บทที่ 2 การจัดการงาน (process)

วัตถุประสงค์ของเนื้อหา

- ศึกษาถึงความหมายของงาน (process) และการจัดการงานในระดับพื้นฐานของระบบปฏิบัติการ
- ศึกษาถึงการจัดสรรเวลาของงาน (CPU scheduling) และวัฏจักรของงาน และสถานะต่างๆ ภายในวัฏจักร
- ศึกษาถึงหลักการการติดต่อระหว่างงานภายในระบบ
- ศึกษาถึงกรณีผู้ผลิต-ผู้บริโภค (Producer-Consumer) ที่เป็นกรณีประเด็นปัญหาพื้นฐานของการสื่อสารระหว่างงาน

สิ่งที่คาดหวังจากการเรียนในบทนี้

- นักศึกษาเข้าใจถึงสถานะต่างๆ ของงานแต่ละงานที่กำลังทำงานภายในระบบ อันเป็นพื้นฐานของระบบปฏิบัติการแบบหลายภารกิจ (Multitasking OS)
- นักศึกษาเข้าใจถึงกลไกการทำงานขั้นพื้นฐานและวัฏจักรของงาน ตั้งแต่การสร้างงาน ไปจนถึงการจบการทำงาน
- นักศึกษาเข้าใจถึงหลักการสื่อสารขั้นพื้นฐานระหว่างงานที่กำลังรันอยู่บนระบบปฏิบัติการ

วัตถุประสงค์ของปฏิบัติการท้ายบท

- นักศึกษาได้ทดลองการสร้างงานใหม่ทั้งบนระบบปฏิบัติการวินโดวส์ และลินุกซ์ และศึกษาถึงความแตกต่างระหว่างการประยุกต์ใช้งานของทั้งสองระบบปฏิบัติการ
- นักศึกษาได้ทดลองกลไกการติดต่อสื่อสารกันระหว่างงานสองงาน ทั้งบนระบบปฏิบัติการวินโดวส์ และลินุกซ์ และศึกษาถึงความเหมือนกันและที่แตกต่างกันระหว่างทั้งสองระบบปฏิบัติการ ทั้งกรณีของการใช้พื้นที่หน่วยความจำร่วม (Shared memory) และการใช้แมสเซสคิว (Message queue) และ ไปป์ (pipe)
- นักศึกษาได้ทดลองกรณีประเด็นปัญหาผู้ผลิต-ผู้บริโภค ซึ่งเป็นพื้นฐานของการสื่อสารระหว่างงาน

สิ่งที่คาดหวังจากปฏิบัติการท้ายบท

- นักศึกษาสามารถแตกงานใหม่จากงานตั้งต้นได้
- นักศึกษาสามารถสั่งให้งานพ่อแม่ รองานลูก และสั่งจบการทำงานของงานพ่อแม่ และงานลูกได้
- นักศึกษาสามารถใช้การติดต่อสื่อสารระหว่างงาน ทั้งในรูปแบบของการใช้พื้นที่หน่วยความจำร่วม การใช้แมสเซสคิว และ ไปป์ โดยอาศัยกรณีประเด็นปัญหาผู้ผลิต-ผู้บริโภค เป็นกรณีตัวอย่าง

เวลาที่ใช้ในการเรียนการสอน

- ทฤษฎี 2 ชั่วโมง
    - พื้นฐานเบื้องต้นของงาน และวัฏจักรของงาน 1 ชั่วโมง
    - การติดต่อสื่อสารระหว่างงาน 1 ชั่วโมง
- ปฏิบัติ 4 ชั่วโมง
    - การเขียนโปรแกรมแตกงานใหม่ และการจบการทำงาน 0.5 ชั่วโมง
    - การสื่อสารระหว่างงานอาศัยพื้นที่หน่วยความจำร่วม และ memory-mapped file 1.5 ชั่วโมง
    - การสื่อสารระหว่างงานอาศัยเมสเซสคิว และไปป์ 2 ชั่วโมง

บทที่ 2 การจัดการงาน (process)

งาน หรือ process คือองค์ประกอบของชุดคำสั่ง ข้อมูลที่ใช้ประกอบการคำนวณ และข้อมูลอื่นๆ ที่ใช้ประกอบร่วมกันที่กำลังปฏิบัติงานอยู่ภายใต้การควบคุมของระบบปฏิบัติการ งานโดยส่วนมากจะมีจุดเริ่มต้นจากตัวโปรแกรมในหน่วยความจำสำรอง ถูกระบบปฏิบัติการโหลดขึ้นมาวางไว้ในพื้นที่หน่วยความจำที่เตรียมไว้เพื่อให้ดำเนินการไปตามชุดคำสั่งในโปรแกรมนั้นๆ ดังนั้นเราอาจกล่าวได้ว่า โพรเซส คือหน่วยการคำนวณพื้นฐานของซอฟต์แวร์ภายใต้ระบบคอมพิวเตอร์ก็ว่าได้

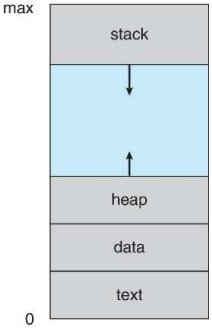
ในสมัยยุคแรกๆ จุดประสงค์ของการบริหารจัดการการคำนวณของโปรแกรมต่างๆ ในคอมพิวเตอร์ เป็นไปเพื่อการเข้าคิวโปรแกรมหลายโปรแกรม เพื่อส่งให้คอมพิวเตอร์ประมวลเป็นชุดๆ (batch system) เราจะเรียกโปรแกรมแต่ละตัวที่ถูกเข้าคิวรอทำงานว่า job (โปรแกรมแต่ละตัวจะถูกโหลดขึ้นหน่วยความจำหลัก แต่ระบบปฏิบัติการจะสั่งรันไปทีละตัวจนกระทั่งจบทั้งคิว)

ในปัจจุบัน การบริหารการจัดการคำนวณของโปรแกรมโดยระบบปฏิบัติการ มักเป็นไปเพื่อการใช้ทรัพยากรในแต่ละส่วนของคอมพิวเตอร์ให้คุ้มค่าที่สุด โปรแกรมที่กำลังทำงาน (ที่ได้มาจากการโหลดโปรแกรมขึ้นมาในหน่วยความจำให้ทำงาน) จะมีอยู่ได้มากกว่าหนึ่งตัวพร้อมๆ กัน โดยอาศัยการจัดแบ่งเวลาเข้าครอบครองซีพียูโดยระบบปฏิบัติการ เรามักจะเรียกแต่ละหน่วยว่า process หรือบางทีก็อาจเรียกว่า task (แต่ละงานถูกสลับกันขึ้นมารันไป โดยไม่ได้รอให้ตัวใดตัวหนึ่งจบเสียก่อน)

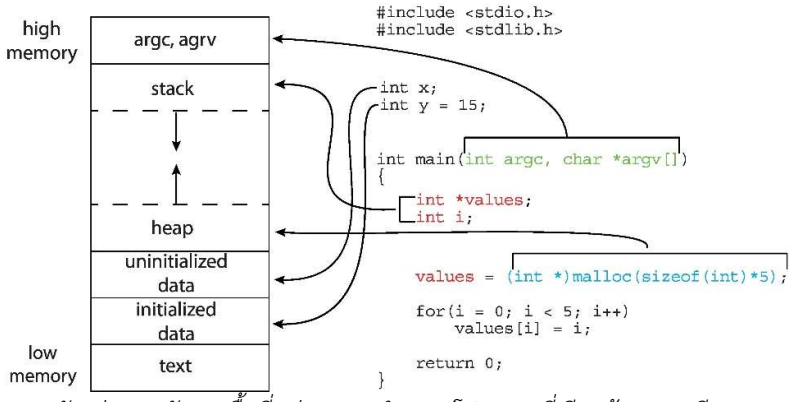
ดังนั้นในการใช้งานปัจจุบัน เราอาจจะพบคำว่า job process หรือ task ในการเรียกถึงหน่วยของงานที่กำลังคำนวณนี้ และเพื่อให้เข้าใจตรงกัน ต่อจากนี้จะขอใช้คำว่า process เป็นหลัก

process แต่ละตัว จะประมวลคำสั่งในลักษณะอ่านคำสั่งต่อเนื่องไปตามข้อกำหนดของคำสั่งเพื่อนำมาประมวลโดยซีพียู (sequential) (ในลักษณะของ von Neumann architecture นั่นเอง) process แต่ละตัวจะประกอบไปด้วย

- **Text Section** หรือส่วนที่จัดเก็บ program code หรือชุดคำสั่งที่จะใช้ประมวล
- **Stack** เป็นพื้นที่หน่วยความจำที่จัดการเข้าถึงในลักษณะของสแต็ก โดยอาศัยเรจิสเตอร์ในซีพียู (สแต็กพอยน์เตอร์) ในการอ้างถึงตำแหน่งยอดของสแต็ก พื้นที่ส่วนนี้โปรแกรมทั่วไปมักใช้เก็บตัวแปรเฉพาะที่ (local variables) และข้อมูลที่ใช้ส่งระหว่างฟังก์ชัน *(อาร์กิวเมนต์ของฟังก์ชัน ค่ากลับคืน และ ค่าตำแหน่งหน่วยความจำที่ของคำสั่งที่จะดำเนินการต่อหลังจากกระโดดกลับจากฟังก์ชัน)*
- **Data Section** เป็นส่วนที่จัดเก็บตัวแปรส่วนกลาง (global variables)
- **Heap** เป็นพื้นที่(หรือขอบเขตตำแหน่งหน่วยความจำ)ที่จะใช้ในการวางพื้นที่หน่วยความจำที่จองมาจากระบบปฏิบัติการ
- **PCB (Process Control Block)** เก็บข้อมูลที่จำเป็นของโพรเซสแต่ละตัว รวมถึงค่าในเรจิสเตอร์ที่จะถูกพักไว้ในระหว่างที่ถูกสลับงานออก ขอบเขตหน่วยความจำที่เข้าถึงได้ รายการไฟล์แฮนเดิลที่กำลังใช้งาน สภาวะปัจจุบันของโพรเซส ค่าตำแหน่งหน่วยความจำของคำสั่งที่กำลังจะประมวล เป็นต้น



โครงสร้างของพื้นที่หน่วยความจำเชิงเส้น (linear memory space) ของโพรเซส (ที่นิยมใช้กันโดยทั่วไป) สังเกตว่าพื้นที่ของสแต็ก และ ฮีป นั้น จะถูกกำหนดพื้นที่ตั้งต้นไว้ขนาดหนึ่ง และจะถูกขยายเพิ่มเติมได้เมื่อต้องการ (กลไกการขยายพื้นที่ หรือการจองพื้นที่เพิ่มเติมนี้ จะได้เรียนในรายละเอียดต่อไปในภายหลัง)



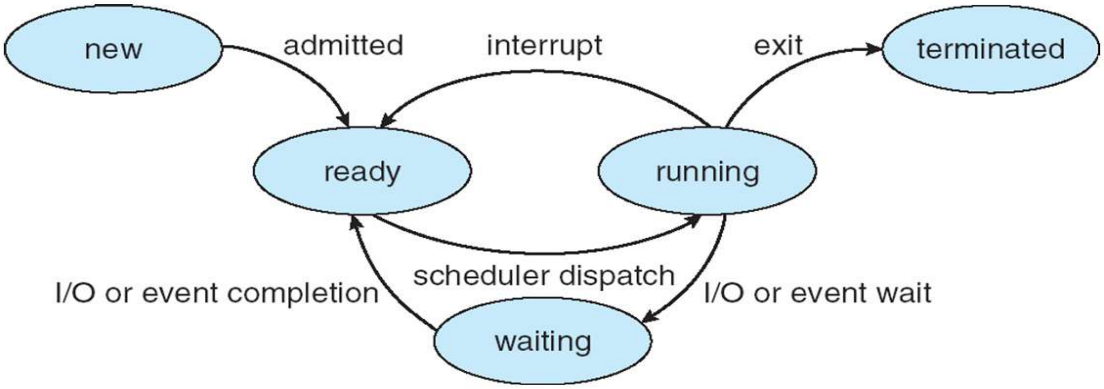
ตัวอย่างการจัดสรรพื้นที่หน่วยความจำ จากโปรแกรมที่เขียนด้วยภาษาซี

2.1 สถานะของโพรเซส (Process State)

คือสถานะของโพรเซสที่เป็นไปได้ในขณะที่อยู่ในระบบ โดยเริ่มจาก

- **New** คือสถานะของโพรเซสที่เพิ่งถูกสร้างและเตรียมพร้อมจะให้เริ่มทำงาน (หลังจากที่โหลดโปรแกรมเข้ามาในหน่วยความจำที่เตรียมไว้ และจัดเตรียม PCB ตั้งต้นเสร็จสิ้น)
- **Ready** เมื่อโพรเซสพร้อมจะทำงานจากสภาวะ new หรือเพิ่งจบจากการถูกร้องขอ (อินเทอร์รัปต์ หรือเมื่อจบการทำ system call) หรือจัดการกับ I/O เสร็จสิ้น โพรเซสก็จะมาอยู่ในสถานะนี้เพื่อเตรียมพร้อมจะทำงานต่อไป
- **Running** จากกลไกการจัดการ CPU scheduling ของระบบปฏิบัติการ ระบบปฏิบัติการจะคัดโพรเซสที่รอคิวเตรียมพร้อมรัน (ready) เข้ามาครอบครองเวลาของซีพียู (สั่งกระโดดเข้าไปทำงานในโค้ดของโพรเซส-**schedular dispatch**) ในช่วงเวลาสั้นๆ โดยเมื่อครบรอบการประมวล ก็จะไปยังสถานะอื่นๆ โดยกลไกการหลุดออกจากสถานะ running อาจมีกรณีเป็นไปได้ดังนี้
  - **(Interrupt)** เมื่อถูกอินเทอร์รัปต์โดยอุปกรณ์ฮาร์ดแวร์ของระบบ หรือเกิดความผิดพลาดในการคำนวณ (trap/exception) ISR ที่เกี่ยวข้องกับการจัดการอินเทอร์รัปต์ (ซึ่งเป็นองค์ประกอบของตัวระบบปฏิบัติการ) ก็จะเข้าแทรกการทำงานในทันที โดยโพรเซสจะถูกเปลี่ยนสถานะเป็น ready
  - **(I/O call or event wait)** โพรเซสปัจจุบันเรียกใช้บริการของระบบปฏิบัติการอาศัย system call (ซึ่งเป็นชุดคำสั่งในส่วนของระบบปฏิบัติการ) ระบบปฏิบัติการจะพักโพรเซสที่กำลังทำงานนั้นไว้ก่อน จนกว่า system call เสร็จสิ้น กรณีนี้โพรเซสจะถูกเปลี่ยนสถานะเป็น waiting (หรือ blocking)
  - **(I/O or event completion)** เมื่อโพรเซสหนึ่งๆ ร้องขอ system call แล้ว ซึ่งอาจจะเป็นการร้องขอติดต่อกับอุปกรณ์ต่างๆ ของระบบ ตามปกติแล้ว การทำงานของอุปกรณ์ต่างๆ นั้น ช้ากว่าการทำงานของซีพียูอยู่มาก ดังนั้นการที่จะให้โพรเซสที่กำลังครอบครองซีพียูในปัจจุบัน รอคอยการทำงานของ I/O จึงไม่เหมาะสม ดังนั้นโพสเซสที่ร้องขอติดต่อ I/O ใดๆ ก็จะถูกโยนเข้าคิว waiting ในทันที และจนกว่า I/O จะทำงานเสร็จสิ้น โดยอาศัยกลไกการอินเทอร์รัปต์ของฮาร์ดแวร์ของระบบ เมื่อ I/O ทำงานเสร็จก็จะเกิดอินเทอร์รัปต์เกิดขึ้น ชุดคำสั่ง ISR ของระบบปฏิบัติการที่เกี่ยวข้องกับอินเทอร์รัปต์นั้นๆ ก็จะถูกสั่งให้ทำงาน เพื่อปิดท้ายกลไกการจัดการของ system call ที่เกิดขึ้นจากในขั้นตอนก่อนหน้า จากนั้นโพรเซสที่ถูกพักไว้ (waiting) นี้ก็จะถูกจับย้ายไปสู่สถานะ ready เพื่อรอคอยคิวเข้าครอบครองซีพียูในรอบการทำงานต่อไป
  - **(Exit)** เมื่อโปรแกรมทำงานเสร็จสิ้น ก็จะจบการทำงาน และเข้าสู่สภาวะ **terminated**

- **Terminated** เมื่อโพรเซสเข้าสู่สภาวะ terminated ไม่ว่าจะเป็นการร้องขอตามปกติจากโพรเซสเอง หรือการบังคับจากระบบปฏิบัติการ (abnormal termination) ระบบปฏิบัติการจะดึงคืนทรัพยากรทั้งหมดที่โพรเซสนั้นใช้กลับคืนมาเพื่อเก็บไว้ให้โพรเซสอื่นที่ต้องการได้ใช้ต่อไป



2.2 Process Control Block

PCB (Process Control Block หรืออาจเรียกว่า Task Control Block) เป็นโครงสร้างข้อมูลที่จัดเก็บค่าต่างๆ ที่จำเป็น ต่อสถานะการทำงานของโพรเซสหนึ่งๆ ซึ่งมีองค์ประกอบที่สำคัญดังเช่น

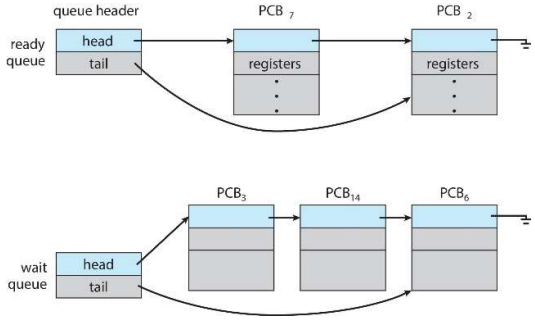
- สถานะปัจจุบันของโพรเซส (ว่าเป็น new, running , ...)
- หมายเลขของโพรเซส (Process identifier หรือ Process ID)
- Program Counter ค่าตำแหน่งหน่วยความจำถัดไปของคำสั่งที่กำลังประมวลอยู่ (บอกให้รู้ว่าเมื่อคำนวณคำสั่งปัจจุบันเสร็จแล้วจะไปทำคำสั่งที่ตำแหน่งไหนต่อ โดยปกติแล้วค่านี้จะอยู่ในเรจิสเตอร์ของซีพียู และเป็นเรจิสเตอร์ตัวหนึ่งที่สำคัญต่อการทำงานของซีพียูด้วย)
- CPU Registers โครงสร้างข้อมูลของเรจิสเตอร์ที่กำลังใช้งานของโพรเซสนั้นๆ ในซีพียูทั้งหมด ค่านี้จะได้มาจากการดัมป์ค่าจากซีพียูก่อนที่โพรเซสจะออกจากสถานะ running เข้าสู่สถานะอื่น และค่านี้จะถูกโหลดกลับให้กับเรจิสเตอร์ของซีพียู เมื่อโพรเซสจะกลับเข้าสู่สถานะ running ในรอบการครอบครองซีพียูครั้งถัดไป ทำให้การทำงานของซีพียูในโพสเซสปัจจุบัน เป็นไปอย่างต่อเนื่อง
- CPU-scheduling information เป็นข้อมูลที่ระบบปฏิบัติการใช้บันทึกพารามิเตอร์ต่างๆ ที่ต้องใช้ในการทำ CPU Scheduling เช่นค่า priority ของโพรเซสเป็นต้น
- Memory-management information ข้อมูลเกี่ยวกับค่าตำแหน่งหน่วยความจำฐาน (อาจจะเป็นค่าเซ็กเมนต์ฐานพร้อมเพจฐาน หรืออื่นใดนั้นขึ้นอยู่กับสถาปัตยกรรมการจัดการทางฮาร์ดแวร์ของระบบ และที่ควบคุมอยู่โดยระบบปฏิบัติการ ดังจะได้เรียนต่อไปในเรื่อง memory management)
- Accounting information เป็นข้อมูลเกี่ยวกับรายละเอียดการทำงานของโพรเซส เช่นซีพียูที่สามารถครอบครองได้ (ในกรณีที่มีซีพียูมากกว่าหนึ่งหน่วย) เวลาที่ใช้ไปในการทำงาน ผู้ใช้เจ้าของโพรเซส และอื่นๆ
- I/O status information เป็นรายละเอียดของทรัพยากร I/O ที่โพรเซสนั้นกำลังครอบครองหรือแบ่งใช้งานอยู่ ณ ปัจจุบัน (เช่น รายการไฟล์แฮนเดิลของไฟล์ที่กำลังเปิดใช้อยู่ของโพรเซสนั้นๆ เป็นต้น)

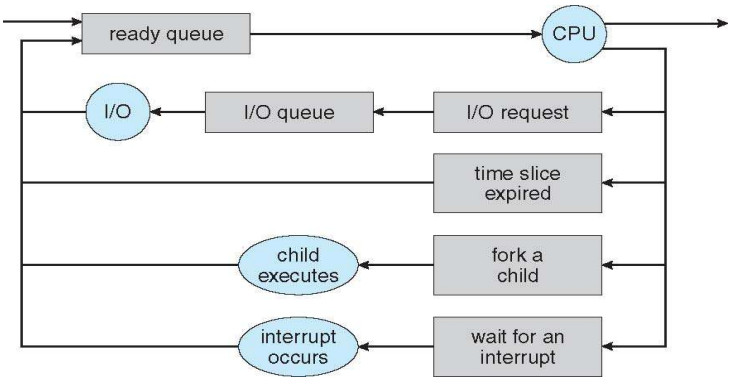
อนึ่ง ในระบบปฏิบัติการที่รองรับเฉพาะโพรเซสที่เป็นแบบเธรดเดียว (single thread) โพรเซสแต่ละตัวก็จะมี PCB เพียงชุดเดียว แต่ในกรณีที่ระบบปฏิบัติการรองรับ multithreading นั้น เมื่อแตกเธรดใหม่ จะสร้าง PCB ชุดใหม่ให้กับเธรดที่เพิ่งเริ่มทำงานด้วย และแต่ละเธรดก็จะมีวัฏจักรการทำงานที่เป็นอิสระต่อกัน (และจะมีสถานะของเธรดที่แตกต่างกัน ดังจะได้เรียนในบทถัดไป)

2.3 การจัดสรรเวลาให้กับโพรเซส (Process Scheduling)

Process Scheduling คือการจัดสรรโพรเซสทั้งหมดที่อยู่ในระบบ ได้มีโอกาสผลัดกันเข้ามาครอบครองเวลาประมวลของซีพียู (อย่างรวดเร็วจนดูเสมือนหนึ่งว่าโพสเซสทั้งหมดทำงานไปพร้อมๆ กัน) ทั้งนี้ก็เพื่อให้ซีพียูถูกใช้งานอย่างตลอดเวลา หรือมีเวลา idle น้อยที่สุด โดยมี Process scheduler เป็นกลไกของระบบปฏิบัติการที่ทำหน้าที่ดังกล่าว ซึ่งจะคอยสลับโพสเซส(หรือพูดในอีกนัยที่ชัดเจนกว่าคือ PCB ของโพรเซส) ต่างๆ ให้เข้าออกคิวต่างๆ ที่ใช้สำหรับการจัดการ คิวรอคอยที่สำคัญ มีดังต่อไปนี้

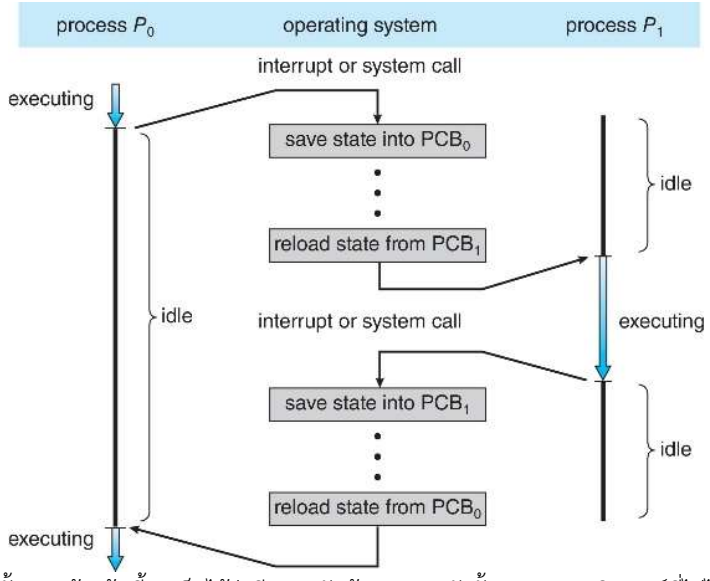
- Job queue เป็นคิวที่จัดเก็บรายการของโพสเซสทั้งหมดที่กำลังทำงานอยู่ในระบบ
- Ready queue เป็นคิวที่จัดเก็บรายการของโพรเซสที่จะรอเพื่อให้เข้าครอบครองเวลาซีพียูเพื่อประมวลผลในรอบต่อไป การดึงโพรเซสจาก ready queue เข้ามาประมวลผลโดยซีพียูเรียกว่า dispatch
- Device queue / Wait queue เป็นคิวที่จัดเก็บรายการของโพรเซสที่ต้องรอการติดต่อกับดีไวซ์ ซึ่งปกติจะมีประสิทธิภาพความเร็วในการทำงานต่ำกว่าซีพียู (การโยนมาให้คิวนี้รับไว้ จึงทำให้โพรเซสอื่นไม่ต้องมารับภาระการรอคอยงานที่ไม่ได้เกี่ยวเนื่องกับการทำงานของซีพียู) ซึ่ง device queue ก็จะมีได้หลายคิวโดยแบ่งไปตามดีไวซ์แต่ละตัว เช่นดิสก์แต่ละตัว หรือการแสดงผลบนจอภาพหรือเทอร์มินัล หรือเหตุการณ์ต่างๆ ที่ระบบปฏิบัติการสร้างขึ้น เป็นต้น





การจัดสรรงาน (Scheduler)

กลไกการจัดสรรงานของระบบปฏิบัติการ กระทำโดยส่วนการทำงานที่เรียกว่า **scheduler** ซึ่งทำหน้าที่จัดสรรงาน (dispatch) โพรเซสที่รออยู่ใน ready queue ให้เข้ามาครอบครองซีพียูตามขั้นตอนคือ เมื่อเกิดอินเทอร์รัปต์ขึ้นภายในระบบปฏิบัติการ (ส่วน ISR ก็จะถูกสั่งขึ้นมาทำงาน) หรือโพรเซสที่กำลังทำงานในปัจจุบันนั้นเรียก system call (เกิดการกระโดดไปยังชุดคำสั่งของระบบปฏิบัติการที่จะให้บริการตาม system call ที่กำหนด) ในจุดเหล่านี้ ก็จะกระโดดไปทำงานในส่วนของ scheduler ก่อน (กลไกทั้งหมดเหล่านี้ทำงานภายใต้ kernel mode) scheduler ก็จะบันทึก PCB ของโพรเซสที่กำลังทำงานในปัจจุบันเก็บไว้ (ใน waiting queue หรือ ready queue ขึ้นอยู่กับแต่ละกรณี) จากนั้น จะเลือกโพรเซสจาก ready queue ที่จะมาทำงานแทน โดยหลังจากทำงานตามที่ต้องกระทำ (ตาม ISR หรือ system call ที่จำเป็นจนเสร็จสิ้นในเบื้องต้น) ก็จะโหลด PCB ของโพรเซสถัดไปเข้ามา (แล้วสั่งลดระดับสิทธิลงเป็น user mode) และสั่งกระกระโดดไปทำงานในโพรเซสถัดไป กลไกทั้งหมดนี้เรียกว่า **context switch**



การทำงานตามขั้นตอนข้างต้นนี้จะเห็นได้ว่ามีความซับซ้อน ดังนั้นระบบคอมพิวเตอร์ที่ไม่ได้มีชุดคำสั่งที่รองรับอย่างเหมาะสม (เช่นการสั่งดัมป์เรจิสเตอร์ของซีพียูทั้งชุด หรือการสั่งสลับชุดเรจิสเตอร์ของซีพียูที่ใช้งานอยู่) จะเสียเวลาในการดำเนินการนี้อย่างมาก

ประเภทของการจัดสรรงาน

กลไกการจัดสรรทรัพยากรอาจกระทำได้ในระดับการเลือกที่จะดึงเอางานไหนเข้ามาทำ โดยงานที่รอคอยมักจะยังคงเก็บไว้ในหน่วยความจำสำรอง รอเวลาที่จะถูกโหลดขึ้นหน่วยความจำหลักเพื่อประมวล กลไกแบบนี้เรียกว่า Long-term scheduler หรือ job scheduler เมื่องานหนึ่งๆ ถูกโหลดขึ้นมาในรูปของโพรเซสแล้ว ระบบปฏิบัติการก็มีอีกกลไกที่จะเลือกว่าโพรเซสใดที่พร้อมที่จะเข้าครอบครองเวลาในซีพียูเพื่อประมวล (allocate CPU) กลไกนี้เรียกว่า CPU Scheduler หรือ Short-term scheduler

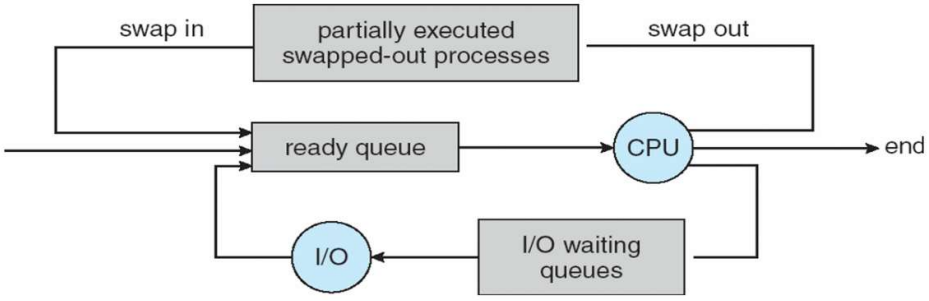
- ระบบปฏิบัติการแบบหลายภารกิจ (multitasking) ที่ใช้ในปัจจุบันโดยทั่วไปจะเป็นแบบ short-term scheduler เป็นหลัก (บางตัวจะมีกลไกนี้เพียงอย่างเดียวเท่านั้น)

- ความเร็วในการสลับงานเข้าครอบครองซีพียูของ short-term scheduler นั้นจะมีความรวดเร็วมากในระดับมิลลิวินาที เช่น อาจจะคำนวณไปเพียงไม่กี่มิลลิวินาทีก่อนถูกตัดจาก running state และอาจจะกลับมารันได้ใหม่ในทุกๆ รอบละ 100 มิลลิวินาที หรือรันๆ หยุดๆ ถึง 10 ครั้งต่อวินาที หรืออาจจะเร็วกว่านั้น เป็นต้น และเวลาที่ใช้ในการสลับโพรเซสเข้าครอบครองซีพียูอาจอยู่ในระดับไมโครวินาที ยกตัวอย่างเช่น หากมีงานกำลังรันอยู่ทั้งหมด 10 โพรเซส และแต่ละโพรเซส ใช้เวลารันเพียง สิบมิลลิวินาที และระบบปฏิบัติการใช้เวลาคัดเลือกงานและสลับโพรเซสเข้าออกด้วยเวลาประมาณหนึ่งมิลลิวินาทีต่อครั้ง เราจะพบว่าเวลาที่ใช้ต่อรอบจะเป็น 100 มิลลิวินาทีต่อ 10 โพรเซส และเวลาสลับอีก 10 มิลลิวินาที จะเห็นว่า เวลาประมาณ หนึ่งในสิบของเวลาที่ใช้กับซีพียู สูญเสียไปกับการทำ CPU scheduling เลยทีเดียว
- ความเร็วในการสลับงานของ long-term scheduler นั้นจะมีช่วงเวลายาวกว่า short-term อย่างมาก เช่นอยู่ในระดับวินาที หรือนาที โพรเซสที่ทำงานในลักษณะนี้ ระบบปฏิบัติการอาจจะยังคงเก็บตัวโพรเซสไว้ในหน่วยความจำหลัก หรืออาจจะโยนโพรเซสไปพักไว้ในหน่วยความจำสำรอง (swapping) แม้ว่าการเขียนหรืออ่านหน่วยความจำสำรองมีกลไกที่ซับซ้อนและเสียเวลามากกว่าการประมวลของซีพียูอย่างมาก แต่ผลที่ได้กลับมานั่นคือ เมื่อโพรเซสหนึ่งๆ ถูกโยนลงหน่วยความจำสำรองทั้งหมด ทำให้ดึงคืนหน่วยความจำหลักมาให้กับโพรเซสอื่นๆ ที่ระบบปฏิบัติการกำลังจัดการอยู่ได้ ส่งผลทำให้คอมพิวเตอร์สามารถรองรับงานทั้งหมดที่ใช้ทรัพยากรหน่วยความจำหลักที่รวมแล้วมีพื้นที่มากเกินกว่าที่ระบบมีได้ (เพราะโพรเซสทุกตัวมิได้ครอบครองหน่วยความจำหลักตลอดเวลา) ดังนั้นผลของการจัดการแบบ long-term scheduling จึงทำให้เราสามารถคุมจำนวนงานที่คอมพิวเตอร์สามารถรองรับได้ เรียกว่า **degree of multiprogramming** (จำนวนโพรเซสที่สามารถทำงานได้ ณ ขณะหนึ่ง ซึ่งรวมถึงโพรเซสที่ถูกพักไว้ในหน่วยความจำสำรองด้วย) ตัวอย่างของโพรเซสในลักษณะนี้อย่างเช่นโปรแกรมตรวจสอบไวรัสที่อาจจะถูกสั่งให้ตรวจสอบไฟล์ในหน่วยความจำสำรองวันละครั้ง เป็นต้น

เมื่อพิจารณาถึงอัตราส่วนของเวลาที่โพรเซสหนึ่ง จะตกอยู่ใน state ใด state หนึ่งขณะทำงาน เราอาจจะแบ่งชนิดของโพรเซสได้เป็น

- **I/O Bound Process** คือโพรเซสที่ใช้เวลาส่วนใหญ่อยู่ในการทำงานของ I/O มากกว่า โดยจะใช้ช่วงเวลาสั้นๆ ในการครอบครองซีพียู
- **CPU Bound Process** คือโพรเซสที่ใช้เวลาส่วนใหญ่ในการคำนวณ มีโอกาสน้อยมากที่จะเข้าสู่ state ของการใช้งาน I/O

ระบบปฏิบัติการปัจจุบัน ได้นำเอากลไกการกระทำ long-term scheduling เข้ามาใช้ร่วมกับการทำ short-term scheduling โดยผสมผสานการผลักโพรเซสใดที่ไม่จำเป็นต่อการทำงานในขณะนั้น (เช่นมีแต่การคำนวณคำสั่งรอคอย wait เป็นเวลานาน หรือมี priority ที่ต่ำ) นำลงเก็บในหน่วยความจำสำรอง (เป็นลักษณะการจัดการแบบ long-term scheduling โดยในที่นี้เราจะเรียกว่า swap out) และค่อยผลัดโหลดคืนจากหน่วยความจำสำรองขึ้นหน่วยความจำหลัก (swap in) เพื่อเข้าสู่ ready queue รอคอยการเข้าครอบครองซีพียูต่อไป แบบนี้เรียกว่า medium term scheduling



การจัดสรรงานในระบบปฏิบัติการสำหรับสมาร์ทโฟนและแท็ปเล็ต

อุปกรณ์ในประเภทสมาร์ทโฟน และแท็ปเล็ต มักจะถูกใช้งานโดยอาศัยพลังงานจากแบตเตอรีเป็นหลัก ระบบปฏิบัติการจึงต้องมีกลไกการประหยัดพลังงานเป็นพิเศษแตกต่างจากระบบปฏิบัติการที่ทำงานบนอุปกรณ์คอมพิวเตอร์ที่ทำงานโดยเชื่อมต่อกับไฟเลี้ยงเป็นหลัก กลไกต่างๆ ที่ถูกนำมาใช้งานกันมีอย่างเช่น

- การยอมให้มีเพียงโพรเซสผู้ใช้เพียงตัวเดียวเท่านั้นที่ทำงานในช่วงเวลาใดเวลาหนึ่ง โดยยึดจากโพรเซสที่กำลังใช้พื้นที่แสดงผล ณ ขณะนั้น ส่วนโพรเซสอื่นๆ จะถูกสั่งหยุดการทำงาน (blocked) ไว้ ทั้งนี้เพื่อให้เวลาที่เหลือจากการทำงานของระบบปฏิบัติการเองและโพรเซสหลักนี้ ถูกนำมาใช้เพื่อเปลี่ยนสถานะของซีพียูให้เข้าโหมดประหยัดพลังงาน ทำให้การใช้พลังงานลดลง ตัวอย่างที่เห็นในลักษณะนี้ก็เช่น iOS และแอนดรอยด์ ในเวอร์ชันแรกๆ
- การสั่งให้ระบบเข้าสู่โหมดประหยัดพลังงาน เมื่อพักหน้าจอ (เมื่อหน้าจอดับลงหลังจากผู้ใช้ไม่ได้ใช้งานสักระยะหนึ่ง) โพรเซสผู้ใช้ที่กำลังทำงานอยู่ก็จะถูกสั่งหยุดการทำงานไว้ชั่วคราว จนกว่าผู้ใช้จะสั่งการใดๆเพื่อใช้งานต่อไป
- การกำหนดให้มี foreground process (โพรเซสที่กำลังแสดงผลอยู่) และ background process (โพรเซสที่กำลังทำงาน แต่ไม่ได้แสดงผลอยู่เนื่องจาก foregroud process ใช้พื้นที่อยู่) ในลักษณะเช่นนี้ background process จะถูกสั่งรันในลักษณะที่จำกัด เช่นแค่ยอมรับอีเวนต์เท่านั้น หรือทำงานอื่นที่จำเป็นเช่น เล่นเสียง (เพลง) หรือให้ช่วงเวลาทำงานสั้นๆ
    - กรณีของแอนดรอยด์ ตัว background process จะอาศัยเซอร์วิส (service) ในการทำงานต่อไป แม้ว่าตัว background process ถูกสั่ง blocked/suspended ไว้
    - ตัวเซอร์วิสที่ไม่ต้องการพื้นที่แสดงผล ก็จะใช้พื้นที่หน่วยความจำน้อยเท่าที่จำเป็น

2.4 การทำงานของโพรเซส

การสร้างโพรเซส (Process Creation)

เมื่อระบบปฏิบัติการเริ่มต้นทำงาน จะสร้างโพรเซสแรกขึ้นมาเพื่อเตรียมพร้อมรับการทำงานของโปรแกรมผู้ใช้ โดยเมื่อผู้ใช้สั่งการ หรือกลไกภายในระบบปฏิบัติการสั่งการให้รันโปรแกรมผู้ใช้หรือโปรแกรมระบบใดๆ ตามสภาวะที่กำหนด จะสร้างโพรเซสใหม่ขึ้นมา และโพรเซสผู้ใช้สามารถสร้างโพรเซสผู้ใช้อื่นขึ้นมาได้อีก เราเรียกโพรเซสผู้สร้างว่า parent process และโพรเซสลูกว่า child process ดังนั้นเราอาจมองโครงสร้างของโพรเซสทั้งหมดมีความสัมพันธ์เหมือนโครงสร้างต้นไม้ ที่ความสัมพันธ์ของผู้สร้างและโพรเซสลูกมีต่อกัน

เมื่อโพรเซสหนึ่งๆ ถูกสร้างขึ้นมา ระบบปฏิบัติการจะกำหนดเลขหมายประจำโพรเซส (Process identifier หรือ Process ID หรือ pid) ขึ้นมาเพื่อใช้อ้างอิง

ลักษณะวิธีการแบ่งปันการใช้ทรัพยากรระหว่างโพรเซสพ่อแม่กับโพรเซสลูก

- โพรเซสพ่อแม่แบ่งปันการใช้ทรัพยากรทั้งหมดให้ลูก นั่นคือทั้งลูกและพ่อแม่ใช้ทรัพยากรร่วมกันทั้งหมด
- โพรเซสพ่อแม่แบ่งทรัพยากรบางส่วนให้ลูกใช้ได้ด้วย (และมีส่วนหนึ่งที่เป็นของตนเองไม่ใช้ร่วมกับลูก)
- ไม่แบ่งปันทรัพยากรระหว่างกันเลย (เป็นลักษณะที่พบเห็นได้โดยทั่วไปในปัจจุบัน)

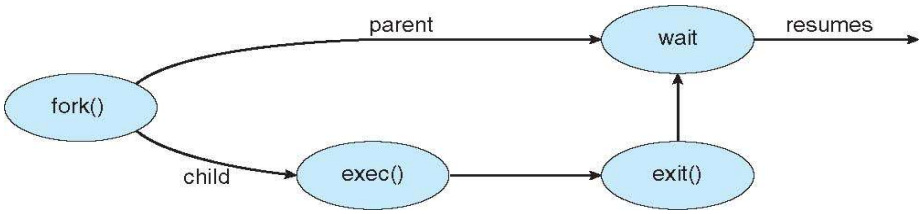
ลักษณะวิธีการประมวลระหว่างโพรเซสพ่อแม่กับโพรเซสลูก

- โพรเซสพ่อแม่ สร้างโพรเซสลูกขึ้นมาเป็นโพรเซสใหม่ และทั้งสองโพรเซสต่างรันต่อไปพร้อมกัน(concurrent processes)
- โพรเซสพ่อแม่ หยุดรอการทำงาน จนกว่าโพรเซสลูกจะจบการทำงาน โพรเซสพ่อแม่จึงทำงานต่อไป

ลักษณะวิธีการครอบครองพื้นที่หน่วยความจำระหว่างโพรเซสพ่อแม่กับโพรเซสลูก

- โพรเซสลูกใช้พื้นที่หน่วยความจำเดียวกันกับพ่อแม่ ทำให้สามารถใช้ข้อมูลและชุดคำสั่งต่างๆ ของพ่อแม่ได้อย่างเต็มที่
- โพรเซสลูกได้รับพื้นที่หน่วยความจำใหม่เป็นของตนเอง ทำให้ข้อมูลและชุดคำสั่งเป็นคนละชุดกับของพ่อแม่ได้ (แม้ว่าในทางปฏิบัติอาจจะได้โค้ดมาจากโปรแกรมเดียวกันก็ได้)

ตัวอย่างการทำงานในลินุกซ์/ยูนิกซ์ ฟังก์ชัน **fork()** จะทำหน้าที่แตกโพรเซสใหม่ขึ้นมา โดยโพรเซสลูกจะได้พื้นที่หน่วยความจำใหม่เป็นของตนเอง แต่ข้อมูลภายในนั้นจะเป็นสำเนาของโพรเซสพ่อแม่ ดังนั้นการทำงานต่อไปจะมีกลไกเหมือนเดิม เว้นเพียงแต่กรณีที่นิยามตัวแปรเพื่อใช้เก็บข้อมูลที่ส่งกลับมาจาก fork() ของโพรเซสพ่อแม่จะได้ค่า pid ของโพรเซสลูก แตกต่างจากจากตัวแปรตัวเดียวกันของโพรเซสลูกที่ฟังก์ชัน fork() ส่งค่า 0 กลับมาให้ ดังที่เห็นจากโค้ดตัวอย่างในปฏิบัติการท้ายบท จะเห็นว่าในการทำงานของโพรเซสลูก เราใช้ฟังก์ชัน **exec()** ซึ่งจะทำหน้าที่โหลดชุดคำสั่งใหม่จากโปรแกรมในหน่วยความจำสำรองมาเริ่มต้นการทำงานใหม่โดยใช้พื้นที่หน่วยความจำเดิม *(จบการทำงานของโค้ดเดิมลงและเริ่มการทำงานโพรเซสใหม่)* ในขณะที่ชุดคำสั่งที่ประมวลต่อเนื่องของโปรแกรมพ่อแม่ อาจเรียกฟังก์ชัน **wait()** เพื่อสั่งให้ตัวโพรเซสหยุดรอไปจนกว่าโพรเซสลูกจบการทำงาน จากนั้นโพรเซสพ่อแม่จึงทำงานต่อเนื่องไปจากที่ค้างไว้ ดังรูป



การจบการทำงานของโพรเซส (Process termination)

เมื่อโพรเซสทำงานคำสั่งสุดท้ายเสร็จสิ้น ก็จะแจ้งให้ระบบปฏิบัติลบโพรเซสตนเองทิ้งด้วย **exit()** *(ในการเขียนโปรแกรมภาษาระดับสูง แม้ว่าเรามิได้ใช้ฟังก์ชัน exit() ให้เห็นเด่นชัด แต่คอมไพเลอร์ก็จะมีกลไกการเติมฟังก์ชันนี้ให้โดยอัตโนมัติ)* ทั้งนี้ โพรเซสดังกล่าวก็จะสามารถส่งค่ากลับคืน (integer value) ไปยังโพรเซสพ่อแม่ *(ซึ่งอาจจะเป็นระบบปฏิบัติการ หรือโพรเซสพ่อแม่ที่ fork() โพรเซสดังกล่าวออกมา)* ในกรณีที่โพรเซสพ่อแม่เป็นผู้ fork โพรเซสลูก โพรเซสพ่อแม่ก็สามารถรับค่ากลับคืนผ่านทางการส่งกลับของฟังก์ชัน wait() เมื่อโปรเซลจบการทำงาน ทรัพยากรทั้งหมดของโพรเซสก็จะถูกดึงคืนกลับโดยระบบปฏิบัติการเพื่อนำไปใช้ในงานอื่นต่อไป

การจบการทำงานของโพรเซสใดๆ อาจจบการทำงานโดยการสั่งการจากโพรเซสอื่นหรือระบบปฏิบัติการได้ ซึ่งการจบการทำงานแบบนี้เป็นลักษณะที่ไม่ปกติ (abort) ซึ่งมีกรณีต่างๆ ดังเช่น

- โพรเซสดังกล่าวจบการทำงานแบบไม่ปกติด้วยตนเอง (เช่นในภาษาซีใช้ฟังก์ชัน **abort()**)
- ในกรณีที่โพรเซสหนึ่งๆ พยายามเข้าถึงทรัพยากรที่ตนมิได้รับอนุญาต หรือการคำนวณเกิด exception ที่ทำให้ระบบปฏิบัติการตัดสินใจหยุดการทำงานของโพรเซส
- โพรเซสพ่อแม่ หรือระบบปฏิบัติการเห็นแล้วว่าโพรเซสที่กำลังรันอยู่นั้น ไม่มีความจำเป็นที่จะต้องรันอีกต่อไปแล้ว (สังเกตว่าการสั่งจบการทำงานโดยโพรเซสอื่นนั้น จะกระทำโดยโพรเซสที่สร้างโพรเซสปัจจุบันขึ้นมา หรือตัวระบบปฏิบัติการ ระบบปฏิบัติการตามปกติจะไม่อนุญาตให้โพรเซสใดๆ จบการทำงานของโพรเซสใดๆ ก็ได้เพราะจะทำให้เป็นจุดอ่อนในการโจมตีระบบได้)
- โพรเซสพ่อแม่จบการทำงาน ในกรณีเช่นนี้โพรเซสลูกก็จะจบการทำงานโดยอัตโนมัติ (ยกเว้นในกรณีที่ระบบปฏิบัติการอนุญาตให้โพรเซสลูกสามารถทำงานต่อไปได้แม้โพรเซสพ่อแม่จบการทำงาน ในกรณีเช่นนี้ ระบบปฏิบัติการก็จะรับ

หน้าที่เป็นโพรเซสพ่อแม่ให้ต่อไป ในโครงสร้างของการออกแบบโพรเซสให้มีโพรเซสลูกและโพรเซสหลาน (ที่สร้างจากลูก) เป็นโครงข่ายซับซ้อน เมื่อโพรเซสพ่อแม่จบ โพรเซสในโครงข่ายทั้งหมดก็จะต้องจบการทำงานตามไปด้วยทั้งหมด เราเรียกว่า cascading termination

- กรณีการจบแบบไม่ปกติ
    - โพรเซสพ่อแม่ไม่รอให้ลูกจบ แต่จบการทำงานไปก่อน เหลือโพรเซสลูกที่ทำงานต่อเนื่องไปโดยไม่มีพ่อแม่ เรียกโพรเซสลูกนี้ว่า orphan
    - โพรเซสพ่อแม่ไม่รอให้ลูกจบ แต่ได้จบการทำงานไปก่อน ส่งผลทำให้โพรเซสลูกจบการทำงานโดยยังคงถือครองทรัพยากรไว้อยู่ เรียกโพรเซสลูกนี้ว่า zombie
- ในกรณีระบบปฏิบัติการสำหรับสมาร์ทโฟนและแท็ปเล็ต ซึ่งยอมให้มีหลายโพรเซสคงทำงานอยู่ในลักษณะของ background process อยู่ จะส่งผลทำให้พื้นที่หน่วยความจำหมดลงไปเรื่อยๆ (ทั้งนี้เนื่องจากอุปกรณ์เหล่านี้ใช้หน่วยความจำสำรองแบบ solidstate ซึ่งไม่เหมาะสมต่อการเขียนทับจำนวนมากๆ จึงไม่มีกลไกการสร้าง swap space ดังเช่นระบบปฏิบัติการโดยทั่วไป) ในลักษณะเช่นนี้ ระบบปฏิบัติการจะเลือกสั่งหยุดการทำงาน background process ที่ไม่จำเป็นลง อย่างกรณีของแอนดรอยด์ จะมีการจัดลำดับความสำคัญดังนี้
    - Foreground process โพรเซสที่ใช้พื้นที่แสดงผลหลัก
    - Visible process โพรเซสที่ใช้พื้นที่แสดงผลร่วมด้วยแต่ไม่ใช่ตัวหลัก
    - Service process เซอร์วิสที่ทำงานเป็นพื้นหลัง เช่นโพรเซสที่คอยดึงข้อมูลหรือรับข้อมูลจากระบบเครือข่าย
    - Background process โพรเซสที่ถูกปิดการแสดงผลลง (เมื่อเราเปิด Foreground process ตัวใหม่ หรือกดปุ่มคืนสู่หน้า UI หลัก)
    - Empty process โพรเซสที่ไม่มีการทำงานใดๆ ต่อไปแล้ว
- สำหรับเบราว์เซอร์ที่มีใช้ในปัจจุบัน บางตัวอาจทำงานในลักษณะของโพรเซสเดียว แต่ส่วนมากจะแบ่งกลไกออกเป็นหลายโพรเซส
    - Browser เป็นโพรเซสหลักที่ทำหน้าที่แสดงหน้าต่างหลักของเบราว์เซอร์ เมนู การจัดการดิสก์และ I/O
    - Renderer โพรเซสที่ทำหน้าที่แสดงหน้าเว็ป จัดการกับภาษา HTML และภาษาอื่นๆ ที่ต้องใช้ประกอบการแสดงผล
    - Plug-in เป็นโพรเซสที่จัดการกับปลักอินเสริมต่างๆ ของเบราว์เซอร์

2.5 การสื่อสารระหว่างโพรเซส (Interprocess communication)

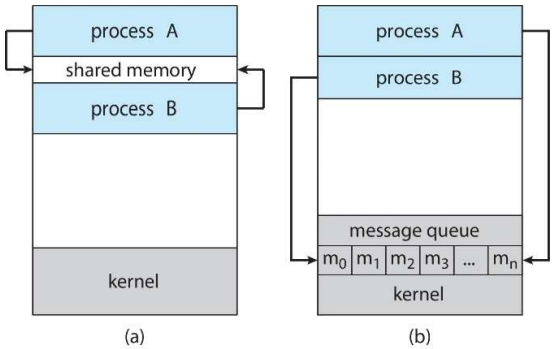
ในระบบปฏิบัติการแบบหลายงาน โพรเซสต่างๆ ในระบบอาจจะทำงานเป็นอิสระจากกันโดยไม่เกิดการรับส่งข้อมูลหรือมีสัญญาณใดๆ ส่งถึงกัน (independent) หรืออาจจะมีโพรเซสสองตัวหรือมากกว่า ทำงานสอดประสานกันเพื่อจุดประสงค์ใดจุดประสงค์หนึ่ง (cooperating) ในการทำงานประสานกันนั้น กลไกการทำงานของโพสเซสหนึ่งๆ ก็จะส่งผลต่อการทำงานของโพรเซสที่ทำงานประสานกันอยู่ด้วย

วัตถุประสงค์ในการประสานงานระหว่างโพรเซส

- Information sharing เพื่อแลกเปลี่ยนข้อมูลระหว่างกัน (เช่นในกรณีตัวอย่างง่ายๆ นาย ก. สร้างเอกสารหนึ่งขึ้นมา นำเอกสารใส่ไฟล์เซอร์เวอร์ เพื่อให้นาย ข. ที่ใช้คอมพิวเตอร์อีกเครื่องหนึ่งอ่านไฟล์ขึ้นมาดู) ระบบปฏิบัติการจำเป็นต้องออกแบบโครงสร้างการสื่อสารส่งต่อข้อมูลระหว่างกัน เพื่อให้ข้อมูลสามารถส่งไปและมาได้อย่างสะดวกและเป็นมาตรฐานเดียวกัน
- Communication Speedup การลดเวลาประมวลข้อมูล ในระบบคอมพิวเตอร์ที่มีซีพียูมากกว่าหนึ่งหน่วย การคำนวณข้อมูลที่มีปริมาณมาก อาจจัดแบ่งข้อมูลบางส่วนไปให้ซีพียูแต่ละตัวช่วยกันคำนวณ โดยสร้างโพรเซสขึ้นหลายโพรเซสเพื่อส่งไปให้ซีพียูแต่ละตัวได้ใช้งาน หรือในกรณีที่งานของผู้ใช้ ประกอบไปด้วยการคำนวณส่วนหนึ่ง (เน้น CPU) กับการรับส่งข้อมูลไปยังดีไวซ์จำนวนมากอีกส่วนหนึ่ง (เน้น I/O) ลักษณะเช่นนี้การแยกโพรเซสก็จะมีความสะดวกในการเขียนโปรแกรมและการจัดการของระบบปฏิบัติการ และเมื่อตัวโพรเซสในส่วนคำนวณไม่ต้องรอส่วน I/O (ตามการออกแบบโพรเซสปกติ เมื่อต้องรอ I/O การคำนวณที่จะตามมาของโพรเซสนั้นๆ ก็หยุดชะงัก รอจนกว่าการประมวล I/O เสร็จสิ้น) ก็จะทำให้ผลลัพธ์คือผู้ใช้เห็นว่าการทำงานของโปรแกรมตนนั้นเร็วขึ้น
- Modularity การแบ่งโปรแกรมออกเป็นโครงสร้างโมดูลด้วยการแบ่งเป็นหลายโพรเซส ในลักษณะเช่นนี้ ทำให้ผู้พัฒนาซอฟต์แวร์สามารถออกแบบโปรแกรมออกเป็นส่วนๆ และบางส่วนอาจจะสามารถใช้โค้ดซ้ำ (เช่นส่วนการคำนวณข้อมูลขนาดใหญ่) ทำให้การพัฒนาโปรแกรมทำได้เป็นระบบและง่ายกว่าการเขียนโปรแกรมเพียงตัวเดียวที่มีขนาดใหญ่
- Convenience การแบ่งโปรแกรมออกเป็นส่วนๆ เพื่อความสะดวกสบายของผู้ใช้งาน เช่นผู้ใช้โปรแกรมพิมพ์เอกสาร เมื่อสั่งพิมพ์เอกสาร จะเกิดโพรเซสเพื่อพิมพ์เอกสารขึ้นมาแยกต่างหาก เพื่อพิมพ์เอกสารไป ในขณะที่ผู้ใช้โปรแกรมก็สามารถอ่านเอกสารนั้นต่อไปได้โดยไม่ต้องหยุดรอจนกว่าเอกสารจะพิมพ์เสร็จ เป็นต้น

การที่โพรเซสสองโพรเซสจะสื่อสารกันได้ ในวิธีที่ง่ายที่สุดก็คือการอ่านเขียนไฟล์ในหน่วยความจำสำรองร่วมกัน แต่วิธีที่เหมาะสมมีประสิทธิภาพ จำต้องมีกลไกการติดต่อสื่อสารระหว่างโพรเซสที่ควบคุมโดยระบบปฏิบัติการ เราเรียกกลไกนี้ว่า Interprocess communication (IPC) โดยกลไกที่นิยมใช้กันอยู่มีสองรูปแบบดังนี้

- การใช้พื้นที่หน่วยความจำ(บางส่วน) ร่วมกันระหว่างโพรเซส (shared memory)
- การใช้ช่องทางการติดต่อสื่อสารผ่านทางระบบปฏิบัติการ (messsage passing)



a) การใช้พื้นที่หน่วยความจำร่วม และ b) การใช้แมสเซสคิวของระบบปฏิบัติการ

<u>ประเด็นปัญหาเรื่อง ผู้ผลิต-ผู้บริโภค (Producer-Consumer Problem</u>

เป็นแนวประเด็นปัญหาเกี่ยวกับการติดต่อรับส่งข้อมูลกันระหว่างโพรเซสที่มีลักษณะคล้ายกับกลไกของผู้ผลิตและผู้บริโภค กล่าวคือ ผู้ผลิต (Producer) ทำหน้าที่ให้ข้อมูล ในขณะที่ ผู้บริโภค (Consumer) ทำหน้าที่รับข้อมูล ดังนั้น ลักษณะความสัมพันธ์ของโพรเซสทั้งสองมีลักษณะที่ข้อมูลจะวิ่งไปในทางเดียวจากผู้ผลิตไปยังผู้บริโภค โดยการส่งผ่านข้อมูลนั้น จะส่งผ่าน

พื้นที่หน่วยความจำที่ใช้ร่วมกัน หรือส่งผ่านแมสเซสคิวที่เปิดเชื่อมถึงกัน เมื่อมองถึงจำนวนพื้นที่พักข้อมูล/คิวที่ใช้ส่งถึงกันนี้ เราสามารถแจงรายละเอียดออกได้เป็น

- Unbounded-buffer พื้นที่ส่งข้อมูลมีโครงสร้างที่รองรับข้อมูลได้ไม่จำกัด
    - ส่งผลทำให้ผู้ผลิตไม่จำเป็นต้องรอ ในกรณีที่พื้นที่พักข้อมูลเต็ม
    - ผู้บริโภคต้องรอรับข้อมูล หากไม่มีข้อมูลในพื้นที่พักข้อมูล
- Bounded-buffer พื้นที่ส่งข้อมูลมีโครงสร้างขนาดจำกัด ทำให้รับฝากข้อมูลได้ในปริมาณที่จำกัด เช่นใช้ อะเรย์ที่จำลองโครงสร้างเป็น Circular queue เป็นต้น
    - ผู้ผลิตต้องรอส่งข้อมูล หากพื้นที่พักข้อมูลเต็ม
    - ผู้บริโภคต้องรอรับข้อมูล หากไม่มีข้อมูลในพื้นที่พักข้อมูล

สิ่งที่ต้องระวังในการจัดการในประเด็นปัญหานี้คือ จะต้องเกิดการประสานกันระหว่างโพรเซส (synchronize) เพื่อให้แน่ใจว่า ผู้ผลิตจะต้องสร้างข้อมูลไปวางไว้ก่อน ที่ผู้บริโภคจะไปหยิบข้อมูลชิ้นนั้นมาได้

<u>การสื่อสารระหว่างโพรเซสด้วยวิธีการใช้พื้นที่หน่วยความจำร่วม</u>

- อาศัยการจองพื้นที่หน่วยความจำเพิ่มเติมมาจากระบบปฏิบัติการ และกำหนดให้ใช้ร่วมกันระหว่างโพรเซส (สามารถเข้าถึงและเขียน/อ่านได้จากทุกโพรเซสที่เกี่ยวข้อง)
- กลไกการควบคุมการเขียน/อ่าน ส่งผ่านข้อมูลระหว่างกัน ดำเนินการโดยโพรเซสที่เกี่ยวข้อง โดยที่ระบบปฏิบัติการไม่เข้าไปยุ่งเกี่ยว (ระบบปฏิบัติการทำหน้าที่เพียงจัดสรรพื้นที่/แมปพื้นที่ และรับมอบพื้นที่คืนเมื่อไม่ใช้งานแล้วเท่านั้น)
- ปัญหาที่จะพบเพิ่มเติมจากกรณีนี้ก็คือจะต้องมีกลไกเพื่อทำให้เกิดการเขียนอ่านที่สามารถส่งข้อมูลถึงกันได้อย่างถูกต้อง (สามารถเข้าจังหวะ - synchronize) กันได้ (กลไกการจัดการการเข้าจังหวะจะได้เรียนต่อไปในบทถัดๆ ไป)

> ***<u>หมายเหตุ</u>*** *ในเอกสารประกอบการเรียนนี้ ใช้การอธิบายทฤษฎีต่างๆ ที่เกี่ยวข้อง ในรูปโครงสร้างที่เลียนแบบภาษาซี การอธิบายกลไกต่างๆ นี้ ไม่ได้หมายถึงจะต้องเขียนโปรแกรมในลักษณะดังกล่าว แต่จะเป็นเพียงการอธิบายหลักการโดยคร่าวๆ เท่านั้น นักศึกษาจะต้องแยกความหมายระหว่างตัวขั้นตอนวิธีที่อธิบายถึงกลไกในรายละเอียด กับแนวคิดของโครงสร้างโปรแกรมออกจากกันด้วย ตัวอย่างเช่น วนรอบ while(true){ } ที่มักจะพบได้ในเอกสาร จะหมายถึง "วัฏจักร" ของการประมวลผลของขั้นตอนวิธี ว่ามีลักษณะวนเวียนเช่นนี้เรื่อยไป (ในทางปฏิบัติเรามักจะมีกลไกที่จะต้องมีเส้นทางออก แต่ในตัวขั้นตอนวิธีที่อธิบายจะไม่ได้เขียนไว้ เป็นต้น)*

<u>ตัวอย่างการส่งผ่านข้อมูลระหว่างโพรเซสผู้ผลิตและผู้บริโภคผ่านพื้นที่หน่วยความจำร่วม</u>

โครงสร้างข้อมูลในพื้นที่หน่วยความจำร่วม

```
#define BUFFER_SIZE 10        //จำนวนหน่วยข้อมูลที่จะจัดเก็บได้สูงสุด
typedef struct{
                ....          //รายละเอียดการนิยามของหน่วยข้อมูลแต่ละตัว
}item;

item buffer[BUFFER_SIZE];     // จองพื้นที่อะเรย์ขนาด BUFFER_SIZE ข้อมูล
int in = 0;                   // ตัวชี้ตำแหน่งนำข้อมูลเข้าบัปเฟอร์
int out = 0;                  // ตัวชี้ตำแหน่งนำข้อมูลออกจากบัปเฟอร์
```

ตัวอย่างต่อไปนี้จะใช้กลไกการสร้างคิววงกลมจากอะเรย์ ซึ่งจากหลักการที่นักศึกษาได้เรียนมาแล้วในรายวิชาโครงสร้างข้อมูลและขั้นตอนวิธี จะพบว่าโครงสร้างนี้จะเก็บข้อมูลได้สูงสุดเท่ากับ BUFFER_SIZE - 1 ตัวด้วยกัน (เพื่อไม่ให้เกิดความสับสนระหว่างการพิจารณาในกรณีที่พื้นที่เต็มกับกรณีพื้นที่ว่าง-ไม่มีข้อมูลเลย)

โพรเซสผู้ผลิต

```
while(true){
    ผลิตข้อมูลขึ้นมาใหม่หนึ่งหน่วย
    while(((in+1) % BUFFER_SIZE) == out)  //ตรวจสอบกรณีคิวเต็ม
        ;    //ดำเนินการการวนรอในขณะที่คิวเต็ม
    buffer[in] = หน่วยใหม่;
    in = (in +1) % BUFFER_SIZE;      //เลื่อนตัวชี้เก็บข้อมูลไปยังตำแหน่งถัดไป
}
```

โพรเซสผู้บริโภค

```
while(true){
    while(in == out)  //ตรวจสอบกรณีคิวว่าง
        ;    //ดำเนินการการวนรอในขณะที่คิวว่าง
    ตัวแปรรับข้อมูลไปใช้งาน = buffer[out];
    out = (out +1) % BUFFER_SIZE;      //เลื่อนตัวชี้อ่านข้อมูลไปยังตำแหน่งถัดไป
   นำข้อมูลจากตัวแปรไปใช้งาน
}
```

จากกรณีข้างต้น เราพบว่าจะต้องเสียพื้นที่เก็บข้อมูลไปหนึ่งหน่วย เนื่องจากข้อจำกัดที่เราไม่สามารถตรวจสอบภายในตัวโครงสร้างข้อมูลว่าสิ่งที่อยู่ภายในเป็นข้อมูลหรือไม่ เราทำได้โดยเช็คจากตัวแปร in และ out ซึ่งถ้าเราเก็บข้อมูลจนเต็ม เราจะพบว่าตัวชี้ in และ out จะกลับมาชี้ตำแหน่งเดียวกัน และทำให้เราไม่สามารถแยกแยะกรณีคิวเต็มและคิวว่างออกจากกันได้

ด้วยเหตุนี้ เราอาจจะอาศัยการนิยามตัวแปรเพิ่มอีกหนึ่งตัวเพื่อใช้นับจำนวนข้อมูลที่มีอยู่ในคิววงกลม โดยการนี้จะทำให้เราสามารถเก็บข้อมูลได้ครบตามขนาดของอะเรย์ที่จองไว้ได้ ขั้นตอนวิธีจะเปลี่ยนไปดังนี้

counter = 0; // กำหนดค่าเริ่มต้นให้กับตัวแปรนับเป็นศูนย์ ก่อนเริ่มทำงาน

โพรเซสผู้ผลิต

```
while(true){
    ผลิตข้อมูลขึ้นมาใหม่หนึ่งหน่วย
    while(counter == BUFFER_SIZE)  //ตรวจสอบกรณีคิวเต็ม
        ;    //ดำเนินการการวนรอในขณะที่คิวเต็ม
    buffer[in] = หน่วยใหม่;
    in = (in +1) % BUFFER_SIZE;      //เลื่อนตัวชี้เก็บข้อมูลไปยังตำแหน่งถัดไป
    counter ++;
}
```

โพรเซสผู้บริโภค

```
while(true){
    while(counter == 0)  //ตรวจสอบกรณีคิวว่าง
          ;     //ดำเนินการการวนรอในขณะที่คิวว่าง
    ตัวแปรรับข้อมูลไปใช้งาน = buffer[out];
    out = (out +1) % BUFFER_SIZE;      //เลื่อนตัวชี้อ่านข้อมูลไปยังตำแหน่งถัดไป
    counter --;
   นำข้อมูลจากตัวแปรไปใช้งาน
}
```

<u>การสื่อสารระหว่างโพรเซสด้วยวิธีการส่งผ่านเมสเสจ</u>

- เป็นกลไกที่อาศัยการบริหารจัดการด้วยระบบปฏิบัติการ ที่จะเป็นผู้สร้างช่องทางการติดต่อสื่อสารระหว่างโพรเซส ดังนั้นจึงไม่ต้องมีการกำหนดพื้นที่หน่วยความจำร่วมกัน
- กลไกของการส่งผ่านเมสเสจ จะมีโอเปอเรชันที่สำคัญสองตัวได้แก่
    - send(message) ซึ่งเมสเสจอาจจะมีขนาดคงที่หรือไม่คงที่แล้วแต่วิธีการ
    - receive(message)
- ขั้นตอนการส่งผ่านเมสเสจ
    - เริ่มจากการสร้างช่องทางของเมสเสจระหว่างโพรเซสทั้สองตัว
    - รับส่งข้อมูลซึ่งกันและกัน
- โครงสร้างของเส้นทางที่ใช้ส่งเมสเสจ
    - อาจจะเป็นเส้นทางแบบกายภาพ เช่นส่งผ่านพื้นที่หน่วยความจำที่ใช้ร่วมกัน หรือส่งผ่านช่องทางติดต่อสื่อสารทางฮาร์ดแวร์เป็นการเฉพาะ (hardware bus)
    - อาจจะเป็นเส้นทางที่จำลองขึ้นโดยซอฟต์แวร์ระบบ (ตัวระบบปฏิบัติการเอง) (logical)
- ในการประยุกต์ใช้งาน นักพัฒนาโปรแกรมจำเป็นต้องคำนึงถึงประเด็นต่างๆ ดังเช่น
    - จะสร้างเส้นทางส่งเมสเสจอย่างไร ใช้กลไกอะไร
    - หากมีโพรเซสมากกว่าสองตัวที่จะติดต่อสื่อสารกัน จะสร้างเส้นทางอย่างไร
    - จะต้องมีเส้นทางกี่เส้นสำหรับการติดต่อระหว่างโพรเซสสองตัว
    - เส้นทางดังกล่าวจะมีขีดความสามารถรองรับปริมาณข้อมูลได้มากน้อยขนาดไหน
    - เส้นทางดังกล่าวจะรับข้อมูลประเภทใด เป็นข้อมูลที่มีขนาดคงที่ หรือมีโครงสร้างที่แปรเปลี่ยนยืดหยุ่นได้
    - เส้นทางเป็นแบบทิศทางเดียวหรือสองทิศทาง
- สำหรับการติดต่อสื่อสารโดยตรง (Direct communication) (ระหว่างโพรเซสกับโพรเซสโดยตรง) จะต้องกำหนดโพรเซสปลายทางที่จะต้องติดต่อสื่อสารด้วย
    - send(P,message) ส่งเมสเซสไปยังโพรเซส P
    - receive (Q,message) รับเมสเซสจากโพรเซส Q
    - ในลักษณะการเรียกใช้การรับส่งเมสเสสแบบนี้ เมื่อโพรเซสผู้ส่ง เรียกใช้ system call เพื่อส่งเมสเซสไปยังโพรเซลปลายทาง ระบบปฏิบัติการจะต้องสร้างเส้นทางเชื่อมต่อโดยอัตโนมัติ และด้วยกลไกในลักษณะนี้ การส่ง

จากโพรเซสหนึ่งไปยังอีกโพรเซสหนึ่งจะกระทำได้เพียงเส้นทางเดียว และมีทิศทางทิศเดียว (หากโพรเซสปลายทางจะส่งข้อมูลกลับ จะสร้างเส้นทางอีกเส้นเพื่อส่งกลับ)

- สำหรับการติดต่อสื่อสารทางอ้อม (Indirect communication) การติดต่อสื่อสารจะอาศัยการฝากข้อความไว้ที่ระบบปฏิบัติการในลักษณะแบบจดหมาย (mailbox หรือ port)
    - กล่องจดหมายแต่ละอันจะมีเลข ID กำกับเป็นการเฉพาะ
    - โพรเซสทั้งสองฝ่ายที่จะติดต่อสื่อสารกัน จะต้องรับรู้เลขหมายของกล่องจดหมายนี้ตรงกัน จึงจะส่งข้อความผ่านไปถึงกันได้
    - ในลักษณะการเรียกใช้การรับส่งเมสเซสแบบนี้ เส้นทางการเชื่อมต่อระหว่างโพรเซสจะเกิดขึ้นอย่างสมบูรณ์เมื่อทั้งสองฝ่ายต่างเชื่อมเส้นทางไปยัง mailbox เดียวกัน แต่ทั้งนี้ ในทางปฏิบัติอาจจะมีโพรเซสที่สามที่สี่ ฯลฯ เข้ามาเชื่อมผ่าน mailbox ตัวนี้ได้ด้วย ทำให้เส้นทางนี้กลายเป็นโครงข่ายที่ส่งข้อมูลถึงกันได้ระหว่างโพรเซสทั้งหมดที่ใช้ mailbox ร่วมกัน และข้อมูลสามารถส่งไปยัง mailbox หรืออ่านจาก mailbox ก่อให้เกิดเส้นทางแบบสองทิศทาง รวมทั้งกลไกการใช้ mailbox เป็นจุดพักข้อมูลกลาง โพรเซสคู่หนึ่งอาจจะติดต่อระหว่างกันโดยใช้ mailbox มากกว่าหนึ่ง ID ได้
    - กลไกการติดต่อ
        - โพรเซสสร้าง mailbox ขึ้นใหม่ (ร้องขอพื้นที่จากระบบปฏิบัติการ)
        - รับส่งข้อมูลโดยอาศัย mailbox ดังกล่าว
        - ทำลาย mailbox เมื่อไม่ใช้งานแล้ว (คืนพื้นที่โครงสร้างกลับให้แก่ระบบปฏิบัติการ)
        - send (A, message) ส่งข้อความไปยัง mailbox A
        - receive (A, message) อ่านข้อความจาก mailbox A
    - ลักษณะการติดต่อกันระหว่างโพรเซสหลายโพรเซสด้วย mailbox สามารถมีความซับซ้อนมากขึ้นได้เช่น มี P1 P2 และ P3 เป็นโพรเซสที่ใช้ mailbox A ร่วมกัน สมมติว่าหาก P1 เป็นผู้ส่ง และ P2 กับ P3 เป็นผู้รับข้อความในลักษณะเช่นนี้จะเห็นได้ว่า จะต้องมีกลไกเพิ่มเติมเพื่อจะตัดสินว่า ข้อความที่ส่งมาจะเป็นของผู้ใดกันแน่ หรืออาจจะเป็นของทั้งสองฝ่าย ทางแก้ไขในประเด็นนี้มีได้ดังเช่น
        - อย่าสร้าง mailbox ที่ใช้ร่วมกันมากกว่าสองโพรเซส
        - หากจะสร้าง mailbox ที่ใช้ร่วมกันมากกว่าสองโพรเซส ควรจะมีโพรเซสเพียงโพรเซสเดียวในช่วงเวลาใดเวลาหนึ่ง ที่ทำหน้าที่ในการอ่าน mailbox (และแน่นอนว่า ผู้ส่งควรจะรับทราบว่าในช่วงเวลานั้นๆ โพรเซสใดกำลังทำหน้าที่รอรับเมสเซส) และกลไกในการจะแจ้งว่าใครจะเป็นผู้รับอาจจะกระทำโดยผู้ส่งเมสเซส เพื่อจะได้ให้แน่ใจว่าใครคือผู้รับ

<u>การเข้าจังหวะระหว่างโพรเซส (process synchronization)</u>

- ในกลไกการ send และ receive เมจเซสนั้น ผู้รับและผู้ส่งจะอยู่ในสภาวะที่จะต้องรอกัน (blocking) หรือไม่รอกัน (non-blocking) ดังเช่น
    - Blocking Send ผู้ส่งเมสเซสจะเข้าสู่สภาวะรอ จนกว่าผู้รับเมสเซสสามารถรับข้อมูลทั้งหมดไปแล้ว
    - Nonblocking Send เมื่อผู้ส่งเมสเซสส่งเมสเซสไปแล้ว จะไปทำงานอื่นๆ ต่อไปได้ในทันที
    - Blocking receive ผู้รับเมสเซสเมื่อพยายามอ่านเมสเซสแล้วพบว่าไม่มีข้อมูล ก็จะหยุดรอจนกว่าจะมีข้อมูลมาถึง จึงอ่านข้อมูลแล้วไปทำสิ่งอื่นๆ ต่อ
    - Nonblocking receive ผู้รับเมสเซสจะอ่านเมสเซสไม่ว่าจะมีข้อมูลรออ่านอยู่หรือไม่ (หากไม่มีข้อมูลรออ่านผลการอ่านก็จะมีค่า null หมายถึงไม่มีข้อมูลปรากฏเลย)

- การผสมผสานกลไกข้างต้น ก่อให้เกิดลักษณะการประสานงานระหว่างโพรเซสดังนี้
  - **การรับส่งข้อมูลแบบเข้าจังหวะ (Synchronous)** ในลักษณะนี้จะเป็น blocking send/blocking receive นั่นคือเมื่อผู้ส่งเมสเซสส่งข้อมูลแล้วก็จะหยุดรอจนกว่าผู้รับจะรับข้อมูล แล้วจึงทำงานต่อ ในขณะที่ผู้รับ เมื่อร้องขอข้อมูล ก็จะรอจนกว่าข้อมูลถูกส่งมาจากผู้ส่ง จึงจะรับข้อมูลแล้วดำเนินการต่อ ทำให้โพรเซสทั้งสองทำงานโดยประสานงานกันในจังหวะของการรับส่งข้อมูลดังกล่าว (ณ เวลาดังกล่าว เราจะทราบได้ว่า โพรเซสทั้งสองกำลังทำงานถึงจุดที่กำหนดของการรับส่งข้อมูล)
  - **การรับส่งข้อมูลแบบไม่เข้าจังหวะ (Asynchronous)** ในลักษณะนี้เป็น nonblocking send/nonblocking receive นั่นคือการรับส่งข้อมูลสามารถกระทำได้โดยไม่ส่งผลกระทบต่อขั้นตอนการทำงานของโพรเซสทั้งสองที่จะมีจุดที่เราทราบได้ว่ากำลังทำงานถึงจุดรับส่งข้อมูลดังกล่าว หรือพูดอีกอย่างว่า แต่ละโพรเซสจะทำงานไปตามขั้นตอนการทำงานของตนไม่ขึ้นต่อกันและกัน
  - หากเราเอากรณีของ producer-consumer มาใช้กับกลไกของเมสเซส การเขียนโค้ดที่ง่ายที่สุด ที่กำหนดว่าจะมีข้อมูลเพียงหนึ่งชุดที่ส่งจาก producer ไปยัง consumer ในแต่ละครั้งของการใช้ข้อมูล เราอาจจะสร้างระบบที่เป็นแบบ synchronous เพื่อให้ producer สร้างข้อมูลแล้วหยุดรอจนกว่า consumer รับข้อมูลไปจัดการแล้ว producer จึงค่อยสร้างข้อมูลใหม่เพื่อส่งต่อไป ในขณะที่ consumer เมื่อนำข้อมูลไปใช้แล้ว ก็จะต้องหยุดรอจนกว่าจะได้ข้อมูลใหม่มาจาก producer

- ในการรับส่งข้อมูลแบบเมสเซสนั้น โครงสร้างของข้อมูลที่ใช้ในการฝากส่งข้อความมักจะอยู่ในรูปของคิว ซึ่งอาจจะมีโครงสร้างที่แตกต่างกันออกไปได้อีกดังเช่น
  - Zero capacity ในกรณีนี้หมายความว่าไม่มีคิวปรากฏอยู่เลย หรือกล่าวในอีกทางหนึ่ง ข้อมูลที่จะส่งมีได้เพียงชุดเดียวเท่านั้น และเมื่อผู้ส่งส่งข้อมูลไปแล้ว ผู้รับจะต้องรับข้อมูลนั้นไปก่อน ผู้ส่งจึงจะสามารถส่งข้อมูลชุดถัดไปได้ มิเช่นนั้นก็จะไปทำข้อมูลเดิม
  - Bounded capacity ในกรณีนี้คิวจะมีขนาดจำกัด ถ้าคิวไม่เต็ม ผู้ส่งก็จะส่งข้อมูลได้ แต่ถ้าคิวเต็มผู้ส่งก็จะไม่สามารถส่งข้อมูลได้อีก(เข้าสู่สถานะ block รอจนกว่าจะมีพื้นที่ว่างจึงจะส่งข้อมูลแล้วไปทำงานต่อ)
  - Unbounded capacity ในกรณีนี้คิวจะมีขนาดไม่จำกัด ผู้ส่งจะสามารถส่งข้อมูลได้เรื่อยๆ โดยไม่เข้าสู่สถานะ block

ตัวอย่างกรณีของการรับส่งข้อมูลแบบเข้าจังหวะ ที่มีลักษณะการรอซึ่งกันและกัน

โพรเซสผู้ผลิต

```
while(true){
    ผลิตข้อมูลขึ้นมาใหม่หนึ่งหน่วย
    send(ข้อมูล);  //กลไกนี้จะรอหากเมสเซสคิวเต็ม
}
```

โพรเซสผู้บริโภค

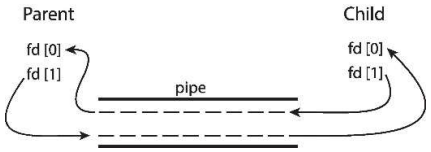
```
while(true){
   receive(&ข้อมูล);   //กลไกนี้จะรอหากคิวว่าง
   นำข้อมูลจากตัวแปรไปใช้งาน
}
```

การสื่อสารระหว่างโพรเซสด้วยวิธีการส่งผ่านไปป์ (Pipe)

ไปป์ เป็นกลไกการสื่อสารระหว่างโพรเซสอีกรูปแบบของระบบปฏิบัติการ มีลักษณะเป็นสตรีมที่นำข้อมูลเข้าและออกได้ในเวลาเดียวกัน

- ระบบปฏิบัติการโดยทั่วไปรองรับการทำงานของไปป์ได้ถึงในระดับทำงานจากชุดคำสั่งในระบบปฏิบัติการ (ใน command line) และผ่านไลบรารีของภาษาโปรแกรมที่ใช้
- กลไกการทำงานของไปป์ ส่วนมากจะรองรับเป็นการรับส่งสองทาง (หรืออาจจะเป็นทางเดียว ขึ้นกับระบบปฏิบัติการ -ในกรณีที่รองรับสองทาง เราก็สามารถปิดช่องทางไปเส้นหนึ่งให้เหลือทางเดียวได้ ดังตัวอย่างในปฏิบัติการท้ายบท)
- การรับส่งสองทางของไปป์ อาจจะเป็น half-duplex (ส่งไปกลับพร้อมกันไม่ได้) หรือ full-duplex (ส่งไปกลับสองทางพร้อมกันในเวลาเดียวกันได้)
- ระบบปฏิบัติการอาจรองรับการใช้งานไปป์ข้ามระบบเครือข่ายหรือไม่
- ประเภทของไปป์
    - Ordinary pipes ไปป์ทั่วไป (หรือไปป์แบบไม่มีชื่อ) เนื่องจากไม่ได้กำหนดชื่อไว้อ้างอิง การเข้าถึงไปป์ที่นิยามในลักษณะนี้จึงกระทำได้เฉพาะจากโพรเซสที่สร้างขึ้นจากกัน เช่นพ่อแม่กับลูกได้ แต่ไม่สามารถใช้งานข้ามระหว่างโพรเซสสองตัวที่ไม่เกี่ยวข้องกัน (เช่นสร้างขึ้นจากระบบปฏิบัติการโดยตรง)
    - Named pipes ไปป์มีชื่อ ในกรณีนี้ เวลาโพรเซสแรกสร้างไปป์ จะกำหนดชื่อไปป์ไว้ให้อ้างอิงได้จากระบบปฏิบัติการ (โดยทั่วไปมักจะอ้างเป็นชื่อไฟล์ภายใต้ระบบโครงสร้างการจัดการไฟล์) ทำให้โพรเซสอื่นๆ สามารถใช้ชื่อนี้ในการอ้างและเข้าถึงไปป์ที่สร้างได้

ไปป์ไม่มีชื่อ (Ordinary Pipes)



- จะเห็นว่าตัวแปรเก็บค่าไอดีของไปป์เป็นอะเรย์ขนาดสองหน่วย โดยหน่วยแรกทำหน้าที่ใช้รับข้อมูลจากไปป์ และหน่วยที่สองทำหน้าที่ส่งข้อมูลเข้าไปป์
- ในกรณีประเด็นปัญหาผู้ผลิต-ผู้บริโภค ผู้ผลิตจะส่งข้อมูลเข้าทางช่องทางเขียน fd[1] ส่วนผู้บริโภคจะอ่านข้อมูลจากช่องทางอ่าน fd[0]
- โพรเซสจะต้องเป็นพ่อแม่/ลูกกัน เพราะต้องอาศัยกลไกการ fork() เพื่อสำเนาโครงสร้าง fd นี้เป็นสองชุด (หรือในกรณีวินโดวส์ ก็จะอาศัยการแตกเธรดเป็นสองเธรด เป็นต้น)

ไปป์มีชื่อ (Named Pipes)

การสื่อสารสองทางแบบไปป์มีชื่อ

- โพรเซสตัวใดตัวหนึ่งขอสร้างไปป์จากระบบปฏิบัติการ ตัวอย่างในยูนิกซ์ และลินุกซ์นั้น จะเป็นการสั่งผ่าน command line (เช่นผ่านฟังก์ชัน system()) เพื่อสร้างไปป์แบบมีชื่อขึ้นมา ซึ่งจะปรากฏเป็นชื่อไฟล์อยู่ในไดเรกตอรีที่โพรเซสที่เกี่ยวข้องจะต้องสามาถเข้าถึงและเขียน/อ่านได้ (เช่นใน /tmp เป็นต้น)
- โพรเซสที่ต้องการใช้งานไปป์ที่เปิดไว้นี้ ก็จะเข้าถึงไปป์และเขียนอ่านได้ราวกับว่าเป็นการเข้าถึงไฟล์ตามปกติ

- แต่จะมีความแตกต่างจากการเข้าถึงไฟล์โดยทั่วไป เพราะการเข้าถึงไฟล์ของโพรเซสหนึ่งๆ จะทำให้เกิดการล็อกไฟล์ไม่สามารถเข้าถึงได้อีกจากโพรเซสอื่นๆ แต่ในกรณีของไปป์ โพรเซสต่างๆ สามารถเปิดไปป์นี้ได้พร้อมๆ กันและเขียน/อ่าน ได้พร้อมกัน

<u>กลไกการสื่อสารกันระหว่างโพรเซสในรูปแบบอื่นๆ</u>

- Socket เป็นการสร้างจุดเชื่อมต่อการสื่อสารกันข้ามระบบเครือข่าย โดยอาศัยข้อมูลที่สำคัญคือเลขไอพี และเลขพอร์ต
- Remote Predudure Calls เป็นการสร้างจุดเชื่อมต่อในลักษณะให้เรียกใช้ในรูปของฟังก์ชัน แต่ตัวฟังช์ชันที่เรียกใช้ทำงานอยู่ในเครื่องอื่นภายในระบบเครือข่าย

ปฏิบัติการ

2.1 การสร้างโพรเซสใหม่ในลินุกซ์

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // for exit()
#include <wait.h>   // for wait()

int main(){
        pid_t  pid=0;

        pid = fork();  //Fork a child process

         if(pid < 0){ //Fork error
                fprintf(stderr,"Fork failed.\n");
                 exit(-1);
         }
         else if(pid==0){ // This is the path of child process
             printf("Before replace the child with other code...\n");
             execlp("/usr/bin/nano","nano",NULL); // call a text editor
        }
        else { // This is the path of parent process
            printf("Before going into the wait  state...\n");
            wait(NULL);
            printf("Child process has terminated\n");
             exit(0);
         }
}
```

2.2 การสร้างโพรเซสใหม่ในวินโดวส์

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

void main(){
        STARTUPINFO si;
        PROCESS_INFORMATION pi;
        TCHAR name[] = _T("c:\\windows\\system32\\notepad.exe");

        ZeroMemory(&si,sizeof(si));
        si.cb = sizeof(si);
        ZeroMemory(&pi,sizeof(pi));

    printf("Before fork a child process...\n");

    if(!CreateProcess(NULL,name,
                NULL,NULL,FALSE,0,NULL,NULL,&si,&pi)){
                        fprintf(stderr,"Create process failed.\n");
                        return ;
        }

        //Parent Process
        printf("Before going into the wait state...\n");
        WaitForSingleObject(pi.hProcess,INFINITE);
        printf("Child process has terminated\n");
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
}
```

2.3 ตัวอย่างประเด็นปัญหา Producer-Consumer ที่จัดการด้วยวิธีการแชร์พื้นที่หน่วยความจำระหว่างโพรเซส สำหรับลินุกซ์

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // for exit()
#include <wait.h>   // for wait()
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/ipc.h>

struct shmareatype{
   int rp;
   int wp;
   char data[256];
};

void consumer(struct shmareatype *shmarea);
void producer(struct shmareatype *shmarea);

int main(){
        int shmid;  // Share memory ID
        struct shmareatype *shmarea; // Pointer to the shared segment
        int shmsize; // share segment size
        pid_t  pid;   // Child Process ID

        shmsize = sizeof(struct shmareatype);
        // Allocate the shared block from OS
        shmid = shmget(IPC_PRIVATE, shmsize, 0666 | IPC_CREAT );

        // Attach the shared block to the pointer so we can use it.
        shmarea = (struct shmareatype *)shmat(shmid, NULL, 0);

       shmarea->rp = shmarea->wp = 0;

        pid = fork();  //Fork a child process

        if(pid < 0){ //Fork error
                fprintf(stderr,"Fork failed.\n");
                exit(-1);
        }
        else if(pid==0){ // This is the path of child process
            consumer(shmarea);
        }
        else { // This is the path of parent process
            producer(shmarea);
            wait(NULL);
            printf("Child process has terminated\n");

          // Detach the shared memory segment
           shmdt(shmarea);
          // Remove the shared memory segment
           shmctl(shmid, IPC_RMID,NULL);
            exit(0);
        }
}

void consumer(struct shmareatype *shmarea){
    int i;

    for(i=0;i<16;i++) // consume data for 16 times
    {
       // Wait 2 minutes if no data in the circular buffer
       while(shmarea->wp==shmarea->rp) sleep(2);
       printf("Data number:%d = %c\n",i,shmarea->data[shmarea->rp]);
       shmarea->rp = ((shmarea->rp+1)%256);
    }
}

void producer(struct shmareatype *shmarea){
    int i;
    char temp[16];

    for(i=0;i<16;i++) // produce data for 16 times
```

```
    {
        printf("Please enter a character No %d :",i);
        fgets(temp,16,stdin);
        printf("Enter data %c into share memory...\n",temp[0]);
         shmarea->data[shmarea->wp]=temp[0];
       // move the write pointer so that the consumer know when to read.
       shmarea->wp = ((shmarea->wp+1)%256);
    }
}
```

2.4 ตัวอย่างประเด็นปัญหา Producer-Consumer ที่จัดการด้วยวิธีการแชร์พื้นที่หน่วยความจำระหว่างโพรเซส สำหรับวินโดวส์

อนึ่ง ระบบปฏิบัติการวินโดวส์ไม่รองรับการแชร์พื้นที่หน่วยความจำหลักโดยตรงระหว่างโพรเซส แต่สามารถแชร์โดยการใช้กลไกการสร้างไฟล์/ไฟล์บัปเฟอร์ แล้วโยงเข้ากับพื้นที่หน่วยความจำหลัก และอ้างถึงข้อมูลภายในโดยใช้ตัวชี้ ในตัวอย่างนี้เป็นการแชร์พื้นที่ใน page file ของวินโดวส์ โดยการกำหนดชื่ออ้างอิง เพื่อให้ใช้ร่วมกันระหว่างสองโพรเซส (ในที่นี้ใช้โปรแกรมเดียวกันแต่รันโปรแกรมสองครั้ง เกิดเป็นสองโพรเซสที่จะทำงานแยกกันอย่างอิสระ)

นอกเหนือจากวิธีนี้แล้ว วินโดวส์ยังมีช่องทางที่เหมาะสมสำหรับการติดต่อสื่อสารระหว่างโพรเซสอื่นๆ อีกเช่น mailslot pipe LPC (Local Procedure Call – undocumented) RPC (Remote Procedure Call) และ WinSock (วิธีสุดท้ายนี้จะได้เรียนในวิชาคอมพิวเตอร์เน็ตเวิร์คและเกมโปรแกรมมิ่งต่อไป)

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

struct shmareatype{
   int rp;
   int wp;
   char data[256];
};

void consumer(struct shmareatype *shmarea);
void producer(struct shmareatype *shmarea);

void main(int argc, char *argv[]){
        TCHAR shmName[] = _T("lab2_4");
        struct shmareatype *shmarea; // Pointer to the shared segment
    int shmsize = sizeof(struct shmareatype); // share segment size
        HANDLE hMapFile;

        if(argc>1){ // If this is the child process it will run this instead
       hMapFile = OpenFileMapping(
                    FILE_MAP_ALL_ACCESS,   // read/write access
                    FALSE,                 // do not inherit the name
                    shmName);              // name of mapping object

     if (hMapFile == NULL) {
      printf("Could not open file mapping object (%d).\n",
             GetLastError());
      return;
     }
 shmarea = (struct shmareatype *) MapViewOfFile(hMapFile,   // handle to map object
                         FILE_MAP_ALL_ACCESS,               // read/write permission
                         0,
                         0,
                         shmsize);

    if (shmarea == NULL)
    {
      printf("Could not map view of file (%d).\n",
             GetLastError());
      return;
    }

        consumer(shmarea);

            UnmapViewOfFile(shmarea);
       CloseHandle(hMapFile);
              return;
        }
```

ชื่ออ้างอิงสำหรับทำ memory file-mapping

```
    // Create Memory Mapping File
    hMapFile = CreateFileMapping(
                 INVALID_HANDLE_VALUE,    // use paging file
                 NULL,                    // default security
                 PAGE_READWRITE,          // read/write access
                 0,                       // max. object size
                 shmsize,                 // buffer size
                 shmName);                // name of mapping object

       if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE){
      printf("Could not create file mapping object...\n");
      return;
    }

       STARTUPINFO si;
       PROCESS_INFORMATION pi;
       TCHAR name[] = _T("lab2_4.exe 1"); // Using the Same code with parameter
       ZeroMemory(&si,sizeof(si));
       si.cb = sizeof(si);
       ZeroMemory(&pi,sizeof(pi));

    printf("Before fork a child process...\n");

    if(!CreateProcess(NULL,name,
               NULL,NULL,FALSE,0,NULL,NULL,&si,&pi)){
                      fprintf(stderr,"Create process failed.\n");
                      return ;
       }
       //Parent Process
       // Map share memory
    shmarea = (struct shmareatype *) MapViewOfFile(hMapFile,   // handle to map object
                        FILE_MAP_ALL_ACCESS, // read/write permission
                        0,
                        0,
                        shmsize);
    if (shmarea == NULL)
    {
      printf("Could not map view of file (%d).\n",
             GetLastError());
      return;
    }
       shmarea->rp = shmarea->wp = 0;
       producer(shmarea);
       printf("Before going into the wait state...\n");
       WaitForSingleObject(pi.hProcess,INFINITE);
       printf("Child process has terminated\n");
       CloseHandle(pi.hProcess);
       CloseHandle(pi.hThread);

       UnmapViewOfFile(shmarea);
    CloseHandle(hMapFile);
}

void consumer(struct shmareatype *shmarea){
    int i;

    for(i=0;i<16;i++) // consume data for 16 times
    {
       // Wait 2 minutes if no data in the circular buffer
       while(shmarea->wp==shmarea->rp) Sleep(2000);
       printf("Data number:%d = %c\n",i,shmarea->data[shmarea->rp]);
       shmarea->rp = ((shmarea->rp+1)%256);
    }
}

void producer(struct shmareatype *shmarea){
    int i;
    char temp[16];

    for(i=0;i<16;i++) // produce data for 16 times
    {
       printf("Please enter a character No %d :",i);
        fgets(temp,16,stdin);
        printf("Enter data %c into share memory...\n",temp[0]);
         shmarea->data[shmarea->wp]=temp[0];
       // move the write pointer so that the consumer know when to read.
       shmarea->wp = ((shmarea->wp+1)%256);
    }
}
```

ไม่มีการสร้างไฟล์ขึ้นจริงในหน่วยความจำสำรอง

ชื่อโปรแกรมที่สร้างขึ้น (อิงจากชื่อโปรเจ็คต์ที่นศ.ใช้)

2.5 ตัวอย่างการใช้ message queue เพื่อส่งข้อมูลระหว่างโพรเซส สำหรับลินุกซ์

อนึ่ง ขนาดสูงสุดของ message queue ของลินุกซ์สามารถเซ็ตค่าได้ที่ /proc/sys/kernel/msgmnb

ตัวอย่างโค้ดผู้ส่งเมสเซส

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

struct msgtype{
        long type;
        char data[256];
};

int main(){
        int qid;
        key_t key;
         char path[] = "/tmp";
         int id = 'a';
         struct msgtype message;

        printf("Message Sender\n==================\n");
        // Acquire a key using a path that user can access to.
        if((key = ftok(path,id))==-1){
                fprintf(stderr,"Error acquire a key...\n");
                 return -1;
         }
         // Get the message queue ID and create it if it does not exist
        if((qid = msgget(key,IPC_CREAT | 0666))==-1){
                fprintf(stderr,"Error creating a message queue...\n");
                return -1;
        }

        do{
              printf("Enter a string or just press ENTER to stop sending :");
              fgets(message.data,256,stdin);
             if(message.data[0]<0x20) message.type=2;  // Notify reader to stop process
              else  message.type=1; // Notify reader to read the message and display it.

                // 256 here is the data size and 0 means WAIT until the message is sent
              if(msgsnd(qid,&message,256,0)==-1){
                fprintf(stderr,"Error sending a message...\n");
                  msgctl(qid,IPC_RMID,NULL); //Remove message queue
                return -1;
             }
         }while(message.type!=2);
         msgctl(qid,IPC_RMID,NULL); //Remove message queue
}
```

ตัวอย่างโค้ดผู้รับเมสเซส

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

struct msgtype{
        long type;
        char data[256];
};

int main(){
        int qid;
        key_t key;
         char path[] = "/tmp";
         int id = 'a';
         struct msgtype message;

        printf("Message Reader\n==================\n");
        // Acquire a key using a path that user can access to.
        if((key = ftok(path,id))==-1){
                fprintf(stderr,"Error acquire a key...\n");
                 return -1;
```

```
        }
        // Get the message queue ID and create it if it does not exist
       if((qid = msgget(key,IPC_CREAT | 0666))==-1){
              fprintf(stderr,"Error creating a message queue...\n");
              return -1;
       }

        printf("Waiting for messages :\n");
       do{
            if(msgrcv(qid,&message,256 // Size of data in the structure
              ,0      // Accept all message  if this value is positive then get the exact type
       // If it is negative it will receive only one less or equal to absolute value of type.
                              ,0)==-1){   // Flag  this means  block receive
              fprintf(stderr,"Error reading a message...\n");
              return -1;
            }
            printf("%s\n",message.data);
            if(message.type==2){
              // Exit the program
               printf("End transmittion...\n");
               return 0;
            }
        }while(1);
}
```

2.6 ตัวอย่างการใช้ socket เพื่อส่งข้อมูลระหว่างโพรเซส สำหรับวินโดวส์ (จาก MSDN)

ตัวอย่างฝั่งเซอร์เวอร์

```
#undef UNICODE

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")

#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "27015"

int __cdecl main(void)
{
    WSADATA wsaData;
    int iResult;

    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET ClientSocket = INVALID_SOCKET;

    struct addrinfo *result = NULL;
    struct addrinfo hints;

    int iSendResult;
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;

    // Resolve the server address and port
    iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
    if ( iResult != 0 ) {
```

```
        printf("getaddrinfo failed with error: %d\n", iResult);
        WSACleanup();
        return 1;
    }

    // Create a SOCKET for connecting to server
    ListenSocket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
    if (ListenSocket == INVALID_SOCKET) {
        printf("socket failed with error: %ld\n", WSAGetLastError());
        freeaddrinfo(result);
        WSACleanup();
        return 1;
    }

    // Setup the TCP listening socket
    iResult = bind( ListenSocket, result->ai_addr, (int)result->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    freeaddrinfo(result);

    iResult = listen(ListenSocket, SOMAXCONN);
    if (iResult == SOCKET_ERROR) {
        printf("listen failed with error: %d\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // Accept a client socket
    ClientSocket = accept(ListenSocket, NULL, NULL);
    if (ClientSocket == INVALID_SOCKET) {
        printf("accept failed with error: %d\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // No longer need server socket
    closesocket(ListenSocket);

    // Receive until the peer shuts down the connection
    do {

        iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
        if (iResult > 0) {
            printf("Bytes received: %d\n", iResult);

        // Echo the buffer back to the sender
            iSendResult = send( ClientSocket, recvbuf, iResult, 0 );
            if (iSendResult == SOCKET_ERROR) {
                printf("send failed with error: %d\n", WSAGetLastError());
                closesocket(ClientSocket);
                WSACleanup();
                return 1;
            }
            printf("Bytes sent: %d\n", iSendResult);
        }
        else if (iResult == 0)
            printf("Connection closing...\n");
        else  {
            printf("recv failed with error: %d\n", WSAGetLastError());
            closesocket(ClientSocket);
            WSACleanup();
            return 1;
        }

    } while (iResult > 0);

    // shutdown the connection since we're done
    iResult = shutdown(ClientSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown failed with error: %d\n", WSAGetLastError());
        closesocket(ClientSocket);
```

```
        WSACleanup();
        return 1;
    }

    // cleanup
    closesocket(ClientSocket);
    WSACleanup();

    return 0;
}
```

ตัวอย่างฝั่งไคลเอนต์ ที่จะร้องขอการเชื่อมต่อเข้าทางเซอร์เวอร์

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>


// Need to link with Ws2_32.lib, Mswsock.lib, and Advapi32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "Mswsock.lib")
#pragma comment (lib, "AdvApi32.lib")


#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "27015"

int __cdecl main()
{
    WSADATA wsaData;
    SOCKET ConnectSocket = INVALID_SOCKET;
    struct addrinfo *result = NULL,
                    *ptr = NULL,
                    hints;
    char *sendbuf = "this is a test";
    char recvbuf[DEFAULT_BUFLEN];
    int iResult;
    int recvbuflen = DEFAULT_BUFLEN;

    // Validate the parameters
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

    ZeroMemory( &hints, sizeof(hints) );
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    // Resolve the server address and port
    iResult = getaddrinfo("127.0.0.1", DEFAULT_PORT, &hints, &result);
    if ( iResult != 0 ) {
        printf("getaddrinfo failed with error: %d\n", iResult);
        WSACleanup();
        return 1;
    }

    // Attempt to connect to an address until one succeeds
    for(ptr=result; ptr != NULL ;ptr=ptr->ai_next) {

        // Create a SOCKET for connecting to server
        ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
            ptr->ai_protocol);
        if (ConnectSocket == INVALID_SOCKET) {
            printf("socket failed with error: %ld\n", WSAGetLastError());
            WSACleanup();
            return 1;
        }

        // Connect to server.
        iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
        if (iResult == SOCKET_ERROR) {
```

```
            closesocket(ConnectSocket);
            ConnectSocket = INVALID_SOCKET;
            continue;
        }
        break;
    }

    freeaddrinfo(result);

    if (ConnectSocket == INVALID_SOCKET) {
        printf("Unable to connect to server!\n");
        WSACleanup();
        return 1;
    }

    // Send an initial buffer
    iResult = send( ConnectSocket, sendbuf, (int)strlen(sendbuf), 0 );
    if (iResult == SOCKET_ERROR) {
        printf("send failed with error: %d\n", WSAGetLastError());
        closesocket(ConnectSocket);
        WSACleanup();
        return 1;
    }

    printf("Bytes Sent: %ld\n", iResult);

    // shutdown the connection since no more data will be sent
    iResult = shutdown(ConnectSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown failed with error: %d\n", WSAGetLastError());
        closesocket(ConnectSocket);
        WSACleanup();
        return 1;
    }

    // Receive until the peer closes the connection
    do {

        iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
        if ( iResult > 0 )
            printf("Bytes received: %d\n", iResult);
        else if ( iResult == 0 )
            printf("Connection closed\n");
        else
            printf("recv failed with error: %d\n", WSAGetLastError());

    } while( iResult > 0 );

    // cleanup
    closesocket(ConnectSocket);
    WSACleanup();

    return 0;
}
```

2.7 ตัวอย่างการใช้ named pipe เพื่อส่งข้อมูลระหว่างโพรเซส สำหรับวินโดวส์ (จาก MSDN)

เซอร์เวอร์ (ตัวนี้ปล่อยให้รันตลอดเวลา แล้วรันตัวไคลเอนต์เพื่อส่งข้อความ)

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

#define PIPE_TIMEOUT 5000
#define BUFSIZE 4096

typedef struct
{
   OVERLAPPED oOverlap;
   HANDLE hPipeInst;
   TCHAR chRequest[BUFSIZE];
   DWORD cbRead;
   TCHAR chReply[BUFSIZE];
   DWORD cbToWrite;
```

```
} PIPEINST, *LPPIPEINST;

VOID DisconnectAndClose(LPPIPEINST);
BOOL CreateAndConnectInstance(LPOVERLAPPED);
BOOL ConnectToNewClient(HANDLE, LPOVERLAPPED);
VOID GetAnswerToRequest(LPPIPEINST);

VOID WINAPI CompletedWriteRoutine(DWORD, DWORD, LPOVERLAPPED);
VOID WINAPI CompletedReadRoutine(DWORD, DWORD, LPOVERLAPPED);

HANDLE hPipe;

int _tmain(VOID)
{
   HANDLE hConnectEvent;
   OVERLAPPED oConnect;
   LPPIPEINST lpPipeInst;
   DWORD dwWait, cbRet;
   BOOL fSuccess, fPendingIO;

// Create one event object for the connect operation.
   hConnectEvent = CreateEvent(
      NULL,    // default security attribute
      TRUE,    // manual reset event
      TRUE,    // initial state = signaled
      NULL);   // unnamed event object

   if (hConnectEvent == NULL)
   {
      printf("CreateEvent failed with %d.\n", GetLastError());
      return 0;
   }

   oConnect.hEvent = hConnectEvent;

// Call a subroutine to create one instance, and wait for
// the client to connect.
   fPendingIO = CreateAndConnectInstance(&oConnect);

   while (1)
   {
   // Wait for a client to connect, or for a read or write
   // operation to be completed, which causes a completion
   // routine to be queued for execution.

      dwWait = WaitForSingleObjectEx(
         hConnectEvent,  // event object to wait for
         INFINITE,       // waits indefinitely
         TRUE);          // alertable wait enabled

      switch (dwWait)
      {
      // The wait conditions are satisfied by a completed connect
      // operation.
         case 0:
         // If an operation is pending, get the result of the
         // connect operation.

         if (fPendingIO)
         {
            fSuccess = GetOverlappedResult(
               hPipe,     // pipe handle
               &oConnect, // OVERLAPPED structure
               &cbRet,    // bytes transferred
               FALSE);    // does not wait
            if (!fSuccess)
            {
               printf("ConnectNamedPipe (%d)\n", GetLastError());
               return 0;
            }
         }

         // Allocate storage for this instance.

            lpPipeInst = (LPPIPEINST) GlobalAlloc(
               GPTR, sizeof(PIPEINST));
            if (lpPipeInst == NULL)
            {
               printf("GlobalAlloc failed (%d)\n", GetLastError());
               return 0;
```

```
            }

            lpPipeInst->hPipeInst = hPipe;

         // Start the read operation for this client.
         // Note that this same routine is later used as a
         // completion routine after a write operation.
            lpPipeInst->cbToWrite = 0;
            CompletedWriteRoutine(0, 0, (LPOVERLAPPED) lpPipeInst);

         // Create new pipe instance for the next client.
            fPendingIO = CreateAndConnectInstance(
               &oConnect);
            break;

      // The wait is satisfied by a completed read or write
      // operation. This allows the system to execute the
      // completion routine.
         case WAIT_IO_COMPLETION:
            break;

      // An error occurred in the wait function.

         default:
         {
            printf("WaitForSingleObjectEx (%d)\n", GetLastError());
            return 0;
         }
      }
   }
   return 0;
}

// CompletedWriteRoutine(DWORD, DWORD, LPOVERLAPPED)
// This routine is called as a completion routine after writing to
// the pipe, or when a new client has connected to a pipe instance.
// It starts another read operation.

VOID WINAPI CompletedWriteRoutine(DWORD dwErr, DWORD cbWritten,
   LPOVERLAPPED lpOverLap)
{
   LPPIPEINST lpPipeInst;
   BOOL fRead = FALSE;

// lpOverlap points to storage for this instance.

   lpPipeInst = (LPPIPEINST) lpOverLap;

// The write operation has finished, so read the next request (if
// there is no error).

   if ((dwErr == 0) && (cbWritten == lpPipeInst->cbToWrite))
      fRead = ReadFileEx(
         lpPipeInst->hPipeInst,
         lpPipeInst->chRequest,
         BUFSIZE*sizeof(TCHAR),
         (LPOVERLAPPED) lpPipeInst,
         (LPOVERLAPPED_COMPLETION_ROUTINE) CompletedReadRoutine);

// Disconnect if an error occurred.

   if (! fRead)
      DisconnectAndClose(lpPipeInst);
}

// CompletedReadRoutine(DWORD, DWORD, LPOVERLAPPED)
// This routine is called as an I/O completion routine after reading
// a request from the client. It gets data and writes it to the pipe.

VOID WINAPI CompletedReadRoutine(DWORD dwErr, DWORD cbBytesRead,
    LPOVERLAPPED lpOverLap)
{
   LPPIPEINST lpPipeInst;
   BOOL fWrite = FALSE;

// lpOverlap points to storage for this instance.

   lpPipeInst = (LPPIPEINST) lpOverLap;

// The read operation has finished, so write a response (if no
```

```
// error occurred).

   if ((dwErr == 0) && (cbBytesRead != 0))
   {
      GetAnswerToRequest(lpPipeInst);

      fWrite = WriteFileEx(
         lpPipeInst->hPipeInst,
         lpPipeInst->chReply,
         lpPipeInst->cbToWrite,
         (LPOVERLAPPED) lpPipeInst,
         (LPOVERLAPPED_COMPLETION_ROUTINE) CompletedWriteRoutine);
   }

// Disconnect if an error occurred.

   if (! fWrite)
      DisconnectAndClose(lpPipeInst);
}

// DisconnectAndClose(LPPIPEINST)
// This routine is called when an error occurs or the client closes
// its handle to the pipe.

VOID DisconnectAndClose(LPPIPEINST lpPipeInst)
{
// Disconnect the pipe instance.

   if (! DisconnectNamedPipe(lpPipeInst->hPipeInst) )
   {
      printf("DisconnectNamedPipe failed with %d.\n", GetLastError());
   }

// Close the handle to the pipe instance.

   CloseHandle(lpPipeInst->hPipeInst);

// Release the storage for the pipe instance.

   if (lpPipeInst != NULL)
      GlobalFree(lpPipeInst);
}

// CreateAndConnectInstance(LPOVERLAPPED)
// This function creates a pipe instance and connects to the client.
// It returns TRUE if the connect operation is pending, and FALSE if
// the connection has been completed.

BOOL CreateAndConnectInstance(LPOVERLAPPED lpoOverlap)
{
   LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");

   hPipe = CreateNamedPipe(
      lpszPipename,             // pipe name
      PIPE_ACCESS_DUPLEX |      // read/write access
      FILE_FLAG_OVERLAPPED,     // overlapped mode
      PIPE_TYPE_MESSAGE |       // message-type pipe
      PIPE_READMODE_MESSAGE |   // message read mode
      PIPE_WAIT,                // blocking mode
      PIPE_UNLIMITED_INSTANCES, // unlimited instances
      BUFSIZE*sizeof(TCHAR),    // output buffer size
      BUFSIZE*sizeof(TCHAR),    // input buffer size
      PIPE_TIMEOUT,             // client time-out
      NULL);                    // default security attributes
   if (hPipe == INVALID_HANDLE_VALUE)
   {
      printf("CreateNamedPipe failed with %d.\n", GetLastError());
      return 0;
   }

// Call a subroutine to connect to the new client.

   return ConnectToNewClient(hPipe, lpoOverlap);
}

BOOL ConnectToNewClient(HANDLE hPipe, LPOVERLAPPED lpo)
{
   BOOL fConnected, fPendingIO = FALSE;

// Start an overlapped connection for this pipe instance.
```

```
   fConnected = ConnectNamedPipe(hPipe, lpo);

// Overlapped ConnectNamedPipe should return zero.
   if (fConnected)
   {
      printf("ConnectNamedPipe failed with %d.\n", GetLastError());
      return 0;
   }

   switch (GetLastError())
   {
   // The overlapped connection in progress.
      case ERROR_IO_PENDING:
         fPendingIO = TRUE;
         break;

   // Client is already connected, so signal an event.

      case ERROR_PIPE_CONNECTED:
         if (SetEvent(lpo->hEvent))
            break;

   // If an error occurs during the connect operation...
      default:
      {
         printf("ConnectNamedPipe failed with %d.\n", GetLastError());
         return 0;
      }
   }
   return fPendingIO;
}

VOID GetAnswerToRequest(LPPIPEINST pipe)
{
   _tprintf( TEXT("[%d] %s\n"), pipe->hPipeInst, pipe->chRequest);
   StringCchCopy( pipe->chReply, BUFSIZE, TEXT("Default answer from server") );
   pipe->cbToWrite = (lstrlen(pipe->chReply)+1)*sizeof(TCHAR);
}
```

ไคลเอนต์

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
#define BUFSIZE 512

int _tmain(int argc, TCHAR *argv[])
{
   HANDLE hPipe;
   LPTSTR lpvMessage=TEXT("Default message from client.");
   TCHAR  chBuf[BUFSIZE];
   BOOL   fSuccess = FALSE;
   DWORD  cbRead, cbToWrite, cbWritten, dwMode;
   LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");

   if( argc > 1 )
      lpvMessage = argv[1];

// Try to open a named pipe; wait for it, if necessary.
   while (1)
   {
      hPipe = CreateFile(
         lpszPipename,   // pipe name
         GENERIC_READ |  // read and write access
         GENERIC_WRITE,
         0,              // no sharing
         NULL,           // default security attributes
         OPEN_EXISTING,  // opens existing pipe
         0,              // default attributes
         NULL);          // no template file

   // Break if the pipe handle is valid.

      if (hPipe != INVALID_HANDLE_VALUE)
         break;

      // Exit if an error other than ERROR_PIPE_BUSY occurs.
```

```
      if (GetLastError() != ERROR_PIPE_BUSY)
      {
         _tprintf( TEXT("Could not open pipe. GLE=%d\n"), GetLastError() );
         return -1;
      }

      // All pipe instances are busy, so wait for 20 seconds.

      if ( ! WaitNamedPipe(lpszPipename, 20000))
      {
         printf("Could not open pipe: 20 second wait timed out.");
         return -1;
      }
   }

// The pipe connected; change to message-read mode.
   dwMode = PIPE_READMODE_MESSAGE;
   fSuccess = SetNamedPipeHandleState(
      hPipe,    // pipe handle
      &dwMode,  // new pipe mode
      NULL,     // don't set maximum bytes
      NULL);    // don't set maximum time
   if ( ! fSuccess)
   {
      _tprintf( TEXT("SetNamedPipeHandleState failed. GLE=%d\n"), GetLastError() );
      return -1;
   }

// Send a message to the pipe server.
   cbToWrite = (lstrlen(lpvMessage)+1)*sizeof(TCHAR);
   _tprintf( TEXT("Sending %d byte message: \"%s\"\n"), cbToWrite, lpvMessage);

   fSuccess = WriteFile(
      hPipe,                  // pipe handle
      lpvMessage,             // message
      cbToWrite,              // message length
      &cbWritten,             // bytes written
      NULL);                  // not overlapped

   if ( ! fSuccess)
   {
      _tprintf( TEXT("WriteFile to pipe failed. GLE=%d\n"), GetLastError() );
      return -1;
   }

   printf("\nMessage sent to server, receiving reply as follows:\n");

   do
   {
   // Read from the pipe.
      fSuccess = ReadFile(
         hPipe,    // pipe handle
         chBuf,    // buffer to receive reply
         BUFSIZE*sizeof(TCHAR),  // size of buffer
         &cbRead,  // number of bytes read
         NULL);    // not overlapped

      if ( ! fSuccess && GetLastError() != ERROR_MORE_DATA )
         break;

      _tprintf( TEXT("\"%s\"\n"), chBuf );
   } while ( ! fSuccess);  // repeat loop if ERROR_MORE_DATA

   if ( ! fSuccess)
   {
      _tprintf( TEXT("ReadFile from pipe failed. GLE=%d\n"), GetLastError() );
      return -1;
   }

   printf("\n<End of message, press ENTER to terminate connection and exit>");
   _getch();

   CloseHandle(hPipe);

   return 0;
}
```

2.8 ตัวอย่างการใช้ pipe เพื่อส่งข้อมูลระหว่างโพรเซสพ่อแม่กับโพรเซสลูก สำหรับลินุกซ์

pipe ของลินุกซ์เป็นกลไกที่เป็นช่องทางแบบสองทิศทางเชื่อมต่อระหว่างโพรเซสพ่อแม่กับโพรเซสลูก โดยจะมีเส้นทางภายในสองเส้น สามารถรับส่งข้อมูลกันได้โดยมองหลักการเป็นสตรีมในภาษาซี

ให้สังเกตว่า pipe ในภาษาซีนั้นถูกจัดการในลักษณะของสตรีม และโครงสร้างของสตรีมในภาษาซี จะรอข้อมูลให้เต็มบัฟเฟอร์เสียก่อนจึงค่อยส่ง ในตัวอย่างนี้เราจึงต้องใช้ fflush() เพื่อสั่งให้เกิดการส่งข้อมูลจากสตรีมออกไปโดยทันที เพื่อให้ปลายทางไม่ต้องเสียเวลารอจนกว่าบัฟเฟอร์ของสตรีมจะเต็ม

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

void consumer(int fds[]);
void producer(int fds[]);

int main(){
        int fds[2];  // Bidirectional pipe
        pid_t  pid;   // Child Process ID

        pipe(fds);

        pid = fork();  //Fork a child process

        if(pid < 0){ //Fork error
                fprintf(stderr,"Fork failed.\n");
                return -1;
        }
        else if(pid==0){ // This is the path of child process
            printf("Child process started..");
            close(fds[1]); // Close the unused end.(not send anything back to parent);
            consumer(fds);
            close(fds[0]); // This will close the read end.
        }
        else { // This is the path of parent process
            close(fds[0]); // Close the read back end. (Not receiving anything from child.
            producer(fds);
            wait(NULL);
            printf("Child process has terminated\n");
            close(fds[1]);  // Close the write end.
            return 0;
        }
}

void consumer(int fds[]){
    int i=0;
    FILE *fr;

    fr = fdopen(fds[0],"r");  // Open the write end to receive data

    for(i=0;i<16;i++){
       if(feof(fr)||ferror(fr)) sleep(2);
       printf("Data number:%d = %c\n",i,fgetc(fr));
    }
}

void producer(int fds[]){
    int i;
    char temp[16];
    FILE *fw;

    fw = fdopen(fds[1],"w");  // Open the write end to send data

    for(i=0;i<16;i++) // produce data for 16 times
    {
        printf("Please enter a character No %d :",i);
        fgets(temp,16,stdin);
        printf("Enter data %c into pipe...\n",temp[0]);
        fputc(temp[0],fw);
        fflush(fw); // Flush data into stream
    }
}
```

2.9 ตัวอย่างการใช้ named pipe เพื่อส่งข้อมูลระหว่างโพรเซสใดๆ ในลินุกซ์

named pipe หรือ FIFO ของลินุกซ์ เป็น pipe ที่อ้างโดยใช้หลักการของไฟล์ในลินุกซ์ การสร้างและลบ namepipe ใช้คำสั่งในลินุกซ์ mkfifo สำหรับการสร้าง namepipe ขึ้นใหม่ และ rm สำหรับลบ namepipe

เนื่องจากสามารถอ้างได้ในรูปของไฟล์ในระบบโครงสร้างไฟล์ ทำให้การติดต่อสามารถกระทำระหว่างโพรเซสใดๆ ภายในลินุกซ์ได้ ตัวอย่างต่อไปนี้นำตัวอย่างที่แล้วมาเปลี่ยนเป็นการใช้ namepipe แทน สังเกตการจัดการ namepipe กระทำแบบเดียวกันกับการจัดการไฟล์ตามปกติ

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

void consumer(void);
void producer(void);

char *pipeName="/tmp/myPipe";

int main(){
        pid_t  pid;   // Child Process ID

        system("mkfifo /tmp/myPipe"); // Create name pipe (FIFO)

        pid = fork();  //Fork a child process

        if(pid < 0){ //Fork error
                fprintf(stderr,"Fork failed.\n");
                return -1;
        }
        else if(pid==0){ // This is the path of child process
            printf("Child process started..");
            consumer();
        }
        else { // This is the path of parent process
            producer();
            wait(NULL);
            printf("Child process has terminated\n");
           system("rm /tmp/myPipe");  // Delete name pipe
            return 0;
        }
}

void consumer(){
    int i=0;
    FILE *fr;

    fr = fopen(pipeName,"r"); // Open the write end to receive data

    for(i=0;i<16;i++){
       if(feof(fr)||ferror(fr)) sleep(2);
       printf("Data number:%d = %c\n",i,fgetc(fr));
    }
    fclose(fr);
}

void producer(){
    int i;
    char temp[16];
    FILE *fw;

    fw = fopen(pipeName,"w"); // Open the write end to send data

    for(i=0;i<16;i++) // produce data for 16 times
    {
        printf("Please enter a character No %d :",i);
        fgets(temp,16,stdin);
        printf("Enter data %c into pipe...\n",temp[0]);
        fputc(temp[0],fw);
        fflush(fw); // Flush data into stream
    }
    fclose(fw);
}
```