บทที่ 4 การจัดสรรเวลาสำหรับโพรเซส

วัตถุประสงค์ของเนื้อหา

- เรียนรู้หลักเหตุและผลของการบริการจัดการเวลาสำหรับโพรเซสต่างๆ ที่ต้องใช้ทรัพยากรซีพียูร่วมกัน
- ศึกษาถึงขั้นตอนวิธีการจัดสรรเวลาให้โพรเซสในรูปแบบต่างๆ
- เรียนรู้ถึงหลักการพื้นฐานของการกระจายโพรเซส/เธรดไปยังคอร์ต่างๆ ของคอมพิวเตอร์แบบหลายคอร์
- เรียนรู้ประเด็นปลีกย่อยต่างๆ ของการบริการจัดการเวลาของโพรเซสและเธรด

สิ่งที่คาดหวังจากการเรียนในบทนี้

- นักศึกษาเข้าใจถึงขั้นตอนวิธีการจัดสรรเวลาให้โพรเซสในรูปแบบต่างๆ ที่เรียนมาในบทนี้
- นักศึกษาเข้าใจถึงความสำคัญของการจัดสรรเวลาของโพรเซส

วัตถุประสงค์ของปฏิบัติการท้ายบท

- -(ไม่มีปฏิบัติการท้ายบท)

สิ่งที่คาดหวังจากปฏิบัติการท้ายบท

- -(ไม่มีปฏิบัติการท้ายบท)

เวลาที่ใช้ในการเรียนการสอน

- ทฤษฎี 2 ชั่วโมง
    - หลักการพื้นฐานของการบริหารจัดการเวลาของโพรเซส 0.25 ชั่วโมง
    - ขั้นตอนวิธีการบริหารจัดการเวลาของโพรเซส 1.5 ชั่วโมง
    - ประเด็นอื่นๆ ที่เกี่ยวข้อง 0.25 ชั่วโมง
- ปฏิบัติ 0 ชั่วโมง
    - -

บทที่ 4 การจัดสรรเวลาสำหรับโพรเซส

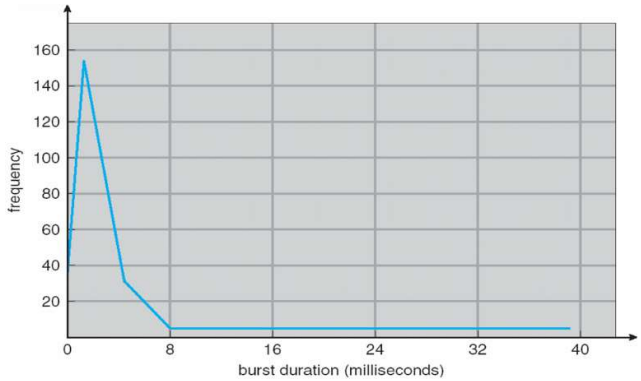
4.1 แนวความคิดเกี่ยวกับการจัดสรรเวลาซีพียูให้กับแต่ละโพรเซส

ในระบบคอมพิวเตอร์ซึ่งประกอบไปด้วยหน่วยประมวลผล หน่วยความจำหลัก และหน่วยรับส่งข้อมูลเข้าออกนั้น ระบบปฏิบัติการที่ดีควรจะสามารถใช้งานองค์ประกอบเหล่านี้ได้อย่างคุ้มค่าที่สุดโดยไม่ต้องมีเวลารอ สมมติกรณีระบบที่มีซีพียูเพียงตัวเดียว เมื่อโปรแกรมตัวหนึ่งกำลังรันตามปกติ ก็จะใช้งานหน่วยประมวลผลในการทำงาน แต่เมื่อต้องการข้อมูลเพิ่มเติมหรือจะส่งข้อมูลออกไปภายนอก ก็จะต้องติดต่อกับหน่วยรับส่งข้อมูลเข้าออก ในจังหวะนี้ โปรแกรมดังกล่าวจะต้องรอการทำงานของหน่วยรับส่งข้อมูลให้ทำงานเสร็จ ปล่อยให้หน่วยประมวลผลอยู่ว่างๆ ไม่ทำอะไร เป็นการเสียเปล่าซึ่งทรัพยากรเวลาของหน่วยประมวลผลโดยใช่เหตุ

ระบบปฏิบัติการแบบหลายงาน ซึ่งรันโปรเซสหลายๆ ตัวในเวลาเดียวกัน โดยจัดสรรหน่วยความจำหลักให้แต่ละโพรเซสเป็นกิจลักษณะ และโดยหลักการของการใช้หน่วยประมวลผลกลางหรือซีพียูให้คุ้มค่า เมื่อโพรเซสหนึ่งต้องติดต่อกับหน่วยรับส่งข้อมูลเข้าออก (I/O) ระบบปฏิบัติการจะดึงคืนเวลาในการครอบครองซีพียูของโพรเซสนั้น (หยุดการทำงานตามคำสั่งของโพรเซสนั้นๆ) เพื่อนำไปให้โพรเซสอื่นได้ใช้ในการประมวลผลต่อ โดยการสลับสับเปลี่ยนชุดคำสั่งการทำงานของแต่ละโพรเซสไปมาเมื่อโพรเซสหนึ่งต้องหยุดรอ ส่งผลให้ซีพียูไม่ต้องหยุดรอ เป็นการใช้ทรัพยากรซีพียูอย่างคุ้มค่า และนี่คือกลไกพื้นฐานสำคัญของการจัดสรรเวลาการครอบครองของซีพียู (Process Scheduling) ของระบบปฏิบัติการโดยทั่วไป

CPU-I/O Burst Cycle

- การทำงานของโพรเซส มักจะมีลักษณะการทำงานเป็นวัฏจักรของการรับข้อมูลเข้ามาจำนวนหนึ่งจาก I/O เพื่อนำมาประมวลอย่างใดอย่างหนึ่ง จากนั้นก็อาจส่งข้อมูลออกไปที่ I/O หรือรับข้อมูลเข้ามาเพิ่มอีก แล้วประมวลต่อไปอีก เกิดเป็นรอบของการคำนวณบนซีพียู (CPU Burst) สลับกับการติดต่อหน่วยรับส่งข้อมูลเข้าออก (I/O Burst) ไปเรื่อยๆ
- หากเราจับเวลาการทำงานระหว่างที่โพรเซสหนึ่งกำลังคำนวณ และเวลาการรอรับเข้าส่งออกข้อมูล แล้วนำเอาคาบเวลาที่ซีพียูรันคำสั่งของโพรเซสหนึ่งๆ ในระหว่างช่วงที่ต้องรอรับส่งข้อมูลแต่ละครั้ง (ซึ่งคือค่า CPU Burst time หรือ CPU burst duration) มาเขียนกราฟเพื่อดูแนวโน้มของช่วงเวลาที่ใช้ไปในการครอบครองซีพียูหรือหน่วยรับส่งข้อมูลได้ ดังกราฟตัวอย่างด้านล่าง



- เราจะเห็นว่าโปรแกรมตัวอย่างนี้จะใช้เวลาคำนวณในแต่ละรอบการคำนวณ (CPU Burst) อยู่ที่ประมาณ 0-8 มิลลิวินาที เป็นต้น
- โพรเซสที่เป็น **CPU-bound** หรือโพรเซสที่**ใช้เวลาส่วนใหญ่ในการทำงานโดยซีพียู** มักจะมีค่าเฉลี่ยในรอบการคำนวณที่

ค่อนข้างยาวนานเมื่อเทียบกับเวลาที่ใช้กับ I/O ในขณะที่โพรเซสที่เป็น **I/O bound** หรือโพรเซสที่**ใช้เวลาส่วนใหญ่ในการรอรับส่งข้อมูลเข้าออก** เราจะพบว่าเวลาส่วนใหญ่จะเป็นการรอคอย I/O ในขณะที่ใช้เวลาในการคำนวณน้อยกว่า และมักจะมีคาบเวลา CPU Burst สั้นๆ

กลไกการจัดสรรเวลาเข้าครอบครองซีพียู (CPU Scheduler)

- หน้าที่ของ CPU Scheduler คือการเลือกโพรเซสที่เข้าคิวรออยู่ใน ready queue ให้เข้าสู่ running state (ครอบครองเวลาการคำนวณของซีพียู) การเลือกว่าโพรเซสใดเหมาะสมจะเข้าสู่ running state ไม่จำเป็นต้องเป็น FIFO แต่จะเป็นแบบใดก็ขึ้นอยู่กับขั้นตอนวิธีที่จะนำมาใช้ในการจัดสรรเวลา
- สำหรับการจัดสรรเวลาเข้าครอบครองซีพียูแบบ**สามารถคัดออกได้ (Preemptive Scheduling)** จะสามารถใช้ช่วงเวลาที่เกิดสถานะดังกรณีต่อไปนี้ เพื่อถือโอกาสตัดสินใจตามขั้นตอนวิธีที่ออกแบบไว้ (เลือกเอาโพรเซสอื่นจาก ready queue เข้ามาทำงานต่อบ้าง)
    1. เมื่อโพรเซสที่อยู่ใน **running state เปลี่ยนสถานะไปสู่ waiting state** (**เรียกใช้ system call** = โพรเซสที่กำลังคำนวณอยู่นั้น มีความจำเป็นต้องติดต่อ I/O และส่งผลให้โพรเซสนั้นต้องเข้าสู่สถานะ waiting state โดยตัวโพรเซสไปเข้า I/O queue ของ I/O ที่กำลังจะติดต่อ)
    2. เมื่อโพรเซสที่อยู่ใน **running state เปลี่ยนสถานะไปสู่ ready state** (**ถูก interrupt** = โพรเซสที่กำลังคำนวณอยู่นั้น ถูกขัดจังหวะจากการอินเทอร์รัปต์ ไม่ว่าจะเป็นแบบฮาร์ดแวร์หรือซอฟต์แวร์ ส่งผลให้เกิดการถอดโพรเซสที่กำลังคำนวณ กลับเข้าไปสู่ ready queue อีกครั้งหนึ่ง)
    3. เมื่อโพรเซสที่อยู่ใน **waiting state กลับเข้ามาสู่ใน ready state** (**ถูก interrupt จาก I/O กรณีทำ I/O สมบูรณ์** = โพรเซสที่รอ I/O นั้นได้รับหรือส่งข้อมูลกับ I/O เป็นที่เรียบร้อย และทำให้ย้ายโพรเซสที่จัดการกับ I/O เสร็จสิ้นแล้ว กลับมาเข้าคิว ready queue อีกครั้ง
    4. เมื่อโพรเซสที่กำลังคำนวณอยู่นั้นจบการทำงาน (**termination**)

จากกรณีทั้งสี่ข้างต้น จะเห็นว่าในจังหวะที่จะเกิดกรณีที่ 1 หรือกรณีที่ 4 นั้น เป็นจังหวะที่โพรเซสที่กำลังคำนวณนั้นๆ ร้องขอ system call ด้วยตนเอง เพื่อส่งต่องานให้ระบบปฏิบัติการทำงานอย่างใดอย่างหนึ่งต่อไป การจัดสรรเวลาการครอบครองของซีพียูจึงมีความจำเป็นต้องเกิดขึ้น เพื่อเอาเวลา(ที่ว่างลงของการครอบครอง)ซีพียูนั้นไปให้โพรเซสอื่นได้ใช้งาน ระบบปฏิบัติการที่ถูกออกแบบให้จัดสรรเวลาเข้าครอบครองซีพียูแค่เฉพาะสองกรณีนี้ จะไม่สามารถแย่งคืนเวลาครอบครองของซีพียู ในขณะที่ซีพียูกำลังประมวลผลอยู่ได้ กล่าวคือ เมื่อโพรเซสใดโพรเซสหนึ่งเข้าครอบครองการทำงานของซีพียูแล้ว ระบบปฏิบัติการจะหมดสิทธิในการเข้าครอบครองการทำงานของซีพียูได้อีกจนกว่าโพรเซสนั้นๆ จะยอมปล่อยการครอบครองซีพียู ผ่านการร้องขอ system call ระบบปฏิบัติการที่จัดสรรเวลาเข้าครอบครองซีพียูในลักษณะนี้ถูกเรียกว่า **Nonpreemptive OS** ซึ่งมีข้อดีคือกลไกการจัดสรรเวลาของซีพียูนั้นกระทำได้ง่าย แต่มีข้อเสียคือ หากโพรเซสที่กำลังครอบครองเวลาซีพียู เกิดตัดสินใจไม่ปล่อยการครอบครองเวลา หรือเกิดการทำงานผิดพลาดเช่นวนลูปอนันต์ ระบบปฏิบัติการหรือโพรเซสอื่นจะทำงานไม่ได้อีกเลย (เครื่องหยุดการตอบสนองใดๆ ส่งผลให้ต้องบูตเครื่องใหม่)

ระบบปฏิบัติการที่สามารถจัดสรรเวลาเข้าครอบครองซีพียูโดยมีกลไกนอกจากกรณีข้อที่ 1 และ 4 แล้ว ยังพิจารณาในกรณีที่ 2 และ 3 ด้วย จะถูกเรียกว่า **Preemptive OS** เหตุด้วยโพรเซสที่กำลังครอบครองเวลาใช้งานซีพียูอยู่ มีสิทธิถูก**ขัดจังหวะ (interrupt)** การทำงานด้วยโพรเซสของระบบปฏิบัติการได้ตลอดเวลา และในจังหวะนั้น ระบบปฏิบัติการก็อาจจะตัดสินใจไม่คืนเวลาเข้าครอบครอง ณ รอบปัจจุบันให้กับโพรเซสนั้นๆ อีก (โดยจับส่งไปเข้า ready queue เพื่อรอรอบต่อไป หรืออาจจะใช้ช่วงเวลานี้ในการบังคับจบการทำงานโพรเซสนั้นๆ ได้ด้วย เป็นต้น)

ข้อเสียของระบบปฏิบัติการแบบ nonpreemptive ที่เห็นได้ชัดอีกกรณีก็คือการแบ่งใช้พื้นที่เก็บข้อมูลร่วมกันระหว่างโพรเซส สมมติว่ามีโพรเซสสองตัวกำลังคำนวณไปพร้อมกัน โพรเซสแรกจะต้องส่งข้อมูลที่คำนวณเสร็จบางส่วนให้โพรเซสตัวที่สอง

ในกรณีของ nonpreemptive โพรเซสแรกจะคำนวณไปเรื่อยๆ จนกว่าจะติดต่อกับ I/O ซึ่งถ้าโพรเซสเป็นแบบ CPU-bound (เช่น การคำนวณเป็นหลัก) ช่วงเวลาที่จะติดต่อ I/O จะน้อยมาก (เช่นอาจจะคำนวณข้อมูลทั้งหมดจนเสร็จก่อน) ดังนั้นเราจะเห็นว่าในกรณีเช่นนี้ โพรเซสที่สองจะไม่มีโอกาส หรือมีโอกาสน้อยมาก ที่จะได้ข้อมูลบางส่วนที่โพรเซสแรกคำนวณเสร็จสิ้น นำมาคำนวณต่อได้

แต่ก็มีข้อเสียของการจัดการแบบ preemptive กรณีหนึ่งที่น่าสนใจก็คือ ในกรณีที่โพรเซสหนึ่งๆ ร้องขอ system call ซึ่งหมายถึงได้ส่งการร้องขอให้กับเคอร์เนิลเพื่อจัดการกับ I/O หรืออื่นใด ในขณะดังกล่าว เคอร์เนิลอาจจะต้องการเข้าถึงพื้นที่หน่วยความจำของโพรเซสเพื่ออ่านหรือเขียนข้อมูลกับ I/O แต่เมื่อโพรเซสถูก preempt การจัดการก็ไม่สมบูรณ์ ทำให้เกิดปัญหาข้อมูลที่จะต้องนำเข้าหรือส่งออกนั้นยังมีไม่ครบถ้วน หรืออาจทำให้การจัดการกับสถานะของ I/O ผิดพลาด ทางแก้ไขของระบบปฏิบัติการบางตัวก็คือ เมื่อโพรเซสที่กำลังทำงานอยู่มีบางเธรดเกิดเรียก system call เธรดอื่นๆ ของโพรเซสนั้นจะไม่ถูก preempt เป็นการชั่วคราวไปจนกว่าจะทำ system call เสร็จสิ้น โดยระบบปฏิบัติการจะออกแบบ system call ให้ใช้เวลาที่สั้นที่สุด และง่ายที่สุด เพื่อกันมิให้ระบบหยุดการตอบสนองในกรณีที่โพรเซสเกิดค้างในระหว่าง system call (และอาจมีกลไกอื่นเพื่อตรวจสอบว่าเกิดความผิดพลาดในการทำงานหรือไม่ประกอบ) แต่การแก้ไขแบบนี้จะทำให้บางรอบของการครอบครองโพรเซสหนึ่งๆ อาจจะกินเวลานานผิดปกติ ทำให้โพรเซสอื่นๆ ที่ต้องทำงานในรอบเวลาที่กำหนด ไม่สามารถทำงานได้ทัน และนี่ส่งผลต่อระบบปฏิบัติการที่รองรับการทำงานแบบทันเวลา (real-time system) ไม่สามารถใช้กลไกดังกล่าวนี้ในการจัดการภายในได้

Dispatcher

Dispatcher คือชุดคำสั่งของเคอร์เนิล ที่ทำหน้าที่คัดสรรโพรเซสที่พร้อมทำงาน (ใน ready queue) เพื่อส่งให้ซีพียูได้ประมวล อันเป็นหัวใจของการ CPU Scheduler (short-time scheduler) มีการทำงานหลักๆ ดังนี้คือ

- การเปลี่ยนข้อมูลประจำโพรเซสที่ซีพียูได้ครอบครองไปยังของโพรเซสที่จะเข้ามาครอบครองแทน (switching context)
- การเปลี่ยนสิทธิการทำงานของซีพียูกลับไปยังโหมดผู้ใช้ (กลไกการแลกเปลี่ยนโพรเซสที่ครอบครองซีพียูนี้เป็นของเคอร์เนิล ดังนั้นขณะปัจจุบัน ซีพียูกำลังรันอยู่ใน kernel mode)
- การส่งค่าตำแหน่งหน่วยความจำที่รันค้างไว้ของโพรเซสใหม่ ไปใส่ในซีพียูโดยใช้คำสั่งกระโดด (jump instruction) เพื่อให้ซีพียูกระโดดไปทำงาน ณ ตำแหน่งที่ค้างอยู่ของโพรเซสใหม่ (เพื่อให้ทำงานต่อจากที่ค้างไว้อย่างถูกต้อง)

เวลาในการกระทำ dispatch นี้โดยทั่วไปจะต้องใช้เวลาให้น้อยมาก เพราะจะต้องเกิดทุกๆ ครั้งเมื่อสลับสับเปลี่ยนโพรเซส (มิเช่นนั้นจะทำให้เวลาการทำงานของซีพียูหมดไปกับการเปลี่ยนโพรเซส แทนที่จะให้โพรเซสของผู้ใช้ได้ทำงาน) ช่วงเวลาในการสับเปลี่ยนโพรเซสนี้เรียกว่า dispatch latency

4.2 ปัจจัยที่เกี่ยวเนื่องกับการจัดสรรเวลาของโพรเซส (Scheduling Criteria)

เราอาจจะออกแบบขั้นตอนวิธีในการจัดสรรเวลาในการเข้าครอบครองซีพียูของโพรเซสได้แตกต่างกันไปในหลายวิธีการ แต่วิธีการที่เราออกแบบมานั้นอาจจะเหมาะสมหรือไม่ในการใช้งานจริง มีปัจจัยหลายอย่างที่เราสามารถพิจารณาได้ดังเช่น

- **สัดส่วนเวลาการคำนวณของซีพียูเมื่อเทียบกับสภาวะรอ (CPU Utilization)** ระบบปฏิบัติการที่ดี จะต้องสามารถสลับเปลี่ยนโพรเซสที่ต้องการครอบครองซีพียูได้เข้าใช้งานให้ได้เต็มเวลาที่สุด หรือในอีกนัยหนึ่ง ให้ค่า utilization เข้าใกล้ 100 เปอร์เซ็นต์ให้ได้มากที่สุด
- **จำนวนโพรเซสที่สามารถทำได้ในช่วงเวลาหนึ่งๆ (Throughput)** โดยอาจวัดเป็นจำนวนโพรเซสที่สามารถสลับ

หมุนเวียนเข้าครองครองซีพียูและทำงานจนเสร็จสิ้นต่อช่วงเวลาหนึ่งๆ

- **เวลาที่โพรเซสต้องใช้งาน**นับตั้งแต่โปรแกรมหนึ่งถูกนำเข้าสู่ระบบ ไปจนกระทั่งโพรเซส(ที่สร้างขึ้นในการรันโปรแกรมหนึ่ง) ถูกถอดออกจากระบบเมื่อทำงานเสร็จ **(Turnaround time)** เวลาดังกล่าวเรานับตั้งแต่ช่วงเวลาที่โปรแกรมหนึ่งได้รับคำสั่งให้รัน ถูกโหลดขึ้นสู่หน่วยความจำหลัก เข้าสู่สภาวะรอ และเริ่มรัน ไปจนกระทั่งประมวลเสร็จสิ้นและจบการทำงาน หรืออาจพิจารณาเป็นช่วงเวลาระหว่างการเข้าสู่รอบ CPU Burst แต่ละครั้งว่าห่างกันเท่าใด (ซึ่งจะรวมเวลาการทำงานตามชุดคำสั่งโดยซีพียู เวลาที่ต้องรอรับส่งข้อมูล และรออยู่ใน ready queue ในแต่ละรอบเข้าด้วย)
- **เวลาที่โพรเซสต้องเสียเวลารออยู่ใน ready queue (Waiting time)**
- **เวลาตอบสนอง (Response time)** ช่วงเวลานับจากเวลา**ร้องขอ**ข้อมูล (เช่น ส่งการร้องขออ่านข้อมูลไปยัง I/O) ไปจนกระทั่ง**ได้รับการตอบสนอง** (ส่งผลลัพธ์กลับออกไปยัง I/O)

ระบบปฏิบัติการที่ดี ควรจะมีค่า CPU Utilization ที่สูง (หมายความว่าสามารถจัดสรรโพรเซสให้ผลัดใช้งานซีพียูได้อย่างคุ้มค่า) ต้องใช้เวลารันโพรเซสให้น้อยที่สุด หมายความว่า มีจำนวนโพรเซสที่ทำงานเสร็จได้สูงที่สุด จำเป็นอย่างยิ่งที่จะต้องมีค่า turnaround time waiting time และ response time ที่ต่ำ สำหรับระบบคอมพิวเตอร์บางระบบ เช่นระบบคอมพิวเตอร์แบบหลายผู้ใช้ ที่มีผู้ใช้ติดต่อร้องขอข้อมูลกับคอมพิวเตอร์หลายคนพร้อมๆ กัน ในกรณีนี้ ค่า response time อาจจะไม่ได้คิดแค่ค่าเฉลี่ยที่ต่ำ แต่ต้องคิดว่า จะไม่มีช่วงเวลารอคอยใดที่อาจจะต้องรอยาวนานมากจนเกินไป (เช่น ร้องขอข้อมูลจากเว็ปเซอร์เวอร์ อาจจะเปิดหน้าเพจได้ช้าหรือเร็วประมาณหนึ่ง แต่ไม่ควรที่จะมีบางครั้งที่ต้องรอเปิดหน้าเพจนานมากๆ) ซึ่งลักษณะเช่นนี้มักจะไม่เป็นที่พอใจต่อผู้ใช้

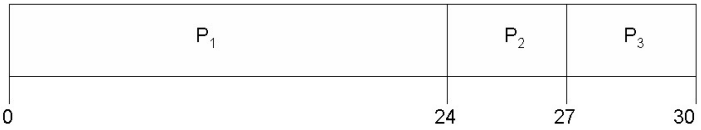
- MAX CPU utilization
- MAX troughput
- Min turnaround time
- Min waiting time
- Min respond time

4.3 ขั้นตอนวิธีจัดสรรเวลาโพรเซส

First-Come, First-Served (FCFS) ใครเข้าคิว ready queue ก่อน ได้รับบริการก่อน

สมมติมีโพรเซสรอเข้าคิวตามลำดับดังนี้

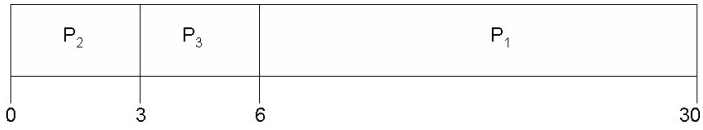
Process	Burst Time
P1	24
P2	3
P3	3



เวลารอของแต่ละโพรเซส *P1* = 0; *P2* = 24; *P3* = 27

เวลารอเฉลี่ยของทุกโพรเซส (0 + 24 + 27)/3 = 17

ในกรณีเช่นนี้ จะเห็นว่าหากโพรเซสที่รอตัวแรกๆ เป็นโพรเซสที่ใช้เวลาทำงานนาน โพรเซสอื่นๆจะเสียเวลารอนานมาก ลองพิจารณากรณีที่ P1 เข้ามารอคิวหลัง P2 และ P3 จะเห็นได้ว่าค่าเวลารอเฉลี่ยจะเปลี่ยนไปได้อย่างมาก



เวลารอแต่ละโพรเซส *P1 = 6; P2 = 0; P3 = 3*

เวลารอเฉลี่ย (6 + 0 + 3)/3 = 3

ข้อด้อยของขั้นตอนวิธีแบบ FCFS ที่เห็นได้ชัดก็คือ ในระบบหลายงานที่มีโพรเซสที่ใช้เวลาคำนวณนาน เช่นโพรเซสที่เป็น CPU-bound อยู่ผสมกับโพรเซสที่เป็น I/O-bound ที่ใช้เวลาคำนวณน้อย แต่ใช้ I/O บ่อย การทำงานภายใต้ FCFS โพรเซสแต่ละตัวก็จะมีช่วงเวลาที่คำนวณสำหรับกับ I/O ทำให้แต่ละโพรเซสเมื่อทำงานจบรอบคำนวณ และจะติดต่อ I/O ตัวโพรเซสก็จะถูกถอดจากการเข้าครอบครองซีพียูไปเข้าคิว waiting queue เพื่อรับบริการจาก I/O เป็นโอกาสให้โพรเซสที่รอใน ready queue เข้าใช้ทรัพยากรเวลาของซีพียูบ้าง ในลักษณะเช่นนี้ เราจะเห็นว่าแต่ละโพรเซสจะได้รับโอกาสเข้าครอบครองซีพียูและทำ I/O สลับวนกันไปเรื่อยๆ ถ้าโพรเซสทั้งหมดใช้เวลา CPU พอๆ กัน เราก็จะพบว่าการตอบสนองการทำงานของโพรเซสทุกๆ ตัวก็จะมีค่าพอๆ กัน แต่ถ้ามี CPU-bound process ปะปนอยู่กับ I/O-bound process เราจะเห็นว่าซีพียูจะใช้เวลาส่วนใหญ่กับโพรเซสที่เป็น CUP-bound ในขณะที่โพรเซสอื่นๆจะต้องมาเข้าคิวกันรอใน ready queue เป็นแถวยาวนานมาก ทำให้โพรเซสที่เป็น I/O-bound ทั้งหมด ทำงานช้า และไม่ตอบสนอง ลักษณะเช่นนี้เรียกว่า **convoy effect**

ลักษณะการสลับโพรเซสโดยอาศัยเพียงแต่การรอโพรเซสให้ทำงานจบรอบการคำนวณแล้วไปหาโพรเซสอื่นใน ready queue ของ FCFS นี้ถือเป็นการจัดสรรเวลาแบบ **nonpreemptive** ชนิดหนึ่ง

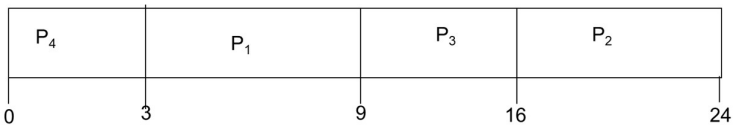
Shortest-Job-First Scheduling การจัดสรรเวลาแบบงานสั้นที่สุดถูกเรียกใช้ก่อน (หรือเรียกได้ว่า Shortest-next-CPU-burst algorithm)

ในการจัดสรรเวลาให้กับโพรเซสแบบนี้ เมื่อโพรเซสหนึ่งๆ ผละจากการครอบครองซีพียูแล้ว ระบบปฏิบัติการจะตรวจดูค่า(ประเมิน)เวลาการเข้าใช้ซีพียู (CPU burst) ว่าโพรเซสใดมีค่า (สถิติ) ดังกล่าวต่ำสุด ก็จะเรียกเอาโพรเซสนั้นๆ มารับบริการก่อน

กลวิธีแบบ SJF นี้มีประสิทธิภาพมากกว่า FCFS เพราะเวลารอคอย (waiting time) โดยเฉลี่ยจะมีค่าต่ำที่สุด แต่ก็มีประเด็นปัญหาเพิ่มเติมก็คือ **เราจะทราบได้อย่างไรว่า CPU Burst ของแต่ละโพรเซสมีค่าต่ำสุด** โดยเฉพาะกรณีการจัดการแบบ long-term scheduling (Job scheduling) เราจะไม่มีข้อมูลใดๆ ของงานแต่ละตัวที่จะทำเลย วิธีที่เป็นไปได้ก็คือให้ผู้ใช้กรอกค่าประเมินเวลาที่ต้องประมวลเข้าไปด้วยตนเอง โดยระบบอาจจะยกเลิกการทำงานของโพรเซสนั้นๆ หากใช้เวลาจริงเกินกว่าที่กำหนด (แล้วส่งไปเข้าคิวรอประมวลตั้งแต่ต้นใหม่)

สมมติมีโพรเซสรอเข้าคิวตามลำดับดังนี้

Process	Burst Time
P1	*6*
P2	8
P3	*7*
P4	*3*



เวลารอแต่ละโพรเซส *P1 = 3; P2 = 16; P3 = 9 ; P4 = 0*

เวลารอเฉลี่ย (3 + 16 + 9 + 0)/4 = 7

เพื่อแก้ไขปัญหาการไม่สามารถทราบถึงค่า CPU burst ของรอบปัจจุบันได้ เราอาจจะอาศัย**<u>ประเมินความเป็นไปได้</u>**โดยพิจารณาจากรอบการทำงานของโพรเซสนั้นๆ ในรอบก่อนหน้า เพื่อมาปรับค่าประเมินเวลา CPU Burst ในรอบถัดไป เช่นการใช้การคำนวณที่เรียกว่า exponential everage โดยมีสูตรคือ

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

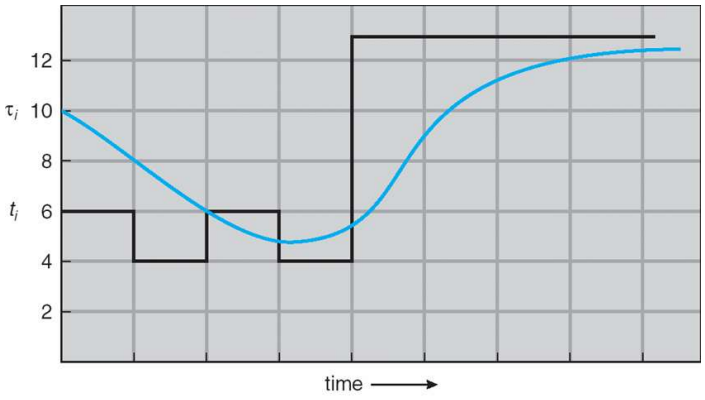
โดยที่

t_n คือเวลา CPU burst ที่ใช้ไปจริง ณ รอบที่ n

τ_{n+1}คือเวลาที่ประเมินสำหรับค่า CPU burst รอบถัดไป (เป็นค่าที่จะถูกนำมาใช้ทำ scheduling)

ค่า α มีค่าระหว่าง 0 ถึง 1

หากกำหนดให้ α =1 เราจะได้ว่า τ_n+1 = τ_n หมายความว่า กำหนดให้ค่าประเมิน CPU Burst ของแต่ละโพรเซสเท่ากับค่า CPU Burst ที่โพรเซสนั้นๆ ได้ใช้ไปจริงในรอบการคำนวณรอบก่อนหน้า ถ้าเราจะใช้หลักการคำนวณสะสมโดยให้น้ำหนักรอบล่าสุดมากสุด แล้วค่อยน้อยๆ ลงไปในรอบก่อนๆ หน้า (exponential average) เราก็กำหนดค่า α ให้มีค่าใดๆ ระหว่าง 0 ถึง 1 ตัวอย่างเช่น สมมติว่า เรากำหนดค่าตั้งต้นของ τ มีค่าเป็น 10 (เนื่องจากการคำนวณในรอบแรก ยังไม่มีสถิติการใช้งานจากรอบการคำนวณก่อนหน้า) และกำหนดให้ α เท่ากับ 0.5 ค่าประเมินของรอบถัดๆ ไปจะเป็นค่าเฉลี่ยระหว่างค่า CPU burst ที่เกิดขึ้นจริงของรอบก่อนหน้ากับค่าประเมิน CPU burst ของรอบก่อนหน้า และทำให้ได้ค่าประเมินในแต่ละรอบเป็นดังรูป (เส้นสีฟ้าคือเส้นบอกค่าประเมิน CPU Burst ของแต่ละรอบ)



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

การจัดการแบบ SJF อาจจะเป็นได้ทั้งแบบ nonpreeamptive หรือ preeamptive ก็ได้ ในกรณีของ nonpreemtive นั้น ก็จะพิจารณาเฉพาะโพรเซสทุกตัว โดยไม่สนใจว่าโพรเซสปัจจุบันมีรอบ CPU burst เป็นเท่าใดและการสลับโพรเซสถัดไปจะกระทำเมื่อโพรเซสปัจจุบันเรียก system call เท่านั้น ในกรณีที่เป็น preemptive นั้น จะนำค่าประเมิน CPU burst ของทุก

โพรเซสใน ready queue มาเปรียบเทียบกับ เวลาที่ยังเหลืออยู่ในการคำนวณของโพรเซสที่กำลังคำนวณปัจจุบัน (ใช้เวลาประเมิน CPU burst ของรอบก่อนหน้า ลบกับเวลาที่ใช้ไปแล้วในรอบปัจจุบัน) หากค่าเวลา CPU burst ที่ประเมินได้ของโพรเซสที่รออยู่มีค่าน้อยกว่าค่าเวลาคำนวณที่เหลืออยู่ของโพรเซสที่กำลังคำนวน โพรเซสที่กำลังคำนวณจะถูกผลักออกไปรอที่ ready queue แล้วดึงโพรเซสที่รอใน ready queue ตัวที่มีค่า CPU burst ต่ำสุดเข้ามาทำงานแทนโดยทันที ลักษณะเช่นนี้เราจึงเรียกว่า **Shortest-remaining-time-first scheduling**

สมมติว่ามีโพรเซสรอการประมวลดังนี้

process	หน่วยเวลาที่มาเข้า ready queue คิวและเป็นหน่วยเวลาที่ dispatcher จะทำงาน	CPU burst time ที่ประเมินได้ และสมมติว่าในการทำงานจริงได้ตามนี้
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

ณ เวลาที่ 0

โพรเซสที่อยู่ในready queue P_1

โพรเซสที่กำลังประมวล (ไม่มี)

ดังนั้น P_1 จะถูกดึงเข้าประมวลโดยทันที

ณ เวลาที่ 1

โพรเซสที่อยู่ใน ready queue P_2 (Burst time = 4)

โพรเซสที่กำลังประมวล P_1 (เวลาที่เหลือ 8-1 = 7)

ดังนั้น P_2 จะถูกดึงเข้ามาประมวลแทน P1 โดย P1 เหลือ Burst time = 7

ณ เวลาที่ 2

โพรเซสที่อยู่ใน ready queue P_1 (Burst time = 7)

P_3 (Burst time = 9)

โพรเซสที่กำลังประมวล P_2 (เวลาที่เหลือ 4-1 = 3)

ดังนั้น P_2 จะยังคงครองเวลาซีพียูอยู่เช่นเดิม

ณ เวลาที่ 3

โพรเซสที่อยู่ใน ready queue P_1 (Burst time = 7)

P_3 (Burst time = 9)

P_4 (Burst time = 5)

โพรเซสที่กำลังประมวล P_2 (เวลาที่เหลือ 4-2 = 2)

ดังนั้น P_2 จะยังคงครองเวลาซีพียูอยู่เช่นเดิม

ณ เวลาที่ 4

โพรเซสที่อยู่ใน ready queue P_1 (Burst time = 7)

P_3 (Burst time = 9)

P_4 (Burst time = 5)

โพรเซสที่กำลังประมวล P_2 (เวลาที่เหลือ 4-3 = 1)

ดังนั้น P_2 จะยังคงครองเวลาซีพียูอยู่เช่นเดิม

ณ เวลาที่ 5

โพรเซสที่อยู่ใน ready queue P_1 (Burst time = 7)

P_3 (Burst time = 9)

P_4 (Burst time = 5)

โพรเซสที่กำลังประมวล P_2 ออกจากสถานะ running ไปสถานะอื่น (สมมติว่าใช้เวลาตรงตามที่ประเมินไว้พอดี

ดังนั้น P_4 จะถูกดึงเข้ามาประมวลโดยทันที

ณ เวลาที่ 6

โพรเซสที่อยู่ใน ready queue P_1 (Burst time = 7)

P_3 (Burst time = 9)

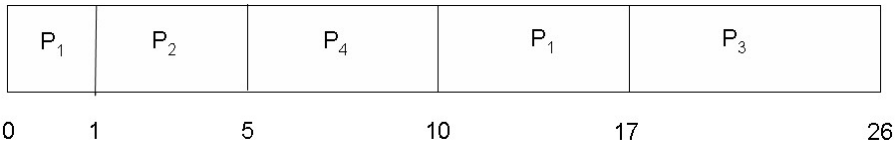
โพรเซสที่กำลังประมวล P_4 (Burst time = 5-1=4)

ดังนั้น P_4 จะยังครองเวลาซีพียูเช่นเดิม

ณ เวลาที่ 7-9 จะเห็นว่าค่าเวลาที่เหลือของ P4 ที่มีค่าน้อยลงเรื่อยๆ ยังคงน้อยกว่าโพรเซสอื่นใน ready queue ดังนั้นโพรเซสอื่นจะต้องรอจนกว่า P_4 เปลี่ยนสถานะจาก running เป็นอย่างอื่น ตัวอย่างนี้ สมมติว่า P_4 ไปยังสถานะอื่นในเวลาพอดี

ณ เวลาที่ 10 จะพบว่า P_1 จะถูกดึงมาประมวลต่อด้วยเวลาที่เหลือคือ 7 หน่วย ซึ่งน้อยกว่า P_3 ที่ยังคงมี 9 หน่วยตั้งแต่ถูกโหลดมารอใน ready queue ตั้งแต่แรก ดังนั้น P_1 จะทำงานต่อไปจนกระทั่งครบช่วง CPU burst จริง และจากนั้นในเวลาที่ 17 P_2 จึงได้โอกาสทำงานต่อไปจนกระทั่งหมด CPU Burst (จากนั้นก็จะเริ่มรอบการทำงานในรอบ CPU burst ถัดไปของทุกโพรเซส)

โดยสรุป เราจะได้ภาพแสดงเวลาการทำงานของโพรเซสต่างๆ ออกมาดังรูป



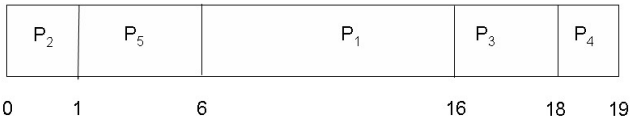
Priority Scheduling การจัดสรรเวลาแบบใครมีสิทธิสูงกว่าจะได้รับการบริการก่อน

ในกลไกแบบนี้ จะจัดลำดับความสำคัญของโพรเซส โดยกำหนดค่าลำดับความสำคัญ (priority) ให้กับทุกโพรเซสที่กำลังทำงานก่อนเริ่มสั่งให้ทำงาน ค่าลำดับความสำคัญนี้ อาจจะมาจากการพิจารณาการใช้ทรัพยากรภายในระบบเอง ซึ่งรวมไปถึงค่าเวลา I/O burst และ CPU burst ด้วยก็ได้ หรืออาจจะได้มาจากการกำหนดจากนอกระบบ เช่นจากผู้ใช้โดยตรง

ขั้นตอนวิธี SJF ที่กล่าวมาก่อนหน้า อาจเรียกได้ว่าเป็นกรณีหนึ่งในขั้นตอนวิธีนี้ โดย SJF อาศัยค่า CPU Burst time เป็นค่า priority นั่นเอง

ตัวอย่างการใช้ priority scheduling

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



กลไกแบบ Priority scheduling สามารถเป็นได้ทั้งแบบ preemptive และ nonpreemptive ในกรณีของ nonpreemptive นั้น เมื่อโพรเซสตัวใดจะเข้าสู่ ready queue ก็จะถูกจับไปเทียบกับสมาชิกอื่นใน queue แล้วก็จะนำไปแทรกไว้ระหว่างโพรเซสที่มีค่า priority ต่ำกว่ากับตัวที่สูงกว่า (ถ้าตัวใหม่ที่เข้ามาเป็นตัวที่มี priority สูงสุดก็จะถูกจับไปวางไว้หน้าคิว รอประมวลในการเรียกของ dispatcher ถัดไปได้เลย)

ในกรณีที่เป็นแบบ preemptive นั้น ก่อนอื่น โพรเซสใหม่ที่เข้ามาใน ready queue จะถูกเทียบกับโพรเซสที่กำลังครอบครองซีพียูอยู่ ถ้าตัวใหม่มีค่า priority สูงกว่าก็จะ preempt ตัวเดิมในทันที โดยตัวเดิมจะถูกนำไปจับเข้า ready queue (ตามกฎการแทรก) แต่ถ้าโพรเซสใหม่ที่เข้ามามี priority ต่ำกว่าตัวที่กำลังรันอยู่ ก็จะส่งไปเข้าคิวตามกฎข้างต้นเช่นเดียวกัน

ปัญหาการใช้ Priority queue โดยเฉพาะแบบ preemptive ก็คือ ในกรณีที่มีโพรเซสที่มีค่า priority สูงจำนวนหนึ่งรันอยู่พร้อมกับโพรเซสที่มีค่า priority ต่ำกว่าจำนวนหนึ่งในระบบ เราจะพบว่าในรอบการทำงานของซีพียูที่เข้าพิจารณาเลือกดึงโพรเซสเข้าไปรันในแต่ละรอบ โอกาสที่โพรเซสที่มี priority สูงกว่าจะได้รับเลือกเสมอ ทำให้โพรเซสที่มีค่า priority ต่ำกว่าจะไม่มีโอกาสได้รันเลย (ถูกบล็อกไว้ไม่ให้รัน blocked) ลักษณะเช่นนี้เราเรียกว่า **indefinite blocking หรือ starvation**

ทางแก้ปัญหาของการ starvation ของโพรเซสที่มีค่า priority ต่ำคือการเพิ่มค่า priority ของแต่ละโพรเซสขึ้นเรื่อยๆ หากโพรเซสดังกล่าวต้องอยู่ใน ready queue นานเป็นพิเศษ (เช่นเพิ่มขึ้นทุกครั้งที่ถูกแซงคิว) กลไกดังกล่าวเรียกว่า aging (เช่น ค่อยๆ เพิ่มค่า priority เฉพาะรอบปัจจุบันที่รออยู่ทีละ 1 หน่วยทุกๆ 2 นาทีที่รอ ดังนั้นเมื่อเวลาผ่านไปช่วงหนึ่ง ค่า priority ของโพรเซสที่รอนาน จะมีค่าสูงจนกระทั่งเอาชนะโพรเซสที่กำลังรันอยู่ได้ และจึงจะสามารถมีสิทธิเข้าครอบครองซีพียูในที่สุด

Round-Robin (RR) Scheduling การผลัดกันเข้าใช้เวลาซีพียูตามคาบเวลาที่กำหนด

ขั้นตอนวิธี Round Robin มีความคล้ายคลึงกันกับแบบ FCFS โดยเพิ่มกลไกการขัดจังหวะของ dispatcher เพื่อตรวจสอบและ preempt โพรเซส ในทุกๆ คาบเวลาที่กำหนด **คาบเวลา (time quantum** หรือ **time slice)** นี้มักจะมีค่าเวลาน้อยกว่าหลักวินาที เช่นประมาณ 10 ถึง 100 มิลลิวินาที หน่วยที่ถูกคัดออกมาจะถูกจับไปวางท้ายคิว เพื่อให้ dispatcher สามารถดึงโพรเซสแต่ละตัววนกันไปได้จนกระทั่งโพรเซสนั้นๆ จบรอบ CPU-burst ไปตามปกติ

ดังนั้น เราจะพบว่า หากมีโพรเซสกำลังสลับรอรันอยู่ใน ready queue (นับรวมตัวที่กำลังรันด้วย) ทั้งหมด n โพรเซส และคาบเวลาเท่ากับ q เราจะพบว่าทุกโพรเซสจะได้รับโอกาสการรันในทุกๆ (n-1)/q (รอประมวลไม่เกินช่วงเวลาที่กำหนด)

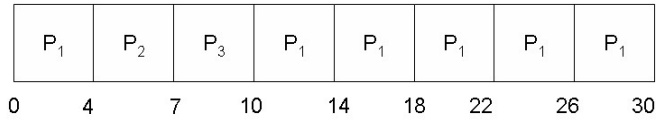
ในกรณีที่โพรเซสที่กำลังทำงานอยู่นั้น ออกจากสถานะ running ก่อนเวลาที่กำหนด dispatcher ก็จะดึงเอาโพรเซสถัดไปที่หัวคิวมาทำงานต่อโดยทันที

ด้วยลักษณะการแบ่งเวลาทำงานระหว่างโพรเซส ถ้าระบบมีจำนวนโพรเซสอยู่มาก ก็จะส่งผลทำให้เวลาเฉลี่ยในการรอ (waiting time) มีค่าสูง เพราะทุกโพรเซสต้องผลัดเปลี่ยนหมุนเวียนกันเข้ามารับการประมวล ทำให้แต่ละโพรเซสเสียเวลารอใน

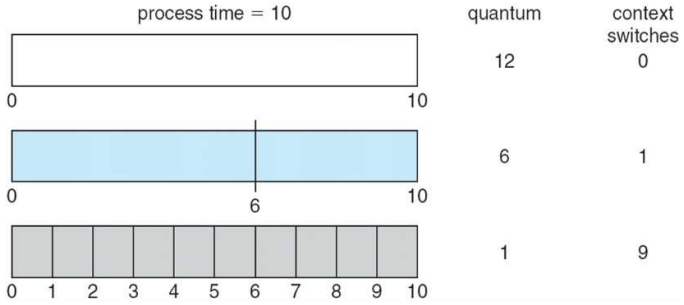
ready queue นาน และถ้า time quantum มีค่าสูงมากๆ ขั้นตอนวิธีนี้ก็จะมีลักษณะการจัดการไม่แตกต่างไปจาก FIFO แต่ถ้าค่า time quantum มีค่าต่ำมากจนเกินไป ก็จะทำให้ประสิทธิภาพในการใช้งานซีพียูต่ำลง เพราะในทุกๆ คาบเวลา ตัว dispatcher ก็จะต้องทำงานหนึ่งครั้งเสมอ การลดคาบเวลาลงมากก็จะทำให้สัดส่วนเวลาที่ใช้ในการทำงานของโพรเซส มีค่ามาเข้าใกล้ค่าเวลาที่ dispatcher ทำงานนั่นเอง

ตัวอย่างการจัดการแบบ RR โดยกำหนดให้ time quantum = 4

Process	Burst Time
P1	24
P2	3
P3	3



- สังเกตว่า ค่า average turnaround (ในที่นี้จะหมายถึงช่วงเวลารอคอยที่จะได้รับประมวลในรอบถัดไป) จะสูงกว่า SJF (เพราะต้องจัดแบ่งทำงานระหว่างทุกโพรเซส) แต่จะมีค่า response time ที่ดีกว่า (เพราะทุกโพรเซสจะได้รับโอกาสการเข้ามาครอบครองซีพียู ไม่มีโพรเซสใดที่ถูกละทิ้งไว้หรือใช้เวลารอนานมากเป็นพิเศษ และโพรเซสที่เป็น I/O bound จะสามารถจบ CPU burst ได้เร็วกว่าเพราะไม่ต้องรอนาน)
- q (time quantum) จะต้องมีค่ามากพอเมื่อเทียบกับเวลาการเปลี่ยนโพรเซส
- q ที่ใช้งานตามปกติมีค่าระหว่าง 10-100 มิลลิวินาที เมื่อเทียบกับเวลาในการสลับโพรเซสที่มีค่าประมาณน้อยกว่า 10 ไมโครวินาที



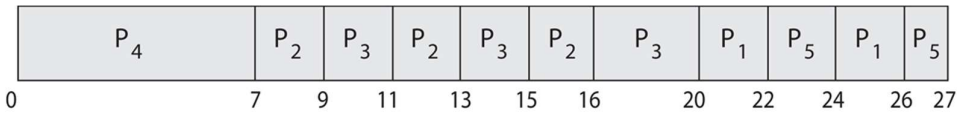
จากรูป จะเห็นว่าหากเราซอย q ลงมีค่าน้อยๆ จำนวนของการเปลี่ยนโพรเซสก็จะมีค่าสูงขึ้น ซึ่งค่าเวลาที่ใช้ในการเปลี่ยนโพรเซสมีค่าเท่าเดิมเสมอ แต่เมื่อมีจำนวนมากขึ้น ก็เท่ากับว่าประสิทธิภาพในการใช้งานซีพียูจะน้อยลง เพราะซีพียูจะเหลือเวลาเอาไปทำงานตามปกติได้น้อยลง

Priority Scheduling with RR

ขั้นตอนวิธีนี้เป็นการผสมผสานกันระหว่างการทำ Round-Robin และการทำ Priority Scheduling โดยพิจารณาแยกค่าระดับความสำคัญของโพรเซส โพรเซสที่มีค่าความสำคัญมากก็จะทำก่อน แต่ถ้าพบว่ามีโพรเซสหลายตัวค่าระดับความสำคัญเท่ากัน โพรเซสเหล่านั้นจะผลัดวนทำงานโดยใช้ RR จนกระทั่งครบรอบ CPU Burst ตัวอย่างเช่น

	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	1

จะได้ Gantt chart ดังรูป (สมมติค่า time quantum เท่ากับ 2)



Multilevel Queue Scheduling การจัดสรรโดยใช้กลไกคิวหลายระดับ

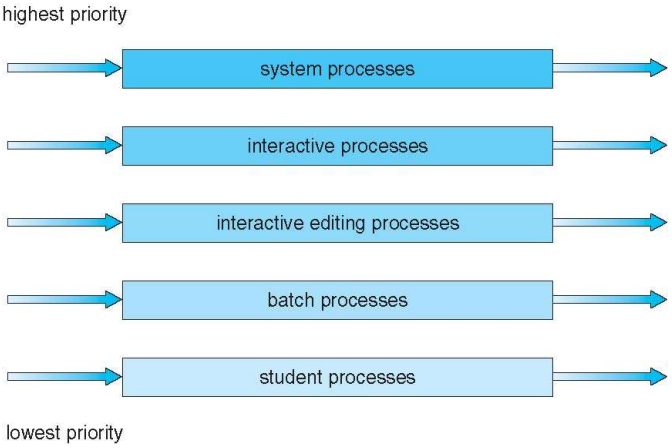
ในลักษณะการจัดการแบบนี้ ready queue จะถูกสร้างขึ้นมากกว่าหนึ่งคิว โดยแต่ละคิวจะรองรับงานที่ระดับความสำคัญแตกต่างกันออกไป เช่น งานที่ผู้ใช้กำลังติดต่ออยู่ หรืองานตามปกติ ถือเป็น foreground ส่วนงานที่ทำหน้าที่ซึ่งไม่มีความจำเป็นเร่งร้อนจะต้องทำให้เสร็จโดยเร็ว ที่มักรันอยู่ภายในระบบแบบอัตโนมัติ (เช่นรันตามช่วงเวลาที่กำหนด) เราเรียกว่า background job

ในลักษณะเช่นนี้ ระบบปฏิบัติการสามารถออกแบบการจัดการ ready queue ของงาน foreground ให้มีลักษณะที่ตอบสนองได้ดี (response time ที่ต่ำ) และมี priority ที่สูงกว่า background โดยโพรเซสแต่ละตัวจะถูกกำหนดประเภทตั้งแต่เวลาที่โพรเซสเริ่มทำงาน ตัวอย่างเช่น ระบบหนึ่งอาจเลือกใช้ RR กับ foreground และ FCFS กับ background เป็นต้น

สำหรับสัดส่วนการให้เวลาซีพียูกับงานในกลุ่ม foreground กับ background นั้น อาจจัดลำดับในลักษณะที่ให้ foreground ก่อนแล้วตามด้วย background อย่างง่ายๆ แต่ทั้งนี้อาจส่งผลให้เกิด starvation กับงานใน background ได้หากงานใน foreground นั้นใช้ซีพียูมากจนไม่มีเวลาเหลือให้ background

ทางเลือกที่ระบบปฏิบัติการอาจใช้ก็คือ การแบ่งสัดส่วนเวลาประมวลในแต่ละคิวไว้ตายตัวแทนการกำหนดเป็น priority เช่น กำหนดเวลา 80 เปอร์เซนต์ของซีพียูในการจัดการงานของ foreground (แล้วในคิวดังกล่าวจะใช้ขั้นตอนวิธีใดก็สุดแล้วแต่) และอีก 20 เปอร์เซนต์ของซีพียูในการจัดการงาน background เป็นต้น

ในระบบที่มีการจัดการซับซ้อนขึ้นไปอีก อาจจะแบ่ง ready queue ออกมากกว่าสองระดับก็ได้



Multilevel Feedback-queue scheduling การจัดการแบบคิวหลายระดับพร้อมการป้อนกลับ

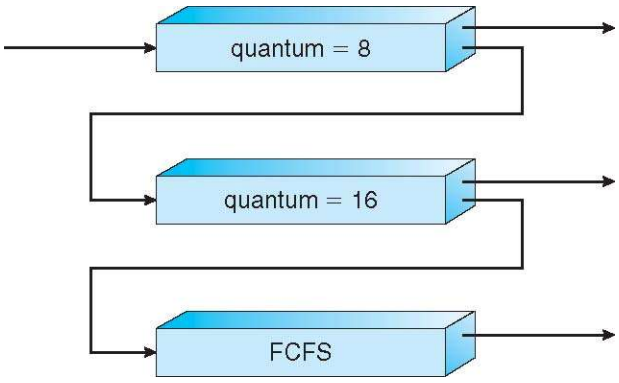
ระบบที่ใช้ขั้นตอนวิธีนี้พัฒนาการต่อมาจากขั้นตอนวิธีแบบคิวหลายระดับ แต่ในกรณีนี้ เราจะอนุญาตให้โพรเซสสามารถเปลี่ยนคิวที่ตนเข้ารอใช้งานได้เมื่อมีสภาวะที่กำหนด เช่น โพรเซสที่อยู่ในคิวที่มีระดับความสำคัญต่ำกว่า หยุดรอนานเกินไป ก็อาจจะยกระดับโดยการย้ายโพรเซสนั้นไปอยู่ในคิวที่มีระดับสูงกว่า (การประยุกต์แนวคิด aging) หรือเมื่อพบว่า มีโพรเซสหนึ่งใช้เวลา CPU มากเกินไป ก็จะถูกลดระดับลงในคิวที่มีระดับต่ำกว่า ซึ่งจะทำให้โพรเซสที่ใช้เวลาของซีพียูพอๆ กัน ไปรวมกันอยู่ในคิวเดียวกัน ทำให้จัดการได้ง่ายขึ้น โดยเฉพาะโพรเซสที่ต้องการ response time ที่สูง ก็จะไปอยู่ในคิวเดียวกันที่จัดการแบบ RR ก็จะทำให้ค่า response time มีค่าที่ดีขึ้น เป็นต้น

ปัจจัยต่างๆ ที่ต้องคำนึงในการออกแบบระบบปฏิบัติการเพื่อจัดการโพรเซสด้วยขั้นตอนวิธีนี้มีเช่น

- จำนวนของคิวที่จะมีในระบบ
- ขั้นตอนวิธีในการจัดการโพรเซสของแต่ละคิว
- ขั้นตอนวิธีในการตัดสินใจว่าจะยกระดับโพรเซสขึ้น (จากคิวที่มีค่าความสำคัญต่ำไปสูงกว่า)
- ขั้นตอนวิธีในการตัดสินใจในการกดระดับโพรเซสลง
- ขั้นตอนวิธีในการเลือกคิวที่จะเข้าบริการเมื่อโพรเซสนั้นกลับมาจาก state อื่นจะเข้าสู่ ready state

ตัวอย่างการประยุกต์

- กำหนดไว้เป็นสามคิว Q_0 สำคัญสูงสุด เป็น RR q=8ms Q_1 สำคัญปานกลาง เป็น RR =16ms และ Q_2 ระดับต่ำสุด เป็น FCFS
- กลไกการจัดการ
    - โพรเซสที่มาจาก state อื่นจะเข้าสู่ ready queue ให้จับเข้า Q_0
        - ตามปกติ ถ้า Burst time น้อยกว่า 8ms ก็จะถูกปลดออกไปสถานะอื่นเมื่อครบ Burst time ที่ใช้ไปจริง
        - ถ้า Burst time มากกว่า 8ms ให้ปลดโพรเซสออก แล้วจับย้ายไปรอใน Q_1
    - โพรเซสที่อยู่ใน Q_1
        - ถ้า Burst time ใช้มากกว่า 16 ms ให้ปลดโพรเซสออก แล้วจับย้ายไปรอใน Q_2
        - โพรเซสที่อยู่ใน Q_2 ประมวลตามกลไก FCFS ไปตามปกติ



4.4 กรณีปลีกย่อยอื่นๆ ในการจัดสรรเวลาใช้งานซีพียู

การจัดสรรเวลาสำหรับเธรด

- ในกรณีที่มีการแบ่งโพรเซสออกเป็นหลายเธรด การร้องขอ system call จากเธรด ซึ่งเป็นการเข้าถึงบริการของระบบปฏิบัติการในส่วนของ kernel thread อาจจะส่งผลต่อการแย่งชิงในการขอรับบริการ kernel thread ในกรณีของการจัดการเธรดแบบ many-to-one หรือ many-to-many เนื่องจากจำนวน kernel thread มีน้อยกว่า user thread
    - ในกรณีที่ระบบปฏิบัติการรองรับ ผู้พัฒนาโปรแกรมสามารถเซ็ตให้บางเธรดมีระดับความสำคัญมากกว่าเธรดอื่นในโพรเซสเดียวกัน เผื่อกรณีที่มีสองเธรดแย่งใช้บริการ kernel thread เธรดที่มีระดับความสำคัญสูงกว่าจะถูกเลือกก่อน (เพื่อไม่ให้เธรดที่เป็นเธรดวิกฤติต้องรอ) กรณีนี้จะเป็นการเซ็ต process-contention scope (PCS) ให้เธรด
- ผู้พัฒนาโปรแกรมยังมีอีกทางเลือก โดยอาจจะพิจารณาว่าเธรดบางเธรดของโพรเซสผู้ใช้นั้น มีความสำคัญมากเมื่อเทียบกับเธรดอื่นๆ ของโพรเซสอื่นๆ ในระบบ ในกรณีนี้จะเป็นการเซ็ต system-contention scope (SCS) ให้กับเธรดนั้นๆ

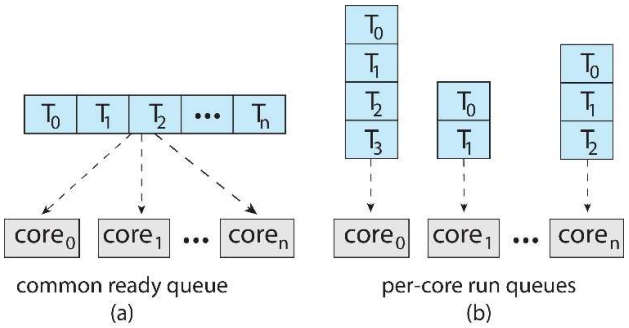
การจัดสรรเธรดในสภาพแวดล้อมแบบหลายคอร์

ในสภาพแวดล้อมที่มีซีพียูหลายคอร์ เราสามารถออกแบบระบบปฏิบัติการให้กระจายงานไปรันบนซีพียูแต่ละตัวที่มีในระบบได้ (load sharing) แต่ทั้งนี้ กลไกการจัดสรรซีพียูแต่ละตัวจะมีความซับซ้อนอย่างไรนั้นขึ้นอยู่กับระบบอย่างเช่น

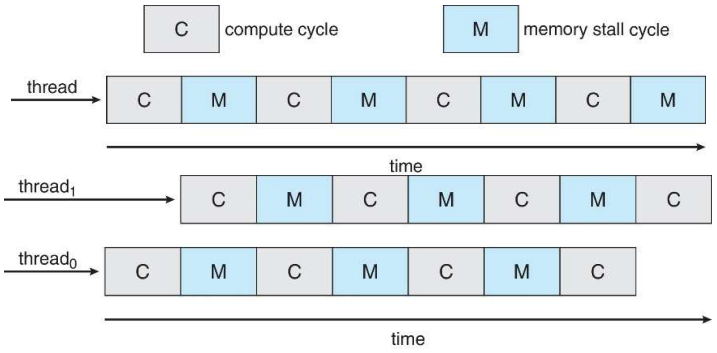
- Multicore CPUs ซีพียูที่มีหลายคอร์ โดยทั่วไปมักจะกระจายงานแบ่งกันไปรันบนคอร์ใดก็ได้โดยไม่มีผลกระทบอื่น
- Multithreading CPU ซีพียูที่คอร์หนึ่งรันได้หลายเธรด (SMT Simultaneous Multithreading) กรณีนี้ในแต่ละเธรดจะแชร์ทรัพยากรการคำนวณและอื่นๆ ระหว่างกัน ทำให้การกระจายงานไปบนเธรดต่างๆ ในคอร์เดียวกันอาจเกิดการแย่งทรัพยากรกัน ทำให้ต้องรอกัน ระบบปฏิบัติการที่รับรู้ SMT จึงมักจะกระจายงานไปรันบนคอร์ต่างๆ ก่อนจนเต็ม แล้วจึงค่อยแบ่งไปรันบนเธรดที่ว่างในคอร์ต่างๆ
- NUMA systems ระบบที่แต่ละคอร์ไม่ได้เข้าถึงหน่วยความจำและ I/O ต่างๆ ที่มีในระบบในลักษณะที่เท่าเทียมกัน ทั้งนี้มักเกิดจากระบบที่สร้างขึ้นจากซีพียูหลายตัวที่แต่ละตัวมีบัสหน่วยความจำ และ I/O แยกกัน แต่มีเส้นทางสื่อสารเชื่อมต่อระหว่างกัน ระบบปฏิบัติการที่รับรู้ NUMA จะพยายามกระจายงานไปรันบนคอร์ต่างๆ ที่อยู่บนซีพียูตัวเดียวกัน เพื่อให้เสียเวลาเข้าถึงหน่วยความจำและ I/O น้อยที่สุด
- Homogenous/Heterogenous Multiprocessing
    - Homogenous Multiprocessing ในลักษณะเช่นนี้ ซีพียูแต่ละคอร์จะมีชุดคำสั่งที่เหมือนกัน และมีประสิทธิภาพที่เท่าเทียมกัน การกระจายงานไปรันบนคอร์ใดๆ จึงทำได้โดยไม่มีผลที่แตกต่างกัน
    - Heterogenous Multiprocessing ซีพียูในลักษณะนี้ จะมีบางคอร์ที่มีประสิทธิภาพสูงกว่า และบางคอร์มีประสิทธิภาพต่ำกว่า และชุดคำสั่งในคอร์เล็ก (ประสิทธิภาพต่ำกว่า)อาจจะมีไม่ครบถ้วนเท่าตัวใหญ่ การส่งงานไปรันบนคอร์ต่างๆ โดยระบบปฏิบัติการ จึงต้องพิจารณาถึงปัจจัยเช่น ต้องการความเร็ว หรือต้องการการประหยัดพลังงาน เป็นต้น

นอกจากนี้ โครงสร้างการบริหารจัดการของระบบปฏิบัติการบนระบบคอมพิวเตอร์หลายซีพียูยังมีผลต่อการออกแบบการจัดสรรทรัพยากรซีพียู อย่างในกรณีของ asymmetric multiprocessing นั้น อาจจะจัดโพรเซสทั้งหมดที่ต้องติดต่อกับ I/O รวมทั้งโพรเซสของระบบปฏิบัติการทั้งหมดมารันบนหน่วยที่เป็น master ในขณะที่หน่วยที่เป็น slave ทั้งหมดจะได้รับโพรเซสของผู้ใช้ไปรันเท่านั้น

ในกรณีของ symmetric multiprocessing การจัดการจะมีความซับซ้อนขึ้น โดยซีพียูแต่ละหน่วยจะต้องรับการจัดสรรโพรเซสจาก ready queue เดียวกัน ในขณะที่กลไกการควบคุมการทำงานภายในซีพียูนั้นเป็นเอกเทศกันอยู่ (เช่น การจัดสรรเวลาภายในอาจแตกต่างกัน อินเทอร์รัพต์ฐานเวลาอาจเหมือนหรือแตกต่างกัน ฯลฯ) และในทางปฏิบัติก็อาจจะมี ready queue ย่อยสำหรับซีพียูแต่ละหน่วยก็ได้ ปัญหาอื่นๆ ที่ตามมาได้อีกก็จะเป็นเรื่องการแชร์ข้อมูลระหว่างโพรเซสที่กำลังรันอยู่บนซีพียูคนละตัว รวมทั้งการชิงโพรเซสเดียวกันไปรันบนซีพียูมากกว่าหนึ่งตัวพร้อมกัน ฯลฯ



ตัวอย่างในกรณีของการจัดการซีพียูที่มีหลายเธรดต่อคอร์ ในลักษณะเช่นนี้ระบบปฏิบัติการสามารถจัดสรรให้แต่ละเธรดเข้าใช้ซีพียูคอร์เดียวกัน โดยคาดหวังกลไกภายในของคอร์ที่ใช้ทรัพยากรภายในซีพียูแต่ละส่วนไม่พร้อมกันอย่างตลอดเวลา (เช่น การรอข้อมูล/ชุดคำสั่งจากหน่วยความจำ กับการคำนวณ ซึ่งจะเป็นลำดับการทำงานที่ต่อเนื่องกันในวัฏจักร Fetch-Decode-Execute-Store ของซีพียูยุคปัจจุบัน)



ปัญหาต่อมาคือเรื่องของการใช้แคชภายในซีพียูอย่างมีประสิทธิภาพ ในระบบคอมพิวเตอร์ที่มีซีพียูหน่วยเดียว โพรเซสหนึ่งๆ จะมีโอกาสรันอยู่บนโพรเซสนั้นเท่านั้น และส่งผลให้แคชภายในซีพียูสามารถเก็บข้อมูลต่างๆ ของโพรเซสไว้ภายในเพื่อรอใช้งานในรอบการคำนวนถัดไปได้ แต่ในระบบหลายซีพียู เมื่อโพรเซสหนึ่งมีโอกาสรันอยู่บนซีพียูตัวหนึ่ง ในรอบถัดไปอาจจะได้ไปรันบนซีพียูอีกตัวหนึ่ง ซึ่งใช้แคชคนละชุด นั่นหมายถึงต้องโหลดข้อมูลขึ้นแคชใหม่ หรือถ้าโพรเซสผลัดใช้ซีพียูไปมา แคชก็ต้องถูกโหลดใหม่ทุกรอบ เป็นการลดประสิทธิภาพการทำงานลงอย่างมาก ในกรณีเช่นนี้ ระบบปฏิบัติการมักกำหนดให้โพรเซสหนึ่งเมื่อมีโอกาสครอบครองซีพียูหนึ่งแล้ว จะให้ใช้ซีพียูตัวนั้นไปตลอด เรียกว่าการจับคู่กันระหว่างโพรเซสและหน่วยซีพียู (process affinity) ระบบปฏิบัติการอาจจะเปิดโอกาสให้โพรเซสหนึ่ง ถูกย้าย (migrate) ไปครอบครองซีพียูตัวอื่นได้ (เช่นในกรณีที่โหลดไม่สมดุลกันระหว่างซีพียูแต่ละตัวเป็นเวลานาน) ลักษณะเช่นนี้เรียกว่า soft affinity กับบางระบบปฏิบัติการ (เช่นลินุกซ์) จะเปิดทางเลือกบังคับ เช่นมี system call ที่กำหนดให้โพรเซสหนึ่งๆ ล็อคอยู่กับซีพียูเพียงหน่วยเดียวก็ได้ เราเรียกว่า hard affinity

การทำสมดุลระหว่างคอร์

ในระบบคอมพิวเตอร์หลายซีพียูที่เป็นแบบ symmetric ซึ่งใช้ ready queue ร่วมกัน ในกรณีนี้ เมื่อซีพียูตัวใดตัวหนึ่งว่างงาน ก็จะสามารถมาดึงเอาโพรเซสที่รออยู่ใน ready queue ได้โดยทันที ดังนั้นในการออกแบบลักษณะนี้ ซีพียูทุกตัวในระบบ

จะถูกกระจายงานให้รันได้อย่างเฉลี่ยเหมาะสม (เมื่อพิจารณาว่ามีงานอยู่ในระบบจำนวนมาก และยอมให้มีการ migration ได้) แต่สำหรับการออกแบบระบบปฏิบัติการที่มี ready queue เพิ่มเติมเฉพาะซีพียูแต่ละหน่วย การ migration จะไม่เกิดขึ้น ดังนั้นจึงต้องมีกลไกการสมดุลโหลดของซีพียู (load balancing) เพิ่มเติมในการคัดโพรเซสจาก ready queue ของซีพียูตัวหนึ่งโอนให้กับตัวที่ว่างงาน เป็นต้น

กลไกการทำ migration ระหว่างซีพียูยังมีได้สองแบบ แบบแรกคือ **push migration** ในลักษณะนี้ จะมีอีกโพรเซสที่คอยตรวจสอบสถิติค่า utilization ของซีพียูทุกตัวในระบบ แล้วคอยคัดโพรเซสจากซีพียูตัวที่ถูกใช้งานมาก ไปให้ซีพียูที่ถูกใช้งานน้อย กับอีกกรณีหนึ่ง **pull migration** ลักษณะเช่นนี้ จะมีโพรเซสประจำซีพียูแต่ละตัวที่จะทำงานเมื่อพบว่าค่า utilization ของตนต่ำกว่าค่าที่กำหนด จะไปดึงเอาโพรเซสที่ซีพียูอื่นที่มีค่า utilization ที่มาก มาเป็นของตนเอง การประยุกต์ใช้ push migration และ pull migration สามารถกระทำได้ทั้งสองแบบในระบบปฏิบัติการหนึ่งๆ

ยังมีเทคโนโลยีทางฮาร์ดแวร์อันหนึ่งที่ถูกนำมาใช้กันอย่างแพร่หลายคือ SMT (Simultaneous Multithreading) (อินเทลใช้ชื่อทางการค้าว่า hyperthreading technology) กลไกการทำ SMT คือการออกแบบซีพียูให้มีการใช้วงจรภายในให้มีลักษณะกึ่งแบ่งปันกันใช้ นั่นคือ จากมุมมองของซอฟต์แวร์แล้ว จะมองเห็นโครงสร้างซีพียูประกอบไปด้วยหน่วยประมวลผลราวกับว่าเป็นหลายคอร์ แต่ในความเป็นจริงนั้นเป็นคอร์เดียวแต่แบ่งการทำงานภายบางส่วนเป็นหลายหน่วย ซีพียูที่นิยมใช้กันในปัจจุบันโดยส่วนมากจะจำลองเป็น 2 หน่วย โดยแต่ละหน่วยมีการจัดการอินเทอร์รัพต์ การเข้าถึงหน่วยความจำ วงจรโหลดและรอทำงานตามคำสั่ง จะแยกกันอิสระ แต่วงจรคำนวณพื้นฐานภายในนั้น อาจใช้ร่วมกันระหว่างหน่วย (และอาจจะมีหลายชุด -เรียกว่า superscalar เพื่อส่งผลการคำนวณแบบคาบเกี่ยว -ตัวหนึ่งเสร็จแล้ว อีกตัวกำลังคำนวณข้อมูลอีกชุด หากใช้ในหน่วยเดียว ทำให้ผลลัพธ์ได้เร็วขึ้น) ดังนั้น แม้ว่าระบบปฏิบัติการจะมองเห็นหน่วยประมวลผลเป็นหลายหน่วย แต่ประสิทธิภาพการทำงานจะไม่ได้สูงเป็นสองเท่า เพราะมีโอกาสสูงที่วงจรภายในอาจถูกแย่งใช้งานกันระหว่างหน่วยย่อย ดังนั้นในทางปฏิบัติการกระจายงานที่แตกต่างกันจะช่วยเพิ่มประสิทธิภาพการใช้งานซีพียูแบบนี้ให้สูงสุด และระบบปฏิบัติการสามารถจัดการได้ดังเช่น ในกรณีที่มีซีพียูที่มี SMT สองตัวอยู่บนระบบ และแต่ละตัวรองรับสองเธรด เวลาที่แบ่งเธรดออกเป็นสองเธรด ระบบปฏิบัติการก็จะส่งแต่ละเธรดไปรันบนซีพียูคนละตัว (จะไม่ไปรันบนซีพียูเดียวกันแต่คนละหน่วยย่อย เพราะนั่นจะเปิดโอกาสให้เกิดการแย่งการใช้งานวงจรภายในซีพียูได้มากขึ้น)

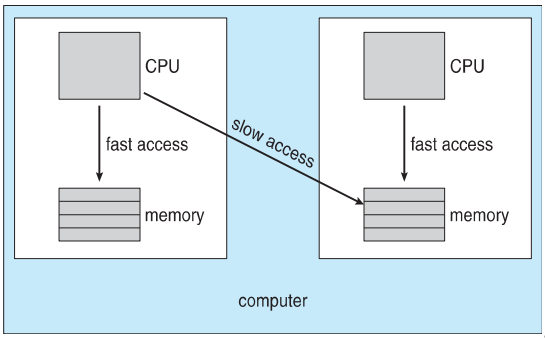
การกำหนดความสัมพันธ์ (affinity) ของเธรดกับซีพียูคอร์

เมื่อเธรดหนึ่งๆ รันบนซีพียู จะพักข้อมูลและชุดคำสั่งที่กำลังคำนวณอยู่บนแคชของซีพียูคอร์นั้นๆ ไปจนถึงหน่วยความจำหลักที่ถูกจัดสรรให้ ก็จะถูกจัดสรรให้เข้าถึงเร็วที่สุด (เช่น เชื่อมต่อตรงกับชิพซีพียูนั้นๆ) ลักษณะนี้ทำให้การรันเธรดนั้นๆ บนซีพียูคอร์นั้นๆ มี**ลักษณะความสัมพันธ์กัน (processor affinity)**

การกระจายงานโดยการย้ายเธรดจากคอร์ที่ใช้งานมาก ไปยังคอร์ที่ใช้งานน้อย จึงเป็นการทำให้ความสัมพันธ์นี้อาจสูญเสียไป เช่น จะต้องไปดึงข้อมูลมาลงในแคชใหม่ (หรือต้องดึงข้อมูลแคชข้ามคอร์ผ่านช่องทางสื่อสารระหว่างซีพียู) หรือในกรณีที่ข้ามชิพ (ระบบคอมพิวเตอร์แบบ NUMA ที่มีหลายชิพซีพียูบนบอร์ดเดียวกัน) การเข้าถึงหน่วยความจำหลักจะเสียเวลามาก ต้องผ่านช่องทางการสื่อสารระหว่างชิพ ส่งผลทำให้ประสิทธิภาพการทำงานของระบบลดลง

ระบบปฏิบัติการจึงมีกลไกที่สามารถกำหนดความสัมพันธ์นี้ได้ในรูป

- **soft affinity** เป็นการยอมให้เธรดต่างๆ ให้ถูกย้ายไปยังคอร์อื่นๆ ใดได้โดยอิสระ
- **hard affinity** เป็นการกำหนดให้เธรดต่างๆ ต้องรันบนคอร์เฉพาะเท่าที่กำหนดให้เท่านั้น (อาจจะเป็นคอร์เดียว หรือกลุ่มของคอร์ที่กำหนดภายในระบบปฏิบัติการ)



ตัวอย่างคอมพิวเตอร์ที่มีซีพียูสองตัว การเข้าถึงหน่วยความจำเป็นแบบ NUMA ในลักษณะเช่นนี้ ระบบปฏิบัติการที่รองรับ NUMA (NUMA-aware) จะกำหนดให้เธรดรันบนคอร์บนชิพซีพียูที่เข้าถึงพื้นที่หน่วยความจำหลักของเธรดนั้นๆ เท่านั้น