

บทที่ 3

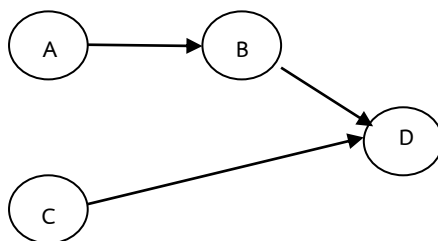
การประสานเวลาของโปรเซส

ในระบบคอมพิวเตอร์ที่มีหน่วยประมวลผลเดียวแต่มีหลายโปรเซสทำงานอยู่ในระบบ โปรเซสต่าง ๆ สลับกันทำงานด้วยความถี่หลายครั้งต่อวินาที คล้ายกับว่าโปรเซสต่าง ๆ ทำงานไปพร้อม ๆ กัน เครื่องที่มีหน่วยประมวลผลมากกว่าหนึ่งหน่วยประมวลผล โปรเซสต่าง ๆ สามารถทำงานไปพร้อมกันได้จริงบนหน่วยประมวลผลคนละตัวกัน สภาพเช่นนี้เรียกว่าภาวะพร้อมกัน (Concurrency) ระบบคอมพิวเตอร์ที่ทำงานแบบมัลติโปรแกรมมิ่ง มีโปรเซสหลายโปรเซสทำงานพร้อมกัน โดยที่แต่ละโปรเซสอาจมีการทำงานเป็นอิสระต่อกัน หรือมีการทำงานที่ขึ้นต่อกัน ดังนั้นหากมีความต้องการที่จะเข้าใช้ทรัพยากรที่มีอยู่พร้อม ๆ กัน เช่น ใช้หน่วยความจำตำแหน่งเดียวกันพร้อมกัน ทำให้เกิดปัญหาการเขียนอ่าน และสำหรับหน่วยความจำที่ใช้ร่วมกัน อาจทำให้ผลของการทำงานคลาดเคลื่อนจากที่ควรจะเป็น

3.1 การประสานเวลาของโปรเซส

โดยทั่วไปทุกโปรเซสต้องการให้มีการประมวลผลอย่างเป็นอิสระ เสมือนมีโปรเซสเดียวที่กำลังทำงานอยู่ ลักษณะความเป็นอิสระนี้เรียกว่า Asynchronous ซึ่งในความเป็นจริงมีหลายโปรเซสเกิดขึ้นในเวลาเดียวกัน และอาจมีความจำเป็นต้องการใช้ทรัพยากรที่มีอยู่อย่างจำกัดพร้อมกัน เช่น หน่วยความจำ ระบบปฏิบัติการจึงเข้ามาช่วยจัดการแต่ละโปรเซส ให้สามารถประมวลผลได้พร้อมกัน (Concurrent processing) และมีการทำงานร่วมกัน (Cooperating) เป็นผลให้โปรเซสในระบบได้มี การใช้ทรัพยากรร่วมกัน ช่วยเพิ่มความเร็วในการทำงานของระบบสูงขึ้นและมีความสะดวกในการทำงาน ในการทำงานของโปรเซสที่ต้องการให้มีการทำงานร่วมกัน จำเป็นต้องมีการสื่อสารกันระหว่างโปรเซส (Inter process communication) จำเป็นต้องให้ทุกโปรเซสทำงานประสานกันเป็นอย่างดี การเข้าจังหวะของโปรเซสหรืออาจเรียกว่าโปรเซสนั้นเกิดการประสานเวลาของโปรเซส (Process synchronization) การประสานเวลาของโปรเซสจะหมายถึง การทำงานของโปรเซสที่ต้องการมีการเกี่ยวข้องกันอาจเป็นเพราะ การใช้ทรัพยากรร่วมกัน หรืออาจจะเป็นการรอให้โปรเซสทำงาน หลังจากที่โปรเซสอื่นทำงานแล้ว

เพื่อความเข้าใจยิ่งขึ้นลองพิจารณาภาพที่ 3.1 จะเห็นว่าโปรเซส B จะเริ่มทำงานได้ก็ต่อเมื่อโปรเซส A ทำงานเสร็จเรียบร้อยแล้ว (ซึ่งต่างกับโปรเซส C ที่ไม่ต้องรอโปรเซสใดเลยก็สามารถทำงานได้ทันที และไม่มีความเกี่ยวข้องกับโปรเซส A และ B เลย) ในทำนองเดียวกันโปรเซส D จะทำงานได้ก็ต่อเมื่อมีการทำงานโปรเซส B และโปรเซส C ทำงานเสร็จเรียบร้อยแล้วนั่นเอง

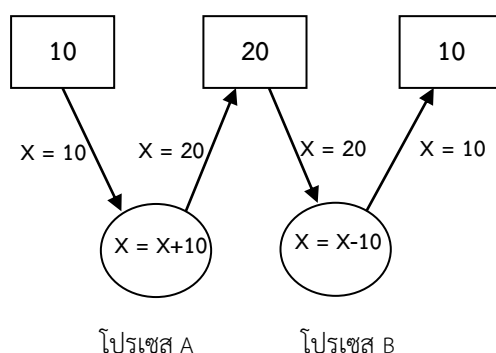


ภาพที่ 3.1 แสดงการประสานเวลาของโปรเซส

3.2 ปัญหาภาวะพร้อมกัน

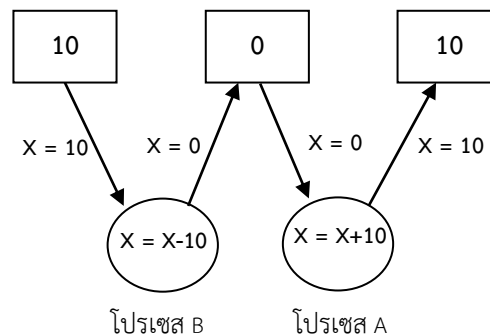
ในระบบที่มีการทำงานบนซีพียูเดียว หรือทำงานบนซีพียูหลายตัว บางครั้งมีความจำเป็นที่ต้องใช้ทรัพยากรร่วมกัน และเป็นทรัพยากรที่ต้องทำงานครั้งละ 1 งาน ไม่สามารถทำงานพร้อมกันได้ จะเรียกทรัพยากรเหล่านี้ว่า ทรัพยากรที่ไม่สามารถใช้ร่วมกันได้ (Non-dedicated resources) ดังนั้น ในขณะที่โปรเซสหนึ่งกำลังใช้ทรัพยากรอยู่ โปรเซสอื่นจะไม่สามารถเข้ามาใช้ทรัพยากรนั้น ๆ ได้ เพื่อให้ได้ผลของการทำงานที่ถูกต้อง เช่น เครื่องพิมพ์ เป็นต้น โปรเซสที่ใช้ข้อมูลร่วมกันหรือร่วมกันทำงาน การใช้ทรัพยากรต่าง ๆ เช่น หน่วยความจำ และอุปกรณ์เข้าออกร่วมกันได้ระหว่างโปรเซสทำให้เกิดปัญหาต่าง ๆ ตามมาดังตัวอย่างต่อไปนี้

ตัวอย่าง ในการใช้ทรัพยากรและข้อมูลร่วมกันโปรเซสสองโปรเซสขึ้นไป อาจก่อให้เกิดปัญหาในการทำงานที่ต้องสมมติโปรเซสทั้งสองเป็นอิสระต่อกัน ทำงานในเวลาพร้อม ๆ กัน และใช้ข้อมูลเดียวกันคือตัวแปร X ซึ่งกำหนดค่าเริ่มต้นไว้ที่ 10 โดยที่ โปรเซส A จะเป็นการเพิ่มค่าเข้าไปอีก 10 ($X = X+10$) ส่วนโปรเซส B จะเป็นการลดค่าจากเดิมออกไป 10 ($X = X-10$)



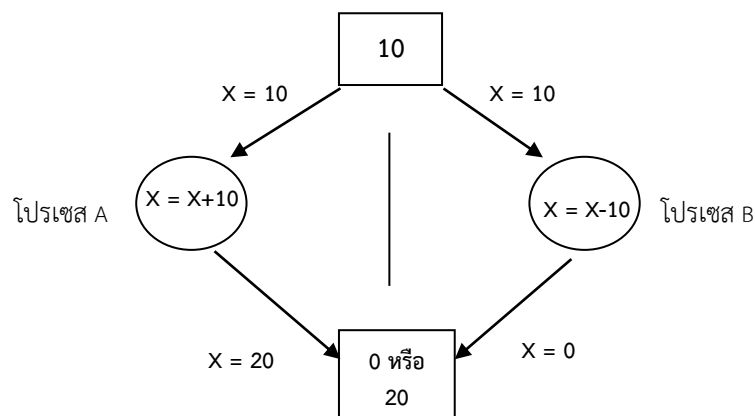
ภาพที่ 3.2 โปรเซส A ทำงานเสร็จก่อน โปรเซส B

จากภาพที่ 3.2 โปรเซส A ทำงานก่อนโปรเซส B โดยอ่านค่า X เข้ามาซึ่งมีค่าเป็น 10 แล้วเพิ่มเข้าไปอีก 10 ทำให้ X มีค่าเป็น 20 แล้วเขียนลงหน่วยความจำ หลังจากนั้นโปรเซส 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 แนนอน ขึ้นอยู่กับว่าโพรเซสใดทำงานเสร็จก่อน

3.3 สถานะการแย่งชิง

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

3.4 การแก้ปัญหาส่วนวิกฤติ

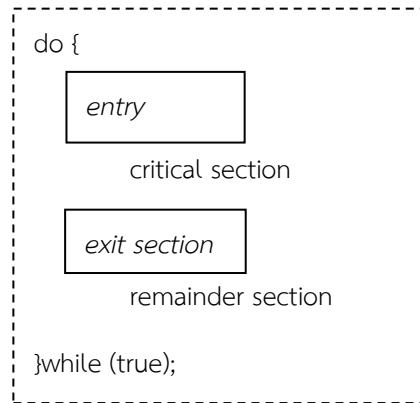
ปัญหาที่เกิดขึ้นจากสถานการณ์ที่มีเงื่อนไขการแย่งชิงนี้ เป็นผลมาจากการที่แต่ละโพรเซสมีการเข้าใช้ข้อมูลบางอย่างร่วมกัน โดยไม่มีการประสานการทำงานระหว่างกัน ต่างฝ่ายต่างทำงานตามคำสั่งไปจนกระทั่งถึงบริเวณที่มีคำสั่งใช้งานข้อมูลร่วมกับอีกฝ่ายหนึ่ง ซึ่งอาจทำให้เกิดผลลัพธ์ที่ผิดพลาดได้ ดังนั้นหากระบบสามารถประสานการเข้าทำงานในบริเวณที่แต่ละโพรเซสมีการใช้ข้อมูลร่วมกันก็จะช่วยแก้ปัญหาที่เกิดขึ้นได้ ลองพิจารณาระบบที่ประกอบด้วย n โพรเซส $\{P_0, P_1, \dots, P_{n-1}\}$ 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 P_i ซึ่งส่วนเริ่มต้นและส่วนจบของการทำงานในส่วนวิกฤติ จะล้อมอยู่ในกรอบรูปสี่เหลี่ยมเพื่อเน้นถึงความสำคัญของส่วนนี้ คำตอบสำหรับการแก้ปัญหาส่วนวิกฤติต้องเป็นไปตามเงื่อนไขทั้งสามข้อต่อไปนี้

3.4.1 การกีดกัน (Mutual exclusion)

ถ้าโพรเซส P_i กำลังปฏิบัติการอยู่ในส่วนวิกฤติแล้ว ต้องไม่มีโพรเซสอื่นใดกำลังอยู่ในส่วนวิกฤติของโพรเซสเหล่านั้นด้วย นั่นคือในขณะที่ขณะหนึ่งมีเพียงโพรเซสเดียวเท่านั้นที่อยู่ในส่วนวิกฤติของตน



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) แต่ไม่มีการกำหนดสมมติฐานที่เกี่ยวกับความเร็วสัมพัทธ์ของโพรเซสทั้งหมด n โพรเซส จากรูปแบบขั้นตอนวิธีสำหรับการแก้ปัญหาส่วนวิกฤติและเงื่อนไขทั้งสามข้อข้างต้นที่ต้องคำนึงถึงในการแก้ปัญหาส่วนวิกฤติเป็นผลให้เกิดแนวคิดการติดตั้งใช้งานในแต่ละโพรเซส ซึ่งจะต้องติดตั้งส่วนเริ่มต้นก่อนเข้าสู่ส่วนวิกฤติเพื่อให้เกิดการกีดกันดังเงื่อนไขข้อ 1 และติดตั้งส่วนจบการทำงานเมื่อออกจากส่วนวิกฤติเพื่อปล่อยให้โพรเซสอื่นได้มีโอกาสเข้าใช้ส่วนวิกฤติของตนเองบ้าง นั่นคือ เป็นไปตามเงื่อนไขข้อที่ 2 และ 3 ซึ่งในโครงสร้างของแต่ละโพรเซสที่มีส่วนเริ่มต้นก่อนเข้าสู่ส่วนวิกฤติ มีจุดประสงค์เพื่อให้โพรเซสทั้งหลายตรวจสอบว่า ขณะนั้นมีโพรเซสใดกำลังอยู่ในส่วนวิกฤติของโพรเซสนั้นหรือไม่ ถ้าไม่มีโพรเซสที่ต้องการเข้าสู่ส่วนวิกฤติของตนเองก็สามารถเข้าทำงานได้ทันที ถ้ามีโพรเซสเหล่านั้นก็ไม่สามารถเข้าสู่ส่วนวิกฤติได้ตามต้องการ ดังนั้นปัญหาที่เกิดขึ้นคือ โพรเซสที่เข้าสู่ส่วนวิกฤติไม่ได้ในขณะนั้นจะดำเนินการรอต่อไปอย่างไร ทั้งนี้แนวคิดในการแก้ปัญหาโดยทั่วไปมี 2 รูปแบบคือ

1. การรอบแบบไม่ว่าง (Preemptive kernels)

ในแต่ละโปรเซสมีการกำหนดเงื่อนไขที่ต้องการใช้ในการตรวจสอบก่อนเข้าสู่ส่วนวิกฤติ ซึ่งถ้าหากไม่สามารถผ่านเงื่อนไขนี้ไปได้ โปรเซสไม่สามารถเข้าสู่ส่วนวิกฤติของตนเองได้ และวนเวียนตรวจสอบเงื่อนไขนั้นอยู่ตลอดเวลาจนกว่าจะได้เข้าสู่ส่วนวิกฤตินั้นคือ ในระหว่างรอโปรเซสนั้นยังต้องทำคำสั่งทำซ้ำ และทดสอบเงื่อนไขอยู่เรื่อย ๆ

2. การขัด (Non-preemptive kernels)

ในการรอบแบบไม่ว่างนั้นไม่ได้ทำให้ระบบเกิดผลงานใด ๆ ขึ้นมา จึงเป็นการใช้วงรอบการทำงานของหน่วยประมวลผลกลางที่ไม่ได้ประโยชน์ และยังต้องแย่งชิงกันเข้าใช้หน่วยประมวลผลกลางระหว่างโปรเซสรวมอื่น ๆ ที่กำลังวนลูปรอเข้าสู่ส่วนวิกฤติ รวมทั้งโปรเซสที่เป็นแบบอิสระทั้งหลายในระบบด้วย ดังนั้นจึงได้มีการพัฒนาให้ทำการบล็อกโปรเซสที่ต้องการเข้าสู่ส่วนวิกฤติแต่ยังไม่อาจเข้าได้ เนื่องจากมีโปรเซสอื่นกำลังทำงานอยู่ในส่วนวิกฤติ ซึ่งการบล็อกนี้ระบบจะนำโปรเซสเหล่านั้นไปรออยู่ในแถวของแต่ละเงื่อนไขและตัวแปรที่ใช้ในการตรวจสอบก่อนเข้าสู่ส่วนวิกฤติ ทำให้โปรเซสเหล่านั้นไม่ต้องเข้ามาแย่งชิงกันใช้ซีพียู และเมื่อใดที่ไม่มีโปรเซสใด ๆ อยู่ในส่วนวิกฤตินั้นคือ โปรเซสล่าสุดที่เข้าสู่ส่วนวิกฤติของตนเองได้ออกจากส่วนวิกฤตินั้นแล้ว จะต้องปลุกหรือใช้คำสั่งบอกให้ระบบทราบ เพื่อให้โปรเซสที่รออยู่ในแถวคอยตามเงื่อนไขที่รอได้กลับเข้ามาทำงานในส่วนวิกฤติของตนเองต่อไป เครื่องมือที่มีการติดตั้งใช้งานในลักษณะนี้ได้แก่ เซมาฟอร์ มอนิเตอร์ และบริเวณวิกฤติ เป็นไปได้ที่สองโปรเซสของเคอร์เนลอยู่ในโหมดรันพร้อมกันในซีพียูที่แตกต่างกัน เพราะเหตุใดโปรเซสจึงชอบเคอร์เนลที่ทำงานก่อนมากกว่าเคอร์เนลที่ทำงานทีหลัง การรอบแบบไม่ว่างเหมาะสมสำหรับโปรแกรมแบบ Real time ซึ่งยอมให้โปรเซสแบบ Real time จองโปรเซสที่กำลังทำงานอยู่ในปัจจุบันบนเคอร์เนล นอกจากนี้การรอบแบบไม่ว่างอาจตอบสนองเพิ่มเติมเนื่องจากมีความเสี่ยงน้อยกว่าโปรเซสของเคอร์เนลโหมดที่ทำงานโดยไม่มีการควบคุม โปรเซสที่มีเวลาการทำงานนานซึ่งได้ทำงานก่อน และทำให้ซีพียูอื่นต้องรอโปรเซสนี้จนกว่าจะทำงานเสร็จ ผลกระทบนี้สามารถลดลงได้ด้วยการออกแบบเคอร์เนลโค้ดที่ไม่ทำงานแบบนี้อย่างแน่นอน

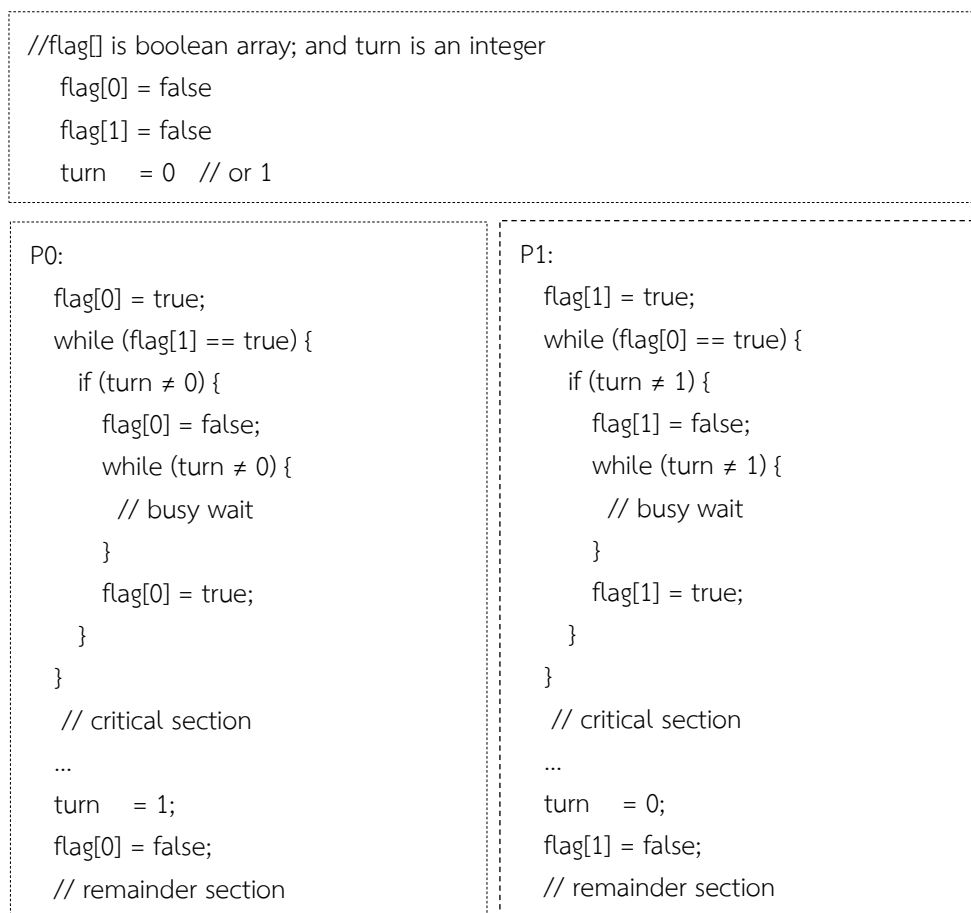
3.5 การประมวลผลพร้อมกันโดยวิธีการทางซอฟต์แวร์

เป็นวิธีการทางซอฟต์แวร์ที่เข้ามาช่วยควบคุมการทำงานของระบบให้มีการประมวลผลพร้อมกัน โดยมีคุณสมบัติของการไม่เกิดร่วม มีอัลกอริทึมที่น่าสนใจคือ

- อัลกอริทึมของเดกเกอร์ (Dekker's algorithm)
- อัลกอริทึมของปีเตอร์สัน (Peterson's algorithm)

3.5.1 อัลกอริทึมของเดกเกอร์

ทำการพัฒนาโปรแกรมที่ประกอบด้วย 2 โปรเซส โดยสามารถทำงานพร้อมกัน และเกิดคุณสมบัติการไม่เกิดร่วม ตัวอย่างประกอบด้วยโปรเซส P0 และ P1 ที่ทำงานพร้อมกัน



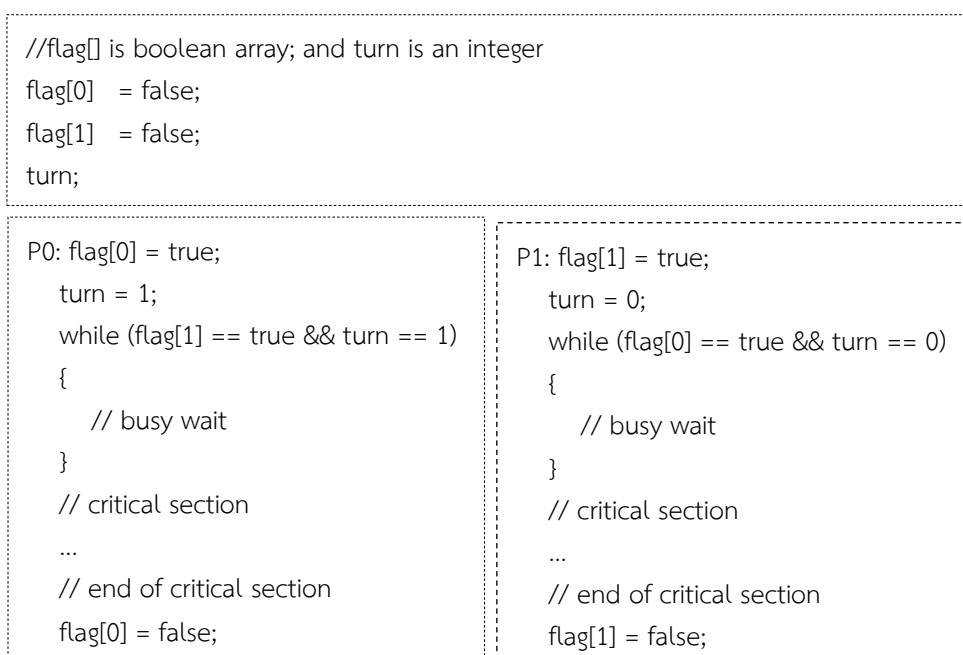
ภาพที่ 3.6 โครงสร้างภาษาที่ใช้อัลกอริทึมของเดกเกอร์ของโปรเซส P0 และ P1

จากโค้ดตัวอย่างตามภาพที่ 3.6 เมื่อเริ่มทำงานตัวแปร turn ถูกกำหนดค่าให้เป็น 0 ทำให้ โปรเซส P0 สามารถเข้าไปทำงานในส่วนวิกฤติส่วนโปรเซส P1 จะต้องวนลูปรอจนกระทั่งโปรเซส P0 ทำงานในส่วนวิกฤติเสร็จ และกำหนดให้ turn มีค่าเป็น 1 โปรเซส P1 จึงจะออกจากการวนลูป และสามารถเข้าไปทำงานในส่วนวิกฤติได้ จะเห็นว่าโปรเซส P0 และ P1 สามารถทำงานพร้อม ๆ กัน และผลัดกันเข้าไปทำงานในส่วนวิกฤติ นั่นคือเกิดคุณสมบัติการไม่เกิดร่วม แต่การทำงานของอัลกอริทึมนี้อาจเกิดปัญหา Busy Waiting ในกรณีที่โปรเซส P0 กำลังทำงานในส่วนวิกฤติ และเกิดหมดเวลาการทำงาน (Timeout) ก่อนที่ P0 จะเปลี่ยนค่าของตัวแปร turn ทำให้ P1 ไม่สามารถเข้าไปทำงานในส่วนวิกฤติได้แม้ว่าจะมีโอกาสเข้าไปทำงานในส่วนวิกฤติ (เพราะ P0 หมดเวลาการทำงาน) ทำให้ P1 ต้องวนลูปตลอดช่วงเวลาการทำงาน

นอกจากนี้ยังอาจเกิดปัญหา Lock Step Synchronization ซึ่งหมายถึง การที่โปรเซสใด ๆ จะเข้าไปทำงานในส่วนวิกฤติ จะต้องทำงานเรียงตามลำดับ เช่น ต้องมีการทำงานเรียงลำดับ P0, P1, P0, P1, Pn แม้ว่าบางโปรเซสอาจไม่ต้องการเข้าไปทำงานในส่วนวิกฤติ

3.5.2 อัลกอริทึมปีเตอร์สันโซลูชัน (Peterson's Solution)

พื้นฐานของการแก้ไขปัญหาคอขวดที่เป็นที่นิยมสำหรับปัญหาส่วนวิกฤติที่รู้จักกันในชื่อของปีเตอร์สันโซลูชัน เนื่องจากสถาปัตยกรรมคอมพิวเตอร์ที่ทันสมัยทำงานโดยใช้คำสั่งภาษาเครื่องพื้นฐาน เช่น โหลดและจัดการ ซึ่งไม่มีการรับประกันว่าอัลกอริทึมปีเตอร์สันโซลูชันจะทำงานได้อย่างถูกต้องในสถาปัตยกรรมนี้ อย่างไรก็ตามการแก้ไขปัญหานี้ให้คำอธิบายอัลกอริทึมที่ดีของการแก้ปัญหาส่วนวิกฤติ และแสดงบางความสัมพันธ์ที่ซับซ้อนในการออกแบบซอฟต์แวร์ อัลกอริทึมนี้จะจำกัดโปรเซสสองโปรเซสที่สลับการดำเนินการระหว่างส่วนวิกฤติและส่วนที่เหลือ ดังภาพที่ 3.7 ที่มีโปรเซสหมายเลข P0 และ P1 และโปรเซสทั้งสองต้องการข้อมูลสองค่าเพื่อใช้ร่วมกันระหว่างกัน



ภาพที่ 3.7 โครงสร้างภาษาที่ใช้อัลกอริทึมของปีเตอร์สันโซลูชันของโปรเซส P0 และ P1

จากโค้ดตัวอย่างตามภาพที่ 3.7 เมื่อเริ่มการทำงาน ตัวแปร flag[0] และ flag[1] ถูกกำหนดค่าให้เป็นเท็จ ถ้าโปรเซสใดต้องการเข้าไปทำงานในส่วนวิกฤติ ให้ทำการกำหนดค่าของตัวแปร flag ของโปรเซสเป็นจริง เช่น ถ้าโปรเซส P0 ต้องการเข้าไปทำงานในส่วนวิกฤติ จะกำหนด flag[0] มีค่าเป็นจริง (true) ทำให้โปรเซส P1 ต้องวนลูปรอ หลังจากที่ P0 ทำงานในส่วนวิกฤติเสร็จแล้ว จะกำหนดค่าของ flag[0] เป็นเท็จ ทำให้โปรเซส P1 หลุดออกจากลูป และสามารถเข้าไปทำงานในส่วนวิกฤติได้ อัลกอริทึมนี้สามารถแก้ปัญหาคอขวดที่เกิดร่วมสำหรับ 2 โปรเซส ที่สามารถพัฒนาให้มีการทำงานเป็น n โปรเซสได้

3.6 ฮาร์ดแวร์ประสานเวลา

การประมวลผลพร้อมกัน และมีคุณสมบัติการไม่เกิดร่วม โดยวิธีการทางฮาร์ดแวร์ สามารถทำได้หลายวิธี ดังนี้

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

3.6.1 การปิดกั้น (Lock)

เมื่อโปรเซสต้องการเข้าไปทำงานในส่วนวิกฤติ โปรเซสจะต้องตรวจสอบก่อนว่า ส่วนวิกฤติถูกล็อก หรือถูกปิดกั้นหรือไม่ หากพบว่าส่วนวิกฤติไม่ถูกล็อก แสดงว่าขณะนั้นไม่มีโปรเซสใดกำลังทำงานในส่วนวิกฤติ โปรเซสสามารถเข้าไปทำงานในส่วนวิกฤติได้ โดยก่อนที่จะเข้าไปทำงานในส่วนวิกฤติ โปรเซสต้องใส่ล็อกเพื่อเป็นการปิดกั้นไม่ให้โปรเซสอื่นเข้าไปทำงานในส่วนวิกฤติได้ จนกระทั่งโปรเซสทำงานในส่วนวิกฤติเสร็จเรียบร้อยแล้วจึงปลดล็อก เพื่อเปิดโอกาสให้โปรเซสอื่นสามารถเข้าทำงานในส่วนวิกฤติได้

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

ภาพที่ 3.8 รูปแบบของคำสั่ง Lock

การทำงานด้วยวิธีนี้ ระบบสามารถมีมากกว่าหนึ่งโปรเซสที่ทำงานพร้อมกันได้ โดยมีคุณสมบัติการไม่เกิดร่วม แต่อาจจะเกิดปัญหาในกรณีที่มากกว่าหนึ่งโปรเซสต้องการเข้าไปทำงานในส่วนวิกฤติพร้อมกัน และทดสอบล็อกแล้วพบว่าล็อกว่าง ก็สามารถเข้าไปทำงานในส่วนวิกฤติได้ ทำให้มีมากกว่าหนึ่งโปรเซสเข้าไปทำงานในส่วนวิกฤติพร้อมกัน

3.6.2 การปิดทางขัดจังหวะ (Disable Interrupt)

สาเหตุหนึ่งที่ทำให้เกิดปัญหาในการทำงานพร้อมกันหลายโปรเซส คือ ขณะที่โปรเซสหนึ่งกำลังทำงานในส่วนวิกฤติ อาจมีโปรเซสอื่นขอขัดจังหวะ เพื่อให้ได้โอกาสเข้าไปทำงานในส่วนวิกฤติ เป็นผลให้โปรเซสที่กำลังทำงานในส่วนวิกฤติต้องหยุดทำงาน โดยที่การทำงานยังไม่เสร็จสมบูรณ์ และสลับให้โปรเซสอื่นเข้าไปทำงานในส่วนวิกฤติแทน โดยที่โปรเซสที่เข้าทำงานภายหลัง อาจแทรกแซงการทำงานหรือเปลี่ยนแปลงค่า หรือมีการประมวลผลใด ๆ ที่ส่งผลกระทบต่อผลลัพธ์ในการทำงานของโปรเซสแรก

บนระบบโปรเซสเซอร์เดี่ยวสามารถระงับการใช้ Interrupts สามารถรับประกันได้ว่าลำดับของชุดคำสั่งจะถูกกระทำโดยปราศจากการบังคับให้ออกจากเวลาของโปรเซสเซอร์ แต่ในการทำงานบน

ระบบมัลติโพรเซสเซอร์ไม่สามารถใช้วิธีนี้ได้ การห้ามขัดจังหวะบนโพรเซสเซอร์หลายตัวนั้นใช้เวลานานมากในการส่งข่าวสารไปยังโพรเซสเซอร์ทุกตัว การส่งข่าวสารจะก่อให้เกิดการหน่วงเวลาไปยังส่วนวิกฤติของทุกโพรเซสเซอร์ซึ่งส่งผลให้ประสิทธิภาพของระบบลดลง เมื่อมีการบันทึกเวลาทุกครั้งที่ถูกเปลี่ยนค่าโดยการขัดจังหวะ

แต่การทำงานด้วยวิธีนี้ไม่เป็นที่ยอมรับ 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 โพรเซสจะเซตค่าล๊อค 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 ระบบคอมพิวเตอร์ที่ทันสมัยให้คำสั่งในเครื่องฮาร์ดแวร์พิเศษที่ช่วยให้สามารถเลือกที่จะทดสอบและปรับปรุงแก้ไขเนื้อหาของคำสั่ง หรือเพื่อที่จะสลับเนื้อหาของคำสั่งสองคำสั่งแบบ Atomically ที่เป็นดังหนึ่ง Uninterruptible Unit เราสามารถใช้คำสั่งพิเศษเหล่านี้เพื่อแก้ปัญหาส่วนวิกฤติ ในลักษณะเชิงสัมพัทธ์ง่าย ๆ แทนที่จะอธิบายเฉพาะหนึ่งคำสั่งสำหรับเฉพาะเครื่องหนึ่ง

คำสั่ง TestAndSet () สามารถกำหนดได้ตามที่แสดงโค้ดภาษาซีในภาพที่ 3.9 ลักษณะที่สำคัญคือ จะทำงานคำสั่งนี้แบบ Atomically ดังนั้นหากคำสั่ง TestAndSet () ที่ทำงานในเวลาเดียวกัน แต่ทำบนซีพียูที่แตกต่างกัน คำสั่งนี้จะถูกรันตามลำดับในบางลำดับแบบสุ่ม หากเครื่องสนับสนุนคำสั่ง TestAndSet () แล้วเราสามารถทำการเอาออกพร้อมกันโดยมีการแจ้งการล๊อคตัวแปร 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 แสดงโครงสร้างโค้ดภาษาซีของโปรเซสในรูปแบบ Test-and-Set

3.6.4 Swap Instruction

คำสั่ง Swap() จะตรงกันข้ามกับคำสั่ง TestAndSet() คำสั่ง Swap() จะทำงานในเนื้อหาของคำสั่งสองคำ กำหนดไว้ตามที่แสดงในภาพที่ 3.11 เช่นเดียวกับคำสั่ง TestAndSet() จะรัน Atomically หากเครื่องสนับสนุนคำสั่ง Swap() แล้วการเอาออกทั้งสองฝ่ายโดยมีเงื่อนไขดังนี้ การล๊อคตัวแปร Global Boolean จะถูกประกาศและทำให้เป็นเท็จในเริ่มต้น นอกจากนี้แต่ละโปรเซสมีคีย์ของตัวเองแปร Local Boolean โครงสร้างของโปรเซส Pi จะปรากฏในภาพที่ 3.12

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

ภาพที่ 3.11 แสดงฟังก์ชัน Swap() ในโครงสร้างโค้ดภาษาซี

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

ภาพที่ 3.12 แสดงโครงสร้างโค้ดภาษาซีการกีดกันโปรเซสโดยการเรียกใช้ฟังก์ชัน Swap()

สำหรับนักออกแบบฮาร์ดแวร์ การใช้กลุ่มคำสั่ง TestAndSet() บนเครื่องประมวลผลจำนวนมากไม่เป็นงานที่สำคัญนัก การทำให้เป็นผลสำเร็จถูกอธิบายไว้ในหนังสือสถาปัตยกรรมคอมพิวเตอร์

3.7 โครงสร้างพื้นฐานสำหรับการซิงโครไนซ์

โปรเซสที่ทำงานร่วมกับโปรเซสอื่นในการดำเนินการ จำเป็นต้องอาศัยการซิงโครไนซ์ที่เหมาะสมเพื่อให้การทำงานถูกต้องตามที่กล่าวมาแล้วนั้น ในการประสานเวลาโปรเซสไม่ใช่เรื่องที่ทำได้ง่าย แม้ว่าจะมีคำสั่งทดสอบและเซต มาเอื้ออำนวยให้ต่อการให้ได้จังหวะกัน แต่ก็ยังไม่สะดวกในการพัฒนาทางด้านนี้ จึงมีโครงสร้างและสิ่งที่อาจเกิดขึ้นในการประสานเวลาที่เรียกว่า คำสั่งพื้นฐาน (Primitive) ที่น่าสนใจดังนี้

3.7.1 เซมาฟอว์ (Semaphore)

วิธีการแก้ปัญหาเขตวิกฤตที่กล่าวมาแล้วทั้งหมด ยังคงไม่สะดวกในการใช้งานกับปัญหาที่ซับซ้อนมากขึ้น เราจึงต้องหาวิธีใหม่ เรียกว่า เซมาฟอว์ ที่โดจสตราได้เสนอขึ้นในปี ค.ศ. 1965 ตามนิยาม เซมาฟอว์แทนได้ด้วย 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 และจะทำการลดค่าลงเมื่อตรวจสอบข้อมูลแล้วพบว่ามีความมากกว่า 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 ที่จะทำงานกับเซมาฟอว์ 2 ตัว คือ คำสั่ง wait(S) ใช้ตรวจสอบค่า รอจน S มีความมากกว่า 0 แล้วจึงลดค่าของ S ลง 1 แล้วทำคำสั่งถัดไป คำสั่ง signal(S) ใช้เพิ่มค่า S ขึ้น 1 แล้วทำคำสั่งถัดไปจะต้องใช้คู่กันเสมอในที่นี้ S คือ เซมาฟอว์ คือ ตัวแปรชนิดจำนวนเต็มที่มีค่าไม่น้อยกว่าศูนย์ คำสั่ง wait(S) จะกั้นโปรเซสไว้หากค่าของ S กลายเป็นศูนย์ คำสั่งทั้งสองนี้เป็นประเภทไม่อาจแบ่งแยกได้ คือไม่สามารถถูกขัดได้ด้วยโปรเซสอื่น

เซมาฟอร์ เป็นเครื่องมือประสานเวลาที่ไม่ต้องการเวลารอคอย เซมาฟอร์ S เป็นตัวแปรจำนวนเต็ม ช่วยลดความซับซ้อนสามารถเข้าถึงได้ผ่านคำสั่งปฏิบัติการมาตรฐานแบบภาวะครบหน่วย (Atomic operation) สองคำสั่ง ได้แก่คำสั่ง Wait และ Signal ความหมายคำสั่งปฏิบัติการแบบภาวะครบหน่วย คือ เมื่อโปรเซสหนึ่งกำลังแก้ไขค่าเซมาฟอร์ จะต้องไม่มีโปรเซสอื่นที่สามารถเข้ามาแก้ไขค่าเซมาฟอร์เดียวกันได้ในเวลาเดียวกัน กรณีของ wait(S) การทดสอบค่าจำนวนเต็มของ S ($S \leq 0$) และการแก้ไขค่า ($S--$) จะต้องกระทำการโดยปราศจากขัดจังหวะ

เซมาฟอร์ เป็นอัลกอริทึมหนึ่งที่ย่างต่อการจัดการทรัพยากรแกโปรเซส โดยเซมาฟอร์นี้จะเปรียบเสมือนการหยิบและใส่ลูกหินในขวดโหล กล่าวคือ จำนวนลูกหินในขวดโหลจะแทนค่าของเซมาฟอร์ในขณะใดขณะหนึ่ง การที่โปรเซสต้องการทรัพยากรก็เปรียบได้กับการหยิบลูกหิน ถ้ามีลูกหินที่ต้องการโปรเซสนั้นก็ทำการหยิบไปลูกหนึ่ง ก็คือโปรเซสนั้นทำคำสั่ง p จากนั้นก็ทำงานของตนต่อไป แต่ถ้าในกรณีที่ไม่มีลูกหินในขวดโหลเลย โปรเซสนั้นต้องรอ (Wait) ให้มีโปรเซสอื่นมาใส่ลูกหินลงไปขวดโหล 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 ไปใช้ในการแก้ไขปัญหาลูกหินของโปรเซส n ตัว โปรเซส n ตัวจะร่วมใช้เซมาฟอร์ ที่ชื่อ mutex (ใช้แทน Mutual exclusion) เริ่มการทำงานที่ 1 โดยที่แต่ละโปรเซส P_i ให้มีโครงสร้างดังภาพที่ 3.14

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

ภาพที่ 3.14 การสร้าง Mutual Exclusion ด้วย Semaphore

เราสามารถนำเซมาฟอร์ในการแก้ไขปัญหาลูกหินของการทำงาน ที่ให้โปรเซสทำงานประสานกันที่ หลากหลาย เช่น เมื่อพิจารณาโปรเซส 2 ตัวที่ทำงานพร้อมกันให้โปรเซส 1 มีสภาวะการทำงานที่ S_1 และโปรเซส 2 มีสภาวะการทำงานที่ S_2 สมมติเราต้องการให้ S_2 ทำงานได้ก็ต่อเมื่อ S_1 ทำงานเสร็จ เราสามารถสร้างวิธีการดังกล่าวได้โดยการใช้โปรเซส 1 และ 2 ใช้เซมาฟอร์ ที่ชื่อ synch ร่วมกัน แล้วเริ่มการ

ทำงานที่ศูนย์ แล้วทำการแทรกเข้าไปในสภาวะการทำงาน S1 ในโปรเซส 1 ในทำนองเดียวกันก็นำไปไว้ที่สภาวะการทำงาน S2 ในโปรเซส 2 นั้นจะทำให้สภาวะ S2 ในโปรเซส 2 ทำงานได้ก็ต่อเมื่อโปรเซส 1 ให้สัญญาณ Signal(synch) เมื่อทำงานสภาวะ S1 เสร็จแล้วเท่านั้น

3.7.3 การสร้างเซมาฟอว์ (Semaphore Implementation)

ข้อเสียของการแก้ไขปัญหการกีดกันนั้นก็คือ ถ้าหากมีโปรเซสใดกำลังทำงานในส่วนวิกฤตอยู่นั้นแล้วมีโปรเซสอื่นต้องการเข้าทำงานก็ต้องวนลูปรอเข้าทำงานไปเรื่อย ๆ เรียกว่า “การวนเวียนรอคอย” (Busy waiting) การรอเช่นนี้ทำให้เสียเวลาอยู่ในสภาวะของ Waiting แทนที่จะสามารถทำอย่างอื่นได้ ทำให้ประสิทธิภาพลดลง ย่อมเป็นเหตุให้เวลาของหน่วยประมวลผลกลางเสียไปโดยเปล่าประโยชน์ เวลาเหล่านี้อาจยกให้กระบวนการอื่นได้ใช้ประโยชน์คุ้มค่ากว่า เหตุการณ์เช่นนี้เรียกว่า Spinlock คือ โปรเซสวนเวียนทำงาน (Spin) อยู่ ขณะที่กำลังรอคอยให้เปิดล็อก ในระบบแบบ Multiprocessor จะเกิดข้อดีเนื่องจากไม่ต้องทำ Context Switch ดังนั้นถ้าการ Lock อยู่ในช่วงเวลาสั้น ๆ Spinlock ก็จะเป็นประโยชน์อย่างยิ่ง

เราสามารถปรับปรุงนิยามของคำสั่ง wait() และ signal() ใหม่ เพื่อให้ไม่มีปัญหการวนเวียนรอคอยเมื่อโปรเซสทำคำสั่ง wait() และพบว่าค่าของเซมาฟอว์น้อยกว่าหรือเท่ากับศูนย์ โปรเซสจะต้องรอคอย โดยการหยุดชั่วขณะ (Block) แทนที่จะวนเวียนรอคอย การหยุดชั่วขณะนี้โปรเซสจะเข้าไปรอในแถวคอย (Waiting queue) ที่เกี่ยวข้องกับเซมาฟอว์นั้น ๆ และเปลี่ยนสถานะเป็นสถานะรอคอย (Waiting state) การควบคุมจะถูกย้ายกลับไปให้ตัวจัดการการทำงานของหน่วยประมวลผลกลาง เพื่อจัดให้โปรเซสอื่นทำงานต่อไป

โปรเซสที่หยุดชั่วขณะโดยรอเซมาฟอว์ S นี้ จะสามารถดำเนินต่อไปได้ เมื่อมีโปรเซสอื่นทำคำสั่ง signal() กับเซมาฟอว์ S ภายในคำสั่ง signal จะมีการปลุก (Wakeup) ให้กระบวนการที่หยุดชั่วขณะกลับดำเนินการต่อไปได้ โดยเปลี่ยนสถานะของกระบวนการนั้นเป็นสถานะพร้อม (Ready state) แล้วย้ายไปต่อแถวพร้อม

การสร้างเซมาฟอว์ตามนิยามใหม่นี้ กำหนดให้เซมาฟอว์เป็นตัวแปรประเภท Structure ในภาษาซี

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

ตัวแปรเซมาฟอว์จะประกอบด้วย ตัวเลขจำนวนเต็ม (ตัวแปร value) และแถวคอยของโปรเซส (ตัวแปร L) เมื่อโปรเซสหนึ่งต้องคอยเซมาฟอว์นี้ มันก็就会被ใส่ชื่อลงในแถวคอย L การใช้คำสั่ง signal จะดึงโปรเซสออกจากหัวแถวคอยนี้ และปลุกโปรเซสที่ได้ให้กลับไปทำงานต่อ 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 จะทำให้โปรเซสที่ทำคำสั่งนี้หยุดชั่วขณะ คำสั่ง wakeup(P) จะปลุกโปรเซส P ให้กลับทำงานต่อ จะสังเกตได้ว่า นิยามดั้งเดิมของเซมาฟอร์ซึ่งมีการวนเวียนรอคอยนั้น ค่าของเซมาฟอร์จะไม่ติดลบ แต่ในนิยามใหม่นี้ค่าของเซมาฟอร์มีค่าเป็นลบได้ ค่าที่เป็นลบแสดงถึงจำนวนกระบวนการที่รอคอยเซมาฟอร์ตัวนั้น ๆ ด้วย

เราต้องไม่ให้มีโปรเซสสองตัวทำคำสั่ง wait และ signal บนเซมาฟอร์ตัวเดียวกันในเวลาเดียวกัน ซึ่งกรณีนี้ก็คือ ปัญหาเซตวิฤตแบบหนึ่งนั่นเอง โดยสามารถเลือกวิธีการจัดการได้ 2 วิธี คือ

1. ถ้าเป็นระบบที่มีหน่วยประมวลผลเพียงตัวเดียว เราก็เพียงแค่ห้ามขัดจังหวะขณะที่กำลังทำงาน คำสั่ง wait และ signal ดังนั้นโปรเซสอื่นจะไม่สามารถมาขัดจังหวะการทำงานได้
2. ถ้าเป็นระบบที่มีหน่วยประมวลผลหลายตัว การห้ามขัดจังหวะจะไม่เพียงพอ เพราะโปรเซสอื่นอาจทำงานอยู่ในหน่วยประมวลผลคนละตัวกับโปรเซสที่ถูกห้ามขัดจังหวะ ดังนั้นอาจจัดการโดยใช้ขั้นตอนวิธีการแก้ไขส่วนวิฤต แล้วเอาคำสั่ง wait และ signal ใส่ไว้ในเซตวิฤตก็จะสามารถป้องกันได้

3.7.4 เซมาฟอร์แบบทวิภาค (Binary Semaphore)

โครงสร้างของเซมาฟอร์ ที่กล่าวมาข้างต้นเป็นลักษณะของเซมาฟอร์แบบนับ (Counting semaphore) ส่วนในเซมาฟอร์แบบทวิภาคจะกำหนดให้ใช้ตัวแปรจำนวนเต็มได้ 2 ค่าคือ 0 และ 1 เท่านั้น ทำให้สามารถสร้างได้ง่ายกว่าเซมาฟอร์แบบนับ ซึ่งสามารถสร้างเซมาฟอร์แบบนับให้อยู่ในรูปแบบทวิภาคได้ โดยการใช้โครงสร้างตามภาพที่ 3.16 เริ่มด้วยค่า $S_1, S_3 = 1$ และ $S_2 = 0$ 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);

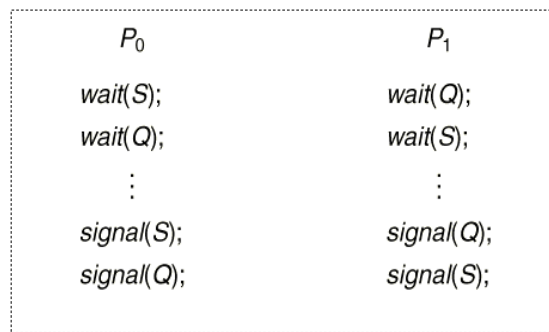
```

ภาพที่ 3.17 รูปแบบของการใช้งาน Semaphore แบบทวิภาค

การทำงานคำสั่ง wait ก่อนที่จะมีการลดค่าของเซมาฟอว์ ต้องการทำ wait(S1) ก่อนลดค่า แล้วก็ตรวจสอบว่าค่าที่ลดน้อยกว่า 0 หรือไม่ ถ้าน้อยกว่าก็จะมีการปลด lock S1 ที่รออยู่ แล้วก็ lock S2 ต่อ แต่ถ้าค่าของ C มากกว่าหรือเท่ากับ 0 ก็ปลด lock S1 อย่างเดียว ส่วนคำสั่งของ Signal จะมีการเพิ่มค่า C เมื่อมีการ wait(S1) ไปแล้ว ซึ่งถ้าค่า C ที่เพิ่มน้อยกว่าหรือเท่ากับ 0 ก็จะส่งสัญญาณไปปลด lock S2 แล้วก็จึงปลด lock S1 การทำงานของสถานะ S1 แทนโปรเซส 1 ส่วน S2 แทนโปรเซส 0 นั้นเอง ดังนั้นทั้งหมดจึงเป็นการเลือกการเข้าทำงานส่วนวิกฤตของสองโปรเซส

3.8 การติดตายและอดตาย

การสร้างเซมาฟอว์ด้วยการให้โปรเซสที่รอการทำงานไปอยู่ในคิว อาจส่งผลให้เกิดเหตุการณ์ที่สองโปรเซสหรือมากกว่านั้นรอแบบไม่มีที่สิ้นสุด เนื่องจากโปรเซสไปรอการทำงานของโปรเซสที่ยังรอการทำงานด้วยเช่นกัน ลักษณะเช่นนี้เรียกว่าการติดตาย (Deadlock)



ภาพที่ 3.18 สถานการณ์ในการใช้งาน 2 โปรเซสแบบ Semaphore และเกิด Deadlock

สถานการณ์ติดตายสามารถอธิบายได้ดังภาพที่ 3.18 ซึ่งมีโปรเซสทำงานอยู่ 2 โปรเซสพร้อมกัน คือ P_0 และ P_1 โดยในแต่ละโปรเซสก็มีการเข้าใช้เซมาฟอร์ 2 ตัวคือ S และ Q โดยให้ค่าเป็น 1 สมมติว่าโปรเซส P_0 ทำงาน `wait(s)` พร้อมกับที่โปรเซส P_1 ทำงาน `wait(Q)` เมื่อโปรเซส P_0 ทำงาน `wait(Q)` มันจะต้องรอนกว่าโปรเซส P_1 ทำการ `signal(Q)` ในทำนองเดียวกันนั้นเมื่อโปรเซส P_1 ทำงาน `wait(S)` มันก็ต้องรอนกว่าโปรเซส P_0 ทำการ `signal(S)` ให้ พบว่าไม่สามารถมีโปรเซสใดทำงานได้ ลักษณะการติดตายจึงเกิดขึ้น หากในการสร้างเซมาฟอร์โดยให้โปรเซสการรอจัดเก็บไว้ในคิวที่มีโครงสร้างแบบ LIFO หรือ เข้าหลังออกก่อน ก็อาจจะให้เกิดปัญหาของการติดตายและอดตายของโปรเซสที่จัดเก็บอยู่ด้านใน

3.9 ปัญหาพื้นฐานของการประสานเวลา

ปัญหาการทำงานของโปรเซส ปัญหาที่เกิดขึ้นกับระบบปฏิบัติการเป็นปัญหาที่น่าสนใจและถกเถียงกันอย่างกว้างขวาง ตลอดจนมีการวิเคราะห์โดยใช้วิธีการด้านประสานเวลา ในที่นี้จะนำปัญหามาพื้นฐานของการประสานเวลาที่เป็นที่กล่าวถึงหรือเป็นต้นแบบดังนี้

3.9.1 ปัญหาที่פקข้อมูลขนาดจำกัด (Bounded-Buffer Problem)

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

<pre>//itemCount is an integer int itemCount = 0;</pre>	
<pre>procedure producer() { while (true) { item = produceItem(); if (itemCount == BUFFER_SIZE) { sleep(); } putItemIntoBuffer(item); itemCount = itemCount + 1; if (itemCount == 1) { wakeup(consumer); } } }</pre>	<pre>procedure consumer() { while (true) { if (itemCount == 0) { sleep(); } item = removeItemFromBuffer(); itemCount = itemCount - 1; if (itemCount == BUFFER_SIZE - 1) { wakeup(producer); } consumeItem(item); } }</pre>

ภาพที่ 3.19 ตัวอย่างการใช้ลอคอริทึม 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)

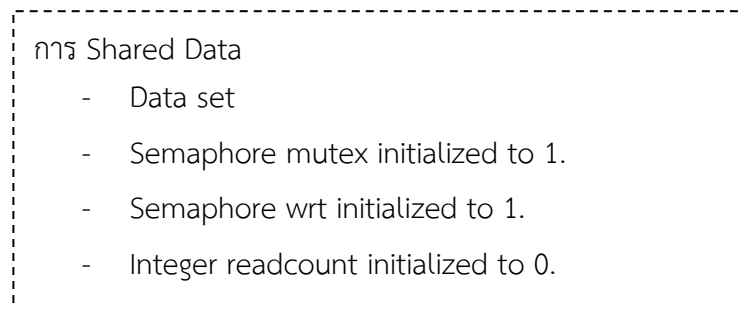
3.9.2 ปัญหาผู้อ่านและผู้เขียน (The Readers/Writers Problem)

ข้อมูลในระบบที่มีการทำงานของโพรเซสพร้อมกันหลายตัวจะถูกใช้ร่วมกันใช้งาน บางโพรเซสต้องการอ่านข้อมูล ในอีกบางโพรเซสต้องการเขียนข้อมูล โดยที่ความแตกต่างของทั้งสองส่วนคือ Readers จะเป็นโพรเซสที่ต้องการจะอ่านข้อมูลเพียงอย่างเดียว ส่วนโพรเซส Writer จะต้องการเพียงแค่การเขียนข้อมูลเท่านั้น สังเกตว่าหากมีหลาย ๆ โพรเซสทำการอ่านข้อมูลพร้อม ๆ กันจะไม่มีปัญหาใดเกิดขึ้น แต่ถ้ามีโพรเซสที่ต้องการเขียนข้อมูลพร้อม ๆ กันจะเกิดปัญหาค้างขึ้นทันที

ปัญหานี้คือ มีพื้นที่ข้อมูลซึ่งใช้ร่วมกันได้ระหว่างกระบวนการทั้งหมด มีโพรเซสจำนวนมากที่ต้องการอ่านพื้นที่ดังกล่าวเพียงอย่างเดียว และมีโพรเซสที่ต้องการเขียนบนพื้นที่ดังกล่าวเพียงโพรเซสเดียวในเวลาเดียวกัน จึงมีการนำเอาเซมาฟอร์ช่วยในการแก้ปัญหาเหล่านี้ โดยให้มีโครงสร้างของการใช้ตัวแปรร่วมกันดังภาพที่ 3.20

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

จากข้อกำหนดข้างต้นอาจนำไปสู่ปัญหาภาวะงูกินหาง กรณีแรกผู้เขียนอาจต้องเป็นฝ่ายรออย่างไม่รู้จบ กรณีหลังผู้อ่านอาจต้องเป็นฝ่ายรออย่างไม่รู้จบ



ภาพที่ 3.20 โครงสร้างตัวแปรที่ใช้ภายใน Semaphore

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

```
do {
    wait (wrt) ;

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

ภาพที่ 3.21 รูปแบบของโปรเซสผู้เขียน

พบว่าเมื่อผู้เขียนกำลังทำงานในส่วนวิกฤต และมีโปรเซสรออยู่เป็นจำนวน n แล้ว เมื่อผู้อ่านถูกนำเข้าไปในคิวด้วย wrt ทำให้จำนวนโปรเซสที่รอลดลงเหลือ $n-1$ ซึ่งจะเข้าคิวด้วยเซมาฟอร์ mutex เมื่อสังเกตจะพบว่าเมื่อผู้เขียนทำ signal(wrt) อาจเกิดกรณีที่ให้โปรเซสผู้อ่านที่รออยู่เข้าทำงาน หรือจะให้สิทธิแก่โปรเซสผู้เขียนที่เหลือเพียงตัวเดียวทำงาน 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) ;

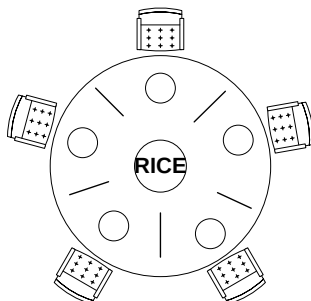
} while (true)

```

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 ตัวอย่างโครงสร้างภาษาในการแก้ปัญหปัญหอาหารเย็นของนักปราชญ์

วิธีนี้อาจป้องกันไม่ให้นักปราชญ์ที่นั่งติดกันกินพร้อมกันได้ แต่อาจเกิดปัญหาวจรอัปชั่นในระบบสมมติว่านักปราชญ์ทั้ง 5 คนหิวพร้อมกัน แล้วหยิบตะเกียบข้างซ้ายของเขาเกือบพร้อมกันหมด จากนั้นแต่ละคนจะพยายามหยิบตะเกียบข้างขวา ซึ่งจะต้องรอกันเองตลอดไป อาจแก้ปัญหาวจรอัปชั่นนี้ได้ดังนี้

1. ให้นักปราชญ์นั่งในโต๊ะได้ ไม่เกิน 4 คน
2. ให้นักปราชญ์หยิบตะเกียบได้ ก็ต่อเมื่อตะเกียบทั้งสองข้าง (ทั้งซ้ายและขวา) วางทั้งคู่ (ต้องทำในเซตวิกฤตด้วย)
3. ใช้วิธีแก้ปัญหาแบบอสมมาตร โดยให้นักปราชญ์คนที่เลขคี่ ให้หยิบตะเกียบข้างซ้ายก่อน ถ้าได้แล้วจึงหยิบตะเกียบข้างขวา
4. นักปราชญ์คนที่เลขคู่ หยิบตะเกียบข้างขวาก่อนแล้วจึงหยิบตะเกียบข้างซ้าย

วิธีการนี้จะรับประกันว่าไม่มีการแย่งชิงที่เป็นสาเหตุให้นักปราชญ์คนหนึ่งเข้าสู่ภาวะติดตาย แต่วิธีนี้ไม่ได้ป้องกันสภาวะการรออย่างไม่รู้จบ

3.10 สรุป

โพรเซสมีความจำเป็นในการเข้าใช้ทรัพยากรและต้องทำงานร่วมกันกับโพรเซสอื่น ดังนั้น โพรเซสหนึ่งอาจจะได้รับผลกระทบจากโพรเซสอื่น การควบคุมให้โพรเซสสองโพรเซสทำงานร่วมกันได้ ที่เรียกว่า การประสานเวลาของโพรเซส หรืออาจเรียกว่า การซิงโครไนซ์โพรเซส การซิงโครไนซ์โพรเซสเป็นเรื่องที่ยุ่งยากและมีความซับซ้อน และเป็นหน้าที่ของระบบปฏิบัติการที่จะต้องจัดการ เช่น การแก้ปัญหาจากการที่โพรเซสเข้าไปใช้ทรัพยากรใดพร้อมกันที่เรียกว่า สถานะการแย่งชิง ระบบการปฏิบัติการจะต้องแน่ใจว่า มีเพียงโพรเซสเดียวเท่านั้น ที่ได้รับการเข้าใช้ทรัพยากรใดในช่วงเวลาใดเวลาหนึ่ง ดังนั้นระบบการปฏิบัติการจะต้องป้องกันการเข้าส่วนวิกฤตให้ดี จำเป็นต้องเป็นไปตามเงื่อนไขของการแก้ไขปัญหาในการเข้าส่วนวิกฤต และการตรวจสอบเงื่อนไขที่กำหนด

การประมวลผลพร้อมกันของโพรเซส เป็นวิธีการทางซอฟต์แวร์ที่เข้ามาช่วยควบคุมการทำงานของระบบให้มีการประมวลผลพร้อมกัน โดยมีคุณสมบัติของการไม่เกิดร่วม มีอัลกอริทึมที่น่าสนใจคือ อัลกอริทึมของเดกเกอร์ และอัลกอริทึมของปีเตอร์สัน การประมวลผลพร้อมกันและมีคุณสมบัติการไม่เกิดร่วม มีวิธีการทางฮาร์ดแวร์สามารถทำได้หลายวิธี เช่นการปิดกั้นไม่ให้โพรเซสใดเข้าใช้ในส่วนวิกฤตในขณะที่มีโพรเซสอื่นใช้งานอยู่ หรือการตรวจสอบส่วนของวิกฤตว่ามีโพรเซสทำงานอยู่หรือไม่ เพื่อเป็นการป้องกันไม่ให้โพรเซสอื่นเข้าไปทำงานในส่วนวิกฤตพร้อมกันได้ ทำให้ระบบสามารถมีโพรเซสจำนวนมากทำงานร่วมกัน โดยมีคุณสมบัติการไม่เกิดร่วม

วิธีการแก้ปัญหาเซตวิกฤต ที่กล่าวมาแล้วทั้งหมด ยังคงไม่สะดวกในการใช้งานกับปัญหาที่ซับซ้อนมากขึ้น ทั้งนี้ได้นำวิธีการเซมาฟอร์เข้ามาช่วยแก้ไขกับปัญหาที่ซับซ้อนมากขึ้น เซมาฟอร์ คือ ตัวแปรชนิด จำนวนเต็ม ช่วยลดความซับซ้อน สามารถเข้าถึงได้โดยผ่านฟังก์ชันมาตรฐานได้แก่ฟังก์ชัน 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

1. ปัญหา Concurrent Problem คืออะไร จงอธิบายพร้อมยกตัวอย่าง รวมทั้งอธิบายถึงการแก้ปัญหาสถานะเงื่อนไขการแย่งชิง
2. ส่วนวิกฤตคืออะไร จงอธิบายมาให้เข้าใจ
3. อัลกอริทึมของเดกเกอร์กับอัลกอริทึมของปีเตอร์สัน แตกต่างกันอย่างไร อธิบายมาให้เข้าใจ
4. จงอธิบายความหมายของคำว่า Busy waiting และมีกี่ประเภทในระบบปฏิบัติการ

5. จงอธิบายว่าทำไม Spinlocks จึงไม่เหมาะกับระบบ Single-Processor แต่กลับถูกใช้กับระบบ Multiprocessor
6. ให้แสดงว่าถ้าเซมาฟอร์ wait() และ signal() ยังไม่ได้ถูกดำเนินการ มีโอกาสที่ลักษณะที่มีการกีดกัน อาจจะไม่สามารถทำตามกฎ
7. สมมติว่าโปรแกรมถูกแก้ไขให้ใช้ Shared binary semaphore S:

โปรเซส A	โปรเซส 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 ก่อนที่ทั้งสองโปรเซสนี้จะเริ่มทำงาน และเมื่อค่า X ถูกตั้งค่าเป็น 5 เมื่อเป็นแบบนี้ X สามารถมีค่าเป็นเท่าไรได้บ้างหลังสองโปรเซสนี้ทำงานเสร็จ?

8. จงอธิบายว่าทำไมการอินเทอร์รัพไม่เหมาะกับการนำไปใช้ใน Synchronization Primitives ในระบบ Multiprocessor
9. จงอธิบาย Circular Buffer ทำงานอย่างไรอธิบายการทำงานและยกตัวอย่าง และเป็นการแก้ปัญหาในเรื่องใด
10. จงอธิบายอัลกอริทึมในการแก้ปัญหาอาหารเย็นของนักปราชญ์ว่าสามารถแก้ไขได้อย่างไร

บรรณานุกรมประจำบทที่ 3

- พิเชษฐ์ ศิริรัตนไพศาลกุล. (2548). **ระบบปฏิบัติการ**. กรุงเทพฯ : ซีเอ็ดยูเคชั่น.
- ยรรยง เต็งอำนวย. (2533). **ระบบปฏิบัติการ**. กรุงเทพฯ : ซีเอ็ดยูเคชั่น.
- สุจิตรา อุดลย์เกษม. (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/>