บทที่ 3 การเขียนโปรแกรมแบบหลายเธรด

วัตถุประสงค์ของเนื้อหา

- ศึกษาถึงความหมายของเธรด และลักษณะของเธรดภายในโพรเซส
- ทราบถึงข้อดีข้อด้อยของการเขียนโปรแกรมแบบหลายเธรด และการใช้งานเธรดในระบบคอมพิวเตอร์ปัจจุบัน
- ศึกษาถึงการเขียนโปรแกรมและการจัดการแบบหลายเธรดในระบบปฏิบัติการปัจจุบัน

สิ่งที่คาดหวังจากการเรียนในบทนี้

- นักศึกษาเข้าใจถึงคุณลักษณะของเธรด และความแตกต่างระหว่างของเธรดกับโพรเซส
- นักศึกษาสามารถพัฒนาโปรแกรมแบบหลายเธรดได้

วัตถุประสงค์ของปฏิบัติการท้ายบท

- นักศึกษาได้ทดลองเขียนโปรแกรมแบบหลายเธรดบนระบบปฏิบัติการวินโดวส์และลินุกซ์
- ทดลองประยุกต์ใช้การเขียนโปรแกรมแบบหลายเธรดในการทำงานแบบ concurrent process

สิ่งที่คาดหวังจากปฏิบัติการท้ายบท

- นักศึกษาสามารถเขียนโปรแกรมแตกเธรด และรวมเธรด บนระบบปฏิบัติการวินโดวส์และลินุกซ์
- นักศึกษาเข้าใจถึงการประยุกต์ประเด็นปัญหาผู้ผลิต-ผู้บริโภค ที่นำมาประยุกต์แบบหลายเธรด เห็นและเข้าใจถึงความแตกต่างระหว่างการประยุกต์แบบหลายโพรเซสและแบบหลายเธรด ว่ามีข้อเด่นข้อด้อยแตกต่างกันอย่างไร

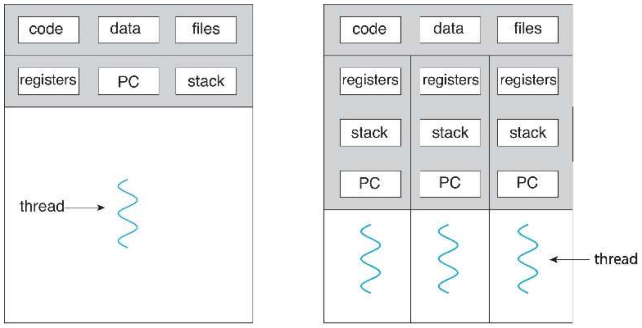
เวลาที่ใช้ในการเรียนการสอน

- ทฤษฎี 2 ชั่วโมง
    - แนวคิด และโครงสร้างของการจัดการเธรดโดยทั่วไป 1 ชั่วโมง
    - การจัดการเธรด การแตกเธรด และการรวมเธรด 1 ชั่วโมง
- ปฏิบัติ 2 ชั่วโมง
    - การเขียนโปรแกรมแบบหลายเธรดบนลินุกซ์ 0.5 ชั่วโมง
    - การเขียนโปรแกรมแบบหลายเธรดบนวินโดวส์ 0.5 ชั่วโมง
    - การประยุกต์ในกรณีประเด็นปัญหาผู้ผลิต-ผู้บริโภคบนสองระบบปฏิบัติการ 1 ชั่วโมง

ครั้งที่ 3 การเขียนโปรแกรมแบบหลายเธรด

3.1 ความนำ

เธรด (Thread) เป็นกลไกการจัดสรรทรัพยากรของซีพียูของโพรเซสในลักษณะของการแบ่งการดำเนินการของตัวโพรเซสออกเป็นหลายๆ ส่วน ที่แต่ละส่วนนั้นทำงานไปพร้อมกัน (แต่ละส่วนถูกจัดสรรเพื่อได้เวลาเข้าครอบครองซีพียูในลักษณะทำนองเดียวกันกับที่แต่ละโพรเซสได้รับจัดสรร) แต่สิ่งที่แตกต่างไปจากกลไกของโพรเซสหลายตัวก็คือ เธรดนั้นมีทรัพยากรเพียงบางอย่างเท่านั้นที่เป็นของตนเอง ซึ่งได้แก่ program counter (ตัวบอกตำแหน่งหน่วยความจำของคำสั่งถัดไปที่จะประมวล) ชุดข้อมูลเรจิสเตอร์ในซีพียู และพื้นที่สแต็ก (ทำให้สามารถจัดเก็บข้อมูลตัวแปรเฉพาะที่ และคำสั่งถัดไปหลังจากการเรียกฟังก์ชัน เป็นอิสระจากกัน) นั่นหมายถึงแต่ละเธรดจะสามารถทำงานไปได้แตกต่างกัน(และเรียกฟังก์ชันใช้งานได้แตกต่างกันไปอย่างอิสระ) โดยมีพื้นที่เก็บข้อมูลเฉพาะส่วนตัว แต่ในขณะเดียวกันนั้น เธรดทุกตัวของโพรเซส จะใช้พื้นที่เก็บชุดคำสั่ง และตัวแปรส่วนกลางร่วมกัน รวมทั้งข้อมูลทั่วไปของโพรเซสได้แก่ข้อมูล environment ของระบบ(ที่ระบบปฏิบัติการเตรียมไว้ให้) ข้อมูลเกี่ยวกับไฟล์และ I/O ที่เปิดใช้งานอยู่ เป็นต้น



โปรแกรมที่พัฒนาใช้งานบนระบบปฏิบัติการยุคใหม่ จึงมักจะถูกออกแบบเป็นแบบหลายเธรด (Multithread) ทั้งนี้เพื่อประโยชน์ในการจัดการกับข้อมูลหลายชิ้นพร้อมๆ กัน หรือจัดการกับ I/O หลายๆ ตัวพร้อมกัน หรือสามารถกระจายงานไปรันบนแต่ละคอร์ของซีพียูที่มีหลายคอร์ได้ ทั้งนี้ในการเขียนโปรแกรมแบบเดิม หากโพรเซสหนึ่งต้องจัดการกับข้อมูลแต่ละตัว หรือ I/O แต่ละตัว ซึ่งต้องเสียเวลารอคอยนาน เส้นทางการเดินของโปรแกรมที่มีเส้นเดียว หมายถึงจะต้องรอคอยข้อมูลไปทีละตัวจนกว่าจะเสร็จ หรือรอคอย I/O ที่ต้องจัดการไปทีละตัวจนกว่าจะเสร็จ ซึ่งจะทำให้เวลารอคอยรวมต้องเสียเวลามาก ในการเขียนโปรแกรมแบบหลายเธรด โปรแกรมจะแบ่งเธรดการติดต่อข้อมูลหรือ I/O นี้แยกออกเป็นเธรดๆ ไป ดังนั้นแต่ละเธรดสามารถร้องขอข้อมูลเฉพาะที่ตนรับผิดชอบ หรือติดต่อเฉพาะ I/O ที่ตนรับผิดชอบ ทำให้เวลารอคอยของแต่ละเธรดไม่ขึ้นต่อกัน และส่งผลทำให้การรอคอยนั้นลดลง เท่ากับโพรเซสโดยรวมทำงานเสร็จเร็วขึ้น ตัวอย่างการที่เราแบ่งเธรดดังเช่น

- การแบ่งเธรดไปใช้ในการปรับปรุงหรือเปลี่ยนการแสดงผลหน้าจอ
- การแบ่งเธรดไปดึงข้อมูลใหม่ๆ เพิ่มเติมเข้ามาใช้ในโพรเซส
- การแบ่งเธรดไปเพื่อจัดการกับข้อมูลบางตัว ที่ไม่ต้องการผลการทำงานอย่างรวดเร็วมาก เช่นแบ่งเธรดเพื่อใช้ตรวจสอบการสะกดคำผิดของโปรแกรมเอกสาร การส่งเอกสารไปพิมพ์ เป็นต้น
- การแบ่งเธรดออกไปเพื่อจัดการกับงานที่ต้องรอคอยนาน หรือทำงานช้ามากๆ เมื่อเทียบกับงานอื่นๆ เช่นการร้องขอ I/O หรือ network ที่จะมีเวลารอคอยสูง

เราอาจเรียกโปรแกรมสมัยเก่าที่ไม่มีการแบ่งเธรดว่า เป็น heavy-weight และโปรแกรมแบบหลายเธรดว่าเป็น light-weight

การแบ่งโพรเซสออกเป็นเธรดๆ เพื่อแบ่งงานกันทำ ยังทำให้การเขียนโปรแกรมนั้นง่ายขึ้น เพราะไม่ต้องเขียนกลไกการตอบสนองข้อมูลผลลัพธ์ที่อาจจะได้คืนมาไม่พร้อมกัน ไม่เป็นลำดับกัน รวมทั้งหากจะเพิ่มประสิทธิภาพ เช่นความเร็วในการตอบสนอง ก็ไม่ต้องค้นหาวิธีการจัดการที่จะต้องสลับสับเปลี่ยนการเฝ้าดู I/O เพื่อรอคอยคำตอบ ในขณะที่กลไกดังกล่าวกระทำได้อย่างง่ายดายโดยการแบ่งเธรดออกเป็นหลายเธรดแล้วให้แต่ละเธรดเฝ้ารอข้อมูลเฉพาะจาก I/O ที่เกี่ยวข้อง นอกจากนี้ยังสามารถเพิ่มความเร็วของการทำงานของโปรแกรม (ดังที่ได้กล่าวมาในตอนต้น) และทำให้ซีพียูถูกใช้งานได้เต็มประสิทธิภาพมากขึ้น (โดยเฉพาะกรณีที่มีโปรแกรมผู้ใช้ทำงานหลักเพียงโปรแกรมเดียว ที่เฝ้าจัดการกับข้อมูลและ I/O ที่หลากหลาย)

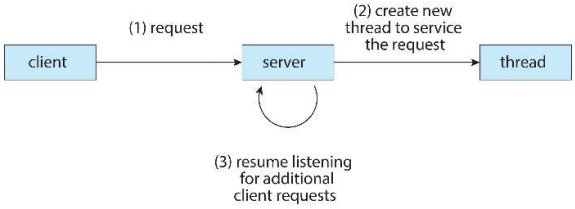
เคอร์เนิลของระบบปฏิบัติการ ในสมัยก่อนอาจจะเป็นแบบเธรดเดียว แต่ในระบบปฏิบัติการสมัยใหม่ โดยส่วนมากจะเป็นแบบหลายเธรดทั้งสิ้น

โดยสรุป ผลของการแบ่งงานเป็นหลายเธรด ได้ประโยชน์คือ

- **การตอบสนองของระบบที่ดีขึ้น ราบรื่นขึ้น (responsiveness)** สำหรับโปรแกรมที่ไม่มีการแตกเธรด และที่มีการเข้าถึงทรัพยากรหลายส่วนนั้น ในขณะที่กำลังรอคอยทรัพยากรบางส่วน (block-หยุดรอข้อมูลอยู่ ทำให้ทำงานต่อไปไม่ได้) จะส่งผลโปรแกรมนั้นไม่ตอบสนองการทำงานอื่นใดกับผู้ใช้ได้อีก แต่หากเป็นโปรแกรมหลายเธรด เราสามารถแบ่งเธรดการจัดการทรัพยากรเหล่านั้นแยกออกไปต่างหาก ในขณะที่เธรดนั้นๆรอคอยอยู่ เธรดอื่นๆ เช่นเธรดที่ใช้ติดต่อกับผู้ใช้ ก็ยังทำงานต่อไปได้
- **การสนับสนุนการแบ่งปันทรัพยากรและการสื่อสารระหว่างงานที่ทำหน้าที่ในกลุ่มเดียวกัน (resource sharing)** ในเนื้อหาการเรียนครั้งก่อน จะเห็นได้ชัดว่า การแบ่งงานออกเป็นหลายโพรเซสเพื่อแบ่งหน้าที่การทำงานนั้น ทรัพยากรของโพรเซสจะถูกจัดสรรแยกกันเป็นการเฉพาะ หากโพรเซสต้องการจะสื่อสารกัน หรือต้องการใช้ทรัพยากรร่วมกัน จะต้องมีกลไกวุ่นวาย จึงเป็นการใช้ทรัพยากรที่ไม่คุ้มค่านัก โดยเฉพาะอย่างยิ่งหากทั้งสองโพรเซสนั้นมีกลไกการทำงานที่เหมือนๆ กัน หรือใช้ I/O ที่ใช้งานไม่พร้อมกัน ในการจัดการแบบหลายเธรด เราสามารถแบ่งใช้พื้นที่ชุดคำสั่งร่วมกันได้ ผลัดกันใช้ทรัพยากร I/O ด้วยกันได้ แลกเปลี่ยนข้อมูลผ่านทางพื้นที่เก็บข้อมูลร่วม(ผ่านตัวแปรส่วนกลาง) ทำให้การพัฒนาโปรแกรมสะดวกมากขึ้น
- **การประหยัดทรัพยากรในการจัดการ (Economy)** จากที่กล่าวมาในข้างต้น การที่เธรดสองตัวขึ้นไป มีชุดคำสั่งที่ทำงานเหมือนกัน สามารถแบ่งใช้พื้นที่เก็บชุดคำสั่งร่วมกันได้ ทำให้ระบบปฏิบัติการไม่ต้องจองพื้นที่หน่วยความจำเพื่อเก็บชุดคำสั่งแยกกันอิสระเหมือนกับกรณีการรันโพรเซสหลายตัวพร้อมๆ กัน ทำให้ประหยัดหน่วยความจำที่ต้องการจากระบบ ระบบปฏิบัติการสามารถนำหน่วยความจำที่ประหยัดได้ไปใช้กับงานอื่น นอกจากนี้ กลไกการสร้างและทำลายโพรเซส มีความซับซ้อนกว่าการสร้างและทำลายเธรด หมายถึงเวลาที่สูญเสียไปในการกระทำดังกล่าวสูงว่า การแบ่งเธรดในการทำงานจึงส่งผลด้านประสิทธิภาพมากกว่าการแบ่งเป็นหลายโพรเซสเพื่อทำงานในลักษณะเดียวกัน
- **การใช้ระบบคอมพิวเตอร์แบบหลายคอร์ให้คุ้มค่า (Utilization of multiprocessor architectures)** โปรแกรมที่แบ่งเธรดเพื่อคำนวณข้อมูลจำนวนมาก จะเอื้ออำนวยให้ระบบปฏิบัติการสามารถแจกงานแต่ละเธรดไปรันบนซีพียูแต่ละคอร์ ในระบบคอมพิวเตอร์ที่มีซีพียูมากกว่าหนึ่งคอร์ ซึ่งนิยมใช้กันในปัจจุบัน ดังนั้นโปรแกรมที่แตกหลายเธรด สามารถใช้ทรัพยากรซีพียูได้คุ้มค่าขึ้น (ไม่มีซีพียูว่างงานแบบในกรณีโปรแกรมแบบเธรดเดียว) และยังส่งผลให้ผู้พัฒนาโปรแกรมสามารถเขียนโปรแกรมที่ให้ประสิทธิภาพการทำงานที่สูงขึ้นได้แบบทวีคูณเมื่อไปรันบนระบบที่มีจำนวนคอร์มากขึ้นได้ด้วย

สำหรับการประยุกต์ใช้งานแบบหลายเธรดกับโพรเซสบนเครื่องให้บริการนั้น จะมีประโยชน์มากจากการที่การให้บริการของเครื่องให้บริการนั้น มักจะเป็นการให้บริการในลักษณะที่เหมือนกันกับผู้ขอรับบริการจำนวนมาก เช่นเครื่องให้บริการเว็บ (web server) ที่รับการเชื่อมต่อร้องขอองค์ประกอบต่างๆ บนหน้าเว็บ เช่นเอกสาร HTML ไฟล์ Javascript รูปภาพ ฯลฯ และมักจะมาจากผู้ใช้งานมากมายต่างแหล่งที่มา ในลักษณะเช่นนี้ ตัวโพรเซสที่ทำหน้าที่ให้บริการ จะแตกเธรดใหม่ออกหนึ่งเธรดในทุกๆ การ

เชื่อมต่อใหม่หนึ่งตัว โดยที่ชุดคำสั่ง(ฟังก์ชันที่ใช้จัดการ) การเชื่อมต่อแต่ละเธรดนั้นจะเป็นชุดคำสั่งเดียวกัน แต่จะจัดการกับข้อมูลที่จะให้บริการแตกต่างกันไป ลักษณะนี้จะทำให้การพัฒนาโปรแกรมเพื่อให้บริการทำได้โดยง่าย และสามารถประหยัดหน่วยความจำ (เพราะชุดคำสั่งที่ให้บริการต่างๆ นั้นใช้ร่วมกันได้)



3.2 โปรแกรมหลายเธรดกับระบบคอมพิวเตอร์หลายหน่วยประมวลผล

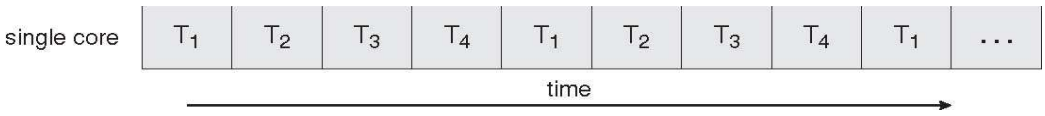
ในปัจจุบัน ระบบคอมพิวเตอร์ที่มีหลายหน่วยประมวลผลแบบ SMP (Symmetric Multi-Processors) ได้กลายเป็นอุปกรณ์ที่คนทั่วไปสามารถหามาใช้งานส่วนตัวได้ และการที่คอมพิวเตอรมีหลายหน่วยประมวลผล จึงทำให้สามารถรองรับกลไกใหม่ๆ ดังนี้

- การทำงานแบบขนาน (Paralellism) ภายใต้แนวคิดการทำงานแบบหลายภารกิจ (multitasking) บนคอมพิวเตอร์ที่มีหน่วยประมวลผลเพียงหน่วยเดียว การทำงานแบบหลายภารกิจจึงอยู่ในรูปของการสลับโพรเซสให้แต่ละโพรเซสได้เข้าครอบครองซีพียูผลัดกันไป การที่มีหลายหน่วยประมวลผล จึงสามารถทำให้มีโพรเซส(หรือเธรด) มากกว่าหนึ่งตัวที่สามารถทำงานได้พร้อมกันอย่างแท้จริง (เช่น หากมี 8 คอร์ ย่อมหมายความว่าสามารถมีโพรเซสหรือเธรดที่ทำงานพร้อมกัน ณ ขณะนั้นๆ ได้ถึง 8 ตัวในเวลาเดียวกัน) และส่งผลทำให้งานที่ต้องการเสร็จด้วยเวลาที่เร็วขึ้น
- การทำงานไปพร้อมกัน (concurrency) เมื่อมองภายใต้มุมมองของโพรเซสหรือเธรดที่จัดการงานต่างๆ ที่ได้ถูกออกแบบให้ทำงานไปพร้อมๆ กันเพื่อให้แต่ละโพรเซส/เธรด บรรลุจุดประสงค์อย่างใดอย่างหนึ่ง การที่คอมพิวเตอร์มีหลายคอร์ จึงทำให้กลไกการทำงานไปพร้อมกัน เกิดขึ้นในลักษณะที่พร้อมกันอย่างแท้จริง

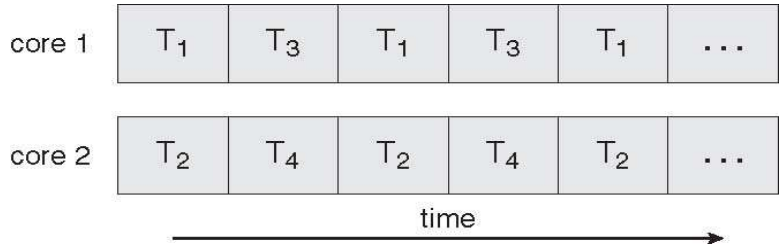
การพัฒนาโปรแกรมในปัจจุบัน มีประเด็นท้าทายในจุดที่จะสามารถรีดประสิทธิภาพสูงที่สุดออกจากระบบคอมพิวเตอร์สมัยใหม่ได้อย่างไร ปัญหาการพัฒนาโปรแกรมที่มีหลายเธรด(หรือหลายโพรเซส)มีดังเช่น

- การจะแบ่งส่วนการทำงานของโปรแกรมออกเป็นหลายส่วนให้ทำงานไปพร้อมๆ กันได้อย่างไร (dividing activities)
- การจะทำให้งานแต่ละส่วนที่แบ่งออกไปนั้น มีอัตราส่วนการแบ่งที่สมดุล ไม่ทำให้ซีพียูตัวใดต้องว่างงาน ในขณะที่ตัวอื่นต้องทำงานหนัก (balance)
- การจะแบ่งข้อมูลให้แก่ส่วนการทำงานของโปรแกรมแต่ละส่วนอย่างไร (data splitting)
- การจะดูว่าซีพียูแต่ละตัวอาจจะต้องใช้ข้อมูลที่ได้จากซีพียูอีกตัวหรือไม่อย่างไร (ต้องส่งผลการคำนวณจากคอร์หนึ่งไปทำงานต่อในอีกงานซึ่งกำลังรันอยู่บนอีกคอร์หนึ่ง รวมถึงข้อมูลที่แบ่งออกไปที่อาจจะต้องมีส่วนซ้ำซ้อนกัน จะระมัดระวังอย่างไรให้ข้อมูลเหล่านั้นต้องเหมือนกันตลอดเวลา หรือระมัดระวังในการแย่งใช้ข้อมูลชุดเดียวกันอย่างไร (Data dependency)
- การทดสอบระบบและแก้บักระบบอย่างไรในสภาพที่มีงานหลายงานต้องรันพร้อมกัน (Testing and Debugging)

ในระบบคอมพิวเตอร์ที่มีซีพียูหน่วยเดียว ซีพียูจะแบ่งทรัพยากรเวลาการเข้าครอบครองซีพียูให้แต่ละเธรดสลับกันเข้าทำงาน ดังรูป



ในระบบคอมพิวเตอร์ที่มีซีพียูหลายคอร์ ระบบปฏิบัติการจะสามารถแบ่งเธรดกระจายงานไปยังซีพียูแต่ละคอร์ ทำให้สามารถเพิ่มความเร็วในการประมวลผลได้อย่างเต็มที่



การจัดการงานแบบขนาน (parallelism) สำหรับโพรเซสหนึ่งๆ นั้น อาจจะแบ่งออกเป็นสองประเภทได้คือ

- **Data Parallelism (การประมวลข้อมูลแบบขนาน)** ในลักษณะเช่นนี้ เราจะแบ่งข้อมูล(ขนาดใหญ่)ที่มีออกเป็นส่วนย่อยๆ เพื่อส่งแต่ละส่วนให้กับเธรดเพื่อประมวลผลไปพร้อมๆ กัน ในลักษณะเช่นนี้ เรามักจะเห็นว่าชุดคำสั่งในแต่ละเธรดมักจะใช้ร่วมกัน (เรียกใช้ฟังก์ชันจัดการเธรดตัวเดียวกัน)
- **Task Parallelism (การประมวลภารกิจแบบขนาน)** การทำงานในลักษณะนี้ เราจะแบ่งงานที่ไม่สัมพันธ์กัน ออกเป็นงานย่อยๆ หลายๆ งาน แล้วกระจายให้แต่ละเธรดไปจัดการงานแต่ละตัว ตัวอย่างเช่น เราอาจจะต้องจัดการกับอินพุต/เอาต์พุต ที่แตกต่างกันหลายประเภท เราก็จะแตกเธรดออกเป็นหลายๆ เธรด แต่ละเธรดจะจัดการกับอินพุตหรือเอาต์พุตแต่ละตัวเป็นอิสระจากกัน เป็นต้น

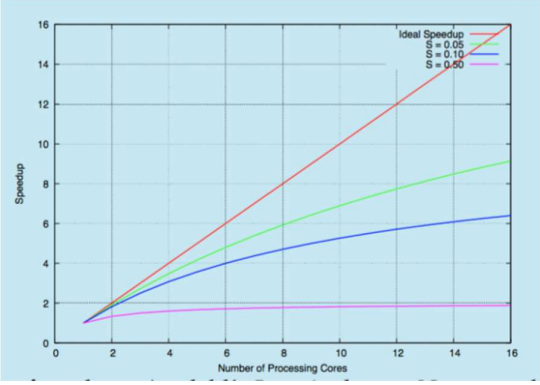
กฎของแอมดาห์ล (Amdahl's Law)

- คำนวณประสิทธิภาพที่เพิ่มขึ้นจากการเพิ่มจำนวนคอร์ประมวลผลให้กับงานที่มีส่วนการทำงานที่ต้องทำงานแบบต่อเนื่องเป็นลำดับ (serial) และที่สามารถทำงานแบบขนานกันไป (parallel)
  - S = ส่วนที่ต้องทำงานไปตามลำดับกัน
  - N = จำนวนคอร์ที่มีให้ใช้

$$Speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

  - ตัวอย่างเช่น สมมติว่างานหนึ่งมีส่วนขนาน 75% และส่วนที่ต้องทำงานแบบเรียงลำดับ 25% การพัฒนาโปรแกรมในส่วนขนานให้สามารถกระจายงานออกไปรันหลายโพรเซส/เธรด บนคอมพิวเตอร์ 2 คอร์ จะทำให้เร็วขึ้นได้ราว 1.6 เท่า
  - ในกรณีที่คอมพิวเตอร์มีจำนวนคอร์มากจนไม่จำกัด ความเร็วที่เพิ่มขึ้นก็จะเท่ากับ 1/S หรือหมายความว่าความเร็วของงานที่จะเร็วขึ้นได้ ขึ้นอยู่กับส่วนงานที่ต้องทำงานต่อเนื่องกันนั้นว่ามีสัดส่วนเท่าใด (หรือกล่าวในอีกนัยหนึ่ง หากงานที่ต้องทำมีแต่ส่วนที่ต้องทำงานต่อเนื่องกันไป การเพิ่มจำนวนคอร์ให้ระบบก็ไม่มีผลให้ทำงานเสร็จเร็วขึ้นแต่อย่างใด)

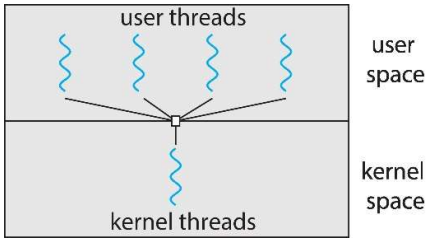
ในทางปฏิบัตินั้น ด้วยกลไก overhead อื่นๆ ของระบบ และความสามารถในการแบ่งงานให้ทำงานแบบขนานได้อย่างเต็มรูปแบบ(และต้องสามารถสมดุลงานที่ส่งให้คอร์ต่างๆได้) ทำให้ความเร็วที่เพิ่มขึ้นจะไม่ถึงตามที่ทฤษฎีได้กล่าวได้ งานแต่ละงานที่ถูกออกแบบประยุกต์เพิ่มส่วนการทำงานแบบขนาน จึงได้ประสิทธิผลที่แตกต่างกัน ตัวอย่างดังรูปด้านล่าง



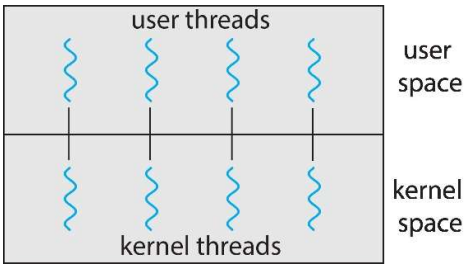
3.3 การจัดการเธรดผู้ใช้ และการจัดการเธรดของระบบปฏิบัติการ

โพรเซสที่รันอยู่บนระบบคอมพิวเตอร์ ถูกแบ่งออกเป็นสองกลุ่ม คือโพรเซสของผู้ใช้ (user process) และโพรเซสระบบ (kernel process) ซึ่งโพรเซสของระบบส่วนหนึ่งก็มีเพื่อรองรับการเรียกใช้งานของโพรเซสผู้ใช้ (เช่นการทำ system call) กลไกการจัดการจัดการเธรดของโพรเซสระบบ และเธรดของโพรเซสผู้ใช้ในระบบปฏิบัติการมีการจัดการแตกต่างกันไปดังนี้

- **การจัดการแบบหลายตัวต่อหนึ่ง (Many-to-One model)** ในการจัดการแบบนี้ เคอร์เนิลจะมีเธรดรองรับการร้องขอของเธรดผู้ใช้งานเพียงเธรดเดียว ต่อเธรดของผู้ใช้ทุกตัวในโพรเซสเดียวกัน ข้อดีของการจัดการแบบนี้คือ ระบบปฏิบัติการจะไม่เปลืองทรัพยากรมากในการสร้างเธรดขึ้นมาตอบสนองต่อการร้องขอของเธรดผู้ใช้ แต่ก็มีข้อเสียคือ หากเธรดผู้ใช้เธรดใดเธรดหนึ่งร้องขอทรัพยากรและรอคอยข้อมูลอยู่

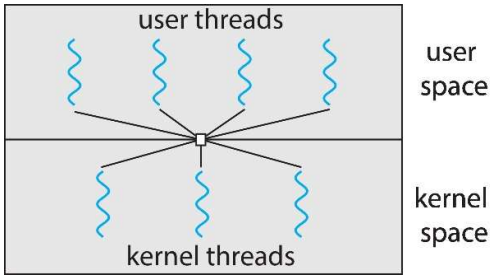


เธรดอื่นๆ ในโปรแกรมก็จะไม่สามารถทำงานร้องขอการติดต่อกับเคอร์เนิลเธรดได้อีก เพราะเคอร์เนิลเธรดไม่ตอบสนองเนื่องจากยังต้องทำงานให้เธรดผู้ใช้ตัวดังกล่าว และยังส่งผลให้ไม่สามารถกระจายเธรดแต่ละตัวของโพรเซสไปยังซีพียูคอร์อื่นๆ ได้เพราะกลไกการทำงานของโพรเซสต้องผูกกับเธรดของเคอร์เนิลตัวเดียว
    - ในกรณีที่เธรดใดเธรดหนึ่งมีปัญหา หรือต้องหยุดทำงาน ก็จะทำให้ทุกเธรดต้องหยุดตามไปด้วย
    - การใช้งานในรูปแบบนี้แทบจะไม่มีให้เห็นในระบบปฏิบัติการในปัจจุบันแล้ว
- **การจัดการแบบหนึ่งต่อหนึ่ง (One-to-One model)** ในการจัดการแบบนี้แก้ปัญหาเรื่องการต้องแบ่งใช้เคอร์เนิลเธรดตัวเดียวทั้งโพรเซส โดยการสร้างเคอร์เนิลเธรดขึ้นมารองรับในแต่ละยูสเซอร์เธรด ดังนั้นหากเธรดผู้ใช้ตัวใดตัวหนึ่งร้องขอบริการจากเคอร์เนิลเธรด เธรดผู้ใช้ตัวอื่นก็ยังสามารถร้องขอบริการจากเคอร์เนิลเธรดของตัวเองต่อไปได้โดยไม่ต้องหยุดรอ ส่งผลทำให้ประสิทธิภาพดีขึ้น และยังเอื้ออำนวยต่อการรันบนระบบหลายซีพียู แต่ข้อเสียก็เป็นเรื่องทรัพยากรระบบเพราะต้องรองรับจำนวนเคอร์เนิลเธรดที่มีมากขึ้น



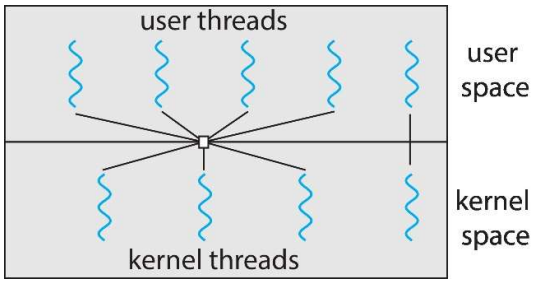
    - ระบบปฏิบัติการที่ใช้กันแพร่หลายในปัจจุบันหลายตัวใช้สถาปัตยกรรมนี้ (ลินุกซ์ และวินโดส์ เป็นต้น)
- **การจัดการแบบหลายตัวต่อหลายตัว (Many-to-Many model)** ในลักษณะเช่นนี้ ระบบปฏิบัติการจะสร้างเคอร์เนิลเธรดขึ้นมาตอบสนองการใช้งานต่อยูสเซอร์เธรดในอัตราส่วนที่อาจจะมีจำนวนน้อยกว่า หรือเท่ากันกับจำนวนเธรดของผู้ใช้ได้ และในการใช้งานนั้น เธรดผู้ใช้หนึ่งๆ ก็จะสามารถเข้าใช้เคอร์เนิลเธรดที่รอรับบริการอยู่ได้ทันที โดยไม่ต้องมีคู่เธรด

ประจำ ส่งผลทำให้โพรเซสสามารถสร้างเธรดจำนวนมากได้ตามต้องการโดยที่ระบบปฏิบัติการไม่จำเป็นต้องเปลือง ทรัพยากรสร้างเคอร์เนิลเธรดจำนวนเท่ากันมารองรับเหมือนแบบ one-to-one และก็ไม่มีปรากฏการณ์ต้องหยุดรอเธรดเหมือนกับกรณี many-to-one เท่ากับมีผลดีของทั้งสองแนวคิดข้างต้น



- ระบบปฏิบัติการที่ต้องจัดสรรทรัพยากรจำนวนมาก และมีงานที่ต้องรันจำนวนมาก มักจะมีตัวเลือกให้ใช้การจัดการเธรดแบบนี้ได้ เช่นระบบปฏิบัติการวินโดวส์เซอร์เวอร์ จะมีโมดูล ThreadFiber ให้เลือกใช้

- **โมเด็ลแบบสองระดับ (two-level model)** เป็นการอนุญาตให้เธรดผู้ใช้บางตัวล็อคขอบริการกับเคอร์เนิลเธรดตัวใดตัวหนึ่งเป็นการเฉพาะ ซึ่งเป็นกลไกเสริมมาจาก many-to-many แต่ก็ยอมให้มีบางยูสเซอร์เธรดทำ one-to-one ได้ด้วย



  - แก้ปัญหาเรื่อง overhead ในการจัดการแบบ many-to-many และแก้ไขปัญหาในกรณีที่อาจจะร้องขอบริการ system call พร้อมๆ กันจนทำให้จำนวนเธรดระบบเต็ม และเกิดการชะงักในการให้บริการกับยูสเซอร์เธรดที่เป็น critical

3.4 ความหลากหลายในการจัดการเธรด

ประเด็นในการจัดการเธรดกับโพรเซสที่ใช้การ fork()

- กลไกของการ fork() ซึ่งหมายถึงการสร้างโพรเซสใหม่ขึ้นหนึ่งโพรเซส ที่ได้เรียนไปในครั้งที่แล้วนั้น ภายใต้ระบบปฏิบัติการแบบหลายเธรด การ fork() มีความเป็นไปได้สองกรณีสำหรับโพรเซสที่ได้แตกเธรดออกเป็นหลายเธรดแล้ว กรณีแรกคือ โพรเซสใหม่ที่ถูกสร้างขึ้นมานั้น มีจำนวนเธรดที่แตกอยู่ในทันทีเท่ากับจำนวนเธรดของโพรเซสพ่อแม่กับอีกกรณีหนึ่ง โพรเซสใหม่จะประกอบไปด้วยเธรดตั้งต้นเพียงตัวเดียว ในระบบปฏิบัติการแต่ละตัวก็อาจจะจัดการในกรณีนี้ในลักษณะที่แตกต่างกัน หรือบางระบบปฏิบัติการก็จะมี fork() ทั้งสองเวอร์ชันให้เลือก
- การ fork() และตามด้วย exec() เพื่อสั่งรันโปรแกรมใหม่ จากเนื้อหาครั้งที่แล้วจะเห็นถึงตัวอย่างที่จำลองโพรเซสใหม่ขึ้นมาอีกตัวหนึ่งแต่ก็มิได้ใช้ชุดคำสั่งเดิมในการประมวล แต่หันไปใช้คำสั่ง exec() เพื่อโหลดโปรแกรมตัวใหม่ขึ้นมาแทนโพรเซสที่เพิ่งจำลองออกมาใหม่ ทำให้เกิดเส้นทางการเดินทางของโพรเซสทั้งสองที่แตกต่างกันไปอย่างสิ้นเชิงเพราะใช้ชุดคำสั่งคนละชุด ในกรณีเช่นนี้ ระบบปฏิบัติการโดยทั่วไปจะไม่สนว่าโพรเซสเดิมนั้นเคยมีเธรดรันอยู่กี่เธรด เพราะในกรณีเช่นนี้ โพรเซสที่จำลองขึ้นใหม่จะถูกสั่งให้จบการทำงานและแทนที่ด้วยโปรแกรมใหม่อยู่แล้ว
- กลไกการยกเลิกการทำงานของเธรดก่อนกำหนด (Thread cancellation) ในการประมวลข้อมูลขนาดใหญ่ เช่นการค้นหาข้อมูลจากดาต้าเบสหลายตัว ผู้พัฒนาโปรแกรมอาจจะเขียนโปรแกรมแตกเธรดออกเป็นหลายๆ ตัวแล้วให้แต่ละตัวเข้าค้นหาดาต้าเบสแต่ละตัวแยกกันโดยอิสระ แล้วพิจารณาผลการค้นหาที่ได้เร็วที่สุด ในลักษณะนี้เมื่อมีเธรดหนึ่งได้รับคำตอบ เธรดอื่นๆ ก็ไม่จำเป็นต้องรันต่อไปให้เสียเวลาอีก หรือในกรณีของเว็บเบราว์เซอร์ เมื่อผู้ใช้เปิดหน้าเว็บและโหลดรูปประกอบหน้าเว็บขึ้นมาแสดง การโหลดรูปนั้นจะเป็นภาระของเธรดหลายตัวที่แบ่งกันไปโหลดรูปของตน ผู้ใช้อาจจะไม่ทนรอให้หน้าเพจโหลดเสร็จสิ้น แต่อาจคลิกไปดูหน้าถัดไป กรณีเช่นนี้เธรดที่กำลังโหลดรูปที่ยังไม่เสร็จ ก็ต้องจบการทำงานก่อนกำหนดทั้งหมด กลไกการยกเลิกเธรดที่ใช้กันในปัจจุบันมีสองวิธีดังนี้

- Asynchronous cancellation การที่เธรดหนึ่งสั่งให้อีกเธรดหนึ่งหยุดทำงานโดยทันที กรณีเช่นนี้จะทำให้ทรัพยากรที่ที่ขอเพิ่มเติมมาโดยเธรดที่ถูกสั่งให้หยุดการทำงาน อาจมิได้รับการพิจารณาสั่งให้คืนแก่ระบบปฏิบัติการอย่างที่ควรจะเป็น
- Deferred cancellation การที่แต่ละเธรดมีกลไกการตรวจเช็คว่าตนควรหยุดการทำงานก่อนกำหนดหรือไม่ และถ้ามีคำตอบว่าให้หยุด เธรดก็จะจบการทำงานด้วยตนเอง (ในลักษณะเช่นนี้ เธรดก็จะสามารถคืนทรัพยากรที่ตนขอเพิ่มเติมมาได้อย่างสะดวก)

- การจัดการเหตุการณ์หรือสัญญาณที่เกิดขึ้นภายในระบบ (Signal Handling) ซึ่งเหตุการณ์บางอย่างเกิดขึ้นจากภายในตัวโพรเซสเอง เช่นมีเธรดหนึ่งพยายามติดต่อพื้นที่หน่วยความจำนอกบริเวณที่กำหนด หรือเกิดการหารด้วยศูนย์ เหตุการณ์ที่เกิดขึ้นนี้เรียกว่า synchronous signals กับอีกกรณีที่เหตุการณ์อาจจะเกิดจากการทำงานภายนอกที่มีจุดมุ่งหมายส่งให้กับโพรเซสแต่มาจากองค์ประกอบของระบบอื่นที่ทำงานคู่ขนานกันไปกับโพรเซสและโพรเซสจะไม่สามารถทราบได้ว่า ณ จุดใดของคำสั่งจะได้รับผลกระทบ เช่น เมื่อผู้ใช้สั่ง CTRL-BREAK หรือ CTRL-C เพื่อหยุดการทำงานของโปรแกรม ลักษณะสัญญาณดังกล่าวเรียกว่า asynchrinous signals

  เมื่อมีสัญญาณเข้ามา ตามปกติแล้วจะถูกตอบสนองด้วยส่วนการทำงาน(เธรดหรือโพรเซส)ของระบบปฏิบัติการ เราเรียกว่า default signal handler แต่ผู้พัฒนาโปรแกรมอาจจะมองเห็นว่าโปรแกรมของตนอาจจะต้องการดักสัญญาณเหล่านี้มาจัดการต่อด้วยตนเอง ก็จะเขียนชุดคำสั่งเพื่อตอบสนองต่อเหตุการณ์เหล่านี้เพิ่มเติม โดยจะสั่งการให้ทำงานแทนที่ (override) การตอบสนองตามปกติของระบบปฏิบัติการ เราเรียกส่วนการทำงานนี้ว่า user-defined signal handler ในโปรแกรมที่พัฒนาแบบเธรดเดียว การส่งต่อสัญญาณจากระบบปฏิบัติการก็กระทำไปยังโพรเซสที่มีเธรดเดียวนั้น แต่ในกรณีของโปรแกรมหลายเธรด ระบบปฏิบัติการจะต้องเลือกส่งสัญญาณให้เธรดใดเธรดหนึ่งตอบสนอง โดยมีแนวคิดต่างๆ ดังเช่น

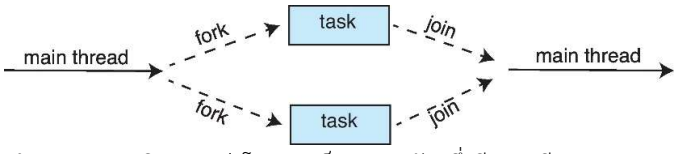
  - ส่งสัญญาณให้เธรดที่เห็นได้ชัดเจนว่าจะรอรับสัญญาณดังกล่าวเพื่อตอบสนองต่อไป
  - ส่งสัญญาณให้กับทุกเธรดของโพรเซสนั้นๆ
  - ส่งสัญญาณให้กับบางเธรดของโพรเซสนั้นๆ
  - เขียนโปรแกรมกำหนดไว้เลยว่าจะให้เธรดใดรอรับสัญญาณทุกตัวที่จะส่งเข้ามายังโพรเซสดังกล่าว ในระบบปฏิบัติการยูนิกซ์ส่วนมาก จะสามารถกำหนดได้ว่าจะรับสัญญาณใด หรือบล็อกสัญญาณใดบ้าง
  - ในกรณีที่ทราบว่าเธรดใดเป็นผู้กำเนิดสัญญาณร้องขอ ก็ส่งสัญญาณกลับไปยังเธรดเจ้าของดังกล่าว

- กลไกของการสร้างแหล่งเก็บรวมรวมเธรดเพื่อไว้รอใช้งาน (Thread Pools) แม้ว่าการสร้างเธรดใหม่ จะเสียเวลาน้อยกว่าการสร้างโพรเซสใหม่ แต่เนื่องด้วยกลไกการออกแบบโปรแกรมแบบหลายเธรด เธรดจำนวนมากอาจจะถูกสร้างขึ้นมาในช่วงเวลาสั้นๆ เพื่อใช้งานเฉพาะเจาะจงที่ใช้เวลาไม่มาก จากนั้นก็จะถูกทำลายไป ทำให้ระบบปฏิบัติการต้องเสียเวลากับการสร้างเธรดและทำลายเธรดไปอย่างมาก เพื่อเพิ่มประสิทธิภาพของโพรเซสโดยการลดจำนวนการสร้างและทำลายเธรด เธรดที่มีหน้าที่การทำงานที่รู้แน่ชัดว่าจะต้องใช้ในอนาคต และมีการใช้งานบ่อยๆ ก็อาจจะถูกสร้างขึ้นล่วงหน้าเลยและนำไปเก็บไว้ใน thread pool โดยเธรดทุกตัวจะอยู่ในสภาวะ wait เพื่อจะไม่กินทรัพยากรเวลาของซีพียู และเมื่อจะใช้งานก็เพียงแต่ดึงเธรดที่รออยู่นี้มาปฏิบัติงาน เมื่อเธรดเหล่านี้ปฏิบัติงานเสร็จก็จะถูกนำไปเก็บใน pool อีกครั้งรอการใช้งานต่อไป ข้อดีของการจัดการในลักษณะนี้ก็คือ
  - การดึงเธรดจาก thread pool มาใช้งานและนำไปเก็บใน pool ทำได้รวดเร็ว ส่งผลทำให้ประสิทธิภาพของโพรเซสดีขึ้น
  - การสร้าง thread pool และการกำหนดจำนวน thread ไว้ล่วงหน้าชัดเจนตั้งแต่เริ่มโปรแกรม ทำให้ผู้พัฒนาโปรแกรมสามารถออกแบบกลไกการจัดการกับข้อมูลหลายอย่างพร้อมกันโดยมีเพดานบนจะไม่มีจำนวนเธรดที่ทำงานพร้อมกันมากไปกว่าที่เตรียมไว้ใน thread pool นี้ ทำให้ระบบคอมพิวเตอร์โดยรวมไม่ต้องพบกับสภาวะที่มีจำนวนเธรดรันในระบบมากจนเกินไปจนทำให้ทรัพยากรไม่พอเพียง

- Thread specific data ในการจัดการโปรแกรมหลายเธรดของระบบปฏิบัติการโดยทั่วไป มักจะยอมให้แต่ละเธรดมีพื้นที่เก็บข้อมูลใช้เฉพาะภายในเธรดของตนเอง
- Scheduler Activations ในการจัดการเธรดแบบ Many-to-Many และ Two-level จะต้องมีกลไกในการจัดการเพื่อให้การสร้างเคอร์เนิลเธรดโดยเธรดผู้ใช้ ไม่ให้มีมากเกินไป ด้วยการกำหนดโครงสร้างข้อมูลในรูปของ Light-Weight Process ที่มีจำนวนจำกัด โดย LWP แต่ละตัวจะติดต่อการทำงานกับเคอร์เนิลเธรดเป็นรายตัว เมื่อเธรดผู้ใช้ใดจะเข้าใช้บริการเคอร์เนิล ก็จะเชื่อต่อผ่านทาง LWP นี้ และในกรณีที่เคอร์เนิลเธรดที่เชื่อมต่อ LWP กำลังถูกขัดจังหวะหรือรอรับข้อมูลอยู่ LWP ก็จะถูกหยุดรอตามไปด้วย กลไกของ LWP สามารถนำไปใช้ในการจัดสรรเคอร์เนิลเธรดในโมเด็ลแบบ Many-to-many หรือ Two-level โดยระบบปฏิบัติการจัดเตรียม LWP (ซึ่งอาจเรียกอีกอย่างเพื่อให้เข้าใจง่ายขึ้นว่า Virtual Processors) ไว้ตามจำนวนที่จะอนุญาตให้มีเคอร์เนิลเธรดได้ เมื่อยูสเซอร์เธรดร้องขอการบริการจากเคอร์เนิลเธรด ในการเรียกใช้ผ่านทาง Thread Library (ชุดคำสั่งที่ใช้ในการจัดการเธรด) ภายในกลไกของไลบรารีจัดการเธรดนั้น จะร้องขอใช้งานเคอร์เนิลเธรด แต่ในการทำงานของเคอร์เนิลเธรดนั้นๆ อาจจะมีสัญญาณหรืออินเทอร์รัพต์ที่พยายามจะเข้าใช้ทรัพยากรร่วมที่เคอรเนิลนั้นใช้อยู่ หรือเหตุผลอื่นใดก็ตาม ที่จะทำให้เคอร์เนิลเธรดนั้นต้องหยุดทำงานลง ก็จะส่งสัญญาณมายัง LWP (upcall) เพื่อแจ้งว่าเคอร์เนิลเธรดดังกล่าวจะไม่พร้อมให้บริการ และจะสามารถเลือกเอาเคอร์เนิลเธรดอื่นที่พร้อมให้บริการ มาให้บริการกับยูสเซอร์เธรดที่กำลังร้องขอบริการนั้นทดแทน ด้วยกลไกนี้จะทำให้สามารถจัดสรรเคอร์เนิลเธรดที่มีอยู่ให้กับยูสเซอร์เธรดไปตามเท่าที่จำนวนเคอร์เนิลเธรดนั้นมีอยู่ และพร้อมจะให้บริการได้

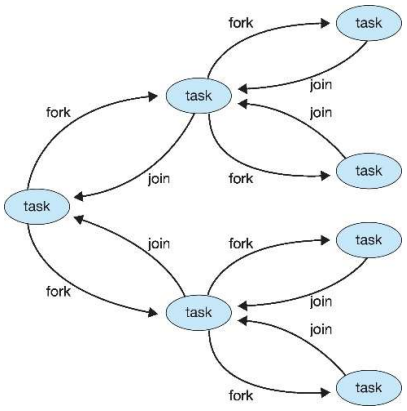
<u>แนวทางการเขียนโปรแกรมจัดการเธรด</u>

การแตกเธรดใหม่และการรวมเธรดในลินุกซ์และยูนิกซ์ ใช้กลไกการ fork เธรด และการ join เธรดดังรูป



เมื่อโพรเซสเริ่มต้นทำงาน เราจะพิจารณาว่าโพรเซสเป็นเธรดหลัก (ซึ่งมีเธรดเดียว) การแตกเธรดหรือการ fork เธรด (pthread_create()) จึงเป็นการสร้างเธรดใหม่ขึ้นใช้งานหนึ่งตัว โดยเราจะส่งค่าอ้างอิงไปยังเธรดฟังก์ชัน ให้กับฟังก์ชันแตกเธรดเพื่อให้เธรดใหม่เริ่มทำงานโดยใช้ขั้นตอนที่กำหนดไว้ในเธรดฟังก์ชัน

เมื่อเธรดฟังก์ชันทำงานเสร็จ ก็จะใช้การกระโดดกลับจากเธรดฟังก์ชัน ในการนี้ เราสามารถกำหนดให้เธรดที่เป็นเมนเธรดรอให้เธรดฟังก์ชันทำงานจบ แล้วจึงค่อยดำเนินการต่อได้โดยการใช้การ join เธรด (pthread_join()) ซึ่งมีลักษณะคล้ายคลึงกันกับการ wait() ของการแตกโพรเซส เธรดแต่ละตัวสามารถแตกเธรดย่อยออกไปได้ไม่จำกัด (และแต่ละเธรดสามารถมีเธรดลูกได้มากมาย ไม่จำกัดแค่สองตัว)



ปฏิบัติการ

1. ตัวอย่างการรันแบบหลายเธรดในลินุกซ์

อนึ่ง ไลบรารีสำหรับการจัดการเธรดตามปกติจะไม่ได้ถูกกำหนดให้ลิงค์เข้ากับโปรแกรม ดังนั้นในกรณีตัวอย่างนี้ จะต้องสั่งให้ linker เพิ่มไลบรารี pthread เข้าไปลิงค์ร่วมด้วย โดยการใส่พารามิเตอร์ -lpthread ตามหลังการเรียกใช้ gcc หรือในกรณีที่ใช้ KDeveloper ให้คลิก Project Obtions แล้วคลิกที่ Configure Options เพิ่ม –lpthread ที่ Linker Flags ด้วย

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

int c[4][4];
int a[4][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};
int b[4][4]={{10,-10,10,-10},{-10,10,-10,10},{10,-10,10,-10},{-10,10,-10,10}};

void showresult();
void *threadFunction(void *selector);

int main(void){
        pthread_t tid1,tid2;          // Thread ID
        pthread_attr_t attr1,attr2;   // Thread attributes
         int section1=0,section2=1;

        pthread_attr_init(&attr1);  // Get default attributes
        pthread_attr_init(&attr2);  // Get default attributes

        // Create 2 threads
        pthread_create(&tid1,&attr1,threadFunction,(void *)&section1);
        pthread_create(&tid2,&attr2,threadFunction,(void *)&section2);

       // Wait until all threads finish
        pthread_join(tid1,NULL);
        pthread_join(tid2,NULL);

        showresult();
        return 0;
}

void showresult(){
        int i,j;

        for(j=0;j<4;j++){
                printf("|");
                for(i=0;i<4;i++){
                        printf("%3d ",c[j][i]);
                }
                printf("|\n");
        }
}
void *threadFunction(void *selector){
        int sel=(int)*((int *)selector);
        int start,stop;
        int i,j;

        start=sel*2;
        stop =start+2;

        for(j=start;j<stop;j++)
                for(i=0;i<4;i++){
                        c[j][i]=a[j][i]+b[j][i];
                        // To Show how thread runs, this will slow thing down...
                        sleep(1);
                        printf("From thread%d i=%d j=%d\n",sel,i,j);
                        fflush(stdout); // Send text to display immediately
                }
        pthread_exit(0);
}
```

2. ตัวอย่างการรันแบบหลายเธรดในวินโดวส์

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

int c[4][4];
int a[4][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};
int b[4][4]={{10,-10,10,-10},{-10,10,-10,10},{10,-10,10,-10},{-10,10,-10,10}};

void showresult();
DWORD WINAPI threadFunction(LPVOID selector);

int main(void){
        DWORD tid1,tid2;                        // Thread ID
        HANDLE th1,th2;                         // Thread Handle
        int section1=0,section2=1;

        // Create 2 threads
        th1 = CreateThread(
                NULL,                           // Default security attributes
                0,                              // Default stack size
                threadFunction,                 // Thread function
                (void *)&section1,      // Thread function parameter
                0,                              // Default creation flag
                &tid1);                 // Thread ID returned.

        th2 = CreateThread(
                NULL,                           // Default security attributes
                0,                              // Default stack size
                threadFunction,                 // Thread function
                (void *)&section2,      // Thread function parameter
                0,                              // Default creation flag
                &tid2);                 // Thread ID returned.

       // Wait until all threads finish
        if((th1 && th2) != NULL){
                WaitForSingleObject(th1,INFINITE);
                WaitForSingleObject(th2,INFINITE);

                showresult();
        }
        return 0;
}

void showresult(){
        int i,j;

        for(j=0;j<4;j++){
                printf("|");
                for(i=0;i<4;i++){
                        printf("%3d ",c[j][i]);
                }
                printf("|\n");
        }
}
DWORD WINAPI threadFunction(LPVOID selector){
        int sel=(int)*((int *)selector);
        int start,stop;
        int i,j;

        start=sel*2;
        stop =start+2;

        for(j=start;j<stop;j++)
                for(i=0;i<4;i++){
                        c[j][i]=a[j][i]+b[j][i];
                        // To Show how thread runs, this will slow thing down...
                        Sleep(1000);
                        printf("From thread%d i=%d j=%d\n",sel,i,j);
                        fflush(stdout); // Send text to display immediately
                }
        return 0;
}
```

3. การประยุกต์กรณีประเด็นปัญหาผู้ผลิต-ผู้บริโภค

ตัวอย่างปฏิบัติการในบทนี้ที่ได้เห็นไปแล้วนั้น เป็นกลไกในลักษณะของการทำ data parallelism ซึ่งเราจะเห็นถึงการใช้เธรดฟังก์ชันตัวเดียวกันกับข้อมูลที่แบ่งออกเป็นสองส่วน

ในกรณีประเด็นปัญหาผู้ผลิต-ผู้บริโภคนั้น ฟังก์ชันผู้ผลิต กับฟังก์ชันผู้บริโภคมีลักษณะงานที่แตกต่างกันออกไป ในการประยุกต์การเขียนโปรแกรมแบบหลายเธรด จึงอาศัยการสร้างเธรดฟังก์ชันแยกกัน เป็นเธรดฟังก์ชันผู้ผลิต และเธรดฟังก์ชันผู้บริโภค เมื่อนักศึกษาประยุกต์โปรแกรมตัวอย่างจากบทที่ 2 มาใช้ในบทนี้ จึงต้องมีสิ่งที่ทำความเข้าใจดังนี้

- นักศึกษาไม่ต้องจองพื้นที่หน่วยความจำร่วม (share memory) เพราะสามารถนิยามตัวแปรที่จะใช้ร่วมกันระหว่างเธรด ให้เป็นตัวแปรส่วนกลางได้ (พื้นที่ตัวแปรส่วนกลางจะถูกเข้าถึงได้จากทุกๆ เธรด)
- การเรียกใช้ฟังก์ชันเพื่อแตกเธรด อาจจะไม่จำเป็นต้องส่งอาร์กิวเมนต์ให้เธรดฟังก์ชัน ในกรณีเช่นนี้ เราใช้การส่งค่าคงตัว NULL ให้ได้เลย เช่น

```
pthread_create(&tid1,&attr1,producer,NULL);
```

- ทั้งนี้ เรายังคงต้องทิ้งการนิยามอาร์กิวเมนต์ที่หัวเธรดฟังก์ชันไว้ตามเดิม เพียงแต่ไม่นำมาใช้งาน