แผนการบริหารการสอนประจำบทที่ 4

เนื้อหาประจำบท

บทที่ 4 การจัดการเธรด

1. แนวคิดในการจัดการเธรดของโปรแกรมที่กำลังดำเนินการในระบบปฏิบัติการ
2. ความแตกต่างของระบบเธรดแบบต่าง ๆ
3. การดำเนินการของเธรด

จุดประสงค์เชิงพฤติกรรม

เมื่อศึกษาบทที่ 4 แล้วนักศึกษาสามารถ

1. เข้าใจความหมายของระบบเธรดเดียว และระบบหลายเธรด ในการดำเนินการของโปรแกรม
2. อธิบายส่วนที่สำคัญต่าง ๆ ของเธรด
3. เข้าใจการดำเนินการของเธรด

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 และจากการทำแบบฝึกหัด
4. ประเมินจากการสอบระหว่างภาคและปลายภาค

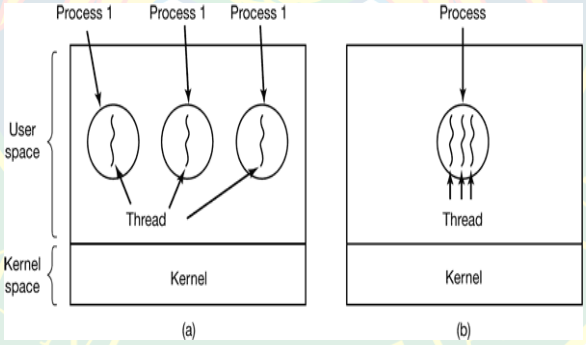
บทที่ 4
การจัดการเธรด

4.1 เธรด

เธรด คือ หน่วยการทำงานย่อยที่อยู่ในโพรเซสที่มีการแบ่งปันทรัพยากรต่าง ๆ ในโพรเซสนั้น ๆ โดยปกติโพรเซสที่มีเพียง 1 เธรดจะถูกเรียกว่า Single thread หรือเรียกอีกชื่อว่า Heavy Weight Process ซึ่งมักพบในระบบปฏิบัติการรุ่นเก่า แต่ถ้า 1 โพรเซสมีเธรดหลายเธรดจะเรียกว่า Light Weight Process (LWP) หรือ Multithread ซึ่งพบได้ในระบบปฏิบัติการรุ่นใหม่ที่ใช้กันในปัจจุบันทั่วไป และ Multithread ก็เป็นที่นิยมมากกว่า Single thread

เหตุที่ต้องมีเธรดคือ การเรียกใช้หน่วยประมวลผลให้เกิดประโยชน์สูงสุด เธรดทำให้การทำงานของโปรแกรมง่ายมีประสิทธิภาพและมีประโยชน์ต่อระบบที่มีหลายหน่วยประมวลผล (Multiprocessor) หรือมีแกนประมวลผลหลายแกน (Multicore) เพราะสามารถเรียกใช้เธรดหลาย ๆ ตัวได้พร้อม ๆ กัน โดยเธรดแต่ละตัวของโพรเซสเดียวกันจะทำงานแตกต่างกัน แต่มีความเกี่ยวข้องกันบางอย่างและต้องทำงานอยู่ภายใต้สภาพแวดล้อมเดียวกัน

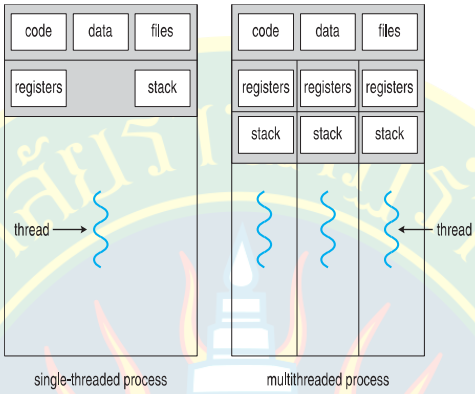
โพรเซสที่กล่าวผ่านมาเป็นการให้โปรแกรมทำงานในลักษณะมีการควบคุมเพียงหนึ่งเธรด (แต่ละโพรเซสจะประกอบด้วยเธรดเพียงเธรดเดียวเท่านั้น) แต่ระบบปฏิบัติการสมัยใหม่ในแต่ละโพรเซสสามารถมีได้หลายเธรด อาจจะกล่าวได้ว่าเธรดก็คือส่วนประกอบย่อยของโพรเซสนั่นเอง จนบางครั้งอาจเรียกเธรดว่า “Light Weight Process” ในภาพที่ 4.1 (a) คุณจะเห็นโพรเซส 3 โพรเซส แต่ละโพรเซสจะมีเลขที่ตำแหน่งเป็นของตนเอง และควบคุมเพียง 1 เท่านั้น ในภาพที่ 4.1 (b) จะเห็นว่าในแต่ละโพรเซสจะควบคุมสามเธรด โดยใช้เลขที่ตำแหน่งเดียวกันอยู่



ภาพที่ 4.1 (a) 3 โพรเซสแต่ละโพรเซสจะมี 1 Thread (b) 1 โพรเซสที่มี 3 Thread
ที่มา: Andrew S. Tanenbaum. Modern Operating System.
http://lovingod.host.sk/tanenbaum/Processes-and-threads.html

เธรดเป็นหน่วยพื้นฐานของการจัดสรรการใช้ประโยชน์ของซีพียู ภายในโพรเซสจะประกอบด้วยเธรดจะมีการแชร์โค้ด ข้อมูล และทรัพยากร เช่น ไฟล์ อุปกรณ์ต่าง ๆ เป็นต้น โพรเซสดั้งเดิม (ที่เรียกว่า Heavy weight) ที่มีการควบคุมเพียง 1 เธรด แสดงว่าทำงานได้ 1 งาน แต่ถ้าโพรเซสมีหลาย

เธรด (อาจเรียกว่า Multithread) จะทำงานได้หลายงานในเวลาเดียวกัน ภาพที่ 4.2 แสดงส่วนประกอบของโพรเซส ที่เป็น Single thread และ Multithread



ภาพที่ 4.2 โพรเซสที่เป็น Single-threaded และ Multithread
ที่มา: Abraham, S. Peter, B. G., & Greg, G. Operating system concepts 9th ed. (2013, p.164)

องค์ประกอบภายในเธรดประกอบด้วย

1) Threads ID หมายเลขเธรดที่อยู่ในโพรเซส
2) Counter ตัวนับเพื่อติดตามคำสั่งที่จะถูกดำเนินการเป็นลำดับถัดไป
3) Register หน่วยความจำเก็บค่าตัวแปรที่ทำงานอยู่ปัจจุบัน
4) Stack เก็บประวัติการทำงาน

4.2 ตัวอย่างการใช้เธรด

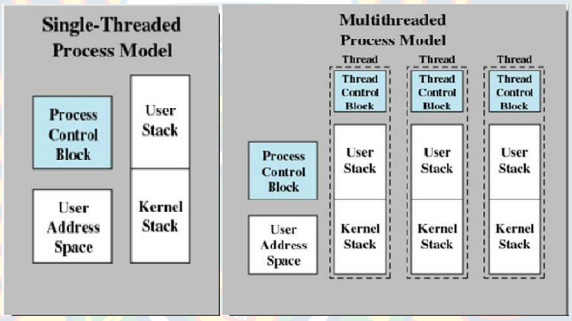
ซอฟต์แวร์ปัจจุบันที่รันกับเครื่องพีซีสมัยใหม่มีการออกแบบให้เป็น Multithread โดยแยกออกเป็นโพรเซสที่ควบคุมหลาย ๆ เธรด เช่น โปรแกรมเว็บบราวเซอร์ที่มีเธรดหนึ่งในการแสดงรูปภาพหรือเขียนข้อความในขณะที่อีกเธรดหนึ่งกำลังดึงข้อมูลจากเน็ตเวิร์ค หรือในโปรแกรมเวิร์ดโพรเซสเซอร์ที่มีหลายเธรด โดยที่เธรดหนึ่งกำลังแสดงภาพกราฟฟิก เธรดที่สองกำลังรอรับคำสั่งจากคีย์บอร์ดจากผู้ใช้ ในขณะที่เธรดที่สามกำลังตรวจสอบคำสะกดและไวยากรณ์ในลักษณะทำงานอยู่เบื้องหลัง เป็นต้น

ในสถานการณ์บางอย่างการทำงานแบบ Single อาจจะต้องการทำงานพร้อมกันหลาย ๆ งาน เช่น web server ยอมรับสิ่งที่เครื่องลูกข่ายต้องการพวก Web page image sound ถ้าเครื่องให้บริการมีระบบการทำงานแบบ single เครื่องลูกข่ายนับพัน ๆ เครื่องที่ติดต่อเข้ามาก็จะได้รับการให้บริการเพียงเครื่องเดียวเท่านั้น วิธีหนึ่งคือให้เซริฟเวอร์เรียกใช้งานโพรเซสขึ้นมาหนึ่งโพรเซสและรอรับการร้องขอ เมื่อได้รับแล้ว จะสร้างโพรเซสแยกออกมาเพื่อให้บริการในทุกการร้องขอที่ขอมา ซึ่งจริง ๆ แล้วการสร้างโพรเซสในลักษณะนี้เป็นวิธีแบบธรรมดาที่เคยใช้กันก่อนที่จะมีระบบเธรดขึ้นมาอีก การสร้างโพรเซสต้องใช้เวลาในการสร้างมากและต้องละเอียดถี่ถ้วน ดังที่แสดงให้ดูในบทก่อนหน้า ถ้ามีโพรเซสใหม่ถูกสร้างขึ้นจะมีการทำงานเหมือนกับโพรเซสเดิมที่มีอยู่แล้ว และทำให้การทำงานแบบนี้ไม่เกิด Overhead

4.3 ความแตกต่างระหว่างโพรเซสกับเธรด

ความหมายโดยทั่วไปของเธรด คือ "Light weight process" ซึ่งหมายถึงโพรเซสที่ใช้ทรัพยากร อย่างพอเพียง ภาพที่ 4.3 แสดงถึงความแตกต่างระหว่างโพรเซสกับเธรด ภาพที่ 4.3 ทางด้านซ้ายแสดงถึงโครงสร้างของโพรเซส ที่มีกับเธรดอยู่หนึ่งตัว (Single Threaded Process model) โดยมี PCB มีเนื้อที่ใช้สอยสำหรับผู้ใช้ มี Kernel stack และ User stack สำหรับจัดการกับข้อมูลต่าง ๆ ของโพรเซส เมื่อมีการประมวลผลในขณะที่โพรเซสกำลังประมวลผลอยู่ รีจีสเตอร์ของโพรเซสจะถูกควบคุมโดยโพรเซสเอง และรีจีสเตอร์เหล่านี้จะถูกเก็บไว้ใช้งาน

เมื่อรีจีสเตอร์ถูกบล็อกในส่วนที่เป็นมัลติเธรดนั้น ยังมี PCB อยู่หนึ่งตัวรวมไปถึงเนื้อที่ใช้สอยของผู้ใช้ แต่จะมีผู้ใช้และ Kernel stacks สำหรับเธรดทุกตัวรวมไปถึง Thread Control Block (TCB) ซึ่งจะเป็นที่เก็บรีจีสเตอร์ ข้อมูลลำดับความสำคัญและข้อมูลอื่น ๆ ที่เกี่ยวข้องกับสถานะภาพของเธรดนั้น ๆ เพราะฉะนั้นเธรดทั้งหมดจึงใช้ทรัพยากรของโพรเซสร่วมกัน มีการรับรู้การเปลี่ยนแปลงต่าง ๆ ที่เกิดขึ้น เช่น ถ้าเธรดตัวหนึ่งเปิดแฟ้มข้อมูลขึ้นมาอ่าน เธรดตัวอื่น ๆ สามารถอ่านแฟ้มข้อมูลเหล่านั้นได้เหมือนกัน



ภาพที่ 4.3 แสดงความแตกต่างระหว่างโพรเซสกับเธรด
ที่มา: Cardiff School of Computer Science & Informatics. Retrieved June 11, 2014 from http://www.cs.cf.ac.uk/Dave/C/node29.html

4.4 สถานะของเธรด

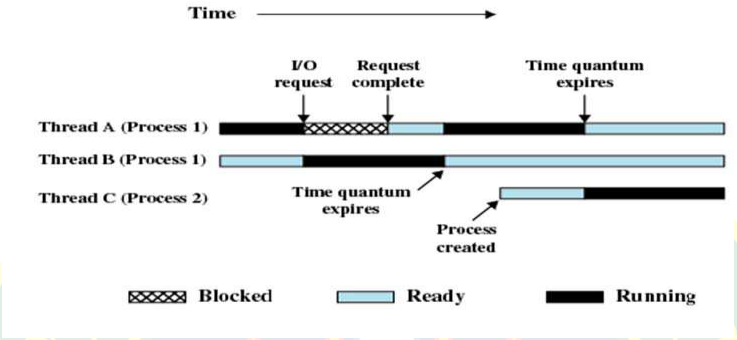
เช่นเดียวกันกับโพรเซสสถานะของเธรด จะมีอยู่สามสถานะคือ Ready, Running, และ Blocked ดังนั้นขั้นตอนที่เกี่ยวข้องกับการเปลี่ยนสถานะของเธรดคือ

1) Spawn หมายถึงการที่โพรเซสตัวหนึ่งสร้างโพรเซสอีกตัวหนึ่งขึ้นมา (ซึ่งทำให้เกิดนิยาม Parent process และ Child process) และเมื่อโพรเซสทำการ Spawn เธรดที่อยู่ในโพรเซสก็ทำการ Spawn ด้วยและเธรดที่อยู่ในโพรเซสก็สามารถที่จะ Spawn เธรดใหม่ได้ด้วย

2) Block เมื่อใดก็ตามที่เธรดต้องรอให้เหตุการณ์ใด ๆ เกิดขึ้น มันก็จะทำการบล็อกซึ่งก็จะทำให้มีการเก็บข้อมูลที่เกี่ยวข้อง เช่น User register, Program counter และ Stack pointer ไว้ และซีพียูจะไปให้บริการแก่เธรดตัวอื่นที่พร้อมสำหรับการบริการต่อไป

3) Unblock เมื่อมีเหตุการณ์ที่เธรดถูกบล็อก เธรดก็จะถูกนำไปเก็บไว้ใน Ready queue

4) Finish เมื่อเธรดสิ้นสุดการทำงานค่าต่าง ๆ ก็จะถูกส่งคืนให้กับระบบภาพแสดงตัวอย่างของเธรดในระบบ Multithreading

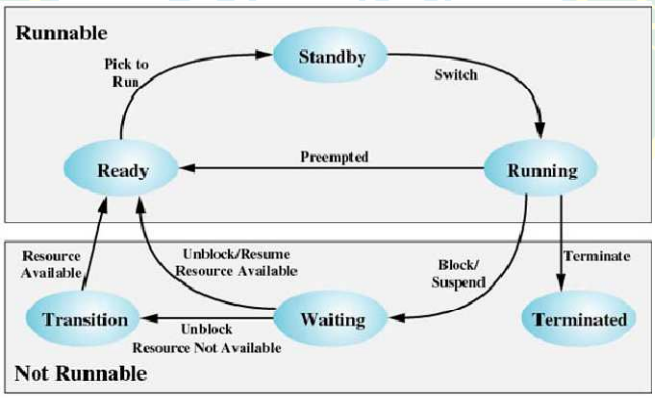


ภาพที่ 4.4 แสดงตัวอย่างของเธรดใน Multithreading system

จากภาพที่ 4.4 เรามีเธรดอยู่สามตัว โดยมีสองโพรเซสที่กำลังทำงานสลับกันไปมาอยู่ในซีพียู การประมวลผลดังกล่าวมีการส่งข้อมูลไปมาระหว่างเธรด เมื่อมีการบล็อกเกิดขึ้นจากเธรดที่กำลังทำงานอยู่ หรือเมื่อเวลาที่กำหนดไว้หมดลง การคัดเลือกเธรดสำหรับการประมวลผลขึ้นอยู่กับ อัลกอริทึ่มของการกำหนดเวลาการใช้ซีพียู (Scheduling algorithm)

4.5 เธรดในระบบปฏิบัติการวินโดว์

เธรดใน Windows นั้นเป็น Object (เช่นเดียวกับโพรเซส) โดยที่โพรเซสจะเป็นส่วนประกอบที่เกี่ยวข้องกับการใช้งานของผู้ใช้ (User job) หรือ Application ที่ต้องการใช้ทรัพยากร เช่น หน่วยความจำ หรือไฟล์ ส่วนเธรดจะเป็นส่วนประกอบสำหรับการประมวลผล (Sequentially) ซึ่งสามารถถูกอินเตอรรัพได้ ซึ่งทำให้ซีพียูสามารถให้บริการแก่เธรดตัวอื่นได้ สถานะของเธรดใน Windows แสดงไดอะแกรมในภาพที่ 4.5 ดังนี้



ภาพที่ 4.5 ไดอะแกรมสถานการณ์ทำงานของ Windows Thread

จากภาพที่ 4. 5 สามารถอธิบายสถานการทำงานของ Windows Thread ดังนี้

1) สถานะ Ready เธรดได้ถูกคัดเลือกให้เข้าทำงาน โดยมีการจัดอันดับด้วย Priority

2) สถานะ Standby เธรดได้รับการบริการจากซีพียูในอันดับถัดไปเมื่อซีพียูว่าง

3) สถานะ Running เธรดกำลังทำงานในซีพียูและจะทำไปจนกว่าจะถูกดึงออก หรือถูกบล็อกใช้เวลาที่กำหนดไว้หมด หรือเสร็จงาน ซึ่งถ้าเป็นในสองกรณีแรก เธรดก็จะถูกนำไปเก็บไว้ใน Ready queue เพื่อรอรับการบริการต่อไป

4) สถานะ Waiting เธรดจะเข้าสู่สถานะรอเมื่อ 1) ถูกบล็อก 2) รอให้งานบางอย่างเสร็จ 3) ระบบบังคับให้เธรดหยุดตัวเองลง เธรดจะกลับเข้าสู่สถานะ Ready เมื่อทรัพยากรที่เธรดต้องการมีให้ใช้

5) Transition เธรดจะเข้าสู่สถานะนี้เมื่อการรอสิ้นสุดลงและพร้อมที่จะเข้าสู่สถานะ Running แต่ทรัพยากรที่มันต้องการยังไม่มี (ระบบยังให้ไม่ได้)

6) Terminated เธรดสามารถที่จะยุติการทำงานของตัวเอง หรือจากเธรดอื่น หรือเมื่อ Parent process ยุติการทำงาน เมื่อระบบเสร็จสิ้นการทำงาน เธรดก็จะถูกดึงออกจากระบบ หรืออาจถูกเก็บไว้เพื่อการทำงานในอนาคต

4.6 ข้อได้เปรียบของ Multithreaded

การที่ระบบปฏิบัติการสนับสนุนระบบ Multithread ทำให้มีข้อได้เปรียบในด้านต่าง ๆ โดยแบ่งออกเป็น 4 ด้านหลัก ๆ ดังนี้

1) การตอบสนอง (Response) ระบบ Multithread ที่โต้ตอบแอปพลิเคชั่นจะยินยอมให้โปรแกรมยังคงดำเนินต่อไป ถึงแม้ว่าจะมีบางส่วนถูกบล็อกหรือมีการปฏิบัติที่ยาวนาน เนื่องจากการเพิ่มการโต้ตอบกับผู้ใช้นั่นเอง ยกตัวอย่างเช่น โปรแกรมเว็บบราวเซอร์ยังคงโต้ตอบกับผู้ใช้ได้ในขณะที่มีหนึ่งเธรดที่กำลังโหลดรูปภาพอยู่

2) การใช้ทรัพยากรร่วมกัน (Share resource) โดยปกติเธรดจะแชร์หน่วยความจำ และทรัพยากรของโพรเซสอยู่แล้ว ข้อได้เปรียบของการแชร์โค้ดจะทำให้แอปพลิเคชั่นสามารถมีกิจกรรมของเธรดได้หลาย ๆ กิจกรรมภายในเลขที่ตำแหน่งเดียวกัน

3) ประหยัด (Economic) การจัดสรรหน่วยความจำและทรัพยากรสำหรับการสร้างโพรเซสมีค่าใช้จ่ายมาก ในทางตรงข้ามเนื่องจากเธรดแชร์ทรัพยากรของโพรเซสที่มันอาศัยอยู่แล้ว ทำให้เกิดการประหยัดในการสร้างเธรดและทำการ Context switch ของเธรดเป็นการยากที่จะวัดความแตกต่างระหว่างการสร้างและคงสภาพโพรเซสที่มีมากกว่าเธรดได้ แต่โดยปกติแล้วจะใช้เวลาในการสร้างและคงสภาพโพรเซสสูงกว่าเวลาที่ใช้กับเธรดใน Solaris2 การสร้างโพรเซสจะช้ากว่าการสร้างเธรด 30 เท่าและ Context switch จะช้ากว่า 5 เท่า

4) รองรับการขยายระบบ (Scalability) ด้วยประโยชน์ของสถาปัตยกรรมที่รองรับการทำงานหลาย ๆ เธรด ทำให้ระบบช่วยเสริมประสิทธิภาพการทำงานของเธรดให้สูงขึ้น โดยแต่ละเธรดสามารถทำงานขนานกันไปในโพรเซสเซอร์อื่นได้ ในโพรเซสที่มีเธรดเดียวสามารถรันเพียงซีพียูเดียวเท่านั้น ไม่ว่าจะมีกี่ซีพียูก็ตามระบบมัลติเธรดในเครื่องที่มีหลายซีพียูจะเพิ่มประสิทธิภาพในการทำงานพร้อม ๆ กันได้มากขึ้น ในสถาปัตยกรรมที่มีซีพียูเดียว ซีพียูจะย้ายแต่ละเธรดให้เร็วมากขึ้น เพื่อให้ทำงานเสมือนว่าขนานกันอยู่ แต่ในความเป็นจริงจะรันเพียงเธรดเดียวเท่านั้นในเวลานั้น ๆ

4.7 เธรดสำหรับผู้ใช้และเธรดสำหรับระบบปฏิบัติการ

เธรดอาจจะแบ่งตามระดับการสนับสนุนได้ 2 แบบที่สัมพันธ์กัน คือ เธรดสำหรับผู้ใช้ง่ายที่จะถูกสร้างและอาจถูกยกเลิกก่อนเข้าเธรดสำหรับระบบปฏิบัติการได้ และเธรดสำหรับระบบปฏิบัติการรองรับเธรดสำหรับผู้ใช้และการปฏิบัติงาน

4.7.1 เธรดสำหรับผู้ใช้ (User Thread)

จะได้รับการสนับสนุนจาก Kernel ด้านบน และอยู่ในไลบรารีของเธรดในระดับของผู้ใช้ ไลบรารียังสนับสนุนการสร้างเธรดการจัดเวลา และการจัดการเธรดโดยไม่ต้องได้รับการสนับสนุนจาก Kernel เนื่องจากไม่ต้องยุ่งเกี่ยวกับเธรดระดับผู้ใช้ การสร้างเธรดและการจัดเวลา เธรดทั้งหมดจะกระทำเสร็จสิ้นภายในพื้นที่ของผู้ใช้โดยไม่จำเป็นต้องใช้ Kernel ดังนั้นเธรดในระดับผู้นำผู้ใช้สามารถสร้างและจัดการได้อย่างรวดเร็ว อย่างไรก็ตามถ้า Kernel เป็น Single thread แล้ว เธรดระดับผู้ใช้จะบล็อก System call จนเป็นเหตุให้ทุกโพรเซสถูกบล็อก ถึงแม้ว่าเธรดอื่นจะยังคงรันอยู่ในแอปพลิเคชั่นก็ตาม ไลบรารีของ User thread รวมถึง POSIX Pthreads, Windows, และ Java. Pthreads

4.7.2 เธรดสำหรับระบบปฏิบัติการ (Kernel Thread)

ได้รับการสนับสนุนโดยตรงจากระบบปฏิบัติการ โดย Kernel จะสร้าง จัดเวลา และจัดการเธรดภายในพื้นที่ของ Kernel เอง เนื่องจากระบบปฏิบัติการเป็นผู้จัดการเกี่ยวกับการสร้างและจัดการเธรดเอง จึงทำให้เธรดสำหรับระบบปฏิบัติการจะสร้างและจัดการได้ช้ากว่าเธรดสำหรับผู้ใช้ อย่างไรก็ตาม เพราะ Kernel จัดการเกี่ยวกับเธรด ดังนั้นถ้าเธรดเกิดการบล็อก System call จะทำให้ Kernel จัดการนำเอาเธรดอื่นในแอปพลิเคชั่นเข้ามาดำเนินการแทนได้ เช่นเดียวกับในสภาวะมัลติโพรเซสเซอร์ที่ Kernel สามารถจัดเธรดลงในโพรเซสเซอร์อื่นได้ ระบบปฏิบัติการที่สนับสนุนเธรดสำหรับระบบปฏิบัติการเช่น ระบบปฏิบัติการวินโดว์ ลินุกส์ แมคโอเอส และ โซลาริส เป็นต้น

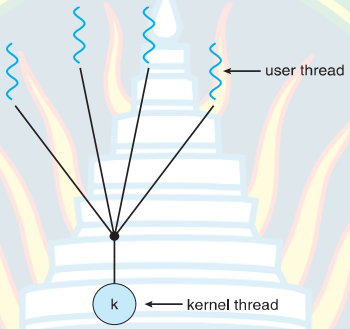
4.8 รูปแบบของเธรด

มีหลายระบบที่สนับสนุนทั้งเธรดสำหรับผู้ใช้และเธรดสำหรับระบบปฏิบัติการ ในรูปแบบที่แตกต่างกัน พิจารณาในรูปแบบทั้งสามของการดำเนินการเธรดดังนี้ เราจะอธิบายการทำงานของเธรดซึ่งมีการพัฒนามาตลอด แต่อย่างไรก็ตามการสนับสนุนการทำงานของเธรดจะขึ้นอยู่กับระดับของผู้ใช้จากเธรดของผู้ใช้หรือจาก Kernel แต่เธรดของผู้ใช้จะสนับสนุนมากกว่า Kernel และสามารถควบคุมโดยไม่ต้องใช้การสนับสนุนจาก Kernel ส่วนเธรดของ Kernel นั้นจะสนับสนุนและควบคุมโดยตรงจากระบบปฏิบัติการ และเกือบทุกรุ่นของระบบปฏิบัติการไม่ว่าจะเป็น ระบบปฏิบัติการวินโดว์ ลินุกส์ แมคโอเอส โซลาริส และยูนิกส์ (เช่น Tru64 UNIX) ที่สนับสนุนการทำงานของ Kernel

ในที่สุดแล้วเธรดของผู้ใช้และเธรดของ Kernel ก็ยังเชื่อมโยงกันอยู่ดี ในส่วนนี้สามารถพิสูจน์ความสัมพันธ์ซึ่งสามารถแบ่งออกเป็น 3 รูปแบบที่เป็นที่รู้จักกันโดยทั่วไป

4.8.1 รูปแบบ Many-to-One

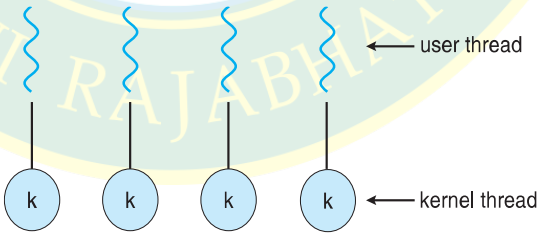
รูปแบบ Many-to-One เป็นรูปแบบที่ใช้เธรดสำหรับระบบปฏิบัติการ 1 หน่วย กับเธรดสำหรับผู้ใช้หลายหน่วย (ดังภาพที่ 4.6) การจัดการเธรดจะอยู่ในพื้นที่ของผู้ใช้ซึ่งมีประสิทธิภาพ แต่ถ้าเธรดบล็อก System call โพรเซสทั้งหมดจะถูกบล็อกไปด้วย เนื่องจากจะมีเพียงเธรดเดียวเท่านั้นที่เข้าถึง Kernel ในเวลาหนึ่ง ๆ เธรดหลาย ๆ เธรด ไม่สามารถรันขนานกันในระบบมัลติโพรเซสเซอร์ได้ ระบบที่ใช้รูปแบบนี้เช่น Green thread ซึ่งเป็นไลบรารีในโซลาริสทู (Solaris 2) นอกจากนี้ในไลบรารีของ User thread ในระบบปฏิบัติการที่ไม่สนับสนุนสำหรับระบบปฏิบัติการ จะใช้รูปแบบ Many-to-One นี้



ภาพที่ 4.6 รูปแบบ Many-to-one
ที่มา: Abraham, S. Peter, B. G., & Greg, G. Operating system concepts 9th ed. (2013, p.169)

4.8.2 รูปแบบ One-to-One

รูปแบบ One-to-One เป็นรูปแบบที่แต่ละเธรดสำหรับผู้ใช้ จะจับคู่กับเธรดสำหรับระบบปฏิบัติการ ในลักษณะ 1 ต่อ 1 (ดังภาพที่ 4.7) ทำให้สามารถทำงานพร้อมกันดีกว่าแบบ Many-to-One โดยยอมให้เธรดอื่นรันได้เมื่อเธรดบล็อก System call นอกจากนี้โมเดลนี้ยังยอมให้หลาย ๆ เธรดทำงานแบบขนานกันได้ในระบบมัลติโพรเซสเซอร์ได้อีกด้วย มีข้อที่คำนึงอยู่ข้อเดียวคือ การสร้างเธรดสำหรับผู้ใช้ จำเป็นต้องสร้างเธรดสำหรับระบบปฏิบัติการที่สัมพันธ์กัน เนื่องจากการสร้างเธรดสำหรับระบบปฏิบัติการเป็นส่วนสำคัญในเพิ่มประสิทธิภาพของแอปพลิเคชัน ระบบที่โมเดลมีข้อจำกัดที่จำนวนเธรดที่สนับสนุนในระบบได้ โมเดลนี้นำมาใช้ในระบบ เช่น ในระบบปฏิบัติการวินโดว์ เป็นต้น

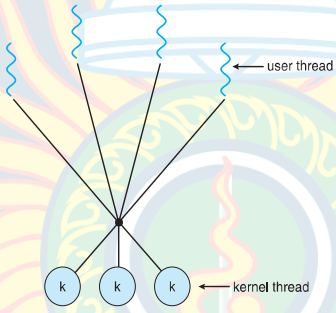


ภาพที่ 4.7 รูปแบบ one-to-one
ที่มา: Abraham, S. Peter, B. G., & Greg, G. Operating system concepts 9th ed. (2013, p.170)

4.8.3 รูปแบบ Many-to-Many

รูปแบบ Many-to-Many เป็นรูปแบบที่อาจจะมีจำนวนเธรดสำหรับผู้ใช้ มากกว่าหรือเท่ากับจำนวนเธรดสำหรับระบบปฏิบัติการก็เป็นได้ (ดังภาพที่ 4.8) จำนวนเธรดสำหรับระบบปฏิบัติการ อาจจะเป็นตัวกำหนดแอปพลิเคชันเฉพาะหรือเครื่องเฉพาะ (แอปพลิเคชั่นอาจจะมีเธรดสำหรับระบบปฏิบัติการบนมัลติโพรเซสเซอร์มากกว่าเธรดสำหรับระบบปฏิบัติการบนโพรเซสเซอร์เดียว) ในขณะที่รูปแบบ Many-to-Many ยอมให้ผู้พัฒนาสร้างเธรดสำหรับผู้ใช้ ได้ตามที่เขาต้องการ แต่จะไม่สามารถทำงานได้พร้อมกัน

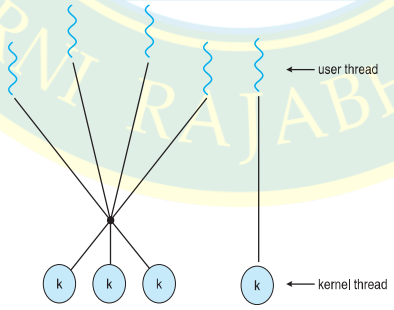
เนื่องจาก Kernel จะจัดเวลาให้ครั้งละเธรดเท่านั้น โมเดล One-to-One ยอมให้รันพร้อมกันได้ดีกว่า แต่ผู้พัฒนาต้องระวังว่าต้องไม่สร้างเธรดมากเกินไปในแอปพลิเคชั่น (ในบางครั้งอาจจะจำกัดจำนวนเธรดที่จะสร้าง) รูปแบบ Many-to-Many จะลดข้อจำกัดของรูปแบบทั้งสอง กล่าวคือ ผู้พัฒนาสามารถสร้างเธรดสำหรับผู้ใช้เท่าที่จำเป็น และสัมพันธ์กับเธรดสำหรับระบบปฏิบัติการที่สามารถรันแบบขนานในระบบมัลติโพรเซสเซอร์ นอกจากนั้น เมื่อเธรดเกิดการบล็อก System call แล้ว Kernel จะจัดเวลาเพื่อนำเธรดอื่นขึ้นมารันก่อนก็ได้ โมเดลนี้เป็นตัวอย่างของระบบปฏิบัติการ ยูนิกส์ เป็นต้น



ภาพที่ 4.8 รูปแบบ Many-to-many
ที่มา: Abraham, S. Peter, B. G., & Greg, G. Operating system concepts 9th ed. (2013, p.170)

4.9 คลังข้อมูลการจัดการเธรด

คลังข้อมูลการจัดการเธรด เป็นข้อกำหนดที่สร้างโดยโปรแกรมเมอร์ สำหรับการจัดสร้างและจัดการเกี่ยวกับเธรดโดยมีสองขั้นตอนในการทำงานคือ



ภาพที่ 4.9 รูปแบบ Two-level
ที่มา: Abraham, S. Peter, B. G., & Greg, G. Operating system concepts 9th ed. (2013, p.171)

1) การเข้าถึงแหล่งของข้อมูลทั้งหมดในเนื้อที่ของผู้ใช้ โดยไม่ผ่าน Kernel ทุกค่าและทุกโครงสร้างข้อมูลของแหล่งข้อมูลในพื้นที่ของผู้ใช้ หรือเป็นการเรียกจากฟังก์ชันเฉพาะในพื้นที่ของผู้ใช้และไม่ใช้ System call

2) การเข้าถึงเครื่องมือที่รับรองแหล่งของ Kernel โดยตรงด้วยระบบปฏิบัติการ ในส่วนนี้โค้ดและโครงสร้างข้อมูลสำหรับแหล่งข้อมูลในพื้นที่ของ Kernel การเรียกฟังก์ชันใน API แหล่งข้อมูลแบบธรรมดาจะส่งผลใน System call ถึง Kernel

Thread Library 3 อย่างหลัก ๆ ที่ใช้ในปัจจุบันคือ

1) POSIX Pthreads

2) Win32

3) Java

Pthreads เป็นเธรดมาตรฐานของ POSIX จะจัดการเกี่ยวกับระดับของผู้ใช้ หรือ ระดับของ Kernel Threads Library ของ Win32 เป็นแหล่งข้อมูลของ Kernel level ที่สะดวกของระบบวินโดวส์ เธรดของจาวาเป็น API ที่ยอมรับในการสร้างและจัดการเธรดโดยตรงในโปรแกรมจาวา อย่างไรก็ตามเพราะว่ากรณีส่วนใหญ่แล้ว JVM จะเป็นตัวทำงานทางด้านบนสุดของระบบปฏิบัติการ เธรดของจาวา API นั้นเป็นรูปแบบเครื่องมือที่ใช้ Threads library ได้สะดวกบนระบบโฮส ซึ่งคล้ายกับระบบปฏิบัติการวินโดวส์ เธรดของจาวาเป็นรูปแบบเครื่องมือที่ใช้ในระบบ Win32 API, UNIX และระบบปฏิบัติการลินุกส์เช่นเดียวกับการใช้ Pthreads

ในส่วนที่เหลือของเรื่องส่วนนี้เราจะอธิบายถึงการสร้างเธรดพื้นฐาน ซึ่งใช้ 3 Thread library และมีภาพประกอบตัวอย่าง เราออกแบบโปรแกรมของรูปแบบหลายเธรดเมื่อทำการหาผลรวมของค่าที่เป็นจำนวนเต็มบวกในเธรดที่ใช้ซึ่งรู้จักกันในรูปฟังก์ชันการหาผลรวม

$$sum = \sum_{i=0}^{N} i$$

จากตัวอย่าง ถ้า N มีค่าเท่ากับ 5 ในฟังก์ชันนี้ค่าผลรวม จาก 0 ถึง 5 คือ 15 โปรแกรมนี้จะทำงานด้วยค่าขอบบนของการหาผลรวมของเธรดมาตรฐาน ถ้าผู้ใช้ใส่ค่า 8 ผลรวมของจำนวนค่าจาก 0 ถึง 8 จะออกมาเป็นอย่างไร

4.9.1 Pthreads

Pthreads เป็นตัวพื้นฐานของ POSIX (IEEE 103.1C) เรียกได้ว่าเป็น API สำหรับการสร้างเธรดและสิ่งที่เกิดขึ้นในเวลาเดียวกัน เป็นตัวบ่งบอกถึงพฤติกรรมของเธรดโดยไม่ใช้เครื่องมือ การออกแบบระบบปฏิบัติการมักจะใช้เครื่องมือในงานที่ต้องการ ระบบปฏิบัติการส่วนใหญ่ใช้ Pthreads เป็นเครื่องมือไม่ว่าจะเป็นระบบปฏิบัติการโซราริส ลินุกส์ แมคโอเอส และยูนิกส์ เป็นเครื่องมือที่สะดวกในการใช้งานทั่วไปสำหรับงานที่หลากหลายซึ่งดีกับระบบปฏิบัติการวินโดวส์

```
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */
int main(int argc, char *argv[])
{
pthread t tid; /* the thread identifier */
pthread attr t attr; /* set of thread attributes */
if (argc != 2) {
fprintf(stderr,"usage: a.out <integer value>\n");
return -1;
}
if (atoi(argv[1]) < 0) {
fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
return -1;
}
/* get the default attributes */
pthread attr init(&attr);
/* create the thread */
pthread create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread join(tid,NULL);
printf("sum = %d\n",sum);
}
/* The thread will begin control in this function */
void *runner(void *param)
{
int i, upper = atoi(param);
sum = 0;
for (i = 1; i <= upper; i++)
sum += i;
pthread exit(0);
}
```

ภาพที่ 4.10 โค้ดภาษาซี ในการเขียน Multithreaded C program โดยการใช้ Pthreads API.
ที่มา: Abraham, S. Peter, B. G., & Greg, G. Operating system concepts 9th ed. (2013, p.173)

โปรแกรมภาษาซีที่แสดงในภาพที่ 4.10 เป็นตัวสาธิตแบบพื้นฐานของ Pthreads API สำหรับสร้างโปรแกรมรูปแบบหลายเธรด ที่เราทำการคำนวณหาผลรวมของจำนวนเต็มบวกในการแบ่งเธรด ในโปรแกรม Pthreads แบ่งเธรดและเริ่มประมวลผล โดยแบ่งออกเป็นฟังก์ชัน ในภาพที่ 4.10 ส่วนของฟังก์ชัน runner( ) เมื่อโปรแกรมเริ่มทำงาน เธรดเส้นหนึ่งจะควบคุมการทำงานของ main() หลัง จากนั้น main( ) จะสร้างเธรดที่สองและเริ่มควบคุมการทำงานของฟังก์ชัน runner()

เธรดสองเส้นสามารถใช้ข้อมูลร่วมกันได้ทั้งหมด เมื่อเรามองลึกลงไปในโปรแกรมจะเห็นว่าทุกโปรแกรม Pthreads นั้นจะถูกรวบรวมไว้ในส่วนของเฮดเดอร์ไฟล์คือ Pthread.h ส่วนของ Pthreads จะรองรับเกี่ยวกับการจำแนกของการสร้างเธรด แต่ละเธรดงานสามารถที่จะกำหนดขนาดของ Attribute และตารางการใช้ข้อมูล

Pthread_attr_t จะเป็นตัวแทน Attribute ของเธรด สามารถกำหนดค่า Attribute ในส่วนของฟังก์ชันเรียก pthread_atr_int(&attr) เพราะเราไม่สามารถกำหนดค่าของ Attribute ได้แน่นอน โดยสามารถใช้ค่า Attribute พื้นฐานได้

การแบ่งเธรดถือเป็นการสร้างเธรดด้วย pthreads_create() ของฟังก์ชันเรียก ในการเพิ่มและผ่านของเธรด สามารถจำแนกเธรดได้ อีกทั้งชื่อของฟังก์ชันเมื่อมีเธรดใหม่เริ่มประมวลผล ในส่วนนี้ฟังก์ชัน runner() สุดท้ายเราจะส่งค่าจำนวนเต็มคงที่ไปยังเธรดที่ควบคุมมัน

จุดนี้โปรแกรมมี 2 เธรดคือ เธรดพ่ออยู่ใน main() และเธรดลูก ทำการรวมการทำงาน ในฟังก์ชัน runner( ) หลังจากการสร้างเธรดลูก เธรดพ่อจะรอคำสั่งสิ้นสุดการทำงานจากฟังก์ชัน pthread_join( ) เธรดลูกจะสิ้นสุดการทำงานเมื่อมันเรียกฟังก์ชัน pthread_exit( ) เธรดลูกสามารถที่จะกลับมาทำงานได้อีกครั้งเมื่อเธรดพ่อส่งข้อมูลไปยังส่วนของข้อมูลที่ใช้ร่วมกัน (Shared data)

4.9.2 เธรดของระบบ Win32

วิธีการสำหรับสร้างเธรดที่ใช้ในระบบ Win32 วีธีการจะคล้าย ๆ Pthreads ในระบบอื่น ๆ เราจะรวมไปถึงเธรดระบบ Win32 ในโปแกรมภาษาซีที่แสดงดังภาพที่ 4.11 จะเห็นได้ว่า เราจะใช้ Windows.h เป็นหัวของไฟล์ เมื่อใช้รูปแบบ Win32 เป็นรูปแบบในการนำเสนอ

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
DWORD Upper = *(DWORD*)Param;
for (DWORD i = 0; i <= Upper; i++)
Sum += i;
return 0;
}
int main(int argc, char *argv[])
{
DWORD ThreadId;
HANDLE ThreadHandle;
int Param;
if (argc != 2) {
fprintf(stderr,"An integer parameter is required\n");
return -1;
}
Param = atoi(argv[1]);
if (Param < 0) {
fprintf(stderr,"An integer >= 0 is required\n");
return -1;
}
/* create the thread */
ThreadHandle = CreateThread(
NULL, /* default security attributes */
0, /* default stack size */
Summation, /* thread function */
&Param, /* parameter to thread function */
0, /* default creation flags */
&ThreadId); /* returns the thread identifier */
if (ThreadHandle != NULL) {
/* now wait for the thread to finish */
WaitForSingleObject(ThreadHandle,INFINITE);
/* close the thread handle */
CloseHandle(ThreadHandle);
printf("sum = %d\n",Sum);
}
}
```

ภาพที่ 4.11 โค้ดภาษาซี ในการสร้าง Multithreaded C program โดยการใช้ Windows API.
ที่มา: Abraham, S. Peter, B. G., & Greg, G. Operating system concepts 9th ed. (2013, p.175)

Pthreads ที่แสดงในภาพที่ 4.11 ข้อมูลถูกแบ่งโดยเธรดที่ทำการแบ่ง ในส่วนนี้ ผลรวมทั้งหมดจะรองรับ (ข้อมูล DWORD) รูปแบบเป็นจำนวนเต็มบวกขนาด 32 บิต เราจะอธิบายฟังก์ชัน summation() นี้เป็นการทำงานในเธรดแบ่ง ฟังก์ชันนี้เป็นการส่ง Pointer ไปยังค่าว่าง ระบบ Win32 ถูกกำหนดโดย LPVOID การทำงานของเธรดนี้ฟังก์ชันที่กำหนดข้อมูลทั้งหมด จากผลรวมไปถึงค่าของผลรวมจาก 0 ถึง ค่าคงที่ที่ผ่านไปถึงฟังก์ชัน summation( )

เธรดที่สร้างขึ้นในระบบ Win32 จะใช้ฟังก์ชัน CreateThread ( ) และใน Pthreads จะกำหนดขนาด Attribute สำหรับเธรดที่แบ่งตัวส่งค่าไปยังฟังก์ชัน attribute ที่รวบรวมข้อมูลที่ปลอดภัย ขนาดของข้อมูลนั้นจะเพิ่มหรือลดสามารถที่จะกำหนดได้ ถ้าเธรดอยู่ในสถานะรอการทำงาน ในโปรแกรมนี้จะใช้ค่าพื้นฐานสำหรับ Attribute นี้ (เป็นการกำหนดค่าของเธรด ในสถานะรอแทนที่เราจะทำมันในตอนที่ ซีพียูทำงาน) ช่วงที่เธรดลูกสร้าง เธรดพ่อจะรอมันทำงานจนสิ้นสุดการทำงาน ก่อนที่จะแสดงค่าผลลัพธ์ออกมา ซึ่งค่านั้นจะถูกกำหนดโดยเธรดลูก จะถูกยกเลิกโดยโปรแกรม Pthreads (ภาพที่ 4.10) เธรดพ่อจะรอเธรดลูกใช้ pthread_join( ) ซึ่งการทำงานคล้ายกับระบบ Win32 ที่ใช้ฟังก์ชัน WaitForSingleObject( ) เพราะการสร้างเธรดจะไปปิดส่วนของการสิ้นสุดการทำงานของเธรดลูก

4.9.3 เธรดของระบบภาษาจาวา (Java Thread)

เธรดเป็นรูปแบบแรกของการประมวลผลโปรแกรมในโปรแกรมจาวา และ ภาษาจาวาเป็นข้อกำหนดลักษณะของกลุ่มสมาชิกขนาดใหญ่ในการสร้างและจัดการเกี่ยวกับเธรด ทุกโปรแกรมจาวาประกอบไปด้วยเธรดเดี่ยวขนาดเล็กทำการควบคุมอยู่ ตัวอย่างคือโปรแกรมจาวาที่มีเมธอด main( ) เพียงอย่างเดียวที่ทำงานแบบเธรดเดี่ยวของ JVM

เรามี 2 วิธีในการสร้างเธรดในโปรแกรมจาวา วิธีที่หนึ่ง คือ การสร้างคลาสใหม่ที่มาจากเธรดคลาสเธรดและผ่านเมธอด run( ) อีกทางเลือกหนึ่ง สามารถควบคุมได้หลายผู้ใช้ซึ่งเป็นวิธีการกำหนดขอบเขตของคลาสที่เป็นตัวทำงานอยู่ ซึ่งตัวที่ทำงานนั้นเราจะใช้เป็นตัวที่เราคิดตาม เมื่อคลาสทำงานคลาสจะเป็นตัวกำหนดเมธอด run( ) โค้ดที่ใช้ในเมธอด run( ) เป็นอะไรที่ทำการแบ่งเธรด

คลาสผลรวมที่ใช้แสดงส่วนที่ทำงานได้ การสร้างเธรดเป็นการทำงานโดยส่วนของการสร้างวัตถุตัวอย่าง ของคลาสเธรดและผ่านการสร้างวัตถุที่สามารถทำงานได้

การสร้างเธรดที่ไม่เจาะจงการสร้างเธรดใหม่ ในทางตรงกันข้ามมันจะเป็นเมธอด start() ที่ทำการสร้างเธรดใหม่อย่างแท้จริง การเรียกเมธอด start( ) เพื่อการทำวัตถุใหม่จะต้องใช้ 2 สิ่งคือ

1) การจัดการหน่วยความจำและการเริ่มใช้เธรด JVM
2) เรียกเมธอด run( ) ให้เธรดทำงานที่เหมาะสมด้วย JVM

เมื่อโปรแกรมทำงานเสร็จ 2 เธรดที่ถูกสร้างโดย JVM อย่างที่หนึ่ง เธรดพ่อเริ่มการทำงานในเมธอด main( ) อย่างที่สอง เธรดจะถูกสร้างเมื่อ เมธอด start( ) ในวัตถุเธรดเป็นตัวเรียก เธรดลูกนี้จะเริ่มทำการประมวลผลในเมธอด run( ) ของคลาส Summation หลังจากส่งค่าผลลัพธ์ เธรดนี้จะถูกกำจัดเมื่อมันออกจากเมธอด run( )

```
class Sum
{
private int sum;
public int getSum() {
return sum;
}
public void setSum(int sum) {
this.sum = sum;
}
}
class Summation implements Runnable
{
private int upper;
private Sum sumValue;
public Summation(int upper, Sum sumValue) {
this.upper = upper;
this.sumValue = sumValue;
}
public void run() {
int sum = 0;
for (int i = 0; i <= upper; i++)
sum += i;
sumValue.setSum(sum);
}
}
public class Driver
{
public static void main(String[] args) {
if (args.length > 0) {
if (Integer.parseInt(args[0]) < 0)
System.err.println(args[0] + " must be >= 0.");
else {
Sum sumObject = new Sum();
int upper = Integer.parseInt(args[0]);
Thread thrd = new Thread(new Summation(upper, sumObject));
thrd.start();
try {
thrd.join();
System.out.println
("The sum of "+upper+" is "+sumObject.getSum());
} catch (InterruptedException ie) { }
}
}
else
System.err.println("Usage: Summation <integer value>"); }
}
```

ภาพที่ 4.12 แสดงวิธีในการสร้างเธรดในโปรแกรมจาวา
ที่มา: Abraham, S. Peter, B. G., & Greg, G. Operating system concepts 9th ed. (2013, p.178)

การแบ่งปันข้อมูลระหว่างเธรดจะเกิดขึ้นง่ายในระบบ Wln32 และ Pthreads ข้อมูลที่ถูกแบ่งปันนั้นจะถูกเปิดเผยทั่วไปอย่างแท้จริงซึ่งเป็นภาษาแนวคิดเชิงวัตถุ จาวาไม่มีแนวคิดของข้อมูลแบบนี้ ถ้าเธรดอย่างน้อยสองเธรดทำการแบ่งปันข้อมูลในโปรแกรมจาวา การแบ่งปันข้อมูลเกิดขึ้นจากการส่งข้อมูลจากแหล่งที่แบ่งปันข้อมูลถึงเธรดที่เหมาะสม ในโปรแกรมจาวาที่แสดงในภาพที่ 4.12 เธรดหลักและเธรดรองทำการแบ่งปันข้อมูลกันในคลาส sum วัตถุที่ถูกแบ่งถูกอ้างอิงมาจากเมธอด getSum( ) และ setSum( ) (มีข้อสงสัยว่าทำไมจึงไม่ใช้วัตถุที่เป็นจำนวนเต็มแทนที่จะใช้การออกแบบคลาสใหม่ มีสาเหตุมาจากคลาสของจำนวนเต็มเปลี่ยนแปลงไม่ได้ นั่นคือ ค่าของข้อมูลในเซตมันไม่สามารถเปลี่ยนแปลงได้)

การเรียกเธรดพ่อใน Pthreads และ Win32 ใช้ pthread_join( ) และ WaitForSingle Object( ) ตามลำดับ รอเธรดลูกเสร็จสิ้นการทำงาน เมธอด join() ในจาวาทำงานคล้ายกัน (ข้อสังเกต join( ) ในจาวาจะไม่ใช้ InterruptedException)

4.10 การยกเลิกเธรด

การยกเลิกเธรดเป็นการทำให้เธรดจบการทำงานก่อนที่จะเสร็จสมบูรณ์ เช่น ถ้ามีหลายเธรดค้นหาข้อมูลในฐานข้อมูลพร้อมกัน แล้วมีเธรดหนึ่งให้ผลลัพธ์ออกมาแล้วเธรดที่เหลือจะถูกยกเลิกในสภาวะอื่น อาจจะเกิดเมื่อผู้ใช้กดปุ่มบนโปรแกรมเว็บบราวเซอร์เพื่อหยุดการโหลดข้อมูล เนื่องจากการโหลดข้อมูลจะใช้เธรดแยกกับการกดปุ่มบนคีย์บอร์ด ดังนั้นเมื่อผู้ใช้มีการกดปุ่ม Stop บนคีย์บอร์ด จึงทำให้โปรแกรมเว็บบราวเซอร์หยุดการโหลดข้อมูล เธรดที่ถูกยกเลิกอาจเรียกว่า Target thread ซึ่งการยกเลิกอาจมี 2 รูปแบบที่ต่างกัน ดังนี้ความยุ่งยากในการยกเลิกจะเกิดในภาวะที่ทรัพยากรถูกกำหนดให้ยกเลิกเธรด หรือมีเธรดหนึ่งถูกยกเลิกในขณะอยู่ระหว่างการบันทึกข้อมูลที่ใช้ทรัพยากรร่วมกันอยู่กับเธรดอื่น สิ่งนี้เป็นกรณีพิเศษของการยกเลิกแบบ Asynchronous ระบบปฏิบัติการมักจะแก้ปัญหาทรัพยากรจากการยกเลิกเธรดแต่ไม่ใช่ทุกทรัพยากร

ดังนั้นการยกเลิกแบบ Asynchronous อาจจะใช้ยกเลิกกับทรัพยากรทั่วไปไม่ได้ ในทางกลับกัน การยกเลิกแบบ Deferred จะทำงานโดยมีเธรดหนึ่งที่กำหนดว่า Target thread ใดจะถูกยกเลิก อย่างไรก็ตาม การยกเลิกนี้จะเกิดขึ้นเฉพาะเมื่อ Target thread นั้นตรวจสอบว่าตัวเองจะถูกยกเลิกหรือไม่ สิ่งนี้ยอมให้เธรดตรวจสอบ เพื่อให้ถูกยกเลิกในจุดที่ปลอดภัย ถ้ายกเลิกจุดที่ว่านี้ใน Pthread API เรียกว่า Cancellation points ในระบบปฏิบัติการส่วนมากจะยอมให้โพรเซสหรือ thread ถูกยกเลิกแบบ Asynchronous แต่ใน Pthread API ยังสนับสนุนการยกเลิกแบบ Defferred อีกด้วย สิ่งนี้หมายความว่าระบบปฏิบัติการที่มี Pthread API จะยอมให้ยกเลิกแบบ Defferred นั่นเอง

4.11 สรุป

เธรด คือ การเรียกใช้หน่วยประมวลผลให้เกิดประโยชน์สูงสุด และทำให้การทำงานของโปรแกรมมีประสิทธิภาพมากขึ้นและมีประโยชน์ต่อระบบในปัจจุบันที่เป็น Multicore เพราะสามารถเรียกใช้เธรดหลาย ๆ เธรดได้พร้อม ๆ กัน ทั้งนี้เธรดของโพรเซสเดียวกันสามารถจะทำงานแตกต่างกัน แต่ยังมีความเกี่ยวข้องกันบางอย่างและต้องทำงานอยู่ภายใต้สภาพแวดล้อมเดียวกัน เธรดเป็นการควบคุมการทำงานภายในของโพรเซส Multithreaded ก็คือโพรเซสจะประกอบไปด้วยเธรดที่ทำหน้าที่ต่างกันแต่ทำงานอยู่บนตำแหน่งหน่วยความจำเดียวกัน ข้อดีของ Multithreaded ประกอบไปด้วย การเพิ่มการตอบสนองต่อผู้ใช้ทั้งแชร์ ทรัพยากรในโพรเซส ความประหยัดและความสามารถของงาน ในสถาปัตยกรรมแบบมัลติโพรเซสเซอร์ ในระดับของเธรดสำหรับผู้ใช้ คือสิ่งที่โปรแกรมเมอร์มองเห็นและระบบปฏิบัติการ Kernel จะสนับสนุนจัดการในระดับ Kernel โดยทั่วไปแล้วเธรดสำหรับผู้ใช้จะเร็วกว่าในการสร้างและจัดการมากกว่า เธรดสำหรับระบบปฏิบัติการและไม่ก้าวก่ายสิ่งที่ Kernel ต้องการ

รูปแบบที่แตกต่างกันระหว่างเธรดสำหรับผู้ใช้กับเธรดสำหรับระบบปฏิบัติการ โดยรูปแบบ Many-to-one หมายถึง เธรดสำหรับระบบปฏิบัติการ 1 หน่วย กับเธรดสำหรับผู้ใช้หลายหน่วย เป็นการ

ออกแบบที่จะยอมให้เธรดเพียงเธรดเดียวที่เข้าถึง Kernel ในกรณีที่เธรดไปบล็อก System call จะทำให้โพรเซสทั้งหมดถูกบล็อกไปด้วย โดยรูปแบบนี้ยอมให้สร้างเธรดสำหรับผู้ใช้ได้ตามต้องการ แต่ไม่สามารถประมวลผลได้พร้อมกัน เพราะยอมให้เข้าใช้เธรดสำหรับระบบปฏิบัติการได้ครั้งละเธรดเท่านั้น

รูปแบบ One-to-one หมายถึง เธรดสำหรับระบบปฏิบัติการ 1 หน่วย กับเธรดสำหรับผู้ใช้ 1 หน่วย ซึ่งระบบปฏิบัติการจะยอมให้เธรดอื่นประมวลผลได้เป็นระบบขนาน ที่ทำงานแบบระบบมัลติโพรเซสเซอร์ มีการใช้หลักการนี้อยู่ในระบบปฏิบัติการวินโดว์ในปัจจุบัน โดยรูปแบบนี้ต้องไม่ยอมให้สร้างเธรดสำหรับผู้ใช้มากเกินไป

รูปแบบ Many-to-many หมายถึง รูปแบบที่ลดข้อจำกัดของ 2 แบบแรก ผู้ใช้สามารถสร้างเธรดสำหรับผู้ใช้เท่าที่จำเป็น และสัมพันธ์กับเธรดสำหรับระบบปฏิบัติการที่รับการทำงานแบบขนานในแบบมัลติโพรเซสเซอร์ เมื่อมีเธรดที่บล็อก System call ทาง Kernel จะจัดเวลาให้เธรดอื่นเข้ามาประมวลผลก่อนได้

เรื่องราวเกี่ยวกับเธรดมีหลายอย่างที่ต้องพิจารณา การยกเลิกเธรดเป็นเรื่องที่ต้องทำความเข้าใจ เพราะการยกเลิกหมายถึง การทำให้เธรดเป้าหมายจบการทำงาน ก่อนที่จะทำงานจนเสร็จสมบูรณ์ การยกเลิกนี้มี 2 วิธีคือ 1) Asynchronous cancellation การยกเลิกที่เธรดอื่นสั่งให้เธรดเป้าหมายหยุดทำงาน 2) Defered cancellation การยกเลิกเธรดเป้าหมาย โดยใช้ตรวจสอบตนเองว่าตนเองต้องถูกยกเลิกด้วยหรือไม่ Pthreads อ้างถึงมาตรฐาน POSIX (IEEE 1003.1c) เพื่อกำหนด API (Application Programming Interface) สำหรับสร้าง และการซินโครไนซ์เซชัน นี่คือการกำหนดสภาพแวดล้อมของเธรด ซึ่งการกำหนดสภาพแวดล้อมของเธรดนี้ถูกจำกัดใน Solaris2 แต่ Pthread ไม่ถูกสนับสนุนใน Windows แม้จะมี Shareware เผยแพร่แล้วก็ตาม

แบบฝึกหัดท้ายบทที่ 4

จงตอบคำถามต่อไปนี้

1) จงอธิบายและยกตัวอย่างโปรแกรม การทำงานแบบ Multithreading ทำงานมีประสิทธิภาพดีกว่า Single threaded

2) จงอธิบายอะไรคือข้อแตกต่างระหว่าง User-Level threads และ Kernel-Level threads

3) รูปแบบ Many-to-Many กับ รูปแบบ One-to-One แตกต่างกันอย่างไรอธิบาย

4) จงอธิบายและยกตัวอย่างโปรแกรมทำงานแบบ Multithreading ที่มีการทำงานด้อยประสิทธิภาพกว่า Single threaded

5) พิจารณาระบบ Multicore และโปรแกรม Multithreaded เขียนโดยใช้รูปแบบเธรด Many-to-Many จำนวนเธรดสำหรับผู้ใช้มากกว่าจำนวนเธรดสำหรับระบบปฏิบัติการ ดังนั้นจำนวนเธรดสำหรับระบบปฏิบัติการ จัดสรรให้โปรแกรมทำงานน้อยกว่าจำนวนแกนของหน่วยประมวลผล (Processing core) ได้หรือไม่

6) จากข้อ 5 จำนวนเธรดสำหรับระบบปฏิบัติการ อธิบายจัดสรรให้โปรแกรมทำงานมากกว่าจำนวนแกนของหน่วยประมวลผล แต่น้อยกว่าจำนวนระดับเธรดสำหรับผู้ใช้

7) จงอธิบายการยกเลิกเธรดแบบ Asynchronous แตกต่างกับการยกเลิกแบบ Deferred อย่างไร

8) องค์ประกอบใดต่อไปนี้ในขณะที่โปรแกรมอยู่ในสถานะการทำงาน สามารถถูกแชร์ทรัพยากรข้ามเธรดได้ในระบบ Multithreaded process

1) Register values
2) Heap memory
3) Global variables
4) Stack memory

เอกสารอ้างอิง

พิเชษฐ์ ศิริรัตนไพศาลกุล. (2548). **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.

Andrew S. Tanenbaum, Maarten van Steen. (2014). **Distributed Systems Principles and Paradigms.** Prentice Hall, Pearson Education International.