บทที่ 2

งานวิจัยและทฤษฎีที่เกี่ยวข้อง

ในบทนี้จะกล่าวถึงทฤษฎีและงานวิจัยที่ใช้ในการศึกษาประกอบการทำวิทยานิพนธ์ฉบับนี้ โดยส่วนแรกเป็นการอธิบายทฤษฎีที่เกี่ยวข้องที่ใช้ในงานวิจัย และส่วนต่อไปเป็นการนำเสนองานวิจัยที่เกี่ยวข้องและตัวอย่างงานวิจัยที่ผ่านมา

2.1 ทฤษฎีที่เกี่ยวข้อง

2.1.1 Graphic Processing Unit (GPU)

Graphic Processing Unit (GPU) นั้นสามารถนำไปใช้งานได้หลายด้านแต่ส่วนมากจะใช้ เพื่อให้การประมวลผลที่เร็วขึ้นมากกว่าปกติที่จากเดิมใช้เพียงซีพียูอย่างเดียว สำหรับการปฏิบัติการด้านคอมพิวเตอร์กราฟิก 2D, 3D รวมทั้ง BitBLT โดยทั่วไปแล้วฮาร์ดแวร์พิเศษที่เรียกว่า Bitter และเป็นตัวดำเนินการในการแสดงผลรูปสี่เหลี่ยม สามเหลี่ยม วงกลม และมุม ซึ่งในปัจจุบันจีพียู สามารถรองรับกราฟิก 3D ในระดับสูง ไม่ว่าจะเป็น Hidefinition Video หรือ เกมส์ต่าง ๆ ที่ล้วนแต่ต้องอาศัยประสิทธิภาพของกราฟิกการ์ดจีพียู

รูปแบบการทำงานของจีพียูเป็นการทำงานแบบ SIMD (Single Instruction Multiple Data) คือการประมวลผลด้วยชุดข้อมูลหลายชุด แต่ทำงานด้วยคำสั่งเดียว ซึ่งเป็นการทำงานแบบคู่ขนาน โดยแต่ละขั้นตอนของการดำเนินงาน อินพุทที่ได้จะมาจากเอาท์พุทของขั้นตอนก่อนหน้า อินพุทดังกล่าวจะเป็นเอาท์พุทที่ถูกส่งไปยังขั้นตอนต่อไป ดังนั้นการคำนวณบนจีพียูจึงคำนวณตามการสั่งของโพรเซสซิ่งสเตท (Processing stage) หรือที่เรียกว่าไปป์ไลน์ (Pipeline) ซึ่งมีการดำเนินการ 3 ขั้นตอนคือ การประมวลผลจุดยอด(Vertex Processing), การยิงแสงสแกนสาดบนหน้าจอ (Rasterization), การประมวลผลแบบแยกออกเป็นชิ้นๆ (Fragment Processing)

2.1.1.1 โครงสร้างการทำงานของจีพียู

การ์ดแสดงผลมีหน้าที่หลักในการรับข้อมูลดิจิตอลมาแปลงเป็นสัญญาณอนาล็อกเพื่อส่งออกไปแสดงผลยังหน้าจอซึ่งสามารถแบ่งการทำงานของการ์ดแสดงผลออกเป็น 2 โหมดคือ โหมดตัวอักษร (Text Mode) และ โหมดกราฟิก (Graphic Mode) การ์ดแสดงผลในปัจจุบันมี

หน้าที่ในการประมวลผลข้อมูลภาพก่อนที่จะส่งไปแสดงผลยังจอมอนิเตอร์ชิพกราฟิกจึงเทียบเท่ากับสมองของการ์ดแสดงผลซึ่งภาพแต่ละเฟรมที่เห็นผ่านจอมอนิเตอร์ต้องผ่านการทำงานของชิพกราฟิกเกือบทั้งหมด โดยทั่วไปสามารถแบ่งชิพกราฟิกได้เป็น 3 ประเภท ดังต่อไปนี้

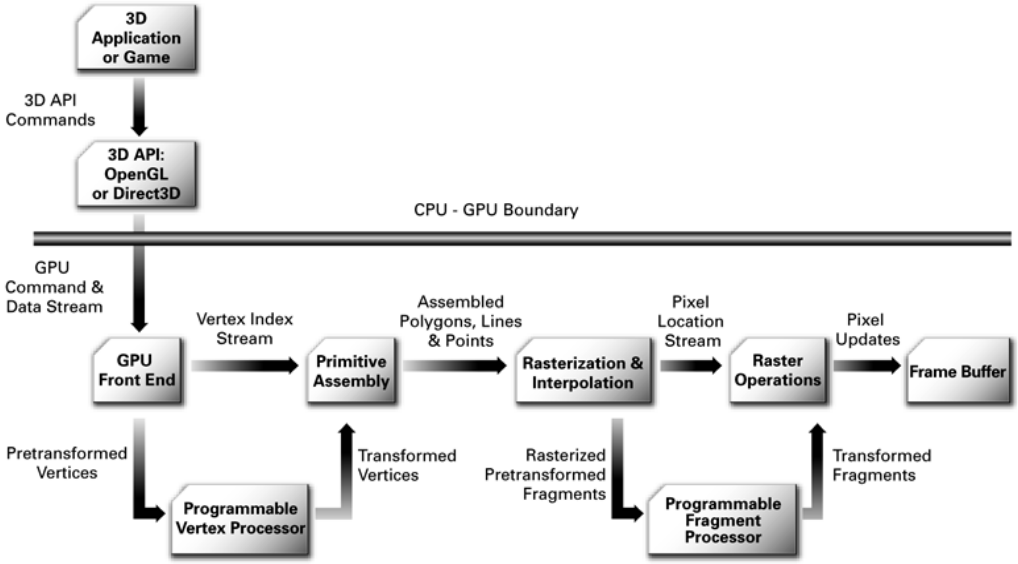
1. เฟรมบัฟเฟอร์ (Frame Buffer) เป็นชิพที่มีการทำงานซับซ้อนน้อยที่สุด เนื่องจากมีหน้าที่เพียงแค่จัดการภาพแต่ละเฟรมที่เก็บไว้ในหน่วยความจำบนการ์ดแล้วส่งข้อมูลไปยังตัวแปลงสัญญาณดิจิตอลให้เป็นอนาล็อก (RAMDAC) เพื่อส่งไปแสดงผลยังหน้าจอมอนิเตอร์ ชิพประเภทนี้ไม่ได้มีหน้าที่ช่วยซีพียูประมวลผลในการสร้างภาพกราฟิกจึงทำให้การประมวลผลด้านกราฟิกอยู่ที่ซีพียูเท่านั้น ส่งผลให้ซีพียูทำงานมากขึ้น

2. Graphics Accelerator เป็นชิพที่ช่วยเร่งความเร็วให้กับการแสดงผล โดยมีหน้าที่หลัก คือรับคำสั่งจากซีพียูมาทำงานเฉพาะด้าน เช่น การสร้างกรอบ การตีเส้น ซึ่งภายในชิพจะมีชุดคำสั่งเก็บไว้ใช้สำหรับงานที่ต้องการแสดงผลบ่อยจากนั้นซีพียูจะทำหน้าที่ตัดสินใจว่าจะให้ชิพ กราฟิกเป็นตัวประมวลผลหรือว่าจะทำการประมวลผลเอง ถึงแม้ว่าชิพตัวนี้จะช่วยลดภาระการทำงานของซีพียูได้ในระดับหนึ่ง แต่ก็มีข้อเสียคือ ชิพยังคงต้องมีการติดต่อกับซีพียูทุกครั้งที่จะทำการแสดงผล ประสิทธิภาพความเร็วของชิพกราฟิกประเภทนี้จึงยังไม่สามารถรองรับงานกราฟิกหนักได้ดีเท่ากับชิพประเภท Graphics Co-Processor

3. Graphics Co-Processor หรือที่เรียกว่าจีพียู (GPU: Graphics Processing Unit) เป็นชิพที่มีความสามารถในการจัดการประมวลผลงานทุกอย่างที่เกี่ยวกับการแสดงผล รวมไปถึงการประมวลผลกราฟิก 3 มิติที่ต้องมีการคำนวณเลขทศนิยมที่มีความละเอียดสูง โดยไม่ต้องพึ่งการทำงานของซีพียูทำให้ซีพียูรับภาระด้านการประมวลผลน้อยลง

2.1.1.2 โครงสร้างการทำงานแบบไปป์ไลน์ของจีพียู

ภาพที่ 2.1 โครงสร้างการทำงานแบบไปป์ไลน์ของจีพียู



ที่มา : “User’s Manual A Developer’s Guide to Programmable Graphics (www.nvidia.com)” โดย nVidia

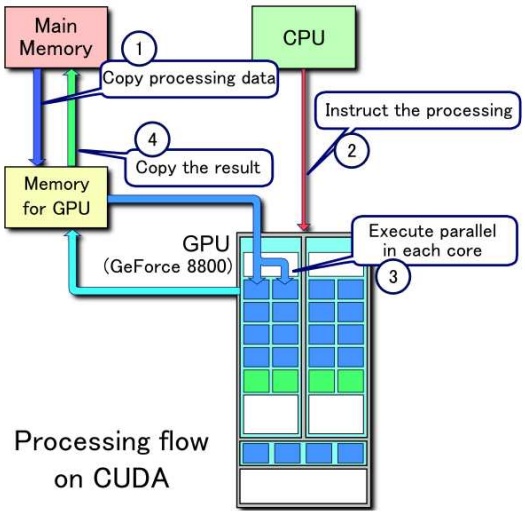
จากภาพที่ 2.1 สามารถอธิบายการทำงานของจีพียูไปป์ไลน์ได้ดังนี้ คือ จีพียูไปป์ไลน์จะทำงานตามลำดับของสเตทไดอะแกรม เริ่มจากส่วนของโปรแกรมเอพีไอ ซึ่งเอพีไอในปัจจุบันจะเป็นพวกโอเพนจีแอล (OpenGL) หรือ ไดเร็คเอ็ก (DirecX) จะทำการส่งข้อมูลมาให้กับหน่วยประมวลผลกราฟิกผ่านทางไดร์เวอร์จากซีพียู ที่ส่งผ่านทางบัสเข้ามาที่จีพียูโดยจีพียูฟรอนท์เอนด์จะรับคำสั่งและข้อมูลจากไดร์เวอร์ผ่านทาง PCI-Express แล้วจึงเข้าสู่กระบวนการของ Vertex Processing ข้อมูลที่เข้ามา เช่น Position Binormal Tangent Textcoord Color และ Psize นอกจากนี้ในกระบวนการของ Vertex Processing อาจมีการรับข้อมูลจากเท็กซ์เจอร์เข้ามาประมวลผลเชดเดอร์ด้วย เมื่อทำการประมวลผลเสร็จจะได้ตำแหน่งและขนาด 2 มิติ จากนั้นจะส่งต่อไปที่ Primitive Assembly ทำการตรวจสอบจุดและเส้นเหลี่ยม อีกทั้งเชื่อมโยงจุดต่างๆ เข้าด้วยกันจนเป็นรูปสามเหลี่ยมจากนั้นขบวนการ การทำให้เป็นจุดภาพ (Rasterization) จะทำการหาความลึก ความสูงเพื่อที่จะคำนวณภาพ 3 มิติออกมา แล้วจึงเข้าสู่กระบวนการ Fragment Processing ซึ่งสามารถโปรแกรมในส่วนนี้ได้ และอาจมีการรับข้อมูลเท็กซ์เจอร์เข้ามา เพื่อคำนวณเชดเดอร์ทำให้ได้ข้อมูลของสี ความลึก และความสูงของภาพ การตรวจสอบเฟรมบัฟเฟอร์นั้น ถ้ามี

ค่าน้อยกว่าแสดงว่าภาพนั้นถูกทับซ้อนอยู่ ด้านหลังขบวนการนี้จะทำการเบลนดิ้งเพื่อหาสีของเฟรมบัฟเฟอร์นั้น

2.1.2 Compute Unified Device Architecture (CUDA)

คูด้าเป็นสถาปัตยกรรมการประมวลผลแบบขนานบนจีพียูที่ถูกพัฒนาขึ้นโดยเอ็นวีเดีย(Nvidia) คูด้ามีความสามารถในการสั่งงานให้จีพียูประมวลผลทั้งในด้านกราฟิกและความสามารถในด้านประผลการคำนวณ โดยมีโครงสร้างการทำงานดังภาพที่ 2.2 และสามารถอธิบายขั้นตอนการทำงานได้ดังนี้

ภาพที่ 2.2 โครงสร้างการทำงานของคูด้า



ที่มา : “en.wikipedia.org/wiki/CUDA” โดย Wikipedia

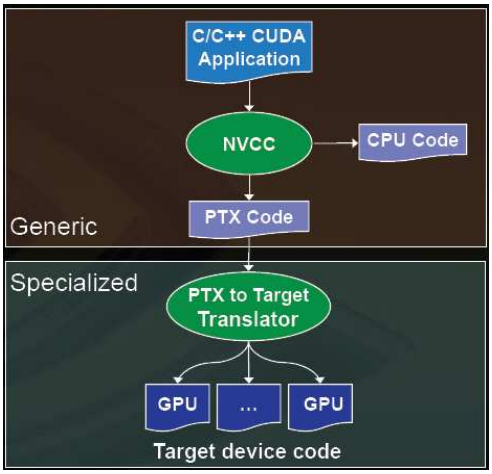
1. เริ่มต้นจากข้อมูลที่อยู่ในหน่วยความจำหลักจะถูกคัดลอกและส่งไปยังหน่วยความจำของจีพียู
2. หลังจากนั้นซีพียูจะส่งคำสั่งสำหรับสั่งงานให้จีพียูประมวลผลไปยังจีพียู
3. เมื่อจีพียูได้รับทั้งข้อมูลและคำสั่งที่ถูกส่งมา จีพียูจะจัดตารางการทำงานและประมวลผลบนจีพียูโดยแบ่งการทำงานไปตามแต่ละแกนหลัก (Core) ที่ยังไม่ได้ถูกเรียกใช้งานซึ่งการประมวลผลการทำงานนี้จะเป็นการประมวลแบบขนาน

4. เมื่อจีพียูประมวลผลจนได้ผลลัพธ์ ผลลัพธ์ดังกล่าวจะถูกคัดลอกกลับไปยังซีพียูเพื่อนำไปใช้งานต่อไป

สถาปัตยกรรมของคูด้าโปรแกรมประกอบด้วย โฮสโปรเซสเซอร์ (Host processor) หน่วยความจำของโฮส (Host memory) และการ์ดจอแสดงผลที่รองรับการทำงานของคูด้าโปรแกรมซึ่งเป็นการ์ดจอของค่ายเอ็นวีเดีย โดยจีพียูที่รองรับการทำงานของคูด้าโปรแกรมจะมีลักษณะการทำงานแบบไปป์ไลน์ โปรเซสเซอร์แรกที่สนับสนุนการทำงานของคูด้าได้แก่ GeForce 8800 ซึ่งเป็นตัวที่ถูกสืบทอดมากจาก GeForce และ Quadro และในปัจจุบันตระกูล Tesla ทั้งหมดก็สนับสนุนสถาปัตยกรรมของคูด้าโปรแกรม ซึ่งคูด้าที่ประมวลผลการทำงานในจีพียูสามารถแบ่งเทรด (Thread) การทำงานเพื่อปฏิบัติงานแบบขนานพร้อมกันได้ครั้งละเป็นพันเทรด และมีลักษณะการทำงานแบบเอสไอเอ็มดี(SIMD) ด้วยหน่วยประมวลผลการคำนวณที่มีเป็นจำนวนมากและฮาร์ดแวร์แบบมัลติเทรดดิ้ง

โปรแกรมภาษาคูด้านั้นถูกพัฒนาขึ้น ซึ่งถูกต่อขยายมาจากโปรแกรมภาษาซีโดยมีวิธีการทำงานในขั้นตอนการประมวลผลแยกกัน คือกรณีฟังก์ชั่นการทำงานถูกเรียกใช้สำหรับสั่งงานในส่วนที่เป็นจีพียูตัวแปรและข้อมูลต่าง ๆ จะถูกนำไปประมวลผลบนจีพียูและถ้าฟังก์ชั่นที่เรียกใช้เป็นคำสั่งสำหรับงานที่อยู่บนซีพียูตัวแปรและข้อมูลต่าง ๆ จะถูกประมวลผลอยู่บนซีพียูเท่านั้น ในการเรียกคูด้าเพื่อใช้งานนั้นจำเป็นจะต้องเรียกผ่านเอพีไอไลบรารีของคูด้าเพื่อไลบรารีดังกล่าวจะทำหน้าที่เข้าถึงอุปกรณ์จีพียูได้โดยตรง โดยมีขั้นตอนการคอมไพล์ คือเมื่อตัวคูด้าคอมไพเลอร์ เอ็นวีซีซี (nvcc) ทำหน้าที่คอมไพล์ไฟล์ที่เป็นคูด้าไฟล์ เอ็นวีซีซีจะประมวลผลแยกโค้ดของไฟล์ที่นำมาคอมไพล์ออกเป็น 2 ส่วน ดังแสดงในภาพที่ 2.3 คือ โค้ดส่วนที่ประมวลผลบนซีพียูกับโค้ดส่วนที่ประมวลผลบนจีพียูในรูปแบบของไบนารีไฟล์ เรียกว่าคูบิน (CUBIN) ไฟล์ สำหรับการเรียกคูด้าเพื่อใช้งานนั้นจำเป็นจะต้องเรียกผ่านเอพีไอไลบรารีของคูด้า โดยเอพีไอถูกแบ่งเป็น 2 ระดับคือ ระดับต่ำถูกเรียกว่า CUDA driver API และระดับสูงเรียกว่า CUDA runtime API เอพีไอดังกล่าวจะทำหน้าที่ติดต่อสื่อสารและเข้าถึงอุปกรณ์จีพียูได้โดยตรง

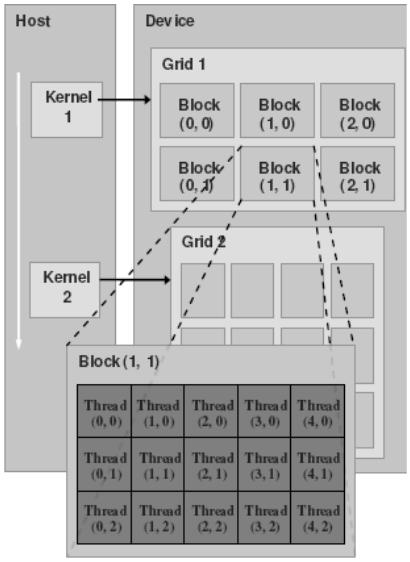
ภาพที่ 2.3 Compiling CUDA



ที่มา : “High Performance Computing with CUDA” โดย Patrick LeGresley, 2008, iCME Colloquium, น.22

การทำงานของคูด้าสำหรับโปรแกรมเมอร์นั้น คูด้าได้ออกแบบให้มีลักษณะการทำงานแบบขนาน โดยแบ่งการทำงานในส่วนที่สั่งงานจีพียูออกเป็นบล็อกและเทรดจากการสั่งงานของโฮส (ซีพียู) ผ่านเคอร์แนล (ฟังก์ชั่น) โดยเมื่อสั่งงานไปยังจีพียูในส่วนของจีพียูจะแบ่งการทำงานออกเป็น กริด บล็อก และเทรด โดย 1 กริดจะประกอบไปด้วยหลายบล็อกและใน 1 บล็อกจะประกอบไปด้วยหลายเทรด ซึ่งกลุ่มของเทรดที่อยู่ในบล็อกจะถูกประมวลผลอยู่บนมัลติโพรเซสเซอร์ นอกจากนั้นมัลติบล็อกสามารถถูกสั่งให้ประมวลผลทำงานบนโพรเซสเซอร์เดียวพร้อม ๆ กันได้ในเวลาเดียวกัน ดังแสดงในภาพที่ 2.4

ภาพที่ 2.4 โครงสร้างการแบ่งบล็อกและเทรดในคูด้า



ที่มา : “NVIDIA CUDA Programming Guide Version 2.3” โดย NVIDIA Corporation, developer.nvidia.com, January 7, 2010

2.1.3 **เวอร์ชวลแมชชีน** (Virtual Machine)

ระบบเวอร์ชวลแมชชีนช่วยให้ฮาร์ดแวร์แพลตฟอร์มของโฮสเครื่องหนึ่งสามารถสนับสนุนเกสโอเอสหลายระบบได้ในเวลาเดียวกันด้วยเทคโนโลยีเวอร์ชวลแมชชีน ผู้ใช้สามารถรันระบบปฏิบัติการที่แตกต่างกันได้บนฮาร์ดแวร์เดียวกัน ความสามารถที่สำคัญของเทคโนโลยีซิสเต็มส์เวอร์ชวลแมชชีนคือการแยกออกจากกัน (Isolation) ของระบบต่าง ๆ ที่รันอยู่ในเวลาเดียวกันบนฮาร์ดแวร์แพลตฟอร์มเดียวกัน นั่นคือถ้าเกสโอเอสระบบหนึ่งเกิดความผิดพลาดขึ้น ซอฟต์แวร์ที่รันอยู่บนเกสระบบอื่นจะไม่ได้รับผลกระทบไปด้วย โดยหลักในระบบเวอร์ชวลแมชชีน VMM จะแบ่งทรัพยากรฮาร์ดแวร์ระหว่างเกสโอเอสต่าง ๆ เช่น ดิสก์ เวอร์ชวลไลเซชั่น โดย VMM จะมีการใช้งานและจัดการทรัพยากรฮาร์ดแวร์ทั้งหมด เกสโอเอสและแอพพลิเคชันโปรเซสจะถูกจัดการภายใต้การควบคุมของ เวอร์ชวลแมชชีนมอนิเตอร์ (VMM : Virtual Machine Monitor) ซึ่งเป็นซอฟต์แวร์เลเยอร์ที่แบ่งฮาร์ดแวร์แพลตฟอร์มออกเป็นเวอร์ชวลแมชชีนหลายเครื่อง เมื่อเกสโอเอสทำคำสั่งพิเศษของระบบหรือทำงานที่ติดต่อกับทรัพยากรฮาร์ดแวร์โดยตรง VMM จะจับการทำงานนั้น ตรวจสอบความถูกต้อง และทำงานนั้นแทนเกสโดยที่ซอฟต์แวร์ของเกสไม่รู้เกี่ยวกับการ

ทำงานนี้ (Smith & Ravi, 2005, p.36) VMs ถูกสร้างขึ้นจากองค์ประกอบที่แตกต่างกัน ดังนี้ (Ruest & Ruest, 2009, pp.30-32)

- คอนฟิกไฟล์ (Configuration File) คือ ไฟล์ที่ประกอบด้วยข้อมูลเกี่ยวกับการตั้งค่าต่าง ๆสำหรับเวอร์ชวลแมชชีน ได้แก่ ขนาด RAM จำนวนโปรเซสเซอร์ จำนวนและประเภทของเน็ตเวิร์คอินเตอร์เฟสการ์ด (NICs) จำนวนและประเภทของเวอร์ชวลดิสก์ โดยแต่ละครั้งที่สร้างเวอร์ชวลแมชชีนเครื่องใหม่คอนฟิกไฟล์ของเวอร์ชวลแมชชีนเครื่องนั้นจะถูกสร้างขึ้น ซึ่งไฟล์นี้จะบอกเวอร์ชวลไลเซชั่นซอฟต์แวร์ ว่าจะจัดสรรทรัพยากรจริงจากโฮสให้กับเวอร์ชวลแมชชีนได้อย่างไร โดยการระบุตำแหน่งที่ฮาร์ดดิสก์ไฟล์อยู่ ขนาด RAM ที่จะใช้วิธีการโต้ตอบกับเน็ตเวิร์คอะแด็ปเตอร์การ์ดและโปรเซสเซอร์ตัวใดบ้างที่ต้องการใช้งาน

- ฮาร์ดดิสก์ไฟล์ คือไฟล์ที่ประกอบด้วยข้อมูลที่โดยปกติจะมีอยู่ในฮาร์ดดิสก์จริงแต่ละครั้งที่สร้างเวอร์ชวลแมชชีน เวอร์ชวลไลเซชันซอฟต์แวร์จะสร้างเวอร์ชวลฮาร์ดดิสก์ขึ้นมา นั่นคือไฟล์ที่จะทำงานเสมือนกับดิสก์ที่มีเซ็กเตอร์ทั่วไป เมื่อติดตั้งระบบปฏิบัติการบนเวอร์ชวลแมชชีนฮาร์ดดิสก์ไฟล์ จะถูกใส่เข้าไปในไฟล์นี้และเหมือนกับระบบจริง คือแต่ละเวอร์ชวลแมชชีนสามารถมีดิสก์ไฟล์ได้หลายไฟล์ เนื่องจากมีจำลองฮาร์ดดิสก์ขึ้นมา ไฟล์นี้จึงสำคัญในด้านขนาด โดยระบบสามารถเริ่มต้นด้วยไฟล์ขนาดเล็ก และค่อย ๆ เพิ่มขนาดขึ้นเมื่อเนื้อหาใหม่ถูกใส่เข้าไปในเวอร์ชวลแมชชีน

- ไฟล์สถานะของเวอร์ชวลแมชชีน เช่นเดียวกับเครื่องจริงเวอร์ชวลแมชชีนสนับสนุนโหมดปฏิบัติการที่คล้ายกับสแตนบายหรือไฮเบอร์เนชั่นในแง่ของเวอร์ชวลไลเซชั่นหมายถึงการหยุดชั่วคราวหรือค้างการทำงานไว้เหมือนกับการบันทึกสถานะของเครื่องเมื่อเครื่องถูกค้างการทำงานไว้ชั่วคราว สถานะที่ถูกหยุดของมันจะถูกบันทึกลงไปในไฟล์ เนื่องจากมีเพียงสถานะของเครื่องเท่านั้น โดยปกติไฟล์นี้จึงเล็กกว่าฮาร์ดดิสก์ไฟล์

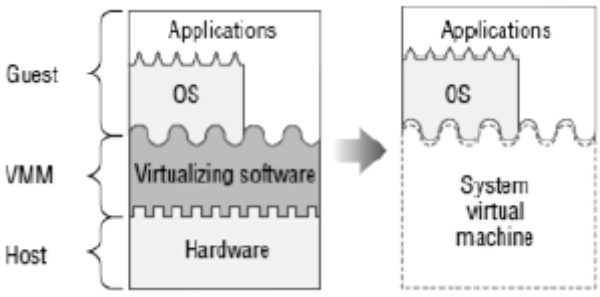
- ไฟล์อื่นๆ คือไฟล์ที่ประกอบด้วยล็อกและข้อมูลที่เกี่ยวกับเวอร์ชวลแมชชีนอื่นๆ

สถาปัตยกรรมของเวอร์ชวลแมชชีน

จากมุมมองของโปรเซสที่เอ็กซิคิวต์ยูเซอร์โปรแกรม แมชชีนจะประกอบไปด้วยพื้นที่แอดเดรสของหน่วยความจำที่จัดไว้ให้กับโปรเซสพร้อมกับคำสั่งระดับยูเซอร์และรีจิสเตอร์ที่ทำให้สามารถเอ็กซิคิวต์โค้ดที่เป็นของโปรเซสได้ ไอโอของเครื่องจะมองเห็นได้ โดยผ่านทางระบบปฏิบัติการเท่านั้น และวิธีการเดียวที่โปรเซสสามารถโต้ตอบกับระบบไอโอคือผ่านทางซิสเต็มส์คอล จากมุมมองของระบบปฏิบัติการและแอพพลิเคชันของตนเอง ระบบทั้งระบบจะรันบนเครื่องจริง ซึ่ง

ระบบคือสภาพแวดล้อมของการเอ็กซิคิวต์ที่สามารถสนับสนุนโปรเซสจำนวนมากได้ในเวลาเดียวกัน โปรเซสเหล่านี้จะใช้ไฟล์ซิสเต็มส์และทรัพยากรไอโออื่น ๆ ร่วมกัน โดยระบบจะจัดสรรหน่วยความจำจริงและทรัพยากรไอโอให้กับโปรเซสต่าง ๆ และยอมให้โปรเซสสามารถโต้ตอบกับทรัพยากรของตนเองได้ เนื่องจากมีมุมมองของโปรเซสและซิสเต็มส์ที่มีต่อ “แมชชีน” เวอร์ชวลแมชชีนจึงมีสองแบบ คือแบบโปรเซสและซิสเต็มส์ โปรเซสเวอร์ชวลแมชชีน (Process VM) คือเวอร์ชวลแพลตฟอร์มที่เอ็กซิคิวต์โปรเซสใดโปรเซสหนึ่ง โดย VM ประเภทนี้มีอยู่เพื่อช่วยเหลือโปรเซสเท่านั้น โดยจะถูกสร้างขึ้นเมื่อโปรเซสถูกสร้างขึ้นและจะจบการทำงานเมื่อโปรเซสจบการทำงาน (ตัวอย่างเช่น Java Virtual Machine) ในทางตรงข้ามระบบเวอร์ชวลแมชชีน (System VM) จะสร้างสภาพแวดล้อมของระบบที่สมบูรณ์และคงอยู่ตลอด ซึ่งสนับสนุนระบบปฏิบัติการพร้อมกับยูเซอร์โปรเซสจำนวนมากของตนเอง และช่วยให้เกสโอเอส (Guest Operating System) สามารถใช้งานทรัพยากรที่เป็นเวอร์ชวลฮาร์ดแวร์ซึ่งรวมทั้งเน็ตเวิร์คไอโอและอาจรวมถึงกราฟิกยูเซอร์อินเตอร์เฟสพร้อมกับโปรเซสเซอร์ และหน่วยความจำด้วยโปรเซสหรือซิสเต็มส์ที่รันบน VM คือเกส (Guest) และแพลตฟอร์มจริงที่สนับสนุน VM คือโฮส (Host) ซอฟต์แวร์เวอร์ชวลไลซ์ที่อิมพลีเมนต์โปรเซสเวอร์ชวลแมชชีนมักตั้งชื่อว่ารันไทม์ ซึ่งมาจากรันไทม์ซอฟต์แวร์ (Runtime Software) ส่วนซอฟต์แวร์เวอร์ชวลไลซ์ในซิสเต็มส์เวอร์ชวลแมชชีนโดยทั่วไปหมายถึงเวอร์ชวลแมชชีนมอนิเตอร์ (Virtual Machine Monitor หรือ VMM)

ภาพที่ 2.5 การแปลง ISA ของซอฟต์แวร์เวอร์ชวลไลซ์ในซิสเต็มส์เวอร์ชวลแมชชีน



ที่มา: “The architecture of virtual machines”
โดย Smith, J. E., & Ravi, N., 2005,*Computer, 38*(5), น. 34.

จากภาพที่ 2.5 ในซิสเต็มส์เวอร์ชวลแมชชีนซอฟต์แวร์ที่ทำเวอร์ชวลไลซ์จะอยู่ระหว่างฮาร์ดแวร์ของโฮสและซอฟต์แวร์ของเกส VMM จะจำลอง ISA ของฮาร์ดแวร์เพื่อที่ซอฟต์แวร์ของ

เกสจะได้สามารถเอ็กซิคิวต์ ISA ที่แตกต่างจากที่อิมพลีเมนต์อยู่บนโฮสได้ อย่างไรก็ตาม ในหลายแอพพลิเคชันซิสเต็มส์เวอร์ชวลแมชชีน VMM ไม่ได้ทำการจำลองคำสั่งแต่หน้าที่หลัก คือจัดสรรทรัพยากรฮาร์ดแวร์ที่ถูกเวอร์ชวลไลซ์ให้กับเกส (Smith & Ravi, 2005, p. 34) จากมุมมองของผู้ใช้ซิสเต็มส์เวอร์ชวลแมชชีนส่วนใหญ่จะทำงานได้เหมือนกัน แต่แตกต่างกันในรายละเอียดการอิมพลีเมนต์ วิธีการแบบเวอร์ชวลแมชชีนมอนิเตอร์ซึ่งเป็นการวาง VMM ไว้บนฮาร์ดแวร์จริงและให้เวอร์ชวลแมชชีนอยู่ด้านบน โดย VMM จะรันในโหมดที่มีสิทธิสูงสุด ขณะที่เกสซิสเต็มส์ทั้งหมดจะรันด้วยสิทธิที่ต่ำกว่าเพื่อให้ VMM สามารถแทรกแซงและจำลองการกระทำของเกสโอเอสทั้งหมดที่จะใช้งานหรือจัดการทรัพยากรฮาร์ดแวร์ได้การอิมพลีเมนต์ซิสเต็มส์เวอร์ชวลแมชชีนแบบโฮสเวอร์ชวลแมชชีน (Host Virtual Machine) จะสร้างซอฟต์แวร์เวอร์ชวลไลซ์ด้านบนโฮสโอเอส ข้อดีของโฮสเวอร์ชวลแมชชีน คือผู้ใช้จะติดตั้งเหมือนกับเป็นแอพพลิเคชันโปรแกรมปกติ นอกจากนั้นซอฟต์แวร์เวอร์ชวลไลซ์ยังสามารถอาศัยโฮสโอเอสให้จัดดีไวซ์ไดรเวอร์ และเซอร์วิสระดับล่างอื่น ๆ ให้ แทนที่จะอาศัย VMM ในซิสเต็มส์เวอร์ชวลแมชชีนปกติซิสเต็มส์ซอฟต์แวร์ของทั้งโฮสและเกสรวมทั้งแอพพลิเคชันซอฟต์แวร์จะใช้ ISA เดียวกับฮาร์ดแวร์จริง อย่างไรก็ตามในบางสถานการณ์ระบบโฮสและเกสไม่ได้มี ISA เดียวกัน ดังนั้นเวอร์ชวลแมชชีนแบบทั้งระบบสามารถแก้ปัญหานี้ได้โดยการเวอร์ชวลไลซ์ซอฟต์แวร์ทั้งหมดซึ่งรวมทั้งระบบปฏิบัติการและแอพพลิเคชันด้วย เนื่องจาก ISAแตกต่างกันเวอร์ชวลแมชชีนจึงต้องจำลองโค้ดทั้งแอพพลิเคชัน และระบบปฏิบัติการเวอร์ชวลแมชชีนซอฟต์แวร์จะเอ็กซิคิวต์เหมือนกับเป็นแอพพลิเคชันโปรแกรมที่ได้รับการสนับสนุน โดยโฮสโอเอสและไม่ใช้โอเปอเรชั่น ISA ของระบบเมื่อโฮสแพลตฟอร์มเป็นมัลติโปรเซสเซอร์ขนาดใหญ่ที่ใช้งานหน่วยความจำร่วมกันจุดประสงค์ที่สำคัญคือการแบ่งระบบขนาดใหญ่ออกเป็นระบบมัลติโปรเซสเซอร์ที่เล็กลงหลาย ๆ ระบบโดยการกระจายทรัพยากรฮาร์ดแวร์ของระบบใหญ่ ด้วยการแบ่งแบบฟิสิคัลทรัพยากรจริงที่เวอร์ชวลซิสเต็มส์หนึ่งใช้จะแยกออกจากทรัพยากรที่ถูกใช้ โดยเวอร์ชวลซิสเต็มส์อื่น ๆ การแบ่งแบบฟิสิคัลนี้ทำให้มีระดับของการแยกออกจากกันสูงดังนั้นปัญหาของซอฟต์แวร์ หรือข้อผิดพลาดของฮาร์ดแวร์บนพื้นที่แบ่งส่วนหนึ่งจะไม่มีผลกระทบต่อโปรแกรมในพื้นที่แบ่งส่วนอื่น ส่วนการแบ่งแบบโลจิคอลทรัพยากรฮาร์ดแวร์จริงจะถูกแบ่งเวลาระหว่างพื้นที่แบ่งต่าง ๆ ซึ่งทำให้การใช้งานทรัพยากรระบบดีขึ้น แต่จะสูญเสียประโยชน์บางอย่างของการแยกออกจากกันของฮาร์ดแวร์ไป โดยปกติเทคนิคการแบ่งพื้นที่ทั้งสองแบบใช้ซอฟต์แวร์ หรือเฟิร์มแวร์พิเศษที่มีการดัดแปลงฮาร์ดแวร์เฉพาะกับพื้นที่แบ่งที่ต้องการการใช้งานฟังก์ชั่นและการรันบนสถาปัตยกรรมที่แตกต่างกันได้เป็นเป้าหมายของระบบเวอร์ชวลแมชชีนส่วนใหญ่ที่ถูกอิมพลีเมนต์บนฮาร์ดแวร์ที่ถูกพัฒนาขึ้นสำหรับ ISA มาตรฐาน ในทางตรงข้ามเวอร์ชวลแมชชีนแบบโคดีไซน์

(Codesigned Virtual Machine) อิมพลีเมนต์ ISA ใหม่ที่มีเป้าหมายที่การปรับปรุงประสิทธิภาพและการใช้พลังงานให้ดีขึ้น โดยอาจมีการสร้าง ISA ของโฮสใหม่ทั้งหมดหรือเพิ่มขยาย ISA ที่มีอยู่ก็ได้ เวอร์ชวลแมชชีนแบบนี้ไม่มีแอพพลิเคชัน ISA จริง แต่ VMM จะกลายเป็นส่วนหนึ่งของการอิมพลีเมนต์ฮาร์ดแวร์มีหน้าที่อย่างเดียว คือจำลอง ISA ของเกส โดย VMM จะอยู่ในพื้นที่ของหน่วยความจำที่ถูกซ่อนจากซอฟต์แวร์ทั้งหมด ซึ่งรวมทั้งตัวแปลงไบนารีที่เปลี่ยนคำสั่งเกสให้เป็นคำสั่ง ISA ของโฮสและแคชคำสั่งเหล่านั้นเก็บไว้ในพื้นที่ของหน่วยความจำที่ถูกซ่อนไว้ (Smith & Ravi, 2005, pp. 36-37)

2.1.4 **อัลกอริทึมของนายธนาคาร** (Banker's Algorithm) (Abraham & Peter Baer)

อัลกอริทึมของนายธนาคารเป็นอัลกอริทึมใช้สำหรับหลีกเลี่ยงปัญหาการติดตาย (Deadlock Avoidance) โดยการหลีกเลี่ยงปัญหาการติดตายนี้จะแตกต่างกับการป้องกันการติดตาย (Deadlock Prevention) คือ การป้องกันการติดตายเป็นการป้องกันเพื่อไม่ให้เกิดการติดตายขึ้น โดยการสร้างข้อกำหนดในการร้องขอทรัพยากร เพื่อให้แน่ใจว่าเงื่อนไขข้อใดข้อหนึ่ง จะไม่เกิดขึ้นอย่างแน่นอน ซึ่งเงื่อนไขดังกล่าวประกอบไปด้วย

1. ห้ามใช้ทรัพยากรร่วมกัน (Mutual Exclusion) เงื่อนไขในข้อนี้ คือ การที่ระบบไม่อนุญาตให้มีการใช้ทรัพยากรร่วมกัน เช่น เครื่องพิมพ์จะไม่สามารถให้กระบวนการ (Process) หลาย ๆ กระบวนการใช้พร้อม ๆ กันได้

2. การถือครองแล้วรอคอย (Hold and Wait) คือ การที่จะไม่ให้เกิดการถือครองแล้วรอคอยขึ้นในระบบ โดยจะต้องกำหนดว่า เมื่อกระบวนการหนึ่งจะร้องขอทรัพยากร กระบวนการนั้นจะต้องไม่ได้ถือครองทรัพยากรใดๆ อยู่ในขณะนั้น ซึ่งอาจทำได้ 2 วิธีการ คือ

(1) ให้กระบวนการร้องขอทรัพยากรที่ต้องการใช้ทั้งหมด (ตลอดการทำงาน) ก่อนที่จะเริ่มต้นการทำงาน เราอาจดำเนินการตามวิธีได้ โดยการกำหนดให้การร้องขอทรัพยากรเป็นคำสั่งเรียกระบบ (System call) ที่ต้องทำก่อนการทำงานใด ๆ ของกระบวนการเสมอ

(2) ยอมให้กระบวนการร้องขอทรัพยากรได้ ก็ต่อเมื่อกระบวนการนั้นมิได้ถือครองทรัพยากรใดไว้เลย ตัวอย่างเช่น กระบวนการหนึ่งอาจร้องขอทรัพยากรบางส่วนและใช้ทรัพยากรนั้นไปก่อน และเมื่อกระบวนการนั้นต้องการทรัพยากรเพิ่มอีก กระบวนการนั้นก็จะต้องคืนทรัพยากรที่ถือครองอยู่กลับสู่ระบบเสียก่อน จึงจะร้องขอใหม่ได้

วิธีการแรกมีข้อเสีย คือ การใช้ทรัพยากรจะมีประสิทธิผลต่ำมาก เพราะกระบวนการจำเป็นต้องร้องขอและถือครองทรัพยากรไว้ทั้งหมดตลอดช่วงเวลาการทำงาน ทั้ง ๆ ที่การใช้ทรัพยากรแต่ละตัวอาจเป็นเพียงช่วงเวลาสั้น ๆ ก็ตาม

นอกจากนั้นอาจมีปัญหา Starvation อีกด้วย โดยถ้ามีบางกระบวนการต้องการใช้ทรัพยากรหลาย ๆ ตัว อาจต้องรอคอยอย่างไม่มีที่สิ้นสุด เพราะทรัพยากรตัวหนึ่งในจำนวนที่ต้องการอาจมีกระบวนการอื่นใช้อยู่ส่วนวิธีการหลังก็จะมีข้อเสีย คือ ต้องคืนทรัพยากรที่ถือครองอยู่ เพื่อที่จะร้องขอกลับมาใหม่อีกร่วมกับทรัพยากรตัวใหม่ ทำให้เสียเวลาโดยเปล่าประโยชน์

3. ห้ามแทรกกลางคัน (No Preemption) เราอาจกำหนดกฎเกณฑ์ ดังนี้ ถ้ากระบวนการหนึ่ง (ที่กำลังถือครองทรัพยากรบางส่วนอยู่) และระบบยังไม่สามารถจัดให้ได้ทันที (แสดงว่ากระบวนการที่ร้องขอจะต้องรอ) เราใช้ทรัพยากรทั้งหมดที่กระบวนการนี้ถือครองอยู่ถูกแทรกกลางคัน นั่นคือ ทรัพยากรที่กระบวนการนี้ถือครองอยู่ทั้งหมดจะถูกปล่อยคืนสู่ระบบโดยปริยาย กระบวนการที่ถูกแทรกกลางคันนี้จะต้องรอคอยทรัพยากร ทั้งที่ร้องขอไว้ตั้งแต่แรกและที่ถูกแทรกกลางคันไป ก่อนที่จะสามารถทำงานต่อไปได้

หรืออาจกล่าวได้ว่า ถ้ามีกระบวนการหนึ่งได้ร้องขอทรัพยากรบางส่วนจากระบบ ในตอนแรกระบบจะตรวจสอบว่า ทรัพยากรที่ร้องขอนั้นว่างอยู่หรือไม่ ถ้าว่างอยู่ ระบบก็จะจัดสรรทรัพยากรเหล่านั้นให้แก่กระบวนการ แต่ถ้ากระบวนการที่ถูกร้องขอนั้นไม่ว่าง ระบบจะตรวจดูก่อนว่าทรัพยากรนั้นไม่ว่างเนื่องจากอะไร ถ้าไม่ว่างเนื่องจากกำลังถูกถือครองโดยกระบวนการอื่น ซึ่งกำลังรอคอยทรัพยากรเพิ่มอยู่ ระบบจะทำการแทรกกลางคันทรัพยากรทั้งหมดของกระบวนการนั้นและจัดสรรทรัพยากรที่ได้มาแก่กระบวนการที่ร้องขอ แต่ถ้ากระบวนการที่ไม่ว่างนั้นไม่ได้ถูกถือครองโดยกระบวนการอื่นที่กำลังรออยู่ ระบบก็จะให้กระบวนการที่ร้องขอทรัพยากรนั้นรอ และขณะที่รออยู่นั้น ทรัพยากรทั้งหมดที่กระบวนการนี้ถือครองอยู่อาจถูกแทรกกลางคันได้ เมื่อมีกระบวนการอื่นร้องขอ และกระบวนการนี้จะสามารถกลับไปทำงานต่อได้ เมื่อได้รับจัดสรรทรัพยากรที่ร้องขอและได้รับทรัพยากรที่อาจถูกแทรกกลางคันไปคืนทั้งหมดเสียก่อน

วิธีการนี้มักใช้กับทรัพยากรที่สามารถเก็บค่าสถานะและติดตั้งค่ากลับคืนมาได้ง่าย เช่น ค่าในรีจิสเตอร์ (ของหน่วยประมวลผลกลาง) เนื้อที่ในหน่วยความจำหลัก เป็นต้น แต่จะไม่สามารถใช้กับทรัพยากรทั่ว ๆ ไปได้ เช่น เครื่องพิมพ์ และหน่วยขับเทป เป็นต้น

4. วงจรรอคอย (Circular Wait) เราอาจป้องกันการเกิดการติดตาย โดยการป้องกันไม่ให้เกิดเงื่อนไขวงจรรอคอย ซึ่งสามารถทำได้โดยการกำหนดลำดับของทรัพยากรทั้งหมดในระบบ และกำหนดให้กระบวนการต้องร้องขอใช้ทรัพยากร เรียงตามเลขลำดับนี้

กำหนดให้ R = { R_1, R_2,..., R_m } โดย R เป็นเซตของทรัพยากรในระบบ และกำหนดให้ทรัพยากรแต่ละประเภทมีค่าลำดับเป็น เลขจำนวนเต็มที่ไม่ซ้ำกัน เขียนแทนด้วย $F(R_i)$ เพื่อให้เราเปรียบเทียบทรัพยากร 2 ประเภทได้ว่าตัวใดมีลำดับก่อน-หลัง ตัวอย่างเช่น ถ้าเซตของทรัพยากร R ประกอบด้วย เครื่องขับเทป เครื่องขับจานบันทึก และเครื่องพิมพ์ ดังนั้นค่าเลขลำดับ $F(R_i)$ อาจถูกกำหนดได้ ดังนี้คือ

F (เครื่องขับเทป) = 1

F (เครื่องขับดิสก์) = 5

F (เครื่องพิมพ์) = 12

และกำหนดวิธีการในการร้องขอทรัพยากรในระบบ ดังนี้

กระบวนการแต่ละตัวสามารถร้องขอทรัพยากรได้ในลำดับที่เพิ่มขึ้นเท่านั้น คือ เริ่มต้นกระบวนการอาจร้องขอทรัพยากรใด ๆ ก็ได้ เช่น ทรัพยากร R_i แต่ต่อจากนี้กระบวนการจะร้องขอทรัพยากร R_j ได้ก็ต่อเมื่อ $F(R_j) > F(R_i)$ ถ้าเป็นการร้องขอทรัพยากรประเภทเดียวกันหลาย ๆ ตัวกระบวนการจะต้องร้องขอทรัพยากรทีละตัว จากตัวอย่าง เลขลำดับทรัพยากรข้างต้น ถ้ากระบวน การหนึ่งต้องการใช้เครื่องขับเทป (F(R) = 1) และเครื่องพิมพ์ (F(R) = 12) กระบวนการนั้นจะต้องร้องขอเครื่องขับเทปก่อน แล้วจึงร้องขอเครื่องพิมพ์จะขอกลับกันไม่ได้ (ระบบไม่อนุมัติ) ในทางตรงกันข้าม ถ้ากระบวนการร้องขอทรัพยากรประเภท R_j กระบวนการจะต้องปล่อยทรัพยากร R_i ซึ่ง $F(R_i) \geq F(R_j)$ คืนสู่ระบบทุกตัวเสียก่อน เช่น ถือครอง R_5 อยู่อยากได้ R_1 ต้องคืน R_5 ก่อน $R_5 \geq R_1$

ตามข้อกำหนดที่กล่าวมา จะเห็นว่าเงื่อนไขวงจรรอคอยจะไม่สามารถเกิดขึ้นได้ สามารถพิสูจน์โดยวิธียกสิ่งตรงข้ามได้ (Proof by Contradiction) ดังนี้

สมมติให้เกิดวงจรรอคอยในระบบ คือ { P_1, P_2, ..., P_n } โดยที่กระบวนการ P_1 รอคอยทรัพยากร R_1 ซึ่งกำลังถูกถือครองโดยกระบวนการ P_{i+1} ถือครองทรัพยากร R_1 อยู่ ขณะที่ร้องขอทรัพยากร R_{i+1} เราจะได้ว่า $F(R_j) < F(R_{i+1})$ สำหรับทุก ๆ ค่าของ I (โดยข้อกำหนดที่ตั้งไว้) ซึ่งหมายความว่า $F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$ ดังนั้น $F(R_0) < F(R_0)$ เอง ซึ่งเป็นไปไม่ได้ สรุปว่าไม่มีวงจรรอคอยในระบบ

เงื่อนไขที่กล่าวมาทั้ง 4 ข้อนี้เป็นการป้องกันการติดตาย แต่สำหรับการหลีกเลี่ยงปัญหาการติดตาย(Deadlock Avoidance) นั้น เราต้องมีข้อมูลเกี่ยวกับการร้องขอทรัพยากรในระบบโดยรวม โดยพิจารณาจากข้อมูลของทรัพยากรที่ถูกร้องขอ เช่นในระบบที่มีเทป และเครื่องพิมพ์อย่างละตัว ดังนั้นเราอาจจัดสรรให้โพรเซส P เข้าใช้งานเทปก่อนแล้วจึงใช้งาน

เครื่องพิมพ์ ในขณะที่โพรเซส Q ใช้งานเครื่องพิมพ์ก่อนแล้วค่อยใช้เทป ดังนั้นนอกจากการร้องขอแล้วยังต้องตรวจว่าทรัพยากรว่างหรือไม่ ทรัพยากรนั้นถูกใช้โดยใคร และโพรเซสไหนจะใช้อะไรก่อนหลัง นอกจากนี้อาจมีข้อมูลที่มากที่สุดที่ขอใช้ทรัพยากรชนิดนั้น รูปแบบอัลกอริทึมของการหลีกเลี่ยงการติดตายจะมีการตรวจสอบสถานะของการใช้ทรัพยากรเพื่อให้แน่ใจว่าจะไม่การเกิดการรอแบบลูปอย่างสม่ำเสมอตลอดเวลา สถานะของการใช้ทรัพยากรถูกกำหนดด้วยจำนวนของทรัพยากรที่ถูกใช้งานและจำนวนทรัพยากรที่มีอยู่ในระบบ (วรรณรัช สันติอมรทัต)

ในระบบที่ทรัพยากร 1 ตัวสามารถให้บริการได้พร้อมกันหลายโพรเซส อัลกอริทึมของนายธนาคาร ซึ่งเป็นอัลกอริทึมที่ใช้งานได้จริงในระบบธนาคารที่ว่า ธนาคารจะไม่จ่ายเงินที่มีอยู่ให้ตามความต้องการของลูกค้าทั้งหมดได้เป็นเวลานาน (หมายถึงมีคนถอนออกอยากเดียว ธนาคารจะไม่สามารถอยู่ได้) ดังนั้นจึงต้องมีระบบเพื่อกำหนดจำนวนสูงสุดที่จะสามารถให้บริการได้ จำนวนนี้อาจไม่จำเป็นต้องเป็นจำนวนทรัพยากรทั้งหมดที่มีอยู่ในระบบ เมื่อผู้ใช้ขอใช้กลุ่มของทรัพยากร ระบบต้องทำการตรวจสอบว่าถ้าให้ใช้แล้วระบบจะยังคงอยู่ในภาวะปลอดภัยหรือไม่ ถ้าไม่ปลอดภัยการให้ใช้ทรัพยากรต้องรอจนกว่าจะมีผู้ใช้รายอื่นทรัพยากรที่ใช้แล้วกลับเข้าสู่ระบบ

โครงสร้างของระบบนายธนาคารมีดังนี้ กำหนดให้มี n โพรเซส มีทรัพยากรในระบบทั้งหมด m ตัว

Available : เป็นเวกเตอร์ของขนาดที่ใช้ชี้จำนวนของทรัพยากรที่สามารถใช้งานได้ Available [j]=k ดังนั้น k คือจำนวนที่ทรัพยากรชนิด j จะสามารถให้บริการได้

Max : เป็นค่าเมตริกซ์ขนาด n x m ที่กำหนดความต้องการสูงสุดของแต่ละ โพรเซส ถ้า Max[i,j] = k แล้วโพรเซส i อาจต้องใช้ทรัพยากร j สูงถึง k บริการ

Allocation เป็นเมตริกซ์ขนาด n x m ที่กำหนดจำนวนของทรัพยากรในแต่ละชนิดที่ให้บริการแต่ละโพรเซสอยู่ได้ ถ้า Allocation[i,j] = k หมายถึงโพรเซส i กำลังใช้งานทรัพยากรชนิด j อยู่เป็นจำนวน k บริการ

Need : เมตริกซ์ขนาด n x m เพื่อบ่งบอกจำนวนทรัพยากรที่เหลือที่ยังต้องการใช้ของแต่ละโพรเซส เช่น Need[i,j] = k หมายถึงโพรเซส i ยังคงต้องการใช้งานทรัพยากร j อยู่อีก k บริการ พบกว่า Need[i,j] = Max[i,j] – Allocation[i,j]

อัลกอริทึมที่ปลอดภัย

อัลกอริทึมที่หาได้ว่าระบบปลอดภัยหรือไม่ สามารถทำได้ดังนี้

1. กำหนดให้ Work และ Finish เป็นเวกเตอร์ที่มีขนาด n x m โดยเริ่มทำงานที่ Work := available และ Finish[i] เป็นเท็จ โดยที่ i มีค่าตั้งแต่ 1, 2 , 3 ... n

2. หาค่า i ทั้ง 2 ฟังก์ชั่น คือ Finish[i] := false , $Need_i$ =< Work ถ้าไม่ตามเงื่อนไข ข้ามไปยังขั้นที่ 4

3. Work := Work + Allocation, Finish[i] := true กลับไปยังขั้นที่ 2

4. ถ้า Finish[i] = true สำหรับทุกค่าของ i แล้วระบบจะอยู่ในภาวะปลอดภัย เวลาในการใช้ทำงานอัลกอริทึมนี้เท่ากับ $m \times n^2$

อัลกอริทึมของการขอใช้ทรัพยากร

กำหนดให้ Request i เป็นเวกเตอร์ของการขอใช้งานสำหรับโพรเซส i ถ้า $Request_i [j] = k$ หมายถึง โพรเซส i ต้องการทำงาน k บริการจากทรัพยากร j เมื่อมีการขอใช้งานทรัพยากรโดยโพรเซส i ก็จะเกิดการทำงานดังต่อไปนี้

1. ถ้า $Request_i$ =< $Need_i$ ไปยังขั้นตอนที่ 2 นอกจากนั้น ให้แสดงข้อความเตือนเนื่องจากโพรเซสมีการใช้ทรัพยากรมากว่าที่คาดการณ์ไว้

2. ถ้า $Request_i$ =< Available ไปยังขั้นตอนที่ 3 นอกจากนี้โพรเซส i ต้องรอเนื่องจากทรัพยากรไม่มีให้ใช้งาน

3. มีการตั้งระบบลวงเพื่อใช้ทรัพยากรที่โพรเซส i ขอใช้ โดยการใส่ค่าในตัวแปรต่อไปนี้

Available := Available – Request;

Allocationi := Allocationi + $Request_i$;

$Need_i$:= $Need_i$ – $Request_i$;

ถ้าผลของการกำหนดค่าการให้ใช้ทรัพยากรออกมาพบว่าระบบอยู่ในภาวะปลอดภัย ก็จะจบสิ้นการทำงานแล้วยอมให้โพรเซส i ใช้งานทรัพยากรนั้นจริงๆ อย่างไรก็ตามถ้าผลออกมาว่าไม่ปลอดภัย โพรเซส i ต้องรอ $Request_i$ แล้วก็จะกลับไปสู่ค่าสถานะเก่า

ภาพที่ 2.6 ตัวอย่างข้อมูลของระบบที่ใช้งาน

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

ที่มา: “Operating System Concepts” โดย Abraham & Peter Baer, 1998, น. 49.

จากภาพที่ 2.6 แสดงข้อมูลการใช้งานของระบบพบกว่ามี 5 โพรเซสในการทำงาน P_0 – P_4 และมีทรัพยากรให้ใช้งานได้ 3 ตัว คือ A, B และ C ทรัพยากร A มีให้ใช้ได้ถึง 10 ตัว ทรัพยากร B ใช้ได้ถึง 5 ตัว และทรัพยากร C ใช้ได้ถึง 7 ตัว สมมติว่าที่เวลา t_0 เกิดเหตุการณ์ดังภาพที่ 2.6 ดังนั้นจะสามารถหา Need := Max – Allocation ดังภาพที่ 2.7 พบว่าระบบอยู่ในภาวะปลอดภัยแน่นอน และลำดับของความปลอดภัยเป็น <P_1,P_3,P_4,P_2,P_0>

ภาพที่ 2.7 ตัวอย่างค่าของ Need := Max – Allocation ของแต่ละโพรเซส

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

ที่มา: “Operating System Concepts” โดย Abraham & Peter Baer, 1998, น. 49.

สมมติให้โพรเซส 1 มีการขอใช้งานทรัพยากรเพิ่ม 1 ตัว ของ A และจากทรัพยากร C อีก 2 ตัว ดังนั้น $Request_1$:= (1,0,2) ซึ่งสามารถตรวจสอบได้จาก $Request_1$ =< Available : (1,0,2) <= (3,3,2) เป็นจริงดังนั้นจึงต้องทำการสร้างระบบจำลองขึ้นมาเพื่อตรวจสอบภาวะของระบบดังภาพที่ 2.8 หลังจากนั้นไปเข้าไปทำอัลกอริทึมที่ปลอดภัยเพื่อหาลำดับความปลอดภัย

และได้ลำดับออกมาดังนี้ <P_1, P_3, P_4, P_0, P_2> แต่เราพบกว่าถ้าโพรเซส P_4 มีการขอใช้ทรัพยากร (3,3,0) จะไม่มีให้ใช้งานได้ และถ้า P_0 ขอใช้งาน (0,2,0) ก็จะใช้ไม่ได้เช่นกันเนื่องจากจะทำให้อยู่ในภาวะไม่ปลอดภัย

ภาพที่ 2.8 ตัวอย่างระบบจำลองที่สร้างเพื่อตรวจสอบภาวะของระบบ

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

ที่มา: “Operating System Concepts” โดย Abraham & Peter Baer, 1998, น. 49.

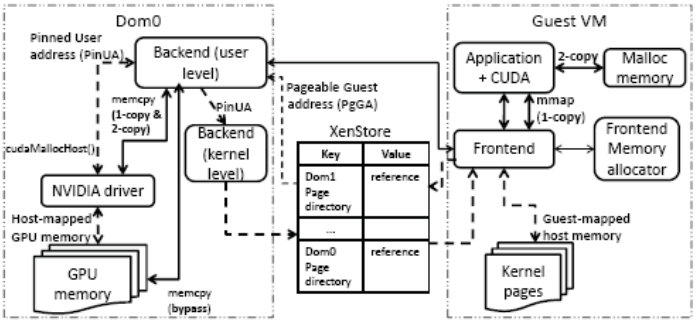
2.2 งานวิจัยที่เกี่ยวข้อง

ส่วนนี้กล่าวถึงงานวิจัยที่เกี่ยวข้อง ซึ่งเกี่ยวกับการเข้าถึงทรัพยากรจีพียู และสั่งงานจีพียูจากเวอร์ชวลแมชชีนในรูปแบบต่าง ๆ โดยมีรายละเอียดดังนี้

งานวิจัย **Vishakha Gupta et al** เสนอการออกแบบระบบจีวิม (GViM) เพื่อให้สามารถใช้ทรัพยากรจีพียูร่วมกันสำหรับเวอร์ชวลแมชชีนโดยใช้การจำลองคูด้าเอพีไอ (CUDA API) ภายใต้ระบบเซน (Xen) และใช้เครื่องมือเซนเสตอร์ (XenStore) ในการแชร์ข้อมูลระหว่างเวอร์ชวลแมชชีนกับเครื่องจริง ข้อดีของจีวิมคือมีค่าโอเวอร์เฮดในการถ่ายโอนข้อมูลระหว่างเวอร์ชวลแมชชีนกับ จีพียูที่น้อยกว่างานอื่น แต่มีข้อจำกัดคือ จีวิมสนับสนุนการใช้งานจีพียูร่วมกันระหว่างเวอร์ชวล แมชชีนที่อยู่บนระบบเซนที่รันอยู่บนเครื่องคอมพิวเตอร์เครื่องเดียวกันเท่านั้น และไม่รองรับแอพพลิเคชันกราฟิก

วิธีการออกแบบระบบของจีวิมเพื่อให้สามารถใช้ทรัพยากรจีพียูร่วมกันสำหรับหลายเวอร์ชวลแมชชีนบน Xen-base ได้แบ่งการทำงานสำหรับการเข้าถึงจีพียูออกเป็น 2 ส่วนการทำงานหลัก คือ 1) การเข้าถึงจีพียูระหว่างโดเมนซีโร่ (Dom0) และเกสเวอร์ชวลแมชชีน 2) การจองพื้นที่หน่วยความจำสำหรับประมวลผลคูด้าแอพพลิเคชันโดเมนซีโร่ (Dom0) และเกสเวอร์ชวลแมชชีน

ภาพที่ 2.9 ภาพรวมของระบบ GViM และการจัดการพื้นที่หน่วยความจำ



ที่มา: “GViM: GPU-accelerated virtual machines”

โดย V. Gupta et al, ACM, 2009.

1. การจัดการ การเข้าถึงจีพียูระหว่างโดเมนซีโร่ (Dom0) และเกสเวอร์ชวลแมชชีน แบ่งการทำงานออกเป็นรายละเอียดดังนี้

• ฟรอนท์เอนด์ไดร์เวอร์ ซึ่งถูกติดตั้งอยู่ที่เกสเวอร์ชวลแมชชีนทำหน้าที่จัดการสำหรับการติดต่อสื่อสารระหว่างเกสเวอร์ชวลแมชชีนและโดเมนซีโร่ โดยใช้ช่องทางในการติดต่อสื่อสารผ่าน Xen-bus และ Xenstore ในการรับแพคเกจข้อมูลจากไลบรารีที่สร้างขึ้นมาสำหรับจัดการแพคข้อมูลและส่งแพคเกจที่ได้จากการแพคส่งไปยังแบคเอ็นด์ที่โดเมนซีโร่เพื่อทำการประมวลผลและรับข้อมูลกลับมาจากการประมวลผล

• แบคเอ็นด์ ถูกติดตั้งอยู่ที่โดเมนซีโร่มีหน้าที่เป็นตัวกลางคอยผสานงานการเข้าถึงจีพียูจากการร้องขอของคูด้า (CUDA) ที่ได้รับมาจากเกสเวอร์ชวลแมชชีนผ่านทาง ฟรอนท์เอนด์เพื่อประมวลผลในจีพียูและส่งค่าที่ได้กลับไปยังฟรอนท์เอนด์

2. การจัดการ การจองพื้นที่หน่วยความจำสำหรับประมวลผลคูด้าแอพพลิเคชันโดเมนซีโร่ (Dom0) และเกสเวอร์ชวลแมชชีน

ในการเรียกใช้งานจีพียูจากเกสเวอร์ชวลแมชชีนเพื่อส่งข้อมูลสำหรับนำไปให้โดเมนซีโร่ส่งไปประมวลผลในจีพียูนั้นงานวิจัยนี้มีการนำเสนอการส่งข้อมูล 3 วิธี โดยมีรายละเอียดการจัดการพื้นที่หน่วยความจำดังนี้

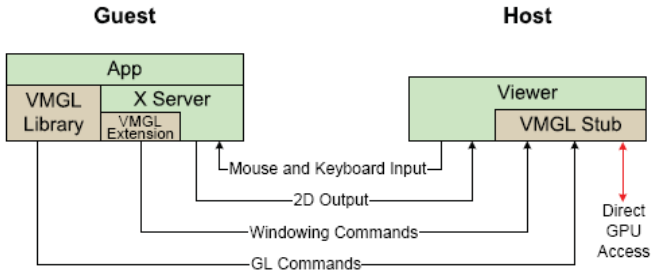
• 2-copy เกสเวอร์ชวลแมชชีนจะจองพื้นที่บนหน่วยความจำตามการร้องขอของคูด้า แอพพลิเคชันบนเกสเวอร์ชวลแมชชีน เมื่อต้องการส่งข้อมูลไปให้โดเมนซีโร่ทำงานจะทำการคัดลอกข้อมูลที่มีอยู่บนหน่วยความจำที่จองไว้ส่งไปยังหน่วยความจำบนโฮสจากนั้นจึงคัดลอกจากหน่วยความจำบนโฮสไปยังจีพียู

• 1-copy เป็นการคัดลอกโดยให้ผู้ใช้เรียก mmap() ที่อยู่ในฟรอนท์เอนด์แทนการเรียก malloc เพื่อลดการคัดลอกข้อมูลจากหน่วยความจำของเกสเวอร์ชวลแมชชีนมาที่หน่วยความจำของโฮส

• Bypass เป็นการลดการคัดลอกข้อมูลระหว่างกันทั้งหมดโดยการผสานพื้นที่หน่วยความจำของเกสเวอร์ชวลแมชชีนและจีพียูเข้าด้วยกันเพื่อลดการคัดลอกระหว่างกัน

งานวิจัย **H. Andres Lagar-Cavilla et al** นำเสนองานวิจัยที่มีชื่อว่าวีเอ็มจีแอล (VMGL) งานวิจัยหลายงานที่สร้างระบบเพื่อให้เข้าถึงทรัพยากรจีพียูจากเวอร์ชวลแมชชีนแต่มีความแตกต่างกับงานวิจัยนี้ ในเรื่องของรูปแบบในการเข้าถึงและลักษณะการจัดสรรทรัพยากรจีพียู งานวิจัยระบบวีเอ็มจีเอล (VMGL) มุ่งเน้นการเข้าถึงจีพียูจากเวอร์ชวลแมชชีนโดยจำลองโอเพนจีแอลเอพีไอ (OpenGL API เป็นเอพีไอมาตรฐานสำหรับการออกคำสั่งให้จีพียูประมวลผลทางด้านกราฟิก) บนเวอร์ชวลแมชชีน ระบบวีเอ็มจีแอลในงานวิจัยนี้อนุญาตให้แอพพลิเคชันที่ต้องการประมวลผลกราฟิกอยู่บนเวอร์ชวลแมชชีนสามารถเรียกใช้งานทรัพยากรจีพียูได้ แต่งานวิจัยนี้ไม่ได้สนับสนุนการเข้าถึงจีพียูเพื่อการประมวลผลสมรรถนะสูงของแอพพลิเคชันที่ใช้คู่ด้าเอพีไอแต่จะมุ่งเน้นไปทางด้านการทำโอเพนจีแอลเวอร์ชวลไลเซชั่น ซึ่งอนุญาตให้แอพพลิเคชันต่าง ๆ ที่ประมวลผลด้านกราฟิกที่ต้องใช้งานจีพียูและอยู่บนเวอร์ชวลแมชชีนสามารถประมวลผลได้ นอกจากนี้ วีเอ็มจีแอลยังมีความสามารถจัดการการทำงานของแอพพลิเคชันที่ทำงานบนจีพียูต่างค่ายกันได้

ภาพที่ 2.10 แสดงโครงสร้างของระบบวีเอ็มจีแอล (VMGL)



ที่มา: “VMM-independent graphics acceleration”
โดย H. A. Lagar-Cavilla et al., ACM, 2007.

โครงสร้างของวีเอ็มจีแอลประกอบไปด้วย 3 ยูสเซอร์-สเปซ โมเดล คือ

1) วีเอ็มจีแอลไลบรารี (VMGL library)

2) วีเอ็มจีแอลสตับ (VMGL stub)

3) วีเอ็มจีแอลเซอร์เวอร์เอ็กซ์เท็นชั่น (VMGL X server extension)

จากการทำวิจัยวีเอ็มจีแอลนี้ได้ผลการทดลองว่าวีเอ็มจีแอลมีประสิทธิภาพในการจัดการการแสดงผลแอพพลิเคชันด้านกราฟิกที่ปฏิบัติงานบนเวอร์ชวลแมชชีนได้ดี

งานวิจัยของ **Dowty และ Sugerman** เสนอการเข้าถึงจีพียูโดยใช้การจำลองจีพียูในระดับของฮาร์ดแวร์แทนที่จะเป็นการจำลองคูด้าหรือโอเพนจีแอลเอพีไอดังเช่นในงานวิจัยนี้ หรือสองงานข้างต้น ทำให้สามารถรองรับทั้งแอพพลิเคชันทางด้านกราฟิกและการประมวลผลสมรรถนะสูงที่ใช้คูด้าเอพีไอได้ แต่งานวิจัยนี้ก็มีความซับซ้อนสูงและขึ้นอยู่กับระบบวีเอ็มแวร์ (vmware) และไม่ได้พิจารณาปัญหาการติดตายเมื่อคูด้าแอพพลิเคชันจากเวอร์ชวลแมชชีนหลายเครื่องต้องการใช้งานจีพียูพร้อม ๆ กัน

จากงานวิจัยที่ผ่านมานั้น การเข้าถึงจีพียูเพื่อสั่งงาน โดยผ่านคูด้าสำหรับเครื่องที่เป็นเวอร์ชวลแมชชีน ส่วนใหญ่จะออกแบบระบบให้มีการทำงานในลักษณะฟรอนท์เอนด์ไลบรารี กับแบ็คเอ็นด์ แต่ยังไม่มีงานวิจัยใดพิจารณาเรื่องระบบการจัดการ การเข้าถึงจีพียูพร้อม ๆ กัน ซึ่งอาจทำให้เกิดปัญหาการติดตาย กรณีรอขอใช้ทรัพยากรจีพียู และการเพิ่มประโยชน์การใช้งานทรัพยากรจีพียูร่วมกัน จึงทำให้มีแนวคิดที่ขอนำเสนอระบบเวอร์ชวลคูด้า เพื่อจัดการ การเข้าถึงจีพียู และการเพิ่มประโยชน์การใช้งานจีพียู ดังจะกล่าวในลำดับถัดไป