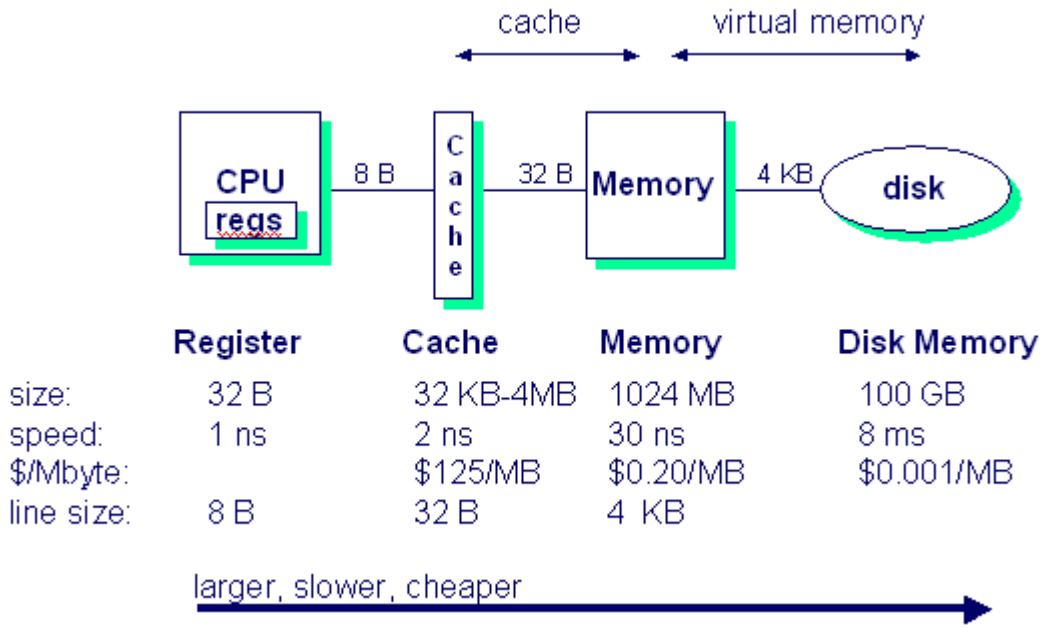
บทที่ 7 หน่วยความจำเสมือน (Virtual Memory)

จากบทที่ผ่านมา เราได้เรียนรู้การทำงานของระบบปฏิบัติการ ในเรื่องของการบริหารหน่วยความจำ (Memory Management) ซึ่งเป็นการบริหารที่ส่วนของหน่วยความจำหลัก ในบทนี้เราจะกล่าวถึงสิ่งที่มีความจำเป็นต่อการทำงานของคอมพิวเตอร์เป็นอย่างมาก โดยเฉพาะการทำงานของโปรแกรมต่างๆในทุกวันนี้ ที่ต้องการทรัพยากรที่มากขึ้น

หลักการทำงานของหน่วยความจำเสมือนก็คือ "**หน่วยความจำของกระบวนการต่างๆจะต้องมีได้อย่างไม่จำกัด**" ระบบปฏิบัติการจะต้องจัดสรรให้ได้ตามความต้องการ แต่ในความคิดของเราอาจคิดว่า จะเป็นไปได้อย่างไร เพราะหน่วยความจำหลักในเครื่องคอมพิวเตอร์มีจำนวนที่จำกัด หากกระบวนการต้องการหน่วยความจำที่มากกว่าหน่วยความจำหลัก จะทำอย่างไร

ในความเป็นจริงแล้ว เรายังมีหน่วยความจำสำรอง ที่สามารถเก็บข้อมูลได้อย่างมากมาย มากกว่าหน่วยความจำหลักหลายเท่า ดังนั้น เราสามารถนำหน่วยความจำสำรอง มาเสริมการทำงานของหน่วยความจำหลัก เพื่อเพิ่มพื้นที่ของหน่วยความจำที่กระบวนการต้องการได้อย่างไร้ขีดจำกัด การทำงานของ VM ต้องการเทคนิคในการบริหารงาน



รูปที่ 7.1 ธรรมชาติของหน่วยความจำในระดับต่างๆ

หลักคิดโดยทั่วไป

เมื่อเราเปิดโปรแกรมหลายๆโปรแกรม ก็จะเกิดกระบวนการขึ้นมาหลายตัว แต่ละกระบวนการก็ต้องการพื้นที่ว่าง (Address Space) ของตนเอง ซึ่งจะต้องใช้หน่วยความจำหลัก ยิ่งเปิดโปรแกรมมาก ก็ยิ่งใช้หน่วยความจำหลักมาก ซึ่งเราสังเกตว่า ในขณะใดๆทุกๆกระบวนการไม่ได้พร้อมที่จะทำงานอย่างเต็มที่ (Active) พร้อมๆกัน เพราะบางกระบวนการอาจจะไม่ถูกเรียกใช้โดยผู้ใช้ ดังนั้น เราสามารถย้ายกระบวนการเหล่านั้นไปไว้ที่หน่วยความจำสำรอง (ฮาร์ดดิสก์) ได้ เพื่อให้หน่วยความจำหลักมีพื้นที่ว่างมากที่สุด

เทคนิคของ Demand Paging

ปัจจุบัน ระบบปฏิบัติการส่วนใหญ่ นิยมการใช้การทำงานของ Demand Paging ในการทำงานของ VM เนื่องจากมันเป็นเทคนิคที่ทำงานได้เร็วและดีที่สุด (ในขณะนี้) หลักการทำงานของ Demand Paging มีดังนี้ กระบวนการจะถูกแบ่งออกเป็นหน้าหรือ Page ในแบบ Logical โดยมีขนาดเท่าๆกันเช่น 500B หรือ 1KB เป็นต้น ซึ่งจะใช้การอ้างแอดเดรสในแบบ Logical โดยมี Page Number และ Offset ตามลำดับ ซึ่งเขียนอยู่ในรูปแบบ Page Number : Offset ดังที่แสดงในเรื่องของ Paging ในบทที่ 6

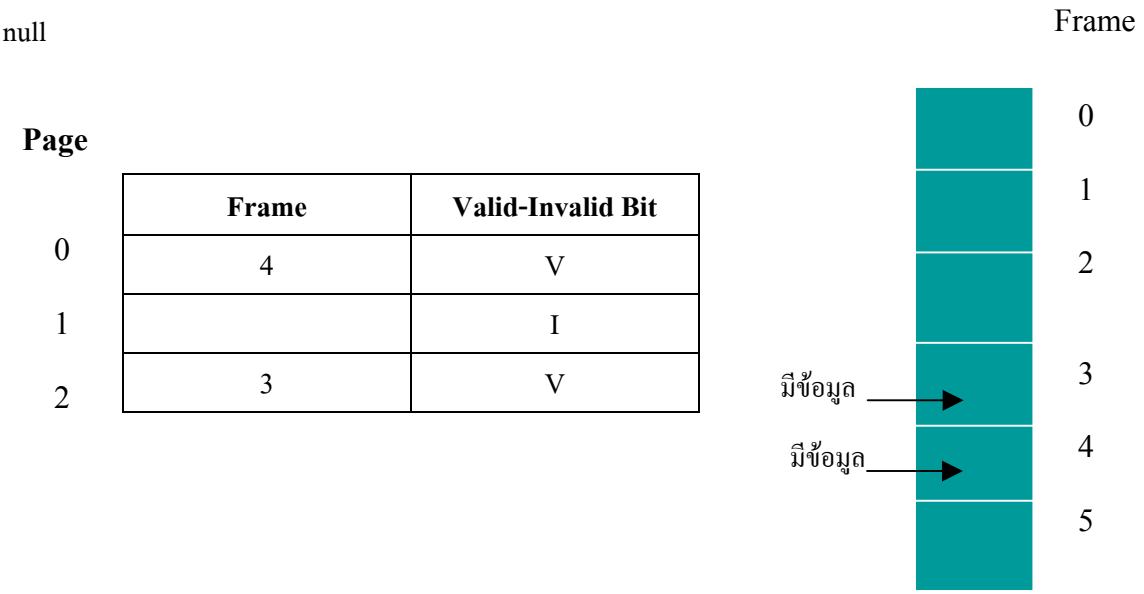
หลักการทำงานของ Demand Paging

หลักการทำงานของ Demand Paging นั้น เป็นหลักการที่ฉลาดมาก โดยมีหลักการง่ายๆว่า “ทุกกระบวนการสามารถทำงานได้ด้วยทรัพยากรเท่าที่ต้องการ” ซึ่งทรัพยากรนั้นก็คือหน่วยความจำหลักนั่นเอง จากเรื่องของกระบวนการ เรารู้ว่าเมื่อโปรแกรมต้องการจะทำงาน มันจะต้องโหลดตัวเองเข้าไปในหน่วยความจำหลัก เพื่อที่จะให้ CPU อ่านเข้าไปทำงาน (Execute) ต่อไป ในอดีตโปรแกรมมีขนาดเล็ก หน่วยความจำเพียงไม่กี่ MB ก็สามารถใช้งานได้ แต่ปัจจุบัน (และอนาคต) หน่วยความจำขนาดนี้ไม่เพียงพอต่อความต้องการอย่างแน่นอน ดังนั้นด้วยวิธีการของหน่วยความจำเสมือน เราก็สามารถใช้หน่วยความจำได้มากเท่าที่ต้องการ เพราะหน่วยความจำเสมือนได้แก่

หน่วยความจำเสมือน = หน่วยความจำหลัก + หน่วยความจำสำรอง

การทำงานของ Demand จะใช้หลักการของ Interrupt โดยจะเรียก Interrupt นี้ว่า Page Fault ซึ่งการเกิดของมันจะเป็นเรื่องปกติในการทำงานแบบ Demand Paging ความหมายของ Page Fault ก็คือ “**Page ของกระบวนการที่กำลังทำงานไม่ได้อยู่ในหน่วยความจำหลัก**” ซึ่งกระบวนการจะไม่สามารถทำงานต่อไปได้ (เพราะกระบวนการจะต้องอยู่ใน RAM เท่านั้น) เมื่อเกิดอินเทอร์รัพท์ Page Fault ขึ้นมาแล้ว OS ก็จะเข้าไปดูว่า Page ไหนที่กระบวนการต้องการ ก็จะโหลด Page นั้นเข้าไปในหน่วยความจำหลัก การตรวจทั้งหมดนี้ OS จะตรวจในตาราง Page (Page Table) ดังรูปที่ 7.2 ซึ่ง v คือ valid หมาย

ถึง Page ข้อมูลนั้นอยู่ในหน่วยความจำเรียบร้อยแล้ว แต่หากไม่อยู่ในหน่วยความจำหลักก็จะมีค่าเป็น null



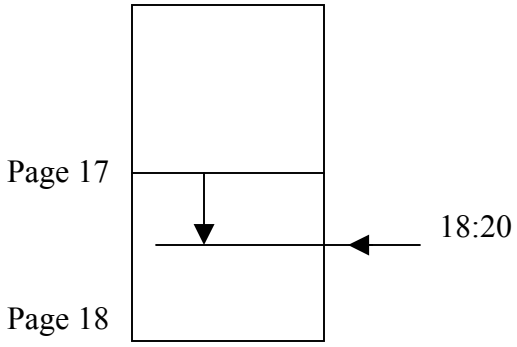
รูปที่ 7.2 แสดงตาราง Page (Page Table)

แอดเดรสเสมือน (Logical Address)

สำหรับหน่วยความจำเสมือน ที่ทำงานในแบบ Demand Paging ในแต่ละเพจนั้น ก็จะมีการใช้หน่วยความจำเสมือน ซึ่งจะประกอบไปด้วยหมายเลขของ Page และ Offset ด้วยการกระทำเช่นนี้ จึงทำให้เราสามารถอ้างถึงแอดเดรสที่เก็บข้อมูลได้อย่างไม่จำกัด ซึ่งในความเป็นจริงแล้วแอดเดรสที่แท้จริงจะรวมทั้งแอดเดรสของหน่วยความจำหลักและหน่วยความจำสำรอง ซึ่งในปัจจุบันฮาร์ดดิสก์ก็มีราคาที่ถูกลงอย่างมาก ทำให้เราใช้ฮาร์ดดิสก์ในความจุที่มากขึ้น หากใช้แอดเดรสแบบเสมือนแล้ว ก็ทำให้เราสามารถรองรับได้อย่างเต็มที่ นอกจากนั้นแล้ว การทำงานของแอดเดรสเสมือนนั้นจะรองรับการทำงานแบบ Demand Paging ได้เป็นอย่างดี เพราะเราสามารถโหลดข้อมูลของเพจต่างๆเข้าหรือออกจากหน่วยความจำได้แบบเป็นพลวัต (Dynamic)

หลักการอ้างแอดเดรสแบบเสมือน

การอ้างแอดเดรสแบบเสมือนนั้น จะใช้ข้อมูล 2 ส่วน ได้แก่ page number และ offset ซึ่งมีลักษณะดังรูปที่ 7.3 หมายเลขของ Page และ Offset จะเริ่มจาก 0 หากเปรียบเทียบแล้ว ก็เหมือนกับการอ่านหนังสือ Page number ก็คือหมายเลขของหน้าของหนังสือ ส่วน Offset number ก็อาจเปรียบได้กับหมายเลขของบรรทัดในแต่ละหน้านั่นเอง



รูปที่ 7.3 แสดงภาพของ Page number และ Offset

ในการที่ระบบปฏิบัติการจะทำการ Paging นั้น ระบบปฏิบัติการจะมองเห็นเป็นแอดเดรสเสมือน ซึ่งมีรูปแบบการเขียนดังนี้

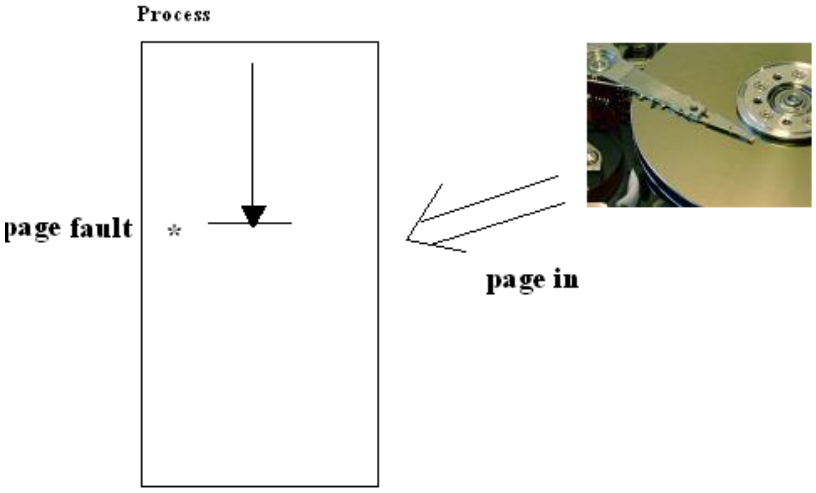
หมายเลขเพจ : หมายเลขออฟเซ็ท

ตัวอย่าง

567C : 2301

หมายถึง เพจที่ 567C (เลขฐาน 16) และหมายเลข offset เป็น 2301 (เลขฐาน 16)

การที่ระบบปฏิบัติการจะทำการ Paging นั้น ก็จะประกอบไปด้วย Page in คือการนำ page จากหน่วยความจำสำรองเข้ามาไว้ในหน่วยความจำหลัก และ Page Out คือการนำข้อมูล page จากหน่วยความจำหลัก กลับไปไว้ที่หน่วยความจำสำรอง ซึ่งจะเห็นว่า การเคลื่อนที่ของข้อมูลเข้าและออกจากหน่วยความจำหลักนั้น จะกระทำในหน่วยของ Page ดังรูปอธิบายด้านล่าง

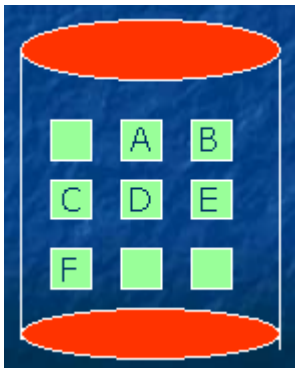


รูปที่ 7.4 การ Page in ของหน่วยความจำเสมือน

เมื่อกระบวนการทำงาน มันจะเริ่มทำงานที่ Page แรก และ Run ไปเรื่อยๆ (ตามลำดับ หรือกระโดดข้ามไปมา) แต่หากว่ามัน Run ไปในส่วนของคำสั่งที่อยู่ใน Page ที่ไม่อยู่ในหน่วยความจำหลัก ก็จะทำให้เกิดการ Interrupt ที่เรียกว่า Page Fault ซึ่งระบบปฏิบัติการก็จะทำการค้นหา Page ที่ต้องการ และโหลดให้ ซึ่งเรียกว่าการ Page in

การทำงานในมุมมองของหน่วยความจำสำรอง

โปรแกรมที่ยังไม่ได้ทำงาน (Run) จะเก็บไว้ในรูปแบบของไฟล์ในหน่วยความจำสำรอง โดยจะแบ่งออกเป็น Page ต่างๆขนาดเท่าๆกัน ดังรูปที่ 7.5 เมื่อโปรแกรมเริ่ม Run จะเริ่มทำงานใน Page ที่ 1 ดังนั้น จะเกิด Page Fault ขึ้นทันที เมื่อเกิด Page Fault แล้ว ระบบปฏิบัติการ ก็จะโหลด Page ที่ต้องการ (page แรก) เข้าไปในหน่วยความจำ และทำงานต่อไป จนกว่าจะจบ page แรก ก็จะเกิด page fault อีก และโหลด page ที่ต้องการอีก เป็นเช่นนี้ไปเรื่อยๆจนกว่าจะจบการทำงาน



รูปที่ 7.5 แสดง Page ต่างๆของโปรแกรมในหน่วยความจำสำรอง

การแทนที่ Page (Page Replacement)

เมื่อมีการ Page in ไปเรื่อยๆ ก็จะทำให้หน่วยความจำมีการใช้งานไปเรื่อยๆจนอาจจะเต็มได้ในที่สุด หากหน่วยความจำหลักเต็มแล้วระบบปฏิบัติการจะทำการหา page ที่จะเป็นเหยื่อ (Victim Page) ที่จะถูกนำออกจากหน่วยความจำหลัก (Page out) และทำให้หน่วยความจำหลักมีพื้นที่ว่างให้ใช้งานได้ วิธีการในการหา Victim Page นั้นเรียกว่า Replacement Algorithm ซึ่งตามทฤษฎีมีอยู่ 3 อัลกอริธึม ได้แก่

1. FIFO
2. Optimal
3. LRU

อัลกอริธึมที่ 1 (First-in First-out)

ความหมายของอัลกอริธึมนี้ได้แก่ มาก่อน-ได้ก่อน ซึ่งมาก่อนได้ก่อนในที่นี้คือ page ไหนเข้ามาในหน่วยความจำก่อน ก็จะต้องถูกเลือกเป็นเหยื่อก่อนนั่นเอง อัลกอริธึมนี้มีหลักการทำงานที่ง่ายที่สุด ดังนั้นการเขียนโปรแกรม (Implementation) ก็ง่ายที่สุดเช่นกัน

ตัวอย่าง

เพื่อให้เข้าใจการทำงานของอัลกอริธึมนี้ เราจะยกตัวอย่างการทำงานของมัน สมมติว่าในหน่วยความจำสามารถบรรจุข้อมูลได้ทั้งหมด 3 page ดังรูปที่ 7.6

รูปที่ 7.6

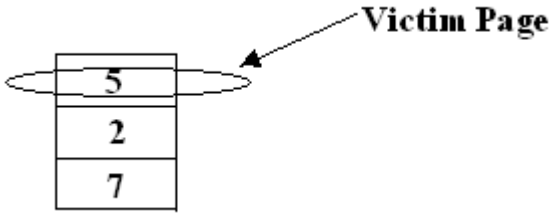
หากว่า page ที่โหลดเข้ามามีลำดับของการเข้ามาดังนี้

page 5, page 2, page 7

ก็จะเข้าไปจับจองพื้นที่ในหน่วยความจำดังรูปที่ 7.7

รูปที่ 7.7

จากรูปที่ 7.7 ก็จะพบว่าหน่วยความจำเต็มแล้ว เมื่อเราโหลดข้อมูลเข้าไป 3 page ดังนั้น หากมี page ที่ 4 เข้ามา ก็จะต้องหา victim page ซึ่งตามอัลกอริธึมของ FIFO แล้ว page ที่เป็นเหยื่อก็คือ page 5 นั่นเอง เพราะเป็น page ที่เข้ามาก่อนอันดับแรก ดังรูปที่ 7.8



รูปที่ 7.8

อัลกอริธึมที่ 2 (Optimal)

การทำงานของอัลกอริธึมนี้ก็คือ มันจะมองไปถึงอนาคตว่า page ใดจะถูกเรียกใช้อีกในอนาคต ก็จะเก็บ page นั้นไว้ และเลือก page ที่ไม่ถูกเรียกใช้ในอนาคตหรือจะถูกเรียกใช้ในอนาคตไกลที่สุด เป็น Victim Page ดังตัวอย่าง

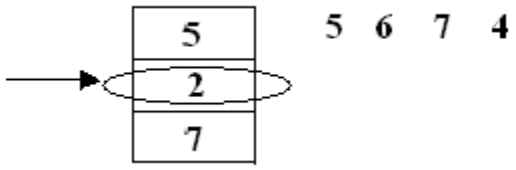
ตัวอย่าง

หากหน่วยความจำเก็บข้อมูลได้ 3 page แล้วมีลำดับการเข้ามาของ page ต่างๆดังนี้

* **

page 5, page 2, page 7, page 5, page 6, page 7, page 4

ตำแหน่งเครื่องหมาย * คือตำแหน่งที่หน่วยความจำเกิดเต็มขึ้นมา แต่ page ต่อไปคือ page 5 แต่ page 5 นั้นอยู่ในหน่วยความจำอยู่แล้ว ดังนั้นจะไม่เกิด page fault และไม่ต้องเลือก Victim Page แต่พอถึงการทำงานของ page 6 (เครื่องหมาย **) จะต้องเลือก Victim Page ซึ่งตาม อัลกอริธึม ก็จะพิจารณาว่า page ใดจะถูกเรียกใช้อีกในอนาคต ซึ่งจากตัวอย่างพบว่า page 7 จะถูกเรียกใช้อีก ดังนั้นจะเลือก page 2 เป็นเหยื่อ ดังรูป 7.9



รูปที่ 7.9

อัลกอริธึมที่ 3 (LRU)

LRU ย่อมาจาก Least Recently Used หมายความว่า “ถูกใช้เป็นปัจจุบันน้อยที่สุด” อัลกอริธึมนี้จะกระทำในสิ่งที่ตรงกันข้ามกับ Optimal คือ ดูจากอดีตจนถึงปัจจุบันว่า page ใดถูกเรียกใช้เป็นปัจจุบันน้อยที่สุด

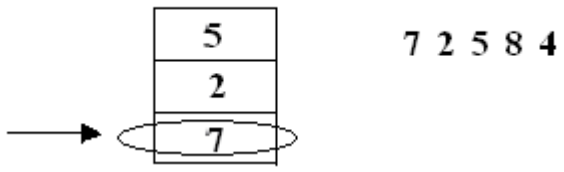
ตัวอย่าง

หากหน่วยความจำเก็บข้อมูลได้ 3 page แล้วมีลำดับการเข้ามาของ page ต่างๆดังนี้

* **

page 5, page 2, page 7, page 7, page 2, page 5, page 8, page 4

ตำแหน่งเครื่องหมาย * คือตำแหน่งที่หน่วยความจำเกิดเต็มขึ้นมา แต่ page ต่อไปคือ page 7 แต่ page 7 นั้นอยู่ในหน่วยความจำอยู่แล้ว และ page ต่อไปคือ page 2 และ page 5 ตามลำดับ ก็อยู่ในหน่วยความจำแล้วเช่นกัน page ต่อไปคือ page 8 ไม่อยู่ในหน่วยความจำ ดังนั้นจะเกิด page fault (เครื่องหมาย **) อัลกอริธึมนี้จะเลือก Victim Page โดยดูว่า page ไหนถูกเรียกเป็นปัจจุบันน้อยที่สุด ซึ่งจากตัวอย่างก็คือ page 7 เพราะมีการเรียกใช้ที่เป็นปัจจุบันน้อยที่สุด ดังรูปที่ 7.10



รูปที่ 7.10

วิเคราะห์อัลกอริธึมทั้ง 3

จากอัลกอริธึมการแทนที่ทั้ง 3 แบบนั้น จะเห็นว่าแต่ละอัลกอริธึมก็มีจุดเด่นและจุดด้อยแตกต่างกันไป ในส่วนนี้จะทำการวิเคราะห์และเปรียบเทียบอัลกอริธึมทั้ง 3 แบบ เพื่อให้เห็นถึงความแตกต่างและสามารถเลือกจุดเด่นที่ต้องการนำไปใช้ได้

เรื่องของความเร็ว

ความเร็วในการทำงานของระบบปฏิบัติการนั้น ขึ้นอยู่กับจำนวนของขั้นตอน (หรือโค๊ดโปรแกรม) ที่ใช้ไปในการทำงาน หากมีขั้นตอนมาก ก็ย่อมทำงานช้าเป็นธรรมดา ในทางตรงกันข้าม หากมีขั้นตอนน้อย ก็จะทำงานได้เร็ว

หากเราวิเคราะห์ดูแล้ว จะพบว่า อัลกอริธึมที่ง่าย มักจะทำงานได้เร็ว เพราะมีขั้นตอนน้อย ดังนั้น อัลกอริธึมที่มีขั้นตอนน้อยที่สุดก็น่าจะเป็นอัลกอริธึมแรกเพราะเก็บข้อมูลแค่ page แรกสุดที่เข้ามาในหน่วยความจำเพียง page เดียวเท่านั้น ส่วนอัลกอริธึมที่ทำงานได้ช้าก็อาจเป็นได้ทั้ง Optimal และ LRU เพราะมีการคำนวณมากทั้งคู่

เรื่องของการเกิด page fault

การทำงานของหน่วยความจำเสมือน หากเกิด page fault บ่อยๆ ก็จะทำให้ระบบทำงานช้าลง เพราะจะต้องทำการคัดลอกข้อมูลจากหน่วยความจำสำรองเข้ามาในหน่วยความจำหลัก เพื่อรองรับการทำงานของกระบวนการ ซึ่งการทำงานของหน่วยความจำสำรอง (เช่น harddisk) มักจะทำงานได้ช้า เมื่อเทียบกับหน่วยความจำหลัก (RAM) อัลกอริธึมที่ดีที่สุดในเรื่องนี้ก็คือ Optimal เพราะเป็นอัลกอริธึมที่รู้ถึงอนาคต ดังนั้นมันจะเก็บ page ที่ต้องการใช้ในอนาคตไว้ในหน่วยความจำ จึงลดปริมาณการเกิด page fault ลงได้

สำหรับอัลกอริธึมที่แย่ที่สุดในเรื่องนี้ก็คืออัลกอริธึม FIFO เพราะใช้เรื่องของเวลาเข้าออกในการประเมินเพียงอย่างเดียว บางครั้ง page ที่เข้ามาเป็น page แรก อาจจะถูกเรียกใช้ในอนาคตอันใกล้อีกก็ได้ ดังนั้น จะมีโอกาสเกิด page fault ได้มาก

แต่อัลกอริธึม Optimal นั้น ไม่สามารถกระทำจริงได้ในทางปฏิบัติ เพราะไม่มีใครสามารถหยั่งรู้อนาคตได้ ดังนั้น อัลกอริธึม LRU จึงเป็นอัลกอริธึมที่น่าจะเหมาะสมที่สุดในทางปฏิบัติ (เร็วและเกิด

page fault น้อย) เนื่องจากว่าอัลกอริธึม LRU นั้น ใช้การ “ทำนายอนาคตจากอดีต” โดยมีความเชื่อว่า page ที่**เพิ่งจะ**ถูกเรียกใช้ไป มีโอกาสจะถูกเรียกใช้อีกในช่วงเวลาที่ใกล้ๆกัน

การเกิดภาวะ Thrashing

ในการทำงานของหน่วยความจำเสมือนด้วยวิธีของ Demand Paging นั้น ใช้การทำงานของ page fault เป็นหลัก (page fault เพื่อโหลดข้อมูล) โดยหลักการแล้ว เราควรลดอัตราการเกิด page fault ให้มีค่าน้อยที่สุด เพราะเป็นสิ่งที่ส่งผลกระทบกับประสิทธิภาพของระบบโดยตรง ซึ่งจากการทดลองพบว่า ระบบสามารถเข้าสู่ภาวะที่มีการเกิด page fault อยู่ตลอดเวลา จนระบบเกิดอาการค้าง (Hang) ซึ่งเรียกภาวะนี้ว่า Thrashing

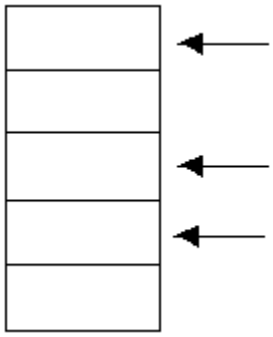
การเกิด Thrashing นั้น ระบบจะทำงานช้าลงอย่างมาก จนอาจจะหยุดการทำงานไปเลยก็ได้ ตัวอย่างของการเกิด Thrashing ได้แก่ สมมติว่ากระบวนการมีการใช้ page ต่างๆดังนี้

page 1, page 2, page 7, page 5, page 1, page 8, page 2

สมมติว่าหน่วยความจำหลักมี 3 frame (บรรจุได้ 3 page) และใช้อัลกอริธึม FIFO ในการทำงาน ดังนั้น เมื่อ page 5 เข้ามาก็จะเกิด page fault ซึ่ง page ที่จะเป็นเหยื่อก็คือ page 1 เพราะเข้ามาก่อน และนำ page 5 ไปแทนที่ page 1 แต่ต่อมาเป็นการทำงานของ page 1 อีก ดังนั้นก็จะเกิด page fault อีกและนำ page1 เข้ามา (ทั้งที่เพิ่งจะถูกเอาออกไปก่อนหน้านี้) แทนที่ page 2 หลังจากนั้น ก็เป็นการทำงานของ page 8 ซึ่งจะไปแทนที่ page 7 จากนั้นก็เป็น page 2 ก็จะเกิด page fault อีก

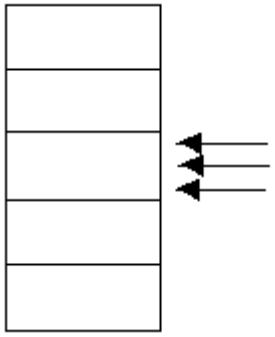
ภาวะการเกิด Thrashing นั้น CPU ไม่สามารถประมวลผลคำสั่งในกระบวนการได้เลย เพราะมัวแต่เสียเวลาไปกับการ page in และ page out ของการเกิด page fault ในปริมาณสูง

สาเหตุหนึ่งของการเกิด Thrashing ก็คือ “การขาดอ้างถึงแบบเฉพาะที่ (Locality of reference)” ของกระบวนการ ความหมายของการอ้างถึงแบบเฉพาะที่ก็คือ การที่กระบวนการทำงานอยู่ในช่วงของโค๊ดคำสั่งที่อยู่ใน page เดิมตลอด ยกตัวอย่างเช่น หากกระบวนการมีพื้นที่จำนวนทั้งหมด 5 KB แบ่งเป็น page มีขนาด page ละ 1 KB ดังนั้นจะได้พื้นที่ทั้งหมดจำนวน 5 page เมื่อกระบวนการทำงานกระโดไปมาระหว่าง page ต่างๆ ดังรูปที่ 7.11 ซึ่งจะทำให้เกิด page fault เป็นจำนวนมาก



รูปที่ 7.11

แต่หากกระบวนการมีการอ้างถึงแบบเฉพาะที่ ก็จะทำให้การทำงานของกระบวนการอยู่ใน page เดิมตลอด ทำให้อัตราการเกิด page fault ลดลง ดังรูปที่ 7.12



รูปที่ 7.12

การแก้ปัญหาการขาดการอ้างถึงเฉพาะที่

จากปัญหาที่พบในหัวข้อที่แล้ว เรื่องของการขาดการอ้างถึงเฉพาะที่ ในการแก้ปัญหานั้นก็มีหลายวิธี วิธีการหนึ่งในการแก้ปัญหาก็คือการใช้ “แบบจำลองชุดทำงาน” (Working Set Model) การใช้แบบจำลองชุดทำงานนั้น มาจากข้อสังเกตที่พบว่า ส่วนใหญ่แล้ว การทำงานของกระบวนการมักจะมีธรรมชาติของตนเอง คือจะมีส่วนการกระจุกตัวของ page ที่ใกล้ๆกัน ในการทำงานของแบบจำลองนี้จะต้องนำ page ทั้งหมดของการทำงานของกระบวนการมาศึกษา เพื่อค้นหาว่ามีส่วนไหนของ ชุดของ page ที่อยู่ในกลุ่ม (Set) เดียวกัน ก็จะโหลดข้อมูลทั้งหมดในชุดนั้นเข้าไปในหน่วยความจำหลักพร้อมๆกัน เพื่อไม่ให้เกิด page fault บ่อยครั้งเกินไป

ตัวอย่าง ชุดของ page ที่กระบวนการหนึ่งจะใช้ในการทำงาน

2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 1 2 3 4 4 4 3 4 4 4 4 1 3 2

สามารถจัดชุดทำงานได้ดังนี้

2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 1 2 3 4 4 4 3 4 4 4 4 1 3 2

ข้อจำกัดของการทำงาน

เนื่องจากว่าการทำงานของแบบจำลองชุดทำงานนั้น ก็จะต้องทราบถึง page ทั้งหมดที่กระบวนการจะต้องใช้ในการทำงานก่อนที่กระบวนการจะทำงานจริง ดังนั้น สิ่งนี้เป็นเรื่องยาก เพราะเราไม่สามารถรู้อนาคตที่แน่นอนของการทำงานได้ ในการใช้งานจริง อาจจะต้องใช้การสุ่มหรือการทำนายหรือการใช้เทคนิคด้านสถิติเข้ามาช่วยก็ได้

Demand Segmentation

หลักการทำงานของ Demand Paging นั้น เป็นสิ่งที่มีความคล่องตัว คือใช้การ page in และ page out เป็นหลักในการทำงาน แต่เมื่อทำงานจริงแล้ว ก็อาจเกิดปัญหาในเรื่องของการขาดการอ้างถึงเฉพาะที่ ดังที่ได้กล่าวถึงมาแล้ว ซึ่งจะทำให้เกิดปัญหาของ Thrashing ตามมา

ในส่วนของ Demand Segmentation นั้น ก็ได้คิดขึ้นมาเพื่อแก้ปัญหานี้ ซึ่งใช้หลักว่า โหลดข้อมูลเป็น Segment แทนที่จะโหลดเป็น page ซึ่งก็ใช้หลักของการเกิด Segment Fault เช่นกัน

เปรียบเทียบ Demand Paging กับ Demand Segmentation

หากการทำงานของกระบวนการใดๆ มีการทำงานที่แบ่งเป็น Segment หรือส่วนของงานอย่างชัดเจน จะทำให้งานต่างๆ จะไปกระจุกตัวอยู่ใน Segment แต่ละอัน ทำให้อัตราการเกิด Segment Fault ลดลง และลดการโหลดข้อมูลลง แต่หากโชคร้าย งานกระโดดไปมาระหว่าง Segment ต่างๆ ก็จะทำให้เกิด Segment Fault สูงขึ้นได้ ข้อเสียอีกอย่างหนึ่งของ Demand Segmentation ก็คือ ใช้พื้นที่ในการทำงานมาก ดังนั้น หน่วยความจำอาจเต็มได้เร็ว แต่การทำงานของ Demand Paging มีการใช้งานกับข้อมูลในขนาดที่เล็กกว่า ทำให้คล่องตัวและรวดเร็วกว่า

สรุป

การใช้งานหน่วยความจำเสมือนทำให้การทำงานของโปรแกรมต่างๆในระบบมีความสามารถในการอ้างถึงหน่วยความจำได้มากขึ้น ทำให้เพิ่มความสามารถในการทำงานของโปรแกรม โปรแกรมสามารถสร้างงาน (Application) ในขนาดที่ใหญ่ขึ้น ซึ่งตามทฤษฎีแล้ว การอ้างถึงหน่วยความจำนั้น ไม่มีข้อจำกัด การทำงานของหน่วยความจำเสมือน อาจใช้หลักการของ Demand Paging ซึ่งใช้การทำงานของ page in และ page out เป็นหลัก โดยให้มีการส่งสัญญาณอินเทอร์รัพท์ที่เรียกว่า page fault ในการโหลดข้อมูลที่ต้องการ ส่วนการทำงานของ Demand Segmentation ก็ใช้หลักการเช่นเดียวกันกับ Demand Paging เพียงแต่โหลดข้อมูลเป็น Segment แทน