอกจากนี้เราสมมติเพิ่มเติมว่า การแสดงผลในที่นี้ ซึ่งจะต้องย้ายตำแหน่งเคอร์เซอร์ไปยังบริเวณที่ต้องการแล้วจึงส่งข้อความไป จะต้องกระทำต่อเนื่อง (มิฉะนั้นจะแสดงผลไม่ถูกตำแหน่ง) เราจึงเพิ่มทรัพยากรการแสดงผลเป็นอีกหนึ่งส่วนที่เป็นส่วนวิกฤติ และเรากำหนดตัวแปร mutex lock ชื่อ display มาเพื่อควบคุมการแสดงส่วนนี้

ให้นักศึกษาเริ่มทดลองโดยการคอมเมนต์บรรทัด #define เป็นดังนี้ก่อน เพื่อให้โปรแกรมทำงานตามปกติ

```
#define _LOCK
//#define _TRYLOCK
//#define _TIMEDLOCK
```

จะเห็นว่าในส่วน producer (chefWork) นั้น เราเริ่มล็อคส่วน mutex (คุมทรัพยากรพื้นที่หน่วยความจำที่เราใช้เขียนอ่านร่วมกันระหว่างโพรเซส) ก่อนแล้วตามด้วย display (คุมทรัพยากรการแสดงผลบนจอภาพที่แต่ละโพรเซสใช้แสดงร่วมกัน) ในส่วน consumer เราเริ่มล็อคส่วน display แล้วตามด้วย mutex เพื่อให้เห็นภาพชัดขึ้น เราจะหน่วงเวลาเล็กน้อยประมาณ 1มิลลิวินาทีด้วย

randomDelay2() สิ่งที่ปรากฏในการทำงานก็คือ เราจะพบว่าเมื่อโปรแกรมทำงานไปได้สักครู่ ก็จะหยุดทำงาน ทั้งนี้เกิดจากที่ producer ได้ครอบครองทรัพยากรพื้นที่หน่วยความจำ (mutex) แล้ว แต่พยายามจะร้องขอการแสดงผล (display) ในขณะที่ consumer ได้ครอบครองทรัพยากรการแสดงผลแล้ว แต่พยายามจะเข้าครอบครองพื้นที่หน่วยความจำร่วม เกิดสภาวะติดตายขึ้นทั้งสองโพรเซส

เพื่อแก้ไขสภาพการณ์ดังกล่าว ในที่นี้เราได้แสดงกลไกสองวิธีการที่มีใช้อย่างแพร่หลาย อย่างแรกคือ***การทดลองว่าทรัพยากรดังกล่าวได้มีผู้ครอบครองแล้วหรือไม่ ถ้ามีผู้ครอบครองแล้ว ในที่นี้เราจะปล่อยทรัพยากรที่ครอบครองทั้งหมดให้โพรเซสอื่นไปจัดการก่อน*** (ในทางปฏิบัติเราสามารถประยุกต์ใช้ได้หลากหลายวิธีการกว่านี้ นี่เป็นวิธีการง่ายๆ วิธีหนึงเท่านั้น) การทดลองแบบนี้ให้นักศึกษาเปลี่ยนคอมเมนต์เป็น

```
//#define _LOCK
#define _TRYLOCK
//#define _TIMEDLOCK
```

การทดสอบว่ามีผู้ล็อคทรัพยากรไว้แล้วหรือไม่ เราใช้ `pthread_mutex_trylock()` ซึ่งจะทำงานคล้ายคลึงกับ `pthread_mutex_lock()` ที่ใช้ขอล็อคทรัพยากรตามปกติ เพียงแต่การใช้ trylock นั้น**หากมีผู้ได้ล็อคไว้ก่อนแล้ว โพรเซสจะไม่หยุดรอ แต่จะส่งค่ากลับมาแจ้งว่าขอล็อคไม่สำเร็จ** ในกรณีนี้เราก็จะสามารถเขียนโค้ดเพื่อจัดการใดๆ ต่อไปได้ตามสะดวก

วิธีข้างต้นมีข้อด้อยที่สำคัญอย่างหนึ่งก็คือ ในกรณีการใช้งานจริง ระบบจะมีจำนวนทรัพยากรมากกว่าหนึ่งหน่วยต่อชนิด ทำให้ต้องใช้ขั้นตอนวิธีของนักการธนาคาร **ซึ่งไม่รับประกันว่าโพรเซสที่หยุดรออยู่นั้นกำลังเกิดสภาวะติดตายอยู่หรือไม่**

ดังนั้น แทนที่เราจะปลดทรัพยากรในทันที เราอาจจะปล่อยโพรเซสที่ถูกเลือกให้หยุดการทำงานหรือถอยหลัง รอไปเป็นชั่วเวลาสั้นๆ เผื่อว่าเป็นการรอคอยทรัพยากรธรรมดาที่ไม่ได้เป็นผลมาจากสภาวะติดตาย เช่นกำหนดว่า **หากรอคอยทรัพยากรเกินเวลาที่กำหนด ให้ปล่อยทรัพยากรทั้งหมดไปก่อน ในตัวอย่างเรารอ 2 วินาที** ถ้ายังไม่สามารถรันต่อไปได้ก็ให้ปล่อยทรัพยากรที่มีออกมา ในกรณีนี้ให้นักศึกษาคอมเมนต์เป็น

```
//#define _LOCK
//#define _TRYLOCK
#define _TIMEDLOCK
```

ในกรณีนี้ เราใช้ `pthread_mutex_timedlock()` ซึ่งจะมีพารามิเตอร์เพิ่มขึ้นมาอีกหนึ่งตัวคือ ฐานเวลานับสิ้นสุด โดยเราจะหาค่าฐานเวลาปัจจุบันมาก่อน แล้วบวกเพิ่มด้วยเวลาที่ต้องการรอคอย จากนั้นส่งเป็นพารามิเตอร์ให้ฟังก์ชัน เมื่อโพรเซสรอจนถึงเวลาตามฐานเวลาที่แจ้งไว้แล้ว ยังมิได้รับล็อคเพื่อเข้าสู่ส่วนวิกฤติ ฟังก์ชันจะจบการทำงานทันทีพร้อมทั้งแจ้งผลกลับว่าเวลารอนั้นหมดแล้ว เราก็สามารถนำเอาค่าส่งกลับมาตรวจสอบเพื่อจัดการใดๆ ต่อไปได้เช่นกัน

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


// Try using one of these options to see how to recover deadlock
//#define _LOCK
//#define _TRYLOCK
#define _TIMEDLOCK


#if defined(_TIMEDLOCK)
struct timespec myTime;
#endif
```

```
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;

pthread_mutex_t mutex;
pthread_mutex_t display;

void randomDelay(void);
void randomDelay2(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

       clrscr();

       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
       pthread_mutex_init(&mutex,NULL);
       pthread_mutex_init(&display,NULL);

       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);

       pthread_mutex_destroy(&mutex);
       pthread_mutex_destroy(&display);

       return 0;
}

void randomDelay(void){
    int stime = ((rand()%10)+1)*1000;
    usleep(stime);
}

void randomDelay2(void){
      usleep(1000);    // Try comment this line to reduce the time between requesting two
resources
}

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);

    pthread_mutex_lock(&display);
```

```
    gotoxy(1,chefNo*3+3);
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);
    pthread_mutex_unlock(&display);

    while(!isFull){

        if(isReady) {
            randomDelay(); // Wait for food to be taken
        }
        pthread_mutex_lock(&mutex);
        randomDelay2();

#if defined(_TRYLOCK)
        if(pthread_mutex_trylock(&display)){  // Try if the lock is a success
                                              // Return 0 is success
                gotoxy(1,chefNo*3+3);
                printf("Chef NO %d Cannot lock the display     \n",chefNo+1);
                fflush(stdout);
                pthread_mutex_unlock(&mutex); // Release the resource
                continue;
        }
#elif defined(_TIMEDLOCK)
        time_t currentTime;
        time(&currentTime);  //Get current time
        myTime.tv_nsec=0;
        myTime.tv_sec=2+currentTime; // `2 seconds waiting from current time
        if(pthread_mutex_timedlock(&display,&myTime)){       // Try if the lock is a success
                                              // Return 0 is success
                gotoxy(1,chefNo*3+3);
                printf("Chef NO %d display timeout..            \n",chefNo+1);
                fflush(stdout);
                pthread_mutex_unlock(&mutex); // Release the resource
                continue;
        }
#else
        pthread_mutex_lock(&display);
#endif
        if((!isFull)&&(!isReady)){
        // Critical Section
                gotoxy(1,chefNo*3+3);
                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

                gotoxy(1,chefNo*3+3);
                printf("Chef NO %d has serve the food       \n",chefNo+1);
                fflush(stdout);
        }
        pthread_mutex_unlock(&display);
        pthread_mutex_unlock(&mutex);

        if(isFull) break;
        randomDelay();
    }
    pthread_exit(0);
}

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

    for(i=0;i<10;i++){

        while(!isReady) randomDelay();

        pthread_mutex_lock(&display);
        randomDelay2();
        pthread_mutex_lock(&mutex);
                // Critical Section
        gotoxy(1,1);
        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;

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

         isReady=false; // Food taken
        pthread_mutex_unlock(&mutex);
        pthread_mutex_unlock(&display);

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

7.2 การหลีกเลี่ยงสภาวะติดตายสำหรับกรณีการใช้ mutex ในวินโดวส์

ในกรณีของระบบปฏิบัติการวินโดวส์นั้น การรอคอย mutex มองเป็นวัตถุ ซึ่งเรารอคอยด้วยฟังก์ชัน `WaitForSingleObject()` ซึ่งฟังก์ชันนี้สามารถเซ็ตพารามิเตอร์บอกเวลาที่รอคอยได้

ในกรณีแรก ถ้าเราเซ็ตเวลารอคอยเป็น `INFINITE` นั่นหมายความว่าการรอคอยทรัพยากรจะไม่มีเวลาจำกัด ถ้าเกิดสภาพการณ์ที่โพรเซสหนึ่งครองทรัพยากรบางส่วน แล้วพยายามไปขอทรัพยากรที่เหลือ ในขณะที่อีกโพรเซสได้ครอบครองไว้แล้ว และพยายามไปครองทรัพยากรที่โพรเซสแรกครอบครองก่อน ก็จะเกิดสภาวะติดตาย

ในกรณีที่สอง ถ้าเราเปลี่ยนพารามิเตอร์เวลารอคอยเป็น 0 ในที่นี้หมายความว่าไม่มีเวลารอคอยเลย ฟังก์ชัน `WaitForSingleObject()` จะจบการทำงานทันทีไม่ว่าจะได้ล็อคหรือไม่ ในที่นี้เราก็จะนำค่าส่งกลับมาตรวจสอบว่า หากเวลาหมดลงก่อน ก็หมายความว่าไม่สามารถล็อก mutex ได้ เราจะปลดทรัพยากรที่ครอบครองไว้ เพื่อให้พ้นจากสภาวะติดตาย ซึ่งจะส่งผลเทียบเท่าการใช้งาน `pthread_mutex_trylock()` ในลินุกซ์

ในกรณีที่สาม ถ้าเราเซ็ตพารามิเตอร์เวลารอคอยเป็นค่าใดๆ เช่นในที่นี้เป็น 2000 มิลลิวินาที หรือสองนาที หมายความว่าเราจะรอว่าจะได้รับ mutex lock ภายในสองวินาทีหรือไม่ หากไม่ได้ในช่วงเวลาดังกล่าว ฟังก์ชัน `WaitForSingleObject()` ก็จะจบการทำงานและแจ้งหมดเวลา ซึ่งเราก็จะสามารถนำมาจัดการใดๆ ต่อไปได้เช่นกัน กรณีนี้เทียบเท่ากับ `pthread_mutex_timedlock()`

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


// Try using one of these options to see how to recover deadlock
//#define _LOCK
//#define _TRYLOCK
#define _TIMEDLOCK

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 mutex,display;

void randomDelay(void);
void randomDelay2(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};

        clrscr();

        mutex = CreateMutex(NULL,false,NULL);
        display = CreateMutex(NULL,false,NULL);

        // 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(mutex);
        CloseHandle(display);
        return 0;
}

void randomDelay(void){
    int stime = ((rand()%10)+1);
    Sleep(stime);
}

void randomDelay2(void){
    Sleep(1);  // Try comment this line to reduce the time between requesting two resources
}

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;
       DWORD dwWaitResult;

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

       WaitForSingleObject(display,INFINITE);
       gotoxy(1,chefNo*3+3);
       printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);
       ReleaseMutex(display);

    while(!isFull){

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

       WaitForSingleObject(mutex,INFINITE);
       randomDelay2();

#if defined(_TRYLOCK)
       if(WaitForSingleObject(display,0)==WAIT_TIMEOUT){    // Try if the lock is a success
              gotoxy(1,chefNo*3+3);
```

```
                printf("Chef NO %d Cannot lock the display     \n",chefNo+1);
                fflush(stdout);
                ReleaseMutex(mutex);   // Release the resource
                continue;
        }
#elif defined(_TIMEDLOCK)
        if(WaitForSingleObject(display,2000L)==WAIT_TIMEOUT){       // wait 2 seconds
                gotoxy(1,chefNo*3+3);
                printf("Chef NO %d time out...        \n",chefNo+1);
                fflush(stdout);
                ReleaseMutex(mutex);   // Release the resource
                continue;
        }
#else
        WaitForSingleObject(display,INFINITE);
#endif
                // Critical Section
                if((!isFull)&&(!isReady)){
                gotoxy(1,chefNo*3+3);
                        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

                gotoxy(1,chefNo*3+3);
                        printf("Chef NO %d has serve the food                  \n",chefNo+1);
                        fflush(stdout);
                }
        ReleaseMutex(display);
        ReleaseMutex(mutex);
        // Remaining Section

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

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

    for(i=0;i<10;i++){

        while(!isReady) randomDelay();

        WaitForSingleObject(display,INFINITE);
        randomDelay2();
        WaitForSingleObject(mutex,INFINITE);
        // Critical Section
                gotoxy(1,1);
                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;

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

                isReady=false; // Food taken

        ReleaseMutex(mutex);
        ReleaseMutex(display);
        // Remaining Section
        }
        isFull = true; // Tell other producer threads to stop;
    return 0;
}
```

7.3 การหลีกเลี่ยงสภาวะติดตายสำหรับกรณีการใช้เซมาฟอร์ในลินุกซ์

ตัวอย่างนี้เป็นการจำลองการทำงานในรูปแบบเดียวกับ 7.1 แต่หันมาใช้เซมาฟอร์แทน mutex lock สังเกตกลไกการจัดการแก้ไขสภาวะติดตายนั้นใช้หลักการเดียวกัน และชื่อฟังก์ชันก็มีลักษณะทำนองเดียวกัน ง่ายต่อการจำ

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

// Try using one of these options to see how to recover deadlock
//#define _LOCK
//#define _TRYLOCK
#define _TIMEDLOCK



#if defined(_TIMEDLOCK)
struct timespec myTime;
#endif

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 mutex;
sem_t display;

void randomDelay(void);
void randomDelay2(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

        clrscr();

        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(&mutex,0,1);
        sem_init(&display,0,1);

        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(&mutex);
        sem_destroy(&display);

        return 0;
}
```

```
void randomDelay(void){
    int stime = ((rand()%10)+1)*1000;
    usleep(stime);
}

void randomDelay2(void){
     usleep(1000);    // Try comment this line to reduce the time between requesting two
resources
}

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);

    sem_wait(&display);
    gotoxy(1,chefNo*3+3);
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);
    sem_post(&display);

    while(!isFull){

        if(isReady) {
            randomDelay(); // Wait for food to be taken
        }
        sem_wait(&mutex);
        randomDelay2();

#if defined(_TRYLOCK)
        if(sem_trywait(&display)){    // Try if the lock is a success
                                              // Return 0 is success
                gotoxy(1,chefNo*3+3);
                printf("Chef NO %d Cannot lock the display     \n",chefNo+1);
                fflush(stdout);
                sem_post(&mutex);      // Release the resource
                continue;
        }
#elif defined(_TIMEDLOCK)
        time_t currentTime;
        time(&currentTime);  //Get current time
        myTime.tv_nsec=0;
        myTime.tv_sec=2+currentTime; // `2 seconds waiting from current time
        if(sem_timedwait(&display,&myTime)){  // Try if the lock is a success
                                              // Return 0 is success
                gotoxy(1,chefNo*3+3);
                printf("Chef NO %d display timeout..            \n",chefNo+1);
                fflush(stdout);
                sem_post(&mutex);      // Release the resource
                continue;
        }
#else
        sem_wait(&display);
#endif
        if((!isFull)&&(!isReady)){
        // Critical Section
                gotoxy(1,chefNo*3+3);
                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

                gotoxy(1,chefNo*3+3);
                printf("Chef NO %d has serve the food       \n",chefNo+1);
                fflush(stdout);
```

```
        }
        sem_post(&display);
        sem_post(&mutex);

        if(isFull) break;
        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(&display);
        randomDelay2();
        sem_wait(&mutex);
                // Critical Section
        gotoxy(1,1);
        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;

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

         isReady=false; // Food taken
        sem_post(&mutex);
        sem_post(&display);
    }
    isFull = true; // Tell other producer threads to stop;
    pthread_exit(0);
}
```

7.4 การหลีกเลี่ยงสภาวะติดตายสำหรับกรณีการใช้เซมาฟอร์ในวินโดวส์

ตัวอย่างนี้ใช้เซมาฟอร์อ็อปเจ็คต์ในการจัดการ ให้นักศึกษาสังเกตถึงการใช้ `WaitForSingleObject()` ที่มีใช้งานเหมือนกันกับกรณีของ mutex lock ทุกประการ

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


// Try using one of these options to see how to recover deadlock
//#define _LOCK
//#define _TRYLOCK
#define _TIMEDLOCK

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,display;

DWORD sem_wait(HANDLE sem);
DWORD sem_trywait(HANDLE sem);
DWORD sem_timedwait(HANDLE sem,int milliSecond);
```

```
DWORD sem_signal(HANDLE sem);

void randomDelay(void);
void randomDelay2(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};

        clrscr();

        sem     = CreateSemaphore(NULL,1,1,NULL);
        display = CreateSemaphore(NULL,1,1,NULL);

        // 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);
        CloseHandle(display);
        return 0;
}

void randomDelay(void){
    int stime = ((rand()%10)+1);
    Sleep(stime);
}

void randomDelay2(void){
    Sleep(1);  // Try comment this line to reduce the time between requesting two resources
}

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;
        DWORD dwWaitResult;

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

        sem_wait(display);
        gotoxy(1,chefNo*3+3);
        printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);
        sem_signal(display);

    while(!isFull){
```

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

        sem_wait(sem);
        randomDelay2();

#if defined(_TRYLOCK)
        if(sem_trywait(display)){     // Try if the lock is a success
                gotoxy(1,chefNo*3+3);
                printf("Chef NO %d Cannot lock the display     \n",chefNo+1);
                fflush(stdout);
                sem_signal(sem);       // Release the resource
                continue;
        }
#elif defined(_TIMEDLOCK)
        if(sem_timedwait(display,2000L)){     // wait 2 seconds
                gotoxy(1,chefNo*3+3);
                printf("Chef NO %d time out...        \n",chefNo+1);
                fflush(stdout);
                sem_signal(sem);       // Release the resource
                continue;
        }
#else
        sem_wait(display);
#endif
                // Critical Section
                if((!isFull)&&(!isReady)){
                gotoxy(1,chefNo*3+3);
                        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

                gotoxy(1,chefNo*3+3);
                        printf("Chef NO %d has serve the food                  \n",chefNo+1);
                        fflush(stdout);
                }
        sem_signal(display);
        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,j;
    char dinner[256]={0};

    for(i=0;i<10;i++){

        while(!isReady) randomDelay();

        sem_wait(display);
        randomDelay2();
        sem_wait(sem);
        // Critical Section
                gotoxy(1,1);
                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;

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

                isReady=false; // Food taken

        sem_signal(sem);
```

```
        sem_signal(display);
        // 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_trywait(HANDLE sem){
    DWORD result = WaitForSingleObject(sem,0);

        if(result == WAIT_TIMEOUT) return 1;
        else return 0;
}

DWORD sem_timedwait(HANDLE sem,int milliSecond){
    DWORD result = WaitForSingleObject(sem,milliSecond);

        if(result == WAIT_TIMEOUT) return 1;
        else return 0;
}
DWORD sem_signal(HANDLE sem){
    return ReleaseSemaphore(sem,1,NULL); // Increase by one
}
```