แผนการบริหารการสอนประจำบทที่ 3

เนื้อหาประจำบท

บทที่ 3 การประสานเวลาของโปรเซส

1. แนวคิดในการทำงานร่วมกันของโปรเซส
2. การประสานเวลาของโปรเซสและอัลกอริทึมในการแก้ไขปัญหา
3. เกณฑ์การกีดกันระหว่างโปรเซสและอัลกอริทึมในการแก้ไขปัญหา

จุดประสงค์เชิงพฤติกรรม

เมื่อศึกษาบทที่ 3 แล้วนักศึกษาสามารถ

1. อธิบายการจัดการปัญหาการเข้าใช้ส่วนวิกฤตและอัลกอริทึมในการแก้ไขปัญหา
2. อธิบายเกี่ยวกับแนวคิดพื้นฐานและความเป็นมาเกี่ยวกับการประสานเวลาของโปรเซส
3. อธิบายการจัดการปัญหาส่วนวิกฤติ การประสานเวลาของฮาร์ดแวร์และซอฟต์แวร์
4. เพื่อให้เข้าใจถึงปัญหาของการประสานเวลา และวิธีการแก้ไขโดยใช้ตัวเฝ้าสังเกต

กิจกรรมการเรียนการสอนประจำบท

1. ผู้สอนอธิบายหลักการทำงานของระบบปฏิบัติการ พร้อมยกตัวอย่างประกอบการบรรยาย
2. ให้ผู้เรียนศึกษาเอกสารประกอบการเรียนการสอน ศึกษาทำความเข้าใจและซักถาม
3. ให้ผู้เรียนทำแบบฝึกหัดและงานที่ได้รับมอบหมาย
4. ทดสอบย่อยหลังจบบทเรียน

สื่อการเรียนการสอน

1. สื่ออิเล็กทรอนิกส์ประกอบการสอนวิชาระบบปฏิบัติการ
2. เอกสารประกอบการสอนวิชาระบบปฏิบัติการ
3. หนังสืออ่านประกอบค้นคว้าเพิ่มเติม

การวัดผลและประเมินผล

1. สังเกตจากการซักถามในระหว่างการเรียน
2. สังเกตจากความสนใจและความตั้งใจ
3. ประเมินจากการอภิปรายกลุ่มย่อย และการทำแบบฝึกหัด
4. ประเมินจากการสอบระหว่างภาคและปลายภาค

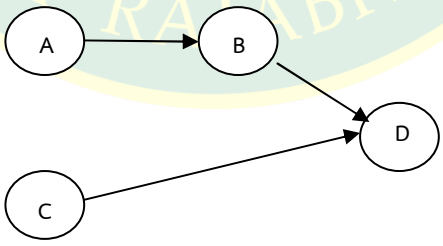
บทที่ 3
การประสานเวลาของโพรเซส

ในระบบคอมพิวเตอร์ที่มีหน่วยประมวลผลเดียวแต่มีหลายโพรเซสทำงานอยู่ในระบบ โพรเซสต่าง ๆ สลับกันทำงานด้วยความถี่หลายครั้งต่อวินาที คล้ายกับว่าโพรเซสต่าง ๆ ทำงานไปพร้อม ๆ กัน เครื่องที่มีหน่วยประมวลผลมากกว่าหนึ่งหน่วยประมวลผล โพรเซสต่าง ๆ สามารถทำงานไปพร้อมกันได้จริงบนหน่วยประมวลผลคนละตัวกัน สภาพเช่นนี้เรียกว่าภาวะพร้อมกัน (Concurrency) ระบบคอมพิวเตอร์ที่ทำงานแบบมัลติโปรแกรมมิ่ง มีโพรเซสหลายโพรเซสทำงานพร้อมกัน โดยที่แต่ละโพรเซสอาจมีการทำงานเป็นอิสระต่อกัน หรือมีการทำงานที่ขึ้นต่อกัน ดังนั้นหากมีความต้องการที่จะเข้าใช้ทรัพยากรที่มีอยู่พร้อม ๆ กัน เช่น ใช้หน่วยความจำตำแหน่งเดียวกันพร้อมกัน ทำให้เกิดปัญหาการเขียนอ่าน และสำหรับหน่วยความจำที่ใช้ร่วมกัน อาจทำให้ผลของการทำงานคลาดเคลื่อนจากที่ควรจะเป็น

3.1 การประสานเวลาของโพรเซส

โดยทั่วไปทุกโพรเซสต้องการให้มีการประมวลผลอย่างเป็นอิสระ เสมือนมีโพรเซสเดียวที่กำลังทำงานอยู่ ลักษณะความเป็นอิสระนี้เรียกว่า Asynchronous ซึ่งในความเป็นจริงมีหลายโพรเซสเกิดขึ้นในเวลาเดียวกัน และอาจมีความจำเป็นต้องการใช้ทรัพยากรที่มีอยู่อย่างจำกัดพร้อมกัน เช่น หน่วยความจำ ระบบปฏิบัติการจึงเข้ามาช่วยจัดการแต่ละโพรเซส ให้สามารถประมวลผลได้พร้อมกัน (Concurrent processing) และมีการทำงานร่วมกัน (Cooperating) เป็นผลให้โพรเซสในระบบได้มี การใช้ทรัพยากรร่วมกัน ช่วยเพิ่มความเร็วในการทำงานของระบบสูงขึ้นและมีความสะดวกในการทำงาน ในการทำงานของโพรเซสที่ต้องการให้มีการทำงานร่วมกัน จำเป็นต้องมีการสื่อสารกันระหว่างโพรเซส (Inter process communication) จำเป็นต้องให้ทุกโพรเซสทำงานประสานกันเป็นอย่างดี การเข้าจังหวะของโพรเซสหรืออาจเรียกว่าโพรเซสนั้นเกิดการประสานเวลาของโพรเซส (Process synchronization) การประสานเวลาของโพรเซสจะหมายถึง การทำงานของโพรเซสที่ต้องมีการเกี่ยวข้องกันอาจเป็นเพราะ การใช้ทรัพยากรร่วมกัน หรืออาจจะเป็นการรอให้โพรเซสทำงาน หลังจากที่โพรเซสอื่นทำงานแล้ว

เพื่อความเข้าใจยิ่งขึ้นลองพิจารณาภาพที่ 3.1 จะเห็นว่าโพรเซส B จะเริ่มทำงานได้ก็ต่อเมื่อโพรเซส A ทำงานเสร็จเรียบร้อยไปแล้ว (ซึ่งต่างกับโพรเซส C ที่ไม่ต้องรอโพรเซสใดเลยก็สามารถทำงานได้ทันที และไม่มีความเกี่ยวข้องกับโพรเซส A และ B เลย) ในทำนองเดียวกันโพรเซส D จะทำงานได้ก็ต่อเมื่อมีการทำงานโพรเซส C และโพรเซส C ทำงานเสร็จเรียบร้อยไปแล้วนั่นเอง

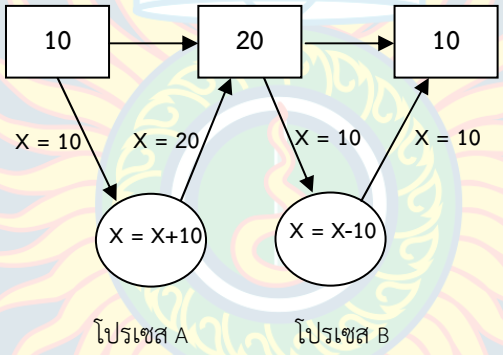


ภาพที่ 3.1 แสดงการประสานเวลาของโพรเซส

3.2 ปัญหาภาวะพร้อมกัน

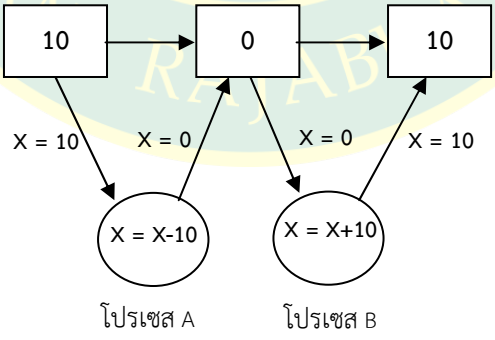
ในระบบที่มีการทำงานบนซีพียูเดียว หรือทำงานบนซีพียูหลายตัว บางครั้งมีความจำเป็นที่ต้องใช้ทรัพยากรร่วมกัน และเป็นทรัพยากรที่ต้องทำงานครั้งละ 1 งาน ไม่สามารถทำงานพร้อมกันได้ จะเรียกทรัพยากรเหล่านี้ว่า ทรัพยากรที่ไม่สามารถใช้ร่วมกันได้ (Non-dedicated resources) ดังนั้น ในขณะที่โพรเซสหนึ่งกำลังใช้ทรัพยากรอยู่ โพรเซสอื่นจะไม่สามารถเข้ามาใช้ทรัพยากรนั้น ๆ ได้ เพื่อให้ได้ผลของการทำงานที่ถูกต้อง เช่น เครื่องพิมพ์ เป็นต้น โพรเซสที่ใช้ข้อมูลร่วมกันหรือร่วมกันทำงาน การใช้ทรัพยากรต่าง ๆ เช่น หน่วยความจำ และอุปกรณ์เข้าออกร่วมกันได้ระหว่างโพรเซสทำให้เกิดปัญหาต่าง ๆ ตามมาดังตัวอย่างต่อไปนี้

ตัวอย่าง ในการใช้ทรัพยากรและข้อมูลร่วมกันโพรเซสสองโพรเซสขึ้นไป อาจก่อให้เกิดปัญหาในการทำงานที่ถูกต้องสมมุติโพรเซสทั้งสองเป็นอิสระต่อกัน ทำงานในเวลาพร้อม ๆ กัน และใช้ข้อมูลเดียวกันคือตัวแปร X ซึ่งกำหนดค่าเริ่มต้นไว้ที่ 10 โดยที่ โพรเซส A จะเป็นการเพิ่มค่าเข้าไปอีก 10 (X = X+10) ส่วนโพรเซส B จะเป็นการลดค่าจากเดิมออกไป 10 (X = X-10)



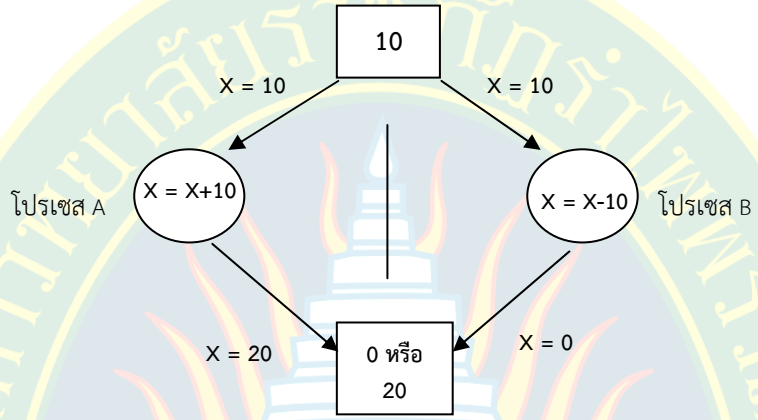
ภาพที่ 3.2 โพรเซส A ทำงานเสร็จก่อน โพรเซส B

จากภาพที่ 3.2 โพรเซส A ทำงานก่อนโพรเซส B โดยอ่านค่า X เข้ามาซึ่งมีค่าเป็น 10 แล้วเพิ่มเข้าไปอีก 10 ทำให้ X มีค่าเป็น 20 แล้วเขียนลงหน่วยความจำ หลังจากนั้นโพรเซส B จึงอ่านค่า X เข้ามาซึ่งมีค่าเป็น 20 แล้วลบค่าออกไป 10 ทำให้ X มีค่าเป็น 10 ลงหน่วยความจำ



ภาพที่ 3.3 โพรเซส B ทำงานเสร็จก่อน โพรเซส A

จากภาพที่ 3.3 การทำงานจะคล้ายกับภาพที่ 3.2 และได้ผลลัพธ์เหมือนกัน แต่โพรเซส B ทำงานก่อนโพรเซส A โดยอ่านค่า X เข้ามาซึ่งมีค่าเป็น 10 แล้วลบค่าออกไป 10 ทำให้ X มีค่าเป็น 0 จากนั้นเขียนลงหน่วยความจำ หลังจากนั้นโพรเซส A จึงอ่านค่า X เข้ามาซึ่งมีค่าเป็น 0 แล้วเพิ่มค่าเข้าไป 10 ทำให้ X มีค่าเป็น 10 แล้วเขียนค่า X ลงหน่วยความจำ



ภาพที่ 3.4 โพรเซส A และ โพรเซส B ทำงานพร้อมกัน

สำหรับภาพที่ 3.4 ทั้งโพรเซส A และ B จะทำงานพร้อม ๆ กัน โดยค่าเริ่มต้น X เป็น 10 ทั้งสองโพรเซสจะได้ผลลัพธ์ที่ต่างกัน โดยโพรเซส A จะได้ค่า X เป็น 20 (จาก 10 เพิ่มอีก 10) ส่วนโพรเซส B จะได้ค่า X เป็น 0 (จาก 10 ลบออกไป 10) ถ้าโพรเซส A ทำงานเสร็จก่อนจะเขียนค่า X ซึ่งเท่ากับ 20 ลงหน่วยความจำ แต่ถ้าโพรเซส B ทำงานเสร็จก่อนจะเขียนค่า X ซึ่งเท่ากับ 0 ลงหน่วยความจำ ซึ่งทำให้เกิดข้อผิดพลาดเนื่องจากค่า X แน่นอน ขึ้นอยู่กับว่าโพรเซสใดทำงานเสร็จก่อน

3.3 สภาวะการแย่งชิง

จากตัวอย่างโพรเซสใช้ตัวแปรตัวหนึ่งในหน่วยความจำร่วมกันทั้งสองโพรเซส การแก้ไขข้อมูลตัวแปรนั้นพบว่า ลำดับของการแก้ไขก่อนหรือหลังมีความสำคัญต่อค่าของข้อมูลตัวนั้น ทำให้โพรเซสสองโพรเซสที่ต้องการใช้ทรัพยากรที่แชร์ไว้พร้อมกันในเวลาเดียวกัน เช่น หน่วยความจำ สื่อจัดเก็บข้อมูล หรือแชร์ไฟล์ ซึ่งโพรเซสทั้งสองสามารถอ่านหรือเขียนทรัพยากรเหล่านี้ได้พร้อมกัน ทำให้ผลลัพธ์อาจจะเกิดการผิดพลาดขึ้นได้ ขึ้นอยู่ว่าโพรเซสใดเข้ามาใช้ก่อน ทำให้ผลลัพธ์อาจจะเกิดการผิดพลาดขึ้นได้ ขึ้นอยู่กับว่าโพรเซสใดทำเสร็จก่อนเรียกสภาวะนี้ว่า สภาวะเงื่อนไขการแย่งชิง (Race condition) ดังนั้นเพื่อป้องกันการเกิดเงื่อนไขการแย่งชิงขึ้นตามตัวอย่าง ระบบต้องแน่ใจว่าในเวลาใด ๆ จะมีเพียงโพรเซสเดียวเท่านั้นที่กำลังเข้าจัดการข้อมูลตัวแปร X โดยต้องอาศัยกลไกบางอย่างในการประสานโพรเซสอย่างได้จังหวะกันจึงสามารถรับประกันความถูกต้องได้

3.4 การแก้ปัญหาส่วนวิกฤติ

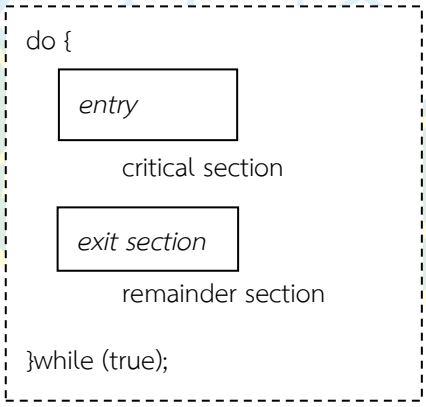
ปัญหาที่เกิดขึ้นจากสถานการณ์ที่มีเงื่อนไขการแย่งชิงนี้ เป็นผลมาจากการที่แต่ละโพรเซสมีการเข้าใช้ข้อมูลบางอย่างร่วมกัน โดยไม่มีการประสานการทำงานระหว่างกัน ต่างฝ่ายต่างทำงานตามคำสั่งไปจนกระทั่งถึงบริเวณที่มีคำสั่งใช้งานข้อมูลร่วมกับอีกฝ่ายหนึ่ง ซึ่งอาจทำให้เกิดผลลัพธ์ที่ผิดพลาดได้ ดังนั้นหากระบบสามารถประสานการเข้าทำงานในบริเวณที่แต่ละโพรเซสมีการใช้ข้อมูลร่วมกันก็จะช่วยแก้ปัญหาที่เกิดขึ้นได้ ลองพิจารณาระบบที่ประกอบด้วย n โพรเซส {P0, P1, ..., Pn-1} โดยแต่ละโพรเซสมีส่วนโปรแกรมซึ่งทำการเปลี่ยนแปลงค่าข้อมูลตัวแปร ปรับปรุงตาราง บันทึกข้อมูลลงแฟ้ม และอื่น ๆ ที่ต้องมีการทำงานร่วมกัน เรียกส่วนโปรแกรมบริเวณนี้ว่า ส่วนวิกฤติ (Critical-section) แล้วกำหนดคุณลักษณะที่สำคัญของระบบคือ ขณะที่โพรเซสหนึ่งกำลังปฏิบัติการอยู่ในส่วนวิกฤติของตนเอง จะต้องไม่มีโพรเซสอื่นได้รับอนุญาตให้เข้าไปปฏิบัติการในส่วนวิกฤติของโพรเซสเหล่านั้น

ดังนั้นการปฏิบัติการในส่วนวิกฤติของโพรเซสต้องเป็นไปในลักษณะที่มีการกีดกัน (Mutual exclusion) โดยที่แนวทางแก้ปัญหาในส่วนวิกฤติคือ การออกแบบข้อตกลงหรือโปรโตคอล (Protocol) เพื่อให้โพรเซสทั้งหลายสามารถทำงานร่วมกันได้อย่างถูกต้อง ทั้งนี้แต่ละโพรเซสต้องร้องขอคำอนุญาตเพื่อเข้าสู่ส่วนวิกฤติของตนเอง ซึ่งอยู่ในรูปแบบของการตรวจสอบตามเงื่อนไขที่กำหนด โดยส่วนโปรแกรมที่ติดตั้งการร้องขอนี้จะเป็นส่วนเริ่มต้นก่อนเข้าสู่ส่วนวิกฤติ และตามด้วยส่วนจบการทำงานซึ่งอยู่ถัดจากส่วนวิกฤติ ท้ายสุดอาจเป็นส่วนโปรแกรมที่ไม่มีความเกี่ยวข้องกับโพรเซสอื่น

การเขียนแสดงรูปแบบขั้นตอนวิธีสำหรับแก้ปัญหาส่วนวิกฤติจะมีการกำหนดเฉพาะตัวแปรที่ใช้เพื่อการประสานอย่างได้จังหวะกันเท่านั้น และบอกถึงโพรเซสเฉพาะบางโพรเซส Pi ซึ่งส่วนเริ่มต้นและส่วนจบของการทำงานในส่วนวิกฤติ จะล้อมอยู่ในกรอบรูปสี่เหลี่ยมเพื่อเน้นถึงความสำคัญของส่วนนี้ คำตอบสำหรับการแก้ปัญหาส่วนวิกฤติต้องเป็นไปตามเงื่อนไขทั้งสามข้อต่อไปนี้

3.4.1 การกีดกัน (Mutual exclusion)

ถ้าโพรเซส Pi กำลังปฏิบัติการอยู่ในส่วนวิกฤติแล้ว ต้องไม่มีโพรเซสอื่นใดกำลังอยู่ในส่วนวิกฤติของโพรเซสเหล่านั้นด้วย นั่นคือในขณะใดขณะหนึ่งมีเพียงโพรเซสเดียวเท่านั้นที่อยู่ในส่วนวิกฤติของตน



ภาพที่ 3.5 โครงสร้างภาษาโดยทั่วไปของการดำเนินการของโพรเซสปกติ

3.4.2 ความก้าวหน้า (Progress)

ถ้าไม่มีโพรเซสใดกำลังปฏิบัติการอยู่ในส่วนวิกฤติแล้วมีบาง โพรเซสเข้าสู่ส่วนวิกฤติของตน ในการเลือกว่าโพรเซสใดจะได้เข้าสู่ส่วนวิกฤติของตนต่อไปนั้น ระบบก็จะพิจารณาจากทุก ๆ โพรเซส ยกเว้นโพรเซสที่กำลังทำงานอยู่ในส่วนที่เหลือนั้น โดยการเลือกนี้ต้องไม่เลื่อนออกไปอย่างไม่มีกำหนด

3.4.3 การรออย่างมีขอบเขต (Bounded waiting)

จะต้องมีขอบเขตของเวลาในการรอ โดยเริ่มนับตั้งแต่เวลาที่โพรเซสอื่นได้รับอนุญาตให้เข้าสู่ส่วนวิกฤติไป หลังจากมีโพรเซสหนึ่งร้องขอจนมาถึงเวลาที่ระบบทำตามคำร้องขอของโพรเซสนั้น

ทั้งนี้ภายใต้ข้อสมมติว่า แต่ละโพรเซสกำลังปฏิบัติการที่ความเร็วขนาดหนึ่ง (ไม่เป็นศูนย์) แต่ไม่ให้มีการกำหนดสมมติฐานที่เกี่ยวกับความเร็วสัมพัทธ์ของโพรเซสทั้งหมด n โพรเซส จากรูปแบบขั้นตอนวิธีสำหรับการแก้ปัญหาส่วนวิกฤติและเงื่อนไขทั้งสามข้อข้างต้นที่ต้องคำนึงถึงในการแก้ปัญหาส่วนวิกฤติเป็นผลให้เกิดแนวคิดการติดตั้งใช้งานในแต่ละโพรเซส ซึ่งจะต้องติดตั้งส่วนเริ่มต้นก่อนเข้าสู่ส่วนวิกฤติเพื่อให้เกิดการกีดกันดังเงื่อนไขข้อ 1 และติดตั้งส่วนจบการทำงานเมื่อออกจากส่วนวิกฤติเพื่อปล่อยให้โพรเซสอื่นได้มีโอกาสเข้าใช้ส่วนวิกฤติของตนเองบ้าง นั่นคือ เป็นไปตามเงื่อนไขข้อที่ 2 และ 3 ซึ่งในโครงสร้างของแต่ละโพรเซสที่มีส่วนเริ่มต้นก่อนเข้าสู่ส่วนวิกฤติ มีจุดประสงค์เพื่อให้โพรเซสทั้งหลายตรวจสอบว่า ขณะนั้นมีโพรเซสใดกำลังอยู่ในส่วนวิกฤติของโพรเซสนั้นหรือไม่ ถ้าไม่มีโพรเซสที่ต้องการเข้าสู่ส่วนวิกฤติของตนเองก็สามารถเข้าทำงานได้ทันที ถ้ามีโพรเซสเหล่านั้นก็ไม่สามารถเข้าสู่ส่วนวิกฤติได้ตามต้องการ ดั้งนั้นปัญหาที่เกิดขึ้นคือ โพรเซสที่เข้าสู่ส่วนวิกฤติไม่ได้ในขณะนั้นจะดำเนินการรอต่อไปอย่างไร ทั้งนี้แนวคิดในการแก้ปัญหาโดยทั่วไปมี 2 รูปแบบคือ

3.4.3.1 การรอแบบไม่ว่าง (Preemptive kernels)

ในแต่ละโพรเซสมีการกำหนดเงื่อนไขที่ต้องการใช้ในการตรวจสอบก่อนเข้าสู่ส่วนวิกฤติ ซึ่งถ้าหากไม่สามารถผ่านเงื่อนไขนี้ไปได้ โพรเซสไม่สามารถเข้าสู่ส่วนวิกฤติของตนเองได้ และวนเวียนตรวจสอบเงื่อนไขนั้นอยู่ตลอดเวลาจนกว่าจะได้เข้าสู่ส่วนวิกฤตินั่นคือ ในระหว่างรอโพรเซสนั้นยังต้องทำคำสั่งทำซ้ำและทดสอบเงื่อนไขอยู่เรื่อย ๆ

3.4.3.2 การขัด (Non-preemptive kernels)

ในการรอแบบไม่ว่างนั้นไม่ได้ทำให้ระบบเกิดผลงานใด ๆ ขึ้นมา จึงเป็นการใช้วงรอบการทำงานของหน่วยประมวลผลกลางที่ไม่ได้ประโยชน์ และยังต้องแย่งชิงกันเข้าใช้หน่วยประมวลผลกลางระหว่างโพรเซสร่วมอื่น ๆ ที่กำลังวนลูปรอเข้าสู่ส่วนวิกฤติ รวมทั้งโพรเซสที่เป็นแบบอิสระทั้งหลายในระบบด้วย ดังนั้นจึงได้มีการพัฒนาให้ทำการบล็อกโพรเซสที่ต้องการเข้าสู่ส่วนวิกฤติแต่ยังไม่อาจเข้าได้เนื่องจากมีโพรเซสอื่นกำลังทำงานอยู่ในส่วนวิกฤติ ซึ่งการบล็อกนี้ระบบจะนำโพรเซสเหล่านั้นไปรออยู่ในแถวของแต่ละเงื่อนไขและตัวแปรที่ใช้ในการตรวจสอบก่อนเข้าสู่ส่วนวิกฤติ ทำให้โพรเซสเหล่านี้ไม่ต้องเข้ามาแย่งชิงกันใช้ซีพียู และเมื่อใดที่ไม่มีโพรเซสใด ๆ อยู่ในส่วนวิกฤตินั่นคือ โพรเซสล่าสุดที่เข้าใช้ส่วนวิกฤติของตนเองได้ออกจากส่วนวิกฤตินั้นแล้ว จะต้องปลุกหรือใช้คำสั่งบอกให้ระบบทราบ เพื่อให้นำโพรเซสที่รออยู่ในแถวคอยตามเงื่อนไขที่รอได้กลับเข้ามาทำงานในส่วนวิกฤติของตนเองต่อไป เครื่องมือที่มีการติดตั้งใช้งานในลักษณะนี้ได้แก่ เซมาฟอร์ มอนิเตอร์ และบริเวณวิกฤติ เป็นไปได้ที่สองโพรเซสของเคอร์เนลอยู่ในโหมดรันพร้อมกันในซีพียูที่แตกต่างกัน เพราะเหตุใดโพรเซสจึงชอบเคอร์เนลที่ทำงานก่อนมากกว่า

เคอร์เนลที่ทำงานทีหลัง การรอแบบไม่ว่างเหมาะสมสำหรับโปรแกรมแบบ Real time ซึ่งยอมให้โพรเซสแบบ Real time จองโพรเซสที่กำลังทำงานอยู่ในปัจจุบันบนเคอร์เนล นอกจากนี้การรอแบบไม่ว่างอาจตอบสนองเพิ่มเติมเนื่องจากมีความเสี่ยงน้อยกว่าโพรเซสของเคอร์เนลโหมดที่ทำงานโดยไม่มีการควบคุมโพรเซสที่มีเวลาการทำงานนานซึ่งได้ทำงานก่อน และทำให้ซีพียูอื่นต้องรอโพรเซสนี้จนกว่าจะทำงานเสร็จ ผลกระทบนี้สามารถลดลงได้ด้วยการออกแบบเคอร์เนลโค้ดที่ไม่ทำงานแบบนี้อย่างแน่นอน

3.5 การประมวลผลพร้อมกันโดยวิธีการทางซอฟต์แวร์

เป็นวิธีการทางซอฟต์แวร์ที่เข้ามาช่วยควบคุมการทำงานของระบบให้มีการประมวลผลพร้อมกัน โดยมีคุณสมบัติของการไม่เกิดร่วม มีอัลกอริทึมที่น่าสนใจคือ

1) อัลกอริทึมของเดกเกอร์ (Dekker's algorithm)
2) อัลกอริทึมของปีเตอร์สัน (Peterson's algorithm)

3.5.1 อัลกอริทึมของเดกเกอร์

ทำการพัฒนาโปรแกรมที่ประกอบด้วย 2 โพรเซส โดยสามารถทำงานพร้อมกัน และเกิดคุณสมบัติการไม่เกิดร่วม ตัวอย่างประกอบด้วยโพรเซส P0 และ P1 ที่ทำงานพร้อมกัน

```
//flag[] is boolean array; and turn is an integer
    flag[0] = false
    flag[1] = false
    turn    = 0   // or 1
```

```
P0:
   flag[0] = true;
   while (flag[1] == true) {
      if (turn ≠ 0) {
         flag[0] = false;
         while (turn ≠ 0) {
           // busy wait
         }
         flag[0] = true;
      }
   }
    // critical section
   …
   turn    = 1;
   flag[0] = false;
   // remainder section
```

```
P1:
   flag[1] = true;
   while (flag[0] == true) {
      if (turn ≠ 1) {
         flag[1] = false;
         while (turn ≠ 1) {
           // busy wait
         }
         flag[1] = true;
      }
   }
    // critical section
   …
   turn    = 0;
   flag[1] = false;
   // remainder section
```

ภาพที่ 3.6 โครงสร้างภาษาที่ใช้อัลกอริทึมของเดกเกอร์ของโพรเซส P0 และ P1

จากโค้ดตัวอย่างตามภาพที่ 3.6 เมื่อเริ่มทำงานตัวแปร turn ถูกกำหนดค่าให้เป็น 0 ทำให้โพรเซส P0 สามารถเข้าไปทำงานในส่วนวิกฤติส่วนโพรเซส P1 จะต้องวนลูปรอจนกระทั่งโพรเซส P0 ทำงานในส่วนวิกฤติเสร็จ และกำหนดให้ turn มีค่าเป็น 1 โพรเซส P1 จึงจะออกจากการวนลูป และสามารถเข้าไปทำงานในส่วนวิกฤติได้ จะเห็นว่าโพรเซส P0 และ P1 สามารถทำงานพร้อม ๆ กัน และผลัดกันเข้าไปทำงานในส่วนวิกฤติ นั่นคือเกิดคุณสมบัติการไม่เกิดร่วม แต่การทำงานของอัลกอริทึมนี้อาจเกิดปัญหา Busy Waiting ในกรณีที่โพรเซส P0 กำลังทำงานในส่วนวิกฤติ และเกิดหมดเวลาการทำงาน (Timeout) ก่อนที่ P0 จะเปลี่ยนค่าของตัวแปร turn ทำให้ P1 ไม่สามารถเข้าไปทำงานในส่วนวิกฤติได้ แม้ว่าจะมีโอกาสเข้าไปทำงานในส่วนวิกฤติ (เพราะ P0 หมดเวลาการทำงาน) ทำให้ P1 ต้องวนลูปรอตลอดช่วงเวลาการทำงาน

นอกจากนี้ยังอาจเกิดปัญหา Lock Step Synchronization ซึ่งหมายถึง การที่โพรเซสใด ๆ จะเข้าไปทำงานในส่วนวิกฤติ จะต้องทำงานเรียงตามลำดับ เช่น ต้องมีการทำงานเรียงลำดับ P0, P1, P0, P1, Pn แม้ว่าบางโพรเซสอาจไม่ต้องการเข้าไปทำงานในส่วนวิกฤติ

3.5.2 อัลกอริทึมปีเตอร์สันโซลูชั่น (Peterson's Solution)

พื้นฐานของการแก้ไขปัญหาซอฟต์แวร์ที่เป็นที่นิยมสำหรับปัญหาส่วนวิกฤติที่รู้จักกันในชื่อปีเตอร์สันโซลูชั่น เนื่องจากสถาปัตยกรรมคอมพิวเตอร์ที่ทันสมัยทำงานโดยใช้คำสั่งภาษาเครื่องพื้นฐาน เช่น โหลดและจัดการ ซึ่งไม่มีการรับประกันว่าอัลกอริทึมปีเตอร์สันโซลูชั่นจะทำงานได้อย่างถูกต้องในสถาปัตยกรรมนี้ อย่างไรก็ตามการแก้ไขปัญหาแบบนี้ให้คำอธิบายอัลกอริทึมที่ดีของการแก้ปัญหาส่วนวิกฤติ และแสดงบางความสัมพันธ์ที่ซับซ้อนในการออกแบบซอฟต์แวร์ อัลกอริทึมนี้จะจำกัดโพรเซสสองโพรเซสที่สลับการดำเนินการระหว่างส่วนวิกฤติและส่วนที่เหลือ ดังภาพที่ 3.7 ที่มีโพรเซสหมายเลข P0 และ P1 และโพรเซสทั้งสองต้องการข้อมูลสองค่าเพื่อใช้ร่วมกันระหว่างกัน

```
//flag[] is boolean array; and turn is an integer
flag[0]   = false;
flag[1]   = false;
turn;
```

```
P0: flag[0] = true;
    turn = 1;
    while (flag[1] == true && turn == 1)
    {
        // busy wait
    }
    // critical section
    ...
    // end of critical section
    flag[0] = false;
```

```
P1: flag[1] = true;
    turn = 0;
    while (flag[0] == true && turn == 0)
    {
        // busy wait
    }
    // critical section
    ...
    // end of critical section
    flag[1] = false;
```

ภาพที่ 3.7 โครงสร้างภาษาที่ใช้อัลกอริทึมของปีเตอร์สันโซลูชั่นของโพรเซส P0 และ P1

จากโค้ดตัวอย่างตามภาพที่ 3.7 เมื่อเริ่มการทำงาน ตัวแปร flag[0] และ flag[1] ถูกกำหนดค่าให้เป็นเท็จ ถ้าโพรเซสใดต้องการเข้าไปทำงานในส่วนวิกฤติ ให้ทำการกำหนดค่าของตัวแปร flag ของโพรเซสเป็นจริง เช่น ถ้าโพรเซส P0 ต้องการเข้าไปทำงานในส่วนวิกฤติ จะกำหนด flag[0] มีค่าเป็นจริง (true) ทำให้โพรเซส P1 ต้องวนลูปรอ หลังจากที่ P0 ทำงานในส่วนวิกฤติเสร็จแล้ว จะกำหนดค่าของ flag[0] เป็นเท็จ ทำให้โพรเซส P1 หลุดออกจากลูป และสามารถเข้าไปทำงานในส่วนวิกฤติได้ อัลกอริทึมนี้สามารถแก้ปัญหาการไม่เกิดร่วมสำหรับ 2 โพรเซส ที่สามารถพัฒนาให้มีการทำงานเป็น n โพรเซสได้

3.6 ฮาร์ดแวร์ประสานเวลา

การประมวลผลพร้อมกัน และมีคุณสมบัติการไม่เกิดร่วม โดยวิธีการทางฮาร์ดแวร์ สามารถทำได้หลายวิธี ดังนี้

1) การปิดกั้น (lock)
2) การปิดทางขัดจังหวะ (disabling interrupt)
3) คำสั่งทดสอบและเซ็ต (test and set instruction)

3.6.1 การปิดกั้น (Lock)

เมื่อโพรเซสต้องการเข้าไปทำงานในส่วนวิกฤติ โพรเซสจะต้องตรวจสอบก่อนว่า ส่วนวิกฤติถูกล็อค หรือถูกปิดกั้นหรือไม่ หากพบว่าส่วนวิกฤติไม่ถูกล็อค แสดงว่าขณะนั้นไม่มีโพรเซสใดกำลังทำงานในส่วนวิกฤติ โพรเซสสามารถเข้าไปทำงานในส่วนวิกฤติได้ โดยก่อนที่จะเข้าไปทำงานในส่วนวิกฤติ โพรเซสต้องใส่ล็อคเพื่อเป็นการปิดกั้นไม่ให้โพรเซสอื่นเข้าไปทำงานในส่วนวิกฤติได้ จนกระทั่งโพรเซสทำงานในส่วนวิกฤติเสร็จเรียบร้อยจึงปลดล็อค เพื่อเปิดโอกาสให้โพรเซสอื่นสามารถเข้าทำงานในส่วนวิกฤติได้

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

ภาพที่ 3.8 รูปแบบของคำสั่ง Lock

การทำงานด้วยวิธีนี้ ระบบสามารถมีมากกว่าหนึ่งโพรเซสที่ทำงานพร้อมกันได้ โดยมีคุณสมบัติการไม่เกิดร่วม แต่อาจจะเกิดปัญหาในกรณีที่มีมากกว่าหนึ่งโพรเซสต้องการเข้าไปทำงานในส่วนวิกฤติพร้อมกัน และทดสอบล็อคแล้วพบว่าล็อคว่าง ก็สามารถเข้าไปทำงานในส่วนวิกฤติได้ ทำให้มีมากกว่าหนึ่งโพรเซสเข้าไปทำงานในส่วนวิกฤติพร้อมกัน

3.6.2 การปิดทางขัดจังหวะ (Disable Interrupt)

สาเหตุหนึ่งที่ทำให้เกิดปัญหาในการทำงานพร้อมกันหลายโพรเซส คือ ขณะที่โพรเซสหนึ่งกำลังทำงานในส่วนวิกฤติ อาจมีโพรเซสอื่นขอขัดจังหวะ เพื่อให้ได้โอกาสเข้าไปทำงานในส่วนวิกฤติ เป็นผลให้โพรเซสที่กำลังทำงานในส่วนวิกฤติต้องหยุดทำงาน โดยที่การทำงานยังไม่เสร็จสมบูรณ์ และสลับให้โพรเซสอื่นเข้าไปทำงานในส่วนวิกฤติแทน โดยที่โพรเซสที่เข้าทำงานภายหลัง อาจแทรกแซงการทำงานหรือเปลี่ยนแปลงค่า หรือมีการประมวลผลใด ๆ ที่ส่งผลกระทบต่อผลลัพธ์ในการทำงานของโพรเซสแรก

บนระบบโพรเซสเซอร์เดียวสามารถระงับการใช้ Interrupts สามารถรับประกันได้ว่าลำดับของชุดคำสั่งจะถูกกระทำการโดยปราศจาการบังคับให้ออกจากเวลาของโพรเซสเซอร์ แต่ในการทำงานบนระบบมัลติโพรเซสเซอร์ไม่สามารถใช้วิธีนี้ได้ การห้ามขัดจังหวะบนโพรเซสเซอร์หลายตัวนั้นใช้เวลานานมากในการส่งข่าวสารไปยังโพรเซสเซอร์ทุกตัว การส่งข่าวสารจะก่อให้เกิดการหน่วงเวลาไปยังส่วนวิกฤติของทุกโพรเซสเซอร์ซึ่งส่งผลให้ประสิทธิภาพของระบบลดลง เมื่อมีการบันทึกเวลาทุกครั้งที่ถูกเปลี่ยนค่าโดยการขัดจังหวะ

แต่การทำงานด้วยวิธีนี้ไม่เป็นที่นิยม เนื่องจากเป็นการไม่ถูกต้องที่ระบบอนุญาตให้มีโพรเซสปิดกั้นการขัดจังหวะของระบบ ในกรณีที่โพรเซสปิดกั้นการขัดจังหวะ แล้วไม่ได้ปล่อยการขัดจังหวะกลับคืนให้ระบบ อาจทำให้ปัญหาอื่นตามมา การปิดทางขัดจังหวะจะมีผลให้ได้ระบบที่เป็นการประมวลผลพร้อมกันและมีคุณสมบัติการไม่เกิดร่วม เฉพาะกรณีที่เป็นการประมวลผลบนซีพียูเดียว แต่กรณีที่ประมวลผลโดยใช้ระบบมัลติโพรเซสเซอร์ แม้ว่าโพรเซสได้ปิดทางขัดจังหวะแล้ว โพรเซสยังมีโอกาสถูกขัดจังหวะการทำงานของโพรเซสจากโพรเซสเซอร์อื่นได้

3.6.3 คำสั่งทดสอบและเซต (Test and Set Instruction)

วิธีนี้พยายามแก้ปัญหาของวิธีการปิดกั้น และการปิดทางขัดจังหวะ ด้วยการใช้คำสั่งทดสอบและเซต ซึ่งเป็นคำสั่งระดับฮาร์ดแวร์ โดยการทำงานของคำสั่งไม่สามารถถูกขัดจังหวะได้ การทำงานของวิธีนี้คือ โพรเซสที่ต้องการเข้าไปทำงานในส่วนวิกฤติจะต้องเรียกใช้คำสั่งทดสอบและเซต เพื่อตรวจสอบว่ามีโพรเซสใดทำงานอยู่ในส่วนวิกฤติหรือไม่ หากพบว่าขณะนั้นไม่มีโพรเซสใดทำงานในส่วนวิกฤติ โพรเซสจะเซตค่าล็อค เพื่อเป็นการป้องกันไม่ให้โพรเซสอื่นเข้าไปทำงานในส่วนวิกฤติพร้อมกันได้ ทำให้ระบบสามารถมีโพรเซสจำนวนมากทำงานร่วมกัน โดยมีคุณสมบัติการไม่เกิดร่วม

```
boolean test_and_set (boolean *target)
   {
       boolean rv = *target;
       *target = TRUE;
       return rv:
   }
```

ภาพที่ 3.9 รูปแบบของคำสั่ง Test-and-Set

ในหลาย ๆ ระบบคอมพิวเตอร์ที่ทันสมัยให้คำสั่งในเรื่องฮาร์ดแวร์พิเศษที่ช่วยให้สามารถเลือกที่จะทดสอบและปรับปรุงแก้ไขเนื้อหาของคำสั่ง หรือเพื่อที่จะสลับเนื้อหาของคำสั่งสองคำสั่งแบบ Atomically ที่เป็นดังหนึ่ง Uninterruptible Unit เราสามารถใช้คำสั่งพิเศษเหล่านี้เพื่อแก้ปัญหาส่วนวิกฤติ ในลักษณะเชิงสัมพัทธ์ง่าย ๆ แทนที่จะอภิปรายเฉพาะหนึ่งคำสั่งสำหรับเฉพาะเครื่องหนึ่ง

คำสั่ง TestAndSet( ) สามารถกำหนดไว้ตามที่แสดงโค้ดภาษาซีในภาพที่ 3.9 ลักษณะที่สำคัญคือจะทำงานคำสั่งนี้แบบ Atomically ดังนั้นหากคำสั่ง TestAndSet( ) ที่ทำงานในเวลาเดียวกัน แต่ทำบนซีพียูที่แตกต่างกัน คำสั่งนี้จะถูกรันตามลำดับในบางลำดับแบบสุ่ม หากเครื่องสนับสนุนคำสั่ง TestAndSet( ) แล้วเราสามารถทำการเอาออกร่วมกันโดยมีการแจ้งการล็อคตัวแปร Boolean ในการเริ่มต้นจะเป็นเท็จ โครงสร้างของโพรเซส Pi จะแสดงในภาพที่ 3.10

```
// Shared boolean variable lock, initialized to FALSE
// Solution:

do {
      while (test_and_set(&lock)) ; /* do nothing */
                /* critical section */
      lock = false;
                /* remainder section */
} while (true);
```

ภาพที่ 3.10 แสดงโครงสร้างโค้ดภาษาซีของโพรเซสในรูปแบบ Test-and-Set

3.6.4 Swap Instruction

คำสั่ง Swap( ) จะตรงกันข้ามกับคำสั่ง TestAndSet( ) คำสั่ง Swap( ) จะทำงานในเนื้อหาของคำสองคำ กำหนดไว้ตามที่แสดงในภาพที่ 3.11 เช่นเดียวกับคำสั่ง TestAndSet( ) จะรัน Atomically หากเครื่องสนับสนุนคำสั่ง Swap( ) แล้วการเอาออกทั้งสองฝ่ายโดยมีเงื่อนไขดังนี้ การล็อคตัวแปร Global Boolean จะถูกประกาศและทำให้เป็นเท็จในเริ่มต้น นอกจากนี้แต่ละโพรเซสมีคีย์ของตัวแปร Local Boolean โครงสร้างของโพรเซส Pi จะปรากฏในภาพที่ 3.12

```
Void Swap (boolean *a, boolean *b)  {
            boolean temp = *a;
            *a = *b;
            *b = temp;
      }
```

ภาพที่ 3.11 แสดงฟังก์ชั่น Swap( ) ในโครงสร้างโค้ดภาษาซี

```
do {
        key = TRUE ;
            while (key = = TRUE )
                    Swap (&lock, &key) ,
                    // critical section
                  Lock = FALSE;
                    // remainder section

} while  (TRUE) ;
```

ภาพที่ 3.12 แสดงโครงสร้างโค้ดภาษาซีการกีดกันโพรเซสโดยการเรียกใช้ฟังก์ชั่น Swap( )

สำหรับนักออกแบบฮาร์ดแวร์ การใช้กลุ่มคำสั่ง TestAndSet( ) บนเครื่องประมวลผลจำนวนมากไม่เป็นงานที่สำคัญนัก การทำให้เป็นผลสำเร็จถูกอธิบายไว้ในหนังสือสถาปัตยกรรมคอมพิวเตอร์

3.7 โครงสร้างพื้นฐานสำหรับการซินโครไนซ์

โพรเซสที่ทำงานร่วมกับโพรเซสอื่นในการดำเนินการ จำเป็นต้องอาศัยการซินโครไนซ์ที่เหมาะสมเพื่อให้การทำงานถูกต้องตามที่กล่าวมาแล้วนั้น ในการประสานเวลาโพรเซสไม่ใช่เรื่องที่ทำได้ง่าย แม้ว่าจะมีคำสั่งทดสอบและเซต มาเอื้ออำนวยให้ต่อการให้ได้จังหวะกัน แต่ก็ยังไม่สะดวกในการพัฒนาทางด้านนี้ จึงมีโครงสร้างและสิ่งที่อาจจะเกิดขึ้นในการประสานเวลาที่เรียกว่า คำสั่งพื้นฐาน (Primitive) ที่น่าสนใจดังนี้

3.7.1 เซมาฟอร์ (Semaphore)

วิธีการแก้ปัญหาเขตวิกฤตที่กล่าวมาแล้วทั้งหมด ยังคงไม่สะดวกในการใช้งานกับปัญหาที่ซับซ้อนมากขึ้น เราจึงต้องหาวิธีใหม่ เรียกว่า เซมาฟอร์ ที่ไดจสตร้าได้เสนอขึ้นในปี ค.ศ. 1965 ตามนิยามเซมาฟอร์แทนได้ด้วย S ซึ่งมีค่าเป็นเลขจำนวนเต็ม (Integer) การเรียกใช้งานจะทำได้ผ่านทาง 2 คำสั่ง คือ Signal (แทนด้วย V ย่อมาจากคำว่า Verhogen) และ Wait (แทนด้วย P ย่อมาจากคำว่า Proberen) ดังโครงสร้างตามภาพที่ 3.13 พบว่าคำสั่ง wait(S) เป็นการทดสอบค่าของ S และจะทำการลดค่าลงเมื่อตรวจสอบแล้ว พบว่ามีค่ามากกว่า 0 ในทางตรงกันข้ามส่วนของ signal(S) เป็นการทำงานเพื่อเพิ่มค่าให้กับ S

```
// S: wait() and signal()
// Originally called P() and V()

wait (S) {
        while S <= 0
                ; // no-op
          S--;
    }
signal (S) {
```

ภาพที่ 3.13 โครงสร้างของคำสั่ง Signal และ Wait ใน Semaphore

สมมติ Operator ที่จะทำงานกับเซมาฟอร์ 2 ตัว คือ คำสั่ง wait(S) ใช้ตรวจสอบค่า รอจน S มีค่ามากกว่า 0 แล้วจึงลดค่าของ S ลง 1 แล้วทำคำสั่งถัดไป คำสั่ง signal(S) ใช้เพิ่มค่า S ขึ้น 1 แล้วทำคำสั่งถัดไปจะต้องใช้คู่กันเสมอ ในที่นี้ S คือ เซมาฟอร์ คือ ตัวแปรชนิดจำนวนเต็มที่มีค่าไม่น้อยกว่าศูนย์ คำสั่ง wait(S) จะกั้นโพรเซสไว้หากค่าของ S กลายเป็นศูนย์ คำสั่งทั้งสองนี้เป็นประเภทไม่อาจแบ่งแยกได้ คือไม่สามารถถูกขัดได้ด้วยโพรเซสอื่น

เซมาฟอร์ เป็นเครื่องมือประสานเวลาที่ไม่ต้องการเวลารอคอย เซมาฟอร์ S เป็นตัวแปรจำนวนเต็ม ช่วยลดความซับซ้อน สามารถเข้าถึงได้ผ่านคำสั่งปฏิบัติการมาตรฐาน 2 คำสั่งปฏิบัติการแบบภาวะครบหน่วย (Atomic operation) ได้แก่ Wait และ Signal ความหมายคำสั่งปฏิบัติการแบบภาวะครบหน่วย เมื่อโพรเซสหนึ่งกำลังแก้ไขค่าเซมาฟอร์ จะต้องไม่มีโพรเซสอื่นที่สามารถเข้ามาแก้ไขค่าเซมาฟอร์เดียวกันได้ในเวลาเดียวกัน กรณีของ wait(S) การทดสอบค่าจำนวนเต็มของ S (S <= 0) และการแก้ไขค่า (S--) จะต้องกระทำการโดยปราศจากขัดจังหวะ

เซมาฟอร์ เป็นอัลกอริทึมหนึ่งที่ง่ายต่อการจัดการทรัพยากรแก่โปรเซส โดยเซมาฟอร์นี้จะเปรียบเสมือนการหยิบและใส่ลูกหินในขวดโหล กล่าวคือ จำนวนลูกหินในขวดโหลจะแทนค่าของเซมาฟอร์ในขณะใดขณะหนึ่ง การที่โปรเซสต้องการทรัพยากรก็เปรียบได้กับการหยิบลูกหิน ถ้ามีลูกหินที่ต้องการโปรเซสนั้นก็ทำการหยิบไปลูกหนึ่ง ก็คือโปรเซสนั้นทำคำสั่ง p จากนั้นก็ทำงานของตนต่อไปแต่ถ้าในกรณีที่ไม่มีลูกหินในขวดโหลเลย โปรเซสนั้นต้องรอ (wait) ให้มีโปรเซสอื่นมาใส่ลูกหินลงไปในขวดโหล ซึ่งก็คือการทำคำสั่ง v ซึ่งเป็นการส่งสัญญาณ (signal) นั่นเอง ต่อจากนั้นเมื่อมีโปรเซสหนึ่งใส่ลูกหินลงไปแล้วก็ไปทำงานของตนต่อไป และถ้าโปรเซสใดๆ รอลูกหินนั้นอยู่ ก็หยิบไปใช้งานได้ แต่ถ้าไม่มีโปรเซสใดๆ รอหยิบ ก็ไม่มีอะไรเกิดขึ้น ในกรณีที่มีหลายๆ โปรเซสรอการหยิบลูกหินอยู่ ก็จะต้องมีเกณฑ์เพื่อเลือกสรรให้โปรเซสใดโปรเซสหนึ่งหยิบลูกหินอย่างยุติธรรม วิธีที่ง่ายที่สุด เช่น วิธีใครมาก่อนได้ก่อน (First-Come-First-Served)

3.7.2 การใช้งานเซมาฟอร์

สามารถนำเซมาฟอร์ ไปใช้ในการแก้ไขปัญหาส่วนวิกฤตของโพรเซส n ตัว โพรเซส n ตัวจะร่วมใช้ เซมาฟอร์ ที่ชื่อ mutex (ใช้แทน Mutual exclusion) เริ่มการทำงานที่ 1 โดยที่แต่ละโพรเซส Pi ให้มีโครงสร้างดังภาพที่ 3.14

```
do {
        wait(mutex);
                critical section
        signal(mutex);
                remainder section
        } while (true);
```

ภาพที่ 3.14 การสร้าง Mutual Exclusion ด้วย Semaphore

เราสามารถใช้เซมาฟอร์ในการแก้ไขปัญหาของการทำงาน ที่ให้โพรเซสทำงานประสานกันที่หลากหลาย เช่น เมื่อพิจารณาโพรเซส 2 ตัวที่ทำงานพร้อมกันให้โพรเซส 1 มีสภาวะการทำงานที่ S1 และโพรเซส 2 มีสภาวะการทำงานที่ S2 สมมติเราต้องการให้ S2 ทำงานได้ก็ต่อเมื่อ S1 ทำงานเสร็จ เราสามารถสร้างวิธีการดังกล่าวได้โดยการใช้โพรเซส 1 และ 2 ใช้เซมาฟอร์ ที่ชื่อ synch ร่วมกัน แล้วเริ่มการทำงานที่ศูนย์ แล้วทำการแทรกเข้าไปในสภาวะการทำงาน S1 ในโพรเซส 1 ในทำนองเดียวกันก็นำไปไว้ที่สภาวะการทำงาน S2 ในโพรเซส 2 นั้นจะทำให้สภาวะ S2 ในโพรเซส 2 ทำงานได้ก็ต่อเมื่อโพรเซส 1 ให้สัญญาณ Signal(synch) เมื่อทำงานสภาวะ S1 เสร็จแล้วเท่านั้น

3.7.3 การสร้างเซมาฟอร์ (Semaphore Implementation)

ข้อเสียของการแก้ไขปัญหาการกีดกันนั่นก็คือ ถ้าหากมีโพรเซสใดกำลังทำงานในส่วนวิกฤตอยู่นั้น แล้วมีโพรเซสอื่นต้องการเข้าทำงานก็จะต้องวนลูปรอเข้าทำงานไปเรื่อย ๆ เรียกว่า “การวนเวียนรอคอย” (Busy waiting) การรอเช่นนี้ทำให้เสียเวลาอยู่ในสภาวะของ Waiting แทนที่จะสามารถทำอย่างอื่นได้ ทำให้ประสิทธิภาพลดลง ย่อมเป็นเหตุให้เวลาของหน่วยประมวลผลกลางเสียไปโดยเปล่าประโยชน์ เวลาเหล่านี้อาจยกให้กระบวนการอื่นได้ใช้ประโยชน์คุ้มค่ากว่า เหตุการณ์เช่นนี้เรียกว่า Spinlock คือ โพรเซสวนเวียนทำงาน (Spin) อยู่ ขณะที่กำลังรอคอยให้เปิดล็อค ในระบบแบบ Multiprocessor จะเกิดข้อดีเนื่องจากไม่ต้องทำ Context Switch ดังนั้นถ้าการ Lock อยู่ในช่วงเวลาที่สั้น ๆ Spinlock ก็จะเป็นประโยชน์อย่างยิ่ง

เราสามารถปรับปรุงนิยามของคำสั่ง wait() และ signal() ใหม่ เพื่อให้ไม่มีปัญหาการวนเวียนรอคอยเมื่อโพรเซสทำคำสั่ง wait() และพบว่าค่าของเซมาฟอร์น้อยกว่าหรือเท่ากับศูนย์ โพรเซสจะต้องรอคอย โดยการหยุดชั่วขณะ (Block) แทนที่จะวนเวียนรอคอย การหยุดชั่วขณะนี้โพรเซสจะเข้าไปรอในแถวคอย (Waiting queue) ที่เกี่ยวข้องกับเซมาฟอร์นั้น ๆ และเปลี่ยนสถานะเป็นสถานะรอคอย (Waiting

state) การควบคุมจะถูกย้ายกลับไปที่ตัวจัดตารางการทำงานของหน่วยประมวลผลกลาง เพื่อจัดให้โพรเซสอื่นทำงานต่อไป

โพรเซสที่หยุดชั่วขณะโดยรอเซมาฟอร์ S นี้ จะสามารถดำเนินต่อได้ เมื่อมีโพรเซสอื่นทำคำสั่ง signal() กับเซมาฟอร์ S ภายในคำสั่ง signal จะมีการปลุก (Wakeup) ให้กระบวนการที่หยุดชั่วขณะกลับดำเนินการต่อได้ โดยเปลี่ยนสถานะของกระบวนการนั้นเป็นสถานะพร้อม (Ready state) แล้วย้ายไปต่อแถวพร้อม

การสร้างเซมาฟอร์ตามนิยามใหม่นี้ กำหนดให้เซมาฟอร์เป็นตัวแปรประเภท Structure ในภาษาซี

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

ตัวแปรเซมาฟอร์จะประกอบด้วย ตัวเลขจำนวนเต็ม (ตัวแปร value) และแถวคอยของโพรเซส (ตัวแปร L) เมื่อโพรเซสหนึ่งต้องคอยเซมาฟอร์นี้ มันก็จะถูกใส่ชื่อลงในแถวคอย L การใช้คำสั่ง signal จะดึงโพรเซสออกจากหัวแถวคอยนี้ และปลุกโพรเซสที่ได้ให้กลับไปทำงานต่อ เราจึงแก้ไขโปรแกรมคำสั่งเป็นดังนี้

```
void wait(semaphore S) {
        S.value - -;
                if (S.value < 0) {
                        add this process to S.L (list);
                        block();
                  }
    }
void signal(semaphore S) {
        S.value ++;
                if (S.value <= 0) {
                  remove a process P from S.L (list);
                  wakeup(P);
                }
    }
```

ภาพที่ 3.15 แสดงตัวอย่าง Semaphore เป็นโครงสร้างโค้ดในภาษาซี

คำสั่ง block จะทำให้โพรเซสที่ทำคำสั่งนี้หยุดชั่วขณะ คำสั่ง wakeup(P) จะปลุกโพรเซส P ให้กลับทำงานต่อ จะสังเกตได้ว่า นิยามดั้งเดิมของเซมาฟอร์ซึ่งมีการวนเวียนรอคอยนั้น ค่าของเซมาฟอร์

จะไม่ติดลบ แต่ในนิยามใหม่นี้ค่าของเซมาฟอร์มีค่าเป็นลบได้ ค่าที่เป็นลบแสดงถึงจำนวนกระบวนการที่รอคอยเซมาฟอร์ตัวนั้น ๆ ด้วย

เราต้องไม่ให้มีโพรเซสสองตัวทำคำสั่ง wait และ signal บนเซมาฟอร์ตัวเดียวกันในเวลาเดียวกัน ซึ่งกรณีนี้ก็คือ ปัญหาเขตวิกฤตแบบหนึ่งนั่นเอง โดยสามารถเลือกวิธีจัดการได้ 2 วิธี คือ

1) ถ้าเป็นระบบที่มีหน่วยประมวลผลเพียงตัวเดียว เราก็เพียงแต่ห้ามขัดจังหวะขณะที่กำลังทำงานคำสั่ง wait และ signal ดังนั้นโพรเซสอื่นจะไม่สามารถมาขัดจังหวะการทำงานได้

2) ถ้าเป็นระบบที่มีหน่วยประมวลผลหลายตัว การห้ามขัดจังหวะจะไม่เพียงพอ เพราะโพรเซสอื่นอาจทำงานอยู่ในหน่วยประมวลผลคนละตัวกับโพรเซสที่ถูกห้ามขัดจังหวะ ดังนั้นอาจจัดการโดยใช้ขั้นตอนวิธีการแก้ไขส่วนวิกฤต แล้วเอาคำสั่ง wait และ signal ใส่ไว้ในเขตวิกฤตก็จะสามารถป้องกันได้

3.7.4 เซมาฟอร์แบบทวิภาค (Binary Semaphore)

โครงสร้างของเซมาฟอร์ ที่กล่าวมาข้างต้นเป็นลักษณะของเซมาฟอร์แบบนับ (Counting semaphore) ส่วนในเซมาฟอร์แบบทวิภาคจะกำหนดให้ใช้ตัวแปรจำนวนเต็มได้ 2 ค่าคือ 0 และ 1 เท่านั้น ทำให้สามารถสร้างได้ง่ายกว่าเซมาฟอร์แบบนับ ซึ่งสามารถสร้างเซมาฟอร์แบบนับให้อยู่ในรูปแบบทวิภาคได้ โดยการใช้โครงสร้างตามภาพที่ 3.16 เริ่มด้วยค่า S1, S3 =1 และ S2 = 0 แล้วใช้งานตามภาพที่ 3.17

```
Var     S1: binary-semaphore;
        S2: binary-semaphore;
        S3: binary-semaphore;
        C: integer;
```

ภาพที่ 3.16 โครงสร้างของการใช้ Semaphore แบบทวิภาค

- *wait* operation

```
wait(S3);
wait(S1);
C := C − 1;
if C < 0
then begin
        signal(S1);
        wait(S2);
     end
else signal(S1);
signal(S3);
```

- *signal* operation

```
wait(S1);
C := C + 1;
if C ≤ 0 then signal(S2);
signal(S1);
```

ภาพที่ 3.17 รูปแบบของการใช้งาน Semaphore แบบทวิภาค

การทำงานคำสั่ง wait ก่อนที่จะมีการลดค่าของเซมาฟอร์ ต้องการทำ wait(S1) ก่อนลดค่าแล้วก็ตรวจสอบว่าค่าที่ลดน้อยว่า 0 หรือไม่ ถ้าน้อยกว่าก็จะมีการปลด lock S1 ที่รออยู่ แล้วก็ lock S2 ต่อ แต่ถ้าค่าของ C มากกว่าหรือเท่ากับ 0 ก็จะปลด lock S1 อย่างเดียว ส่วนคำสั่งของ Signal จะมีการเพิ่มค่า C เมื่อมีการ wait(S1) ไปแล้ว ซึ่งถ้าค่า C ที่เพิ่มน้อยกว่าหรือเท่ากับ 0 ก็จะส่งสัญญาณไปปลด lock S2 แล้วก็จึงปลด lock S1 การทำงานของสภาวะ S1 แทนโพรเซส 1 ส่วน S2 แทนโพรเซส 0 นั่นเอง ดังนั้นทั้งหมดจึงเป็นการเลือกการเข้าทำงานส่วนวิกฤตของสองโพรเซส

3.8 การติดตายและอดตาย

การสร้างเซมาฟอร์ด้วยการให้โพรเซสที่รอการทำงานไปอยู่ในคิว อาจส่งผลให้เกิดเหตุการณ์ที่สองโพรเซสหรือมากกว่านั้นรอแบบไม่มีที่สิ้นสุด เนื่องจากโพรเซสไปรอการทำงานของโพรเซสที่ยังรอการทำงานด้วยเช่นกัน ลักษณะเช่นนี้เรียกว่าการติดตาย (deadlock)

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*);	*signal*(*S*);

ภาพที่ 3.18 สถานการณ์ในการใช้งาน 2 โพรเซสแบบ Semaphore และเกิด Deadlock

สถานการณ์ติดตายสามารถอธิบายได้ดังภาพที่ 3.18 ซึ่งมีโพรเซสทำงานอยู่ 2 โพรเซสพร้อมกันคือ P0 และ P1 โดยในแต่ละโพรเซสก็มีการเข้าใช้เซมาฟอร์ 2 ตัวคือ S และ Q โดยให้ค่าเป็น 1 สมมติว่าโพรเซส P0 ทำงาน wait(s) พร้อมกับที่โพรเซส P1 ทำงาน wait(Q) เมื่อโพรเซส P0 ทำงาน wait(Q) มันจะต้องรอจนกว่าโพรเซส P1 ทำการ signal(Q) ในทำนองเดียวกันนั้นเมื่อโพรเซส P1 ทำงาน wait(S) มันก็ต้องรอจนกว่าโพรเซส P0 ทำการ signal(S) ให้ พบว่าไม่สามารถมีโพรเซสใดทำงานได้ ลักษณะการติดตายจึงเกิดขึ้น หากในการสร้างเซมาฟอร์โดยให้โพรเซสการรอจัดเก็บไว้ในคิวที่มีโครงสร้างแบบ LIFO หรือ เข้าหลังออกก่อน ก็อาจจะให้เกิดปัญหาของการติดตายและอดตายของโพรเซสที่จัดเก็บอยู่ด้านใน

3.9 ปัญหาพื้นฐานของการประสานเวลา

ปัญหาการทำงานของโพรเซส ปัญหาที่เกิดขึ้นกับระบบปฏิบัติการเป็นปัญหาที่น่าสนใจและถกเถียงกันอย่างกว้างขวาง ตลอดจนมีการวิเคราะห์โดยการใช้วิธีการด้านประสานเวลา ในที่นี้จะนำปัญหาพื้นฐานของการประสานเวลาที่เป็นที่กล่าวถึงหรือเป็นต้นแบบดังนี้

3.9.1 ปัญหาที่พักข้อมูลขนาดจำกัด (Bounded-Buffer Problem)

บางทีเรียกว่าปัญหาผู้ผลิต-ผู้บริโภค (Producer/Consumer problem) เป็นปัญหาที่นิยมใช้ในการทดสอบประสิทธิภาพของเทคนิคการประสานเวลา มีผู้ผลิตหนึ่งคนหรือมากกว่าทำการผลิตข้อมูลบางชนิด (ระเบียน, อักขระ) และใส่ลงในบัฟเฟอร์ และมีผู้บริโภคเพียงคนเดียวเท่านั้นที่สามารถหยิบข้อมูลนั้นออกจากบัฟเฟอร์ในขณะใดขณะหนึ่ง ระบบจะต้องกำหนดเงื่อนไขเพื่อป้องกันไม่ให้เกิดการทับซ้อนกันในการปฏิบัติการบนบัฟเฟอร์นั้น หมายความว่า ณ ขณะใดขณะหนึ่งจะมี (ผู้ผลิตหรือผู้บริโภค) เพียงคนเดียวเท่านั้นที่สามารถเข้าถึงบัฟเฟอร์ได้

```
//itemCount is an integer
int itemCount = 0;
```

```
procedure producer() {
    while (true) {
        item = produceItem();
        if (itemCount == BUFFER_SIZE) {
            sleep();
        }
        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}
```

```
procedure consumer() {
    while (true) {
        if (itemCount == 0) {
            sleep();
        }
        item = removeItemFromBuffer();
        itemCount = itemCount - 1;
        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }
        consumeItem(item);
    }
}
```

ภาพที่ 3.19 ตัวอย่างการใช้อัลกอริทึม Sleep and Wakeup ในการแก้ปัญหา Bounded-Buffer Problem

เนื่องจากมีสองโพรเซสใช้บัฟเฟอร์ร่วมกันโดยที่โพรเซสของผู้ผลิต (Producer process) เป็นผู้ส่ง Information ให้กับบัฟเฟอร์ และโพรเซสของผู้บริโภค (Consumer process) เป็นผู้เรียกใช้ Information นั้น และปัญหาจะเกิดเมื่อโพรเซสต้องการเพิ่ม Information ในบัฟเฟอร์ที่เต็ม (ไม่สามารถใส่เพิ่มได้) วิธีการแก้ปัญหานี้ทางหนึ่งคือโพรเซสของผู้ผลิตต้อง Sleep จนกว่าบัฟเฟอร์จะมีที่ว่าง (wakeup เมื่อบัฟเฟอร์มีที่ว่างให้เพิ่มข้อมูลได้อีก) และในทำนองเดียวกันถ้าบัฟเฟอร์ว่างโพรเซสของผู้บริโภคต้อง Sleep ว่างจนกว่าจะมีข้อมูลเพิ่มเติม (Wakeup เมื่อมีข้อมูลเข้ามาสู่บัฟเฟอร์)

3.9.2 ปัญหาผู้อ่านและผู้เขียน (The Readers/Writers Problem)

ข้อมูลในระบบที่มีการทำงานของโพรเซสพร้อมกันหลายตัวจะถูกร่วมกันใช้งาน บางโพรเซสต้องการอ่านข้อมูล ในอีกบางโพรเซสต้องการเขียนข้อมูล โดยที่ความแตกต่างของทั้งสองส่วนคือ Readers

จะเป็นโพรเซสที่ต้องการจะอ่านข้อมูลเพียงอย่างเดียว ส่วนโพรเซส Writer จะต้องการเพียงแค่การเขียนข้อมูลเท่านั้น สังเกตว่าหากมีหลาย ๆ โพรเซสทำการอ่านข้อมูลพร้อม ๆ กันจะไม่มีปัญหาใดเกิดขึ้น แต่ถ้ามีโพรเซสที่ต้องการเขียนข้อมูลพร้อม ๆ กันจะเกิดปัญหาขึ้นทันที

ปัญหานี้คือ มีพื้นที่ข้อมูลซึ่งใช้ร่วมกันได้ระหว่างกระบวนการทั้งหมด มีโพรเซสจำนวนมากที่ต้องการอ่านพื้นที่ดังกล่าวเพียงอย่างเดียว และมีโพรเซสที่ต้องการเขียนบนพื้นที่ดังกล่าวเพียงโพรเซสเดียวในเวลาเดียวกัน จึงมีการนำเอาเซมาฟอร์ช่วยในการแก้ไขปัญหาเหล่านี้ โดยให้มีโครงสร้างของการใช้ตัวแปรร่วมกันดังภาพที่ 3.20

การจัดการจะต้องเข้ากันได้กับเงื่อนไขดังต่อไปนี้

1) ผู้อ่านสามารถอ่านแฟ้มได้พร้อมกันหลายคน
2) มีผู้เขียนเพียงคนเดียวเท่านั้นที่สามารถเขียนแฟ้มได้ ณ เวลาใดเวลาหนึ่ง
3) ถ้ามีผู้เขียนรอที่จะเขียนแฟ้ม จะต้องไม่มีผู้อ่านคนใดสามารถอ่านแฟ้มนั้นได้

จากข้อกำหนดข้างต้นอาจนำไปสู่ปัญหาภาวะงูกินหาง กรณีแรกผู้เขียนอาจต้องเป็นฝ่ายรออย่างไม่รู้จบ กรณีหลังผู้อ่านอาจต้องเป็นฝ่ายรออย่างไม่รู้จบ

การ Shared Data
- Data set
- Semaphore mutex initialized to 1.
- Semaphore wrt initialized to 1.
- Integer readcount initialized to 0.

ภาพที่ 3.20 โครงสร้างตัวแปรที่ใช้งานใน Semaphore

พบว่ามีการใช้เซมาฟอร์ 2 ตัวคือ mutex และ wrt โดยให้มีการเริ่มค่าที่ 1 ส่วนตัวแปร readcount จะเป็นจำนวนเต็มที่เริ่มการทำงานที่ 0 เซมาฟอร์ wrt จะถูกใช้งานทั้งโพรเซสอ่านและเขียน ส่วนเซมาฟอร์ mutex ใช้ในการแก้ไขปัญหาด้วย Mutual Exclusion เมื่อตัวแปร readcount ถูกเปลี่ยนแปลงค่าโดยที่ตัวแปรนี้ใช้เก็บข้อมูลจำนวนโพรเซสที่กำลังทำการอ่านข้อมูลอยู่ เซมาฟอร์ wrt ใช้เพื่อทำ Mutual ในส่วนของการเขียนซึ่งถูกใช้งานโดยผู้อ่านตัวแรกหรือตัวสุดท้ายที่เข้าทำงานส่วนวิกฤต แต่จะไม่ถูกใช้ในกรณีที่ผู้อ่านตัวนั้นกำลังรอจะเข้าทำงานส่วนวิกฤต แต่มีผู้อ่านตัวอื่นค้างอยู่ในส่วนวิกฤตนั้น โค้ดของโพรเซสผู้อ่านและผู้เขียนแสดงไว้ดังภาพที่ 3.21 และ 3.22

```
do  {
        wait (wrt) ;

          //    writing is performed
        signal (wrt) ;
    } while (true)
```

ภาพที่ 3.21 รูปแบบของโพรเซสผู้เขียน

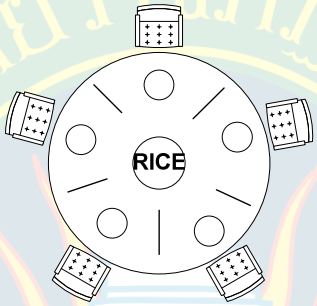
พบว่าเมื่อผู้เขียนกำลังทำงานในส่วนวิกฤต และมีโพรเซสรออยู่เป็นจำนวน n แล้ว เมื่อผู้อ่านถูกนำเข้าไปในคิวด้วย wrt ทำให้จำนวนโพรเซสที่รอลดลงเหลือ n-1 ซึ่งจะเข้าคิวด้วยเซมาฟอร์ mutex เมื่อสังเกตจะพบว่าเมื่อผู้เขียนทำ signal(wrt) อาจเกิดกรณีที่ให้โพรเซสผู้อ่านที่รออยู่เข้าทำงาน หรือจะให้สิทธิแก่โพรเซสผู้เขียนที่เหลือเพียงตัวเดียวทำงาน ทั้งนี้ผู้จัดตารางจะเป็นผู้กำหนดและจัดการ

```
do  {
    wait (mutex) ;
    readcount ++ ;

    if (readercount == 1)  wait (wrt) ;
    signal (mutex)

            // reading is performed
    wait (mutex) ;
    readcount  - - ;

    if redacount  == 0)  signal (wrt) ;
    signal (mutex) ;

  } while (true)
```

ภาพที่ 3.22 รูปแบบของโพรเซสผู้อ่าน

3.9.3 ปัญหาอาหารเย็นของนักปราชญ์ (The Dining Philosophers Problem)

ปัญหาอาหารเย็นของนักปราชญ์ โดย Edsger Dijkstra (ปี 1995) เป็นปัญหาที่ต้องการให้มีการจัดการเรื่องการใช้ทรัพยากรที่มีอยู่อย่างจำกัดร่วมกันระหว่างหลายโพรเซส ปัญหาการติดตายจะเกิดเมื่อนักปราชญ์ทั้งห้าหยิบตะเกียบคนละข้างพร้อมกัน ปัญหาการอดตาย การที่นักปราชญ์ทางด้านซ้ายและด้านขวาของนักปราชญ์ผู้โชคร้ายสลับกันทานอาหาร



ภาพที่ 3.23 ปัญหาอาหารเย็นของนักปราชญ์

ปัญหาอาหารเย็นของนักปราชญ์นี้นับเป็นปัญหาแม่แบบปัญหาหนึ่ง เพราะสามารถใช้แทนปัญหาต่าง ๆ ในระบบการทำงานแบบขนานได้มากมาย การแก้ปัญหานี้ทำได้โดยการกำหนดตัวแปรร่วมดังนี้

1) กำหนดเซมาฟอร์เป็นอาร์เรย์ชื่อ chopstick[5];
2) ให้ตะเกียบ (อาร์เรย์ chopstick) เป็นเซมาฟอร์โดยนักปราชญ์
3) หยิบตะเกียบด้วยคำสั่ง wait
4) วางตะเกียบด้วยคำสั่ง signal
5) ค่าเริ่มต้นของตะเกียบ (chopstick) = 1

```
do {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
                ...
                eat
                ...
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
                ...
                think
                ...
} while (true);
```

ภาพที่ 3.24 ตัวอย่างโครงสร้างภาษาในการแก้ปัญหาปัญหาอาหารเย็นของนักปราชญ์

วิธีนี้อาจป้องกันไม่ให้นักปราชญ์ที่นั่งติดกันกินพร้อมกันได้ แต่อาจเกิดปัญหาวงจรอับขึ้นในระบบ สมมติว่านักปราชญ์ทั้ง 5 คนหิวพร้อมกัน แล้วหยิบตะเกียบข้างซ้ายของเขาเกือบพร้อมกันหมด จากนั้นแต่ละคนจะพยายามหยิบตะเกียบข้างขวา ซึ่งจะต้องรอกันเองตลอดไป อาจแก้ปัญหาวงจรอับนี้ได้ดังนี้

1) ให้นักปราชญ์นั่งในโต๊ะได้ ไม่เกิน 4 คน

2) ให้นักปราชญ์หยิบตะเกียบได้ ก็ต่อเมื่อตะเกียบทั้งสองข้าง (ทั้งซ้ายและขวา) ว่างทั้งคู่ (ต้องทำในเขตวิกฤตด้วย)

3) ใช้วิธีแก้ปัญหาแบบอสมมาตร โดยให้นักปราชญ์คนที่เลขคี่ ให้หยิบตะเกียบข้างซ้ายก่อน ถ้าได้แล้วจึงหยิบตะเกียบข้างขวา

4) นักปราชญ์คนที่เลขคู่ หยิบตะเกียบข้างขวาก่อนแล้วจึงหยิบตะเกียบข้างซ้าย

วิธีการนี้จะรับประกันว่าไม่มีการแย่งชิงที่เป็นสาเหตุให้นักปราชญ์คนหนึ่งเข้าสู่ภาวะติดตาย แต่วิธีนี้ไม่ได้ป้องกันสภาวะการรออย่างไม่รู้จบ

3.10 สรุป

โพรเซสมีความจำเป็นในการเข้าใช้ทรัพยากรและต้องทำงานร่วมกันกับโพรเซสอื่น ดังนั้นโพรเซสหนึ่งอาจจะได้รับผลกระทบจากโพรเซสอื่น การควบคุมให้โพรเซสสองโพรเซสทำงานร่วมกันได้ที่เรียกว่า การประสานเวลาของโพรเซส หรืออาจเรียกว่า การซินโครไนซ์โพรเซส การซินโครไนซ์โพรเซสเป็นเรื่องที่ยุ่งยากและมีความซับซ้อน และเป็นหน้าที่ของระบบปฏิบัติการที่จะต้องจัดการ เช่น การแก้ปัญหาจากการที่โพรเซสเข้าไปใช้ทรัพยากรใดพร้อมกันที่เรียกว่า สภาวะการแย่งชิง ระบบการปฏิบัติการจะต้องแน่ใจว่า มีเพียงโพรเซสเดียวเท่านั้น ที่ได้รับการเข้าใช้ทรัพยากรใดในช่วงเวลาใดเวลาหนึ่ง ดังนั้นระบบการปฏิบัติการจะต้องป้องกันการเข้าส่วนวิกฤตให้ดี จำเป็นต้องเป็นไปตามเงื่อนไขของการแก้ไขปัญหาในการเข้าสู่ส่วนวิกฤต และการตรวจสอบเงื่อนไขที่กำหนด

การประมวลผลพร้อมกันของโพรเซส เป็นวิธีการทางซอฟต์แวร์ที่เข้ามาช่วยควบคุมการทำงานของระบบให้มีการประมวลผลพร้อมกัน โดยมีคุณสมบัติของการไม่เกิดร่วม มีอัลกอริทึมที่น่าสนใจคือ อัลกอริทึมของเดกเกอร์ และอัลกอริทึมของปีเตอร์สัน การประมวลผลพร้อมกันและมีคุณสมบัติการไม่เกิดร่วม มีวิธีการทางฮาร์ดแวร์สามารถทำได้หลายวิธี เช่นการปิดกั้นไม่ให้โพรเซสใดเข้าใช้ในส่วนวิกฤตในขณะที่มีโปสเซสอื่นใช้งานอยู่ หรือการตรวจสอบส่วนของวิกฤตว่ามีโพรเซสทำงานอยู่หรือไม่ เพื่อเป็นการป้องกันไม่ให้โพรเซสอื่นเข้าไปทำงานในส่วนวิกฤติพร้อมกันได้ ทำให้ระบบสามารถมีโพรเซสจำนวนมากทำงานร่วมกัน โดยมีคุณสมบัติการไม่เกิดร่วม

วิธีการแก้ปัญหาเขตวิกฤต ที่กล่าวมาแล้วทั้งหมด ยังคงไม่สะดวกในการใช้งานกับปัญหาที่ซับซ้อนมากขึ้น ทั้งนี้ได้นำวิธีการเซมาฟอร์เข้ามาช่วยแก้ไขกับปัญหาที่ซับซ้อนมากขึ้น เซมาฟอร์ คือ ตัวแปรชนิด จำนวนเต็ม ช่วยลดความซับซ้อน สามารถเข้าถึงได้โดยผ่านฟังก์ชั่นมาตรฐานได้แก่ฟังก์ชั่น wait() และ signal() ซึ่งเป็นคำสั่งปฏิบัติการแบบภาวะครบหน่วยกล่าวคือ เมื่อโพรเซสหนึ่งกำลังแก้ไขค่าเซมาฟอร์จะต้องไม่มีโพรเซสอื่นที่สามารถแก้ไขค่าเซมาฟอร์เดียวกันได้ในเวลาเดียวกัน

นอกจากนี้ยังสามารถใช้เซมาฟอร์แก้ปัญหาการประสานงาน (Synchronization Problem)

แบบต่าง ๆ ทั้งนี้สามารถที่จะนำเซมาฟอร์นี้ นำไปใช้แก้ปัญหาพื้นฐานที่ใช้ทดสอบวิธีการแก้ปัญหาการประสานเวลาให้ตรงกันได้ เช่น ปัญหาการ Bounded-Buffer, ปัญหาการ Readers – Writers และปัญหาการทำ Dining-Philosophers ที่มีความสำคัญหลัก เนื่องจากปัญหาเหล่านี้คือตัวอย่างปัญหาหลักพื้นฐานที่ถูกใช้ในการทดสอบเกือบจะทุกโครงการเลยทีเดียว

แบบฝึกหัดท้ายบทที่ 3

จงตอบคำถามต่อไปนี้

1) ปัญหา Concurrent Problem คืออะไร จงอธิบายพร้อมยกตัวอย่าง รวมทั้งอธิบายถึงการแก้ปัญหาสภาวะเงื่อนไขการแย่งชิง

2) ส่วนวิกฤติคืออะไร จงอธิบายมาให้เข้าใจ

3) อัลกอริทึมของเดกเกอร์กับอัลกอริทึมของปีเตอร์สัน แตกต่างกันอย่างไร อธิบายมาให้เข้าใจ

4) จงอธิบายความหมายของคำว่า Busy waiting และมีกี่ประเภทในระบบปฏิบัติการ

5) จงอธิบายว่าทำไม Spinlocks จึงไม่เหมาะกับระบบ Single-Processor แต่กลับถูกใช้กับระบบ Multiprocessor

6) ให้แสดงว่าถ้าเซมาฟอร์ wait() และ signal() ยังไม่ได้ถูกดำเนินการ มีโอกาสที่ลักษณะที่มีการกีดกันอาจจะไม่สามารถทำตามกฎ

7) สมมติว่าโปรแกรมถูกแก้ให้ใช้ Shared binary semaphore S:

โพรเซส A	โพรเซส B
int Y;	int Z;
wait(S);	wait(S);
A1: Y = X*2;	B1: Z = X+1;
A2: X = Y;	B2: Z = X;
signal(S);	signal(S);

ค่า T ถูกตั้งค่าเป็น 1 ก่อนที่ทั้งสองโพรเซสนี้จะเริ่มทำงาน และเมื่อค่า X ถูกตั้งค่าเป็น 5 เมื่อเป็นแบบนี้ X สามารถมีค่าเป็นเท่าไรได้บ้างหลังสองโพรเซสนี้ทำงานเสร็จ?

8) จงอธิบายว่าทำไมการอินเตอร์รัพไม่เหมาะกับการนำไปใช้ใน Synchronization Primitives ในระบบ Multiprocessor

9) จงอธิบาย Circular Buffer ทำงานอย่างไรอธิบายการทำงานและยกตัวอย่าง และเป็นการแก้ปัญหาในเรื่องใด

10) จงอธิบายอัลกอริทึ่มในการแก้ปัญหาอาหารเย็นของนักปราชญ์ว่าสามารถแก้ไขได้อย่างไร

เอกสารอ้างอิง

พิเชษฐ์ ศิริรัตนไพศาลกุล. (2548). **ระบบปฏิบัติการ**. กรุงเทพฯ : ซีเอ็ดยูเคชั่น.

ยรรยง เต็งอำนวย. (2533). **ระบบปฏิบัติการ.** กรุงเทพฯ : ซีเอ็ดยูเคชั่น.

สุจิตรา อดุลย์เกษม. (2552). **ทฤษฎี ระบบปฏิบัติการ Operating Systems**. กรุงเทพฯ : โปรวิชั่น.

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. (2013). **Operating System Concepts. 9th ed.** Wiley & Sons, Inc.

Andrew S. Tanenbaum, Herbert Bos. (2004). **Modern Operating Systems. 4th ed.** Prentice Hall, Pearson Education International.

M.Sc TU. Blog. Retrieved April 2, 2014 from http://msctu.blogspot.com/2010/03/