สำนักหอสมุดกลาง พระจอมเกล้าลาดกระบัง

โครงสร้างระบบปฏิบัติการลีนุกซ์

LINUX INTERNAL

นาย วิษณุ ตู้แก้ว

นาย อดุลย์สิทธิ์ เพ็งพา

เลขหมู่.....
เลขทะเบียน 46180
วัน, เดือน, ปี 20 มิ.ย. 2546

.b.....
.i.....

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร์บัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2544

DMA 83334

โครงสร้างระบบปฏิบัติการลีนุกซ์

LINUX INTERNAL

โดย

นาย วิษณุ ตู้แก้ว

นาย อดุลย์สิทธิ์ เพ็งพา

อาจารย์ที่ปรึกษา

รศ.ดร. บุญธีร์ เครือตราชู

อาจารย์ กฤตวัน เครือตราชู

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร์บัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2544

ปริญญานิพนธ์ปีการศึกษา 2544

ภาควิชา วิศวกรรมคอมพิวเตอร์

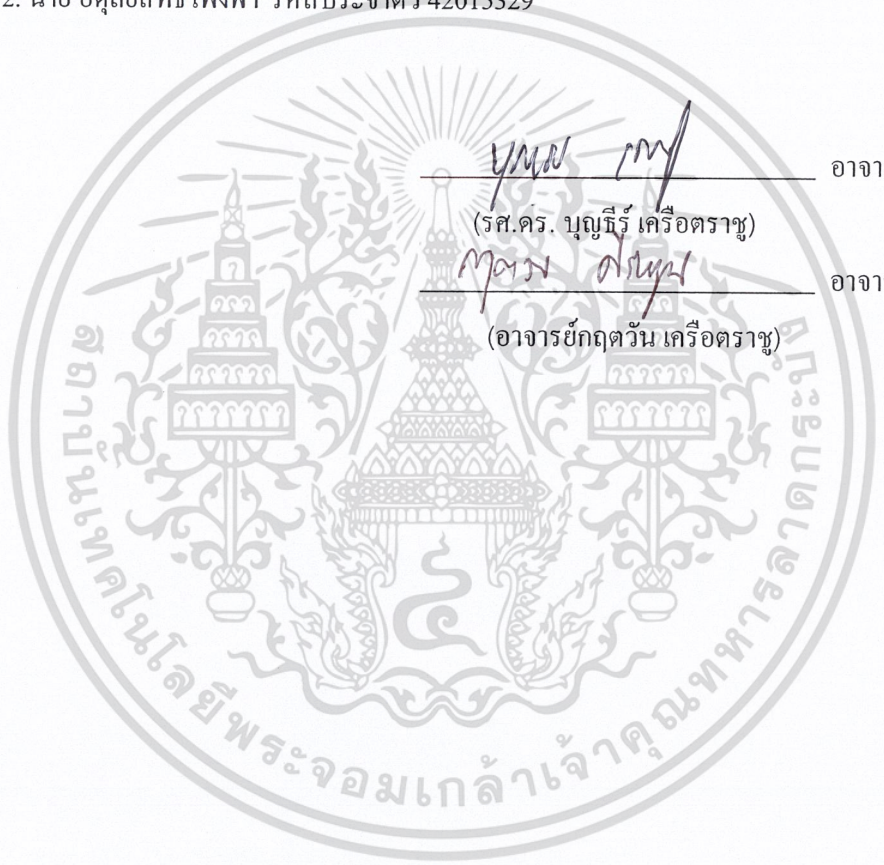
คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง โครงสร้างระบบปฏิบัติการลีนุกซ์

LINUX INTERNAL

ผู้จัดทำ

1. นาย วิษณุ ตู้แก้ว รหัสประจำตัว 42015317
2. นาย อดุลย์สิทธิ์ เพ็งพา รหัสประจำตัว 42015329



อาจารย์ที่ปรึกษา

(รศ.ดร. บุญธีร์ เครือตราชู)

อาจารย์ที่ปรึกษา

(อาจารย์กฤตวัน เครือตราชู)

โครงสร้างระบบปฏิบัติการลีนุกซ์

นาย วิษณุ ตู้แก้ว 42015317
นาย อดุลสิทธิ์ เพ็งพา 42015329
รศ.ดร.บุญธีร์ เครือตราชู อาจารย์ที่ปรึกษา
อ. กฤตวัน เครือตราชู อาจารย์ที่ปรึกษา
ปีการศึกษา 2544

บทคัดย่อ

ปริญญานิพนธ์เล่มนี้มีเนื้อหาเกี่ยวกับโครงสร้างระบบปฏิบัติการลีนุกซ์ ซึ่งได้เจาะจงไปตามเนื้อเรื่องที่สำคัญ อาทิ เช่น การคอมไพล์เคอร์เนล การดีบักโปรแกรม การเรียกใช้สัญญาณ ซิสเต็มคอล (System Call) จากยูสเซอร์โหมดซึ่งปกติฟังก์ชันบางฟังชันเรียกได้เฉพาะส่วนของเคอร์เนลโหมดเท่านั้น นอกจากนั้นเนื้อหายังประกอบด้วยโมดูลที่สำคัญที่สุด คือ การจัดการโปรเซสซึ่งแสดงให้เห็นถึงอัลกอริทึมการจัดการโปรเซสของระบบปฏิบัติการลีนุกซ์ซึ่งส่วนท้ายแสดงถึงอัลกอ ริทึมการจัดการโปรเซสจากคิวเดี่ยวซึ่งเป็นของระบบปฏิบัติการลีนุกซ์ดั้งเดิมมาเป็นอัลกอริทึมการจัดการโปรเซสแบบหลายคิวโดยการทดลองทั้งหมดกระทำบนเคอร์เนลเวอร์ชั่น 2.4.2

ในระบบปฏิบัติการลีนุกซ์เป็นระบบปฏิบัติการที่มีรายละเอียดมากมาย วิทยานิพนธ์เล่มนี้ได้คัดเอามาเพียงบางส่วนเท่านั้นและทำการออกแบบการทดลองสำหรับเรื่องที่สำคัญดังที่กล่าวมาแล้วข้างต้น จำนวน 3 การทดลอง

Linux Internal

Mr. Wisanu Tukaew 42015317
Mr. Adulsit Phengpha 42015329
Dr. Boontee Kruatrachue Advisor
Kritawan Kruatrachue Advisor
2544

Abstract

This thesis presented about Linux internal structure , so purpose with specification such as recompile kernel , debug kernel with GDB , implement most interfaces betweeen User Mode process and hardware device by system call issued to the kernel because some fuction use in Kernel Mode only.

According , This thesis contains about important modules such as process management and process scheduling. Modification of Linux process scheduling using new algorithm because a single run-queue may become a bottleneck while running. As this project to build a kernel on the kernel version 2.4.2

Therefore , Linux operating system consists several detail. We select specification with core important detail and design laboratory for student to learning about Linux operating system. We use Linux operating systam because it opensource.

กิตติกรรมประกาศ

ปริญญานิพนธ์ฉบับนี้คงไม่อาจเสร็จได้ด้วยดี หากไม่ได้รับความช่วยเหลือ และความร่วมมือจากหลายๆ ฝ่ายด้วยกัน บุคคลแรกที่ต้องกล่าวถึง เพราะเป็นส่วนสำคัญที่ทำให้ปริญญานิพนธ์นี้เสร็จลงได้ก็คือ รศ.ดร. บุญธีร์ เครือตราชู อาจารย์ที่ปรึกษานิพนธ์ ที่ให้ความเอาใจใส่ แนะนำและช่วยเหลือเสมอมาซึ่งต้องขอขอบคุณเป็นอย่างมาก

และต้องขอขอบพระคุณบุคคลสำคัญที่สุดที่ทำให้ข้าพเจ้ามีวันนี้ ก็คือ บิดา มารดา อันเป็นที่เคารพรักยิ่ง ซึ่งได้เลี้ยงดูผู้เขียนมาเป็นอย่างดี พร้อมทั้งให้โอกาสในการศึกษาอย่างเต็มที่และยังให้กำลังใจเอาใจใส่เสมอมา ในทุกๆ ด้านอันหาที่เปรียบมิได้ ข้าพเจ้าขอระลึกในพระ คุณอันสุดประมาณ และขอกราบขอบพระคุณมา ณ. ที่นี้

วิษณุ ตู้แก้ว
อดุลย์สิทธิ์ เพ็งพา

สารบัญ

หน้าที่

บทคัดย่อภาษาไทย I

บทคัดย่อภาษาอังกฤษ II

กิตติกรรมประกาศ III

สารบัญ IV

สารบัญภาพ VII

บทที่ 1 บทนำ

1.1 ความสำคัญและที่มา 1

1.2 วัตถุประสงค์ของปริญญานิพนธ์ 2

1.3 ขอบเขตของปริญญานิพนธ์ 2

1.4 วิธีการดำเนินงาน 2

1.5 ขั้นตอนการดำเนินงาน 3

บทที่ 2 สถาปัตยกรรมเคอร์เนลและองค์ประกอบต่าง ๆ ของระบบปฏิบัติการลีนุกซ์

2.1 ความหมายของเคอร์เนล 4

2.2 เป้าหมายของเคอร์เนล (Kernel Goal) 4

2.3 โครงสร้างการทำงานของเคอร์เนล (Look at the kernel 's architecture) 5

2.4 คุณสมบัติที่สำคัญของระบบปฏิบัติการลีนุกซ์ 6

2.5 สถาปัตยกรรมเคอร์เนล (Kernel Architecture) 6

- สถาปัตยกรรมไมโครเคอร์เนล (Microkernel Architecture) 6
- สถาปัตยกรรมโมโนลิทิก (Monolithic Kernel Architecture) 8

2.6 ซอร์ดโค้ดที่สำคัญในระบบปฏิบัติการลีนุกซ์ (The Linux Kernel source) 9

บทที่ 3 การคอมไพล์เคอร์เนลและการดีบักด้วยจีดีบี

(Compile Kernel And Debug With GDB)

3.1 การคอมไพลเคอร์เนล 14

3.2 การใช้ GCC บนระบบปฏิบัติการลีนุกซ์ 16

3.3 การดีบักโปรแกรมด้วย GDB 17

3.4 การทดลองที่ 1 เรื่องการรีคอมไพล์เคอร์เนล, การดีบักเคอร์เนล 21

สารบัญ (ต่อ)

หน้าที่

บทที่ 4 ซิสเต็มคอล (System Call)

4.1 การทำงานของซิสเต็มคอล 28

4.2 การติดต่อของซิสเต็มคอล (System Call linkage) 29

4.3 การสร้างซิสเต็มคอลสตับ (Defining the System Call Number) 30

4.4 การสร้างซิสเต็มคอลสตับ (Generating a System Call Stub) 30

4.5 เคอร์เนลฟังก์ชั่นออเกไนซ์เซชั่น (Kernel Function Organization) 31

บทที่ 5 โปรเซสและการจัดการโปรเซส

5.1 ความหมายของโปรเซส 36

5.2 สถานะของโปรเซส (Process state) 36

5.3 โครงสร้างข้อมูลของโปรเซส (Process Descriptor) 37

5.4 โปรเซสในเคอร์เนลโหมด (Storing a process descriptor) 44

5.5 ความสัมพันธ์ระหว่างโปรเซส (Process Relationship) 46

5.6 ลิสต์ของโปรเซส (Process List) 47

5.7 ลิสต์ของเวทคิว (Wait Queue) 48

5.8 การทำงานของโปรเซสและตารางการจัดการโปรเซส 49

5.8.1 การสร้างโปรเซส 49

5.8.2 โปรเซสไอดี (PID) 50

5.8.3 ไพออริตี้ของโปรเซสในลีนุกซ์ 50

5.8.4 การเก็บงานไว้รันภายหลังเมื่อมีเหตุการณ์บางอย่างเกิดขึ้น(อินเตอร์รัพ) 51

5.8.5 โปรเซสสวิทชิ่ง (Process Switching) 51

5.8.6 ฮาร์ดแวร์คอนเทก (Hardware Context) 52

5.8.7 เทรด (Thread) 52

5.8.8 ตารางการจัดการโปรเซส (Process Scheduling) 53

5.8.9 การทดลองที่ 3 เรื่อง โปรเซสและการจัดการโปรเซส 55

5.8.10 การเปลี่ยนอัลกอริทึมการจัดการโปรเซสจากคิวเดี่ยวเป็นหลายคิว 60

(Modification of Linux process scheduling using multilevel priority queues)

บทที่ 6 การจัดการหน่วยความจำ

6.1 ความต้องการของหน่วยความจำ (Memory Management Requirement) 66

6.2 การแบ่งหน่วยความจำ (Memory Partitioning) 67

6.3 ตำแหน่งของหน่วยความจำ (Memory Address) 69

สารบัญ (ต่อ)

	หน้าที่
6.4 เซกเมนต์ในฮารฺดแวร์ (Segment in Hardware)	69
6.5 เซกเมนต์ในลีนุกซ์ (Segment in Linux)	72
บทที่ 7 บทสรุปและวิจารณ์	
7.1 บทสรุปและวิจารณ์	79
7.2 ปัญหาที่เกิดขึ้น	79
7.3 วิธีการแก้ปัญหา	80
7.4 แนวทางการพัฒนา	80
ภาคผนวก ก. - วิธีการในการติดตั้งลีนุกซ์ Redhat 7.1	81
ภาคผนวก ข. – ตอบคำถามท้ายการทดลอง	90
บรรณานุกรม	94

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

สารบัญรูปภาพ

หน้าที่

รูปที่ 2-1 แสดง รายละเอียดโครงสร้างภายในของเคอร์เนล 5

รูปที่ 2-2 แสดง สถาปัตยกรรมไมโครเคอร์เนล 8

รูปที่ 2-3 แสดง สถาปัตยกรรมโมโนลิทิก 9

รูปที่ 3-1 แสดงไฟล์หลังจากการขยายไฟล์ของเคอร์เนล 14

รูปที่ 3-2 แสดงโปรแกรมที่ใช้ในการดีบัก 22

รูปที่ 3-3 แสดง การคอมไพล์โปรแกรมด้วย gcc โดยเพิ่มออปชัน –g เข้าไปด้วย 22

รูปที่ 3-4 แสดง การเรียกใช้โปรแกรม gdb 23

รูปที่ 3-5 แสดง การใช้คำสั่งเบรกใน gdb 23

รูปที่ 3-6 แสดง การรันโปรแกรมภายใต้ gdb 23

รูปที่ 3-7 แสดง การใช้ disp แสดงค่าตัวแปรที่ต้องการ 23

รูปที่ 3-8 แสดง การทำงานทีละบรรทัดโดยเข้าไปในฟังก์ชันย่อยด้วย 24

รูปที่ 3-9 แสดง การทำงานทีละบรรทัดแต่ไม่เข้าไปในฟังก์ชันย่อย 24

รูปที่ 3-10 แสดง การใช้คำสั่ง make menuconfig 25

รูปที่ 3-11 แสดง การ enable CONFIG_DEBUGSYM และ CONFIG_PT_PROXY 25

รูปที่ 3-12 แสดง ไฟล์ลีนุกซ์ที่ได้หลังจากการคอมไพล์ 26

รูปที่ 3-13 แสดง การดีบักลีนุกซ์ 26

รูปที่ 3-14 แสดง การใช้เบรกพอยนต์ในเคอร์เนล 26

รูปที่ 3-15 แสดง การรันไฟล์ที่ทำการดีบัก 26

รูปที่ 3-16 แสดง การรันทีละสเต็บป์ 27

รูปที่ 3-17 แสดง ตำแหน่งเบรกพอยนต์ 27

รูปที่ 4-1 แสดง การทำงานของซิสเต็มคอลที่ User Mode ติดต่อไปยัง Kernel Mode 28

รูปที่ 4-2 แสดงการแก้ไขไฟล์ entry.S 32

รูปที่ 4-3 แสดงการแก้ไขไฟล์ unistd.h 32

รูปที่ 4-4 แสดงการแก้ไขไฟล์ time.c 33

รูปที่ 4-5 แสดงการเขียนโปรแกรมเรียกใช้ฟังก์ชันในเคอร์เนลโดยเรียกโดยตรง 33

รูปที่ 4-6 แสดงการทดสอบการรันโปรแกรมโดยตรง 34

รูปที่ 4-7 แสดงการเขียนโปรแกรมเรียกใช้ฟังก์ชันในเคอร์เนลโดยเรียกโดยผ่านสตับ 34

รูปที่ 4-8 แสดงการทดสอบการรันโปรแกรมโดยผ่านสตับ 35

รูปที่ 5-1 แสดง การจัดเก็บโครงสร้างข้อมูลของโปรเซสในเคอร์เนลสแต็ก 44

รูปที่ 5-2 แสดง การทำงานของโปรเซสในการติดต่อกับส่วนของเคอร์เนลโหมด 46

สารบัญรูปภาพ(ต่อ)

	หน้าที่
รูปที่ 5-3 แสดง ความสัมพันธ์ระหว่างโปรเซส	47
รูปที่ 5-4 แสดงโปรเซสลิส (Process List)	48
รูปที่ 5-6 แสดง เวทคิว (Wait Queue)	49
รูปที่ 5-7 แสดง การจัดการโปรเซสแบบมาก่อนได้ก่อน (FCFS)	53
รูปที่ 5-8 แสดงการจัดการโปรเซสแบบวนรอบ (Round-Robin)	54
รูปที่ 6-1 แสดงตัวอย่างการจัดสรรแมมโมรี่ของแต่ละโปรเซส	68
รูปที่ 6-2 แสดงการเปลี่ยนลอจิคอลแอดเดรสเป็นฟีซิคอลแอดเดรส	69
รูปที่ 6-3แสดง รายละเอียดของเซกเมนต์รีจิสเตอร์ (Segment Registor)	72
รูปที่ 6-4 แสดง การใช้ Selector ชี้ไปยัง Segment	73
รูปที่ 6-5 แสดง การหาตำแหน่งใน Segment โดยบวกกับค่า Offset	73
รูปที่ 6-6 แสดง การใช้ Ti ในการเลือก Descriptor Table	74
รูปที่ 6-7 แสดง Linux Paging Model	77
รูปที่ 6-8 แสดง การจองหน่วยความจำของเคอร์เนล	78
รูปที่ ก-1 แสดง หน้าจอในการติดตั้ง Linux Red Hat	81
รูปที่ ก-2 แสดง หน้าจอในการเลือกภาษาในการติดตั้ง	82
รูปที่ ก-3 แสดง หน้าจอในการคอนฟิกูเรชัน คีย์บอร์ด	82
รูปที่ ก-4 แสดง หน้าจอในการคอนฟิกูเรชัน เมาส์	83
รูปที่ ก-5 แสดง หน้าจอในการเลือกวิธีในการลงโปรแกรม	83
รูปที่ ก-6 แสดง หน้าจอในการเลือกวิธีในการแบ่งพาร์ทิชั่น	84
รูปที่ ก-7 แสดง หน้าจอในการแบ่งพาร์ทิชั่นโดยใช้ Disk Druid	84
รูปที่ ก-8 แสดง หน้าจอในการแบ่งพาร์ทิชั่นโดยใช้ Fdisk	85
รูปที่ ก-9 แสดง หน้าจอในขณะที่ใช้ Fdisk	85
รูปที่ ก-10 แสดง หน้าจอในการคอนฟิกูเรชันระบบเน็ตเวิร์ก	86
รูปที่ ก-11 แสดง หน้าจอในการคอนฟิกูเรชันระบบไฟร์วอลล์	86
รูปที่ ก-12 แสดง หน้าจอในการเลือกภาษาที่ใช้ในระบบปฏิบัติการลีนุกซ์	87
รูปที่ ก-13 แสดง หน้าจอในการเลือกโซนของเวลา	87
รูปที่ ก-14 แสดง หน้าจอในการเลือกการคอนฟิกูเรชันแอคเคาท์	88
รูปที่ ก-15 แสดง หน้าจอในการเลือกกลุ่มของแพกเกจที่ต้องการติดตั้ง	88
รูปที่ ก-16 แสดง หน้าจอในการเริ่มการติดตั้ง	89

บทที่ 1
บทนำ

1.1 ความสำคัญและที่มา

ในปัจจุบันนี้ระบบเทคโนโลยีสารสนเทศและระบบคอมพิวเตอร์มีผลต่อชีวิตประจำวันเป็นอย่างมากและเทคโนโลยีทางด้านนี้มีความเจริญก้าวหน้าอย่างรวดเร็วซึ่งส่วนหนึ่งของความเจริญก้าวหน้านี้ผู้ที่มีบทบาททางด้านเทคโนโลยีสารสนเทศทุกแขนงรวมทั้งผู้ที่ทำการคิดค้นระบบปฏิบัติการรุ่นแรก ๆ จนกระทั่งถึงปัจจุบันระบบปฏิบัติการได้ถูกคิดค้นขึ้นมาโดยนักวิชาการทางด้านคอมพิวเตอร์อย่างมากมายซึ่งในปัจจุบันระบบปฏิบัติการที่มีอยู่ตามท้องตลาดและในองค์การต่าง ๆ มีความแตกต่างกันทางประสิทธิภาพ ความเร็วขึ้นอยู่กับการใช้งานในแต่ละเหตุการณ์ซึ่งมีระบบปฏิบัติการหนึ่งซึ่งมีคุณลักษณะเฉพาะตัวมีความสามารถไม่แพ้ระบบปฏิบัติการชนิดอื่นคือระบบปฏิบัติการลีนุกซ์

คุณสมบัติเด่นของระบบปฏิบัติการลีนุกซ์ที่เห็นได้ชัดที่สุด คือ เป็นระบบปฏิบัติการที่เปิดซอร์ดโค้ดและเป็นระบบปฏิบัติที่สามารถใช้งานได้ฟรี ผู้ที่ใช้งานและผู้ที่จะศึกษาสามารถศึกษาหลักการของระบบปฏิบัติการได้ง่ายขึ้นกว่าในยุคแรก ๆ และระบบปฏิบัติการลีนุกซ์มีการพัฒนาอย่างต่อเนื่องโดยนักวิชาการคอมพิวเตอร์ทั่งโลกและโปรแกรมเมอร์ที่สำคัญ คือ นาย ลีนุส ทอร์วัลด์ (Linus Trovalds) เป็นผู้เริ่มต้นในการคิดระบบปฏิบัติการนี้ขึ้นมา ระบบปฏิบัติการชนิดนี้ผู้ใช้สามารถทำการแก้ไขซอร์ดโค้ดของระบบปฏิบัติการนี้ได้ด้วยตัวเองและทำการคอมไพล์ใหม่เพื่อปรับปรุงให้สามารถใช้งานได้หลากหลายมากขึ้นรวมทั้งระบบปฏิบัติการชนิดนี้มีการออกแบบเคอร์เนลที่สามารถทำการดาวน์โหลดมาใช้งานได้ตลอดเวลา ในปัจจุบันได้มีผู้พัฒนาระบบปฏิบัติการลีนุกซ์ออกมาหลายรายด้วยกันซึ่งแสดงให้เห็นว่าระบบปฏิบัติการนี้ได้รับความนิยมมากขึ้นเรื่อย ๆ จากผู้ใช้คอมพิวเตอร์ทั่วโลก

ในปัจจุบันนี้ผู้ที่มีความรู้ทางด้านระบบปฏิบัติการมีน้อยมากในประเทศไทย โดยส่วนใหญ่เป็นผู้ที่ใช้งานมากกว่าผู้ที่มีความคิดในด้านการสร้างระบบปฏิบัติการ เนื่องจากสาเหตุมาจากขาดแหล่งข้อมูลที่ดีและผู้ที่มีความรู้ทางด้านนี้อย่างลึกซึ่งมีน้อยซึ่งจุดริเริ่มที่เข้าใจระบบปฏิบัติที่ดี คือ อ่านจากแหล่งหนังสือหรือตามเว็บไซต์ต่าง ๆ และแนวทางที่ดีสำหรับความเข้าใจในระบบปฏิบัติการ คือ การแก้ไขซอร์ดโค้ดของระบบปฏิบัติการ ซึ่งทำให้ผู้ศึกษาสามารถเข้าใจระบบปฏิบัติการนั้นได้เป็นอย่างดี ระบบปฏิบัติการลีนุกซ์ได้รับการพัฒนามาจากระบบปฏิบัติการยูนิกซ์มีคุณสมบัติหลายอย่างที่มีความคล้ายคลึงกับระบบปฏิบัติการยูนิกซ์แต่ระบบปฏิบัติการลีนุกซ์ได้รับความนิยมบนเครื่องพีซีมากกว่าระบบปฏิบัติการยูนิกซ์ทำให้เป็นโอกาสที่ดีของผู้ใช้พีซีในการศึกษาการทำงานของระบบปฏิบัติมากกว่าการใช้งานระบบปฏิบัติการเพียงอย่างเดียว

การเข้าไปแก้ไขซอร์ดโค้ดของระบบปฏิบัติการลีนุกซ์จำเป็นต้องเข้าใจหลักการทำงานของระบบปฏิบัติการชนิดนี้ก่อนซึ่งส่วนที่สำคัญอย่างมาก คือ การจัดการโปรเซส (Process Management) , การจัดการหน่วยความจำ (Memory Management) , ทามเมอร์ (Timer) และ อินเตอร์รัพ (Interrupt) เป็นต้น เมื่อทำการศึกษาหลักการทำงานของระบบปฏิบัติการลีนุกซ์แล้วทำให้สามารถเข้าไปแก้ไขซอร์ดโค้ดในแต่ละส่วนได้ดีขึ้น เพราะซอร์ดโค้ดดั้งเดิมนั้นเป็นซอร์ดโค้ดภาษาซีซึ่งเข้าใจได้ยากถ้าผู้ทำการศึกษาไม่เข้า

ใจหลักการของระบบปฏิบัติการก่อนจะทำให้เข้าไปแก้ไขซอร์ดโค้ดได้ยาก นอกจากนั้นแล้วผู้ที่ทำการศึกษาระบบปฏิบัติการยังสามารถเข้าใจระบบปฏิบัติการได้ดีขึ้นเมื่อตอนใช้งานระบบปฏิบัติการนั้นด้วย

ลีนุกซ์อินเทอร์นอล (Linux Internal) นั้นได้กล่าวถึงงานด้านต่าง ๆ ของระบบปฏิบัติการ อาทิ เช่น สถาปัตยกรรมการออกแบบเคอร์เนลของลีนุกซ์ซึ่งเป็นชนิดสถาปัตยกรรมโมโนลิทิกรวมถึงการทำการรีคอมไพล์เคอร์เนลจากเคอร์เนลเดิมที่มีอยู่ การทำความเข้าใจเกี่ยวกับโปรเซส/เคอร์เนลโมเดล (The Process / Kernel Model) , สัญญาณซิสเต็มคอล (system call) , โปรเซสและการสร้างโปรเซส , การจัดการหน่วยความจำ

1.2 วัตถุประสงค์

- เพื่อศึกษาการทำงานของโครงสร้างภายในของระบบปฏิบัติการลีนุกซ์
- เพื่อให้สามารถทำการเขียนซอร์ดโค้ดในระบบปฏิบัติการลีนุกซ์ได้
- เพื่อศึกษาถึงความสามารถของระบบปฏิบัติการลีนุกซ์
- เพื่อศึกษาถึงการทำงานของระบบปฏิบัติการ

1.3 ขอบเขตของโครงงาน

- ศึกษาและทำความเข้าใจระบบปฏิบัติการลีนุกซ์
- ศึกษาว่าโปรเซสทำงานอย่างไรในระบบปฏิบัติการลีนุกซ์
- ศึกษาว่าลีนุกซ์ใช้เมมโมรี่อย่างไรในระบบปฏิบัติการลีนุกซ์
- เข้าใจซอร์ดโค้ดลีนุกซ์ว่าแต่ละไฟล์ใช้ทำงานอะไร
- แก้ไขซอร์ดโค้ดบางส่วนของระบบปฏิบัติการลีนุกซ์

1.4 วิธีการดำเนินงาน

การดำเนินงานเริ่มต้นโดยการศึกษาถึงการทำงานของเคอร์เนลในระบบปฏิบัติการลีนุกซ์ซึ่งเป็นชนิดสถาปัตยกรรมโมโนลิทิก (Monolithic Architecture) รวมถึงการทำงานของระบบปฏิบัติการลีนุกซ์โดยศึกษาควบคู่ไปกับซอร์ดโค้ดของส่วนต่าง ๆ อาทิ เช่น โปรเซส หน่วยความจำ เป็นต้น รวมทั้งการดาวน์โหลดเคอร์เนลรุ่นใหม่ ๆ เพื่อทำการรีคอมไพล์เคอร์เนล ท้ายที่สุดแล้วต้องทำการเข้าไปแก้ไขซอร์ดโค้ดและทำการคอมไพล์เคอร์เนลจากการแก้ไขซอร์ดโค้ดด้วยตนเองตลอดจนการออกแบบแล็ปวิชาระบบปฏิบัติการเพื่อเป็นแบบทดสอบให้กับผู้ที่ทำการศึกษาลีนุกซ์ อินเทอร์นอลต่อไปสำหรับส่วนหลักที่ทำการศึกษา คือ การจัดการโปรเซส (Process Mangement),ตารางการจัดการโปรเซส(Process Scheduling) โปรเซสและสถานะต่าง ๆ ของโปรเซส , ความสัมพันธ์ระหว่างโปรเซสพาเรนท์และโปรเซสไชล์ หลักการทำงานของเวทคิว (wait queue) , การจัดการหน่วยความจำ (Memory Management) ซึ่งสามารถสรุปขั้นตอนการทำงานต่าง ๆ ได้ดังนี้

1.5 ขั้นตอนการดำเนินงาน

1. ศึกษาถึงการทำงานของระบบปฏิบัติการทั่วไป
2. ศึกษาถึงเคอร์เนลของระบบปฏิบัติการลีนุกซ์
3. ศึกษาถึงหลักการทำงานของระบบปฏิบัติการลีนุกซ์
4. ทำการเปลี่ยนแปลงซอร์ดโค้ด
5. ทำการรีคอมไฟล์เคอร์เนลทั้งที่เป็นส่วนของเคอร์เนลที่ดาวน์โหลดมาและเคอร์เนลที่ทำการเปลี่ยนแปลง
6. ทำการออกแบบแลบที่เกี่ยวกับการทำงานของระบบปฏิบัติการลีนุกซ์

บทที่ 2
สถาปัตยกรรมเคอร์เนลและองค์ประกอบต่าง ๆ

2.1 ความหมายของเคอร์เนล

เคอร์เนลเป็นโมดุลต่าง ๆ ของลีนุกซ์ที่ทำหน้าที่จัดการทำงานด้านต่าง ๆ ของซีพียูเพื่อประกอบกันขึ้นมาเป็นระบบปฏิบัติการมีหน้าที่ดูแลส่วนต่าง ๆ ของระบบปฏิบัติการ (operating system) เช่น ดูแลในส่วนของ การจัดการโปรเซส (process management) , การจัดการหน่วยความจำ (memory management) , การจัดการเวลา (time sharing) , อินพุต (I/O) , การทำงานด้านต่าง ๆ ของโปรเซส (Time Multitasking), เน็ตเวิร์ค (network) , ดีไวซ์ไดรเวอร์ (device driver) หรือระบบไฟล์ที่มีใช้งานบนลีนุกซ์ เป็นต้น ซึ่งเคอร์เนลนั้นจะทำการโหลดเข้าไปยังหน่วยความจำตั้งแต่เริ่มสั่งให้คอมพิวเตอร์ทำงาน(start up)

2.2 เป้าหมายของเคอร์เนล (Kernel Goal)

ความเข้าใจ (Clarity)

เป้าหมายสูงสุดของเคอร์เนล คือ ความเร็ว (Fast) และความถูกต้อง(Correct) ส่วน ความเข้าใจนั้น (Clarity) ถือว่าเป็นเป้าหมายรองลงมา แต่ถ้าพูดถึงในแง่ที่จะพัฒนาโปรแกรมแล้ว ความเข้าใจและความถูกต้องนั้น ถือเป็นเป้าหมายแรกที่ให้ความสำคัญ รองลงมาคือความเร็ว ความเข้าใจก็เป็นส่วนหนึ่งของความถูกต้อง ถ้ามันเข้าใจง่าย ก็ง่ายที่จะพิสูจน์ความถูกต้อง และง่ายที่จะแก้ไขเมื่อเกิดข้อผิดพลาด

ความเข้ากันได้ (Compatibity)

ลีนุกซ์สามารถเข้ากันได้กับระบบอื่น เช่น ดอส ในตัวลีนุกซ์จะใช้ DOSEMU และความเข้ากันได้กับวินโดส์ในเรื่องของไฟล์ หรือการบริการพิมพ์โดยใช้แซมบ้า (SAMBA) รวมทั้งระบบการจัดการไฟล์ซึ่งลีนุกซ์จะสนับสนุน ext2 , ISO-9660 (ใช้ใน CDROM) ,MSDOS, NFS (ใช้กับระบบ Sun) และ AFFS (ใช้กับ Amiga) และเข้ากับระบบเน็ตเวอร์กต่างๆ ทั้ง TCP/IP, IPX, SPX, NCP

ความสามารถในการเข้ากันได้กับหลายรูปแบบ (Portability)

เป้าหมายอีกอย่างคือ ความเข้ากันได้กับฮาร์ดแวร์ มันเป็นความสามารถที่จะรันลีนุกซ์เคอร์เนลได้บนฮาร์ดแวร์ได้หลายรูปแบบ ในลีนุกซ์สามารถรันได้บนซีพียูทั้ง Alpha, ARM, Hitachi, SuperH, LA-64, MIPS, MIPS-64, Motorola 680x0, PA-RISC, PowerPC, SPARC และ SPARC-64

ความแข็งแกร่ง และ ความปลอดภัย (Robustness and Security)

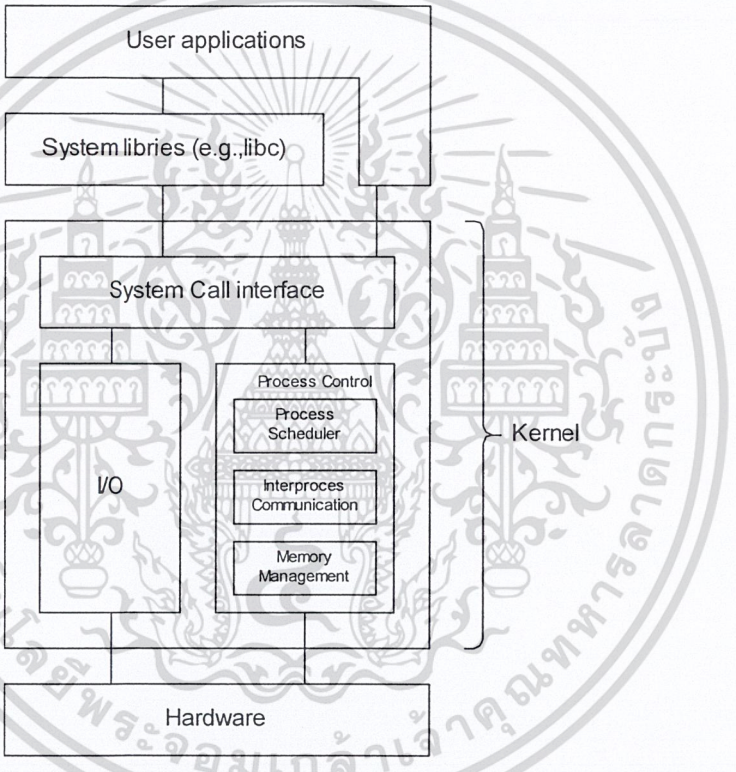
เป็นเป้าหมายที่สำคัญอีกอย่างหนึ่ง คือจะต้องไม่มีจุดผิดพลาด และสามารถป้องกันโปรเซสจากโพสอื่น และเช่นเดียวกัน ต้องสามารถป้องกันระบบออกจากระบบอื่นด้วย ซึ่งระบบลีนุกซ์เป็นระบบที่เปิด ทุกๆบรรทัดของเคอร์เนลจะถูกตรวจสอบจากนักพัฒนาทั่วโลก ซึ่งหนึ่งในนั้นอาจค้นพบจุดพิดพลาดและกระจายข่าวการผิดพลาดนี้ออกไปทำให้ระบบลีนุกซ์แก้ไขจุดผิดพลาดนี้ได้อย่างรวดเร็ว

ความเร็ว (Speed)

เป็นเป้าหมายที่สำคัญอีกอย่างหนึ่งในการออกแบบ การที่จะทำให้บรรลุวัตถุประสงค์ตามเป้าหมายความเร็วก็มีส่วนสำคัญ แม้ว่ามันจะอยู่ภายใต้ ความแข็งแกร่งและความปลอดภัย การทำให้ เคอร์เนลโค้ดให้ได้ดีที่สุด ความเร็วก็จะเพิ่มขึ้นด้วย เวลาที่จะแก้ไขโค้ดก็จำเป็นที่จะต้องคำนึงถึงความเร็วด้วย

2.3 โครงสร้างการทำงานของเคอร์เนล (Look at the Kernel's Architecture)

เคอร์เนลจะเป็นตัวแบ่งระหว่างแอพพลิเคชั่นและฮาร์ดแวร์ และระหว่างแอพพลิเคชั่นและเคอร์เนลก็จะมี Stand C library (libc) เป็นตัวคั่นอีกที โดยปกติแล้วผู้ใช้โปรแกรมจะไม่ได้ติดต่อกับเคอร์เนลโดยตรง แต่จะติดต่อผ่านทาง Standard C library (libc) แต่ในบางกรณีก็จะไม่ผ่าน ดังรูปที่ 2-1



รูปที่ 2-1 รายละเอียดโครงสร้างภายในของเคอร์เนล

โปรเซสจะติดต่อกับเคอร์เนลดังนี้

1. ผู้ใช้โปรแกรมจะร้องขอซิสเต็มคอลล์ (System call) โดยปกติจะผ่านทาง libc
2. การคอลล์จะผ่านทางฟังก์ชั่น system_call ซึ่งจะส่งความต้องการผ่านไปยังฟังก์ชั่นอื่น ตามที่ต้องการออกไป
3. ฟังก์ชั่นจะติดต่อกับโค้ดภายในของโมดูล (Module) และอาจติดต่อกับโมดูลอื่นหรือกับฮาร์ดแวร์ด้วยก็ได้
4. ผลลัพท์จะถูกส่งกลับมาทางเดิมที่ใช้ในการติดต่อ

2.4 คุณสมบัติที่สำคัญของระบบปฏิบัติการลีนุกซ์

นอกจาก Linux เป็นระบบปฏิบัติการที่เปิดซอร์ดโค้ดแล้วยังสามารถมีข้อดีข้ออื่นด้วย คือ

มัลติทาส์กิง (Multitasking)

คือ การที่ระบบสามารถทำการรันโปรเซสได้หลายโปรเซสพร้อมกันในช่วงระยะเวลาใดเวลาหนึ่งโดยซีพียูแบ่งช่วงเวลาให้โปรเซสทำการครอบครองซีพียูซึ่งในระบบปฏิบัติการลีนุกซ์มีความสามารถในการทำมัลติทาสคิ้งหรือมัลติโปรแกรมมิง

การทำเพียงโปรเซสเดียวในช่วงเวลาหนึ่งนั้นเป็นสิ่งที่ทำให้ระบบปฏิบัติการมองดูไม่มีประสิทธิภาพ ซึ่งทำให้โปรเซสต้องหยุดรอบ่อย ๆ เช่น รออินพุทจาก I/O Device , คีย์บอร์ดหรือจากดิสก์เป็นต้น ซึ่งคุณสมบัติในการทำมัลติทาสกิ้งช่วยให้ระบบปฏิบัติการทำงานได้อย่างมีประสิทธิภาพยิ่งขึ้นเพราะทำให้ซีพียูทำงานได้อย่างเต็มประสิทธิภาพของเวลาที่เสียไป

มัลติยูสเซอร์ (Multiuser)

คือ สามารถใช้งานพร้อมกันได้หลาย ๆ คนในเวลาเดียวกัน คุณลักษณะนี้โดยมากมักมีเฉพาะกับเครื่องใหญ่ เช่น มินิคอมพิวเตอร์ หรือ เครื่องเมนเฟรม

2.5 สถาปัตยกรรมของเคอร์เนล (Kernel Architecture)

ระบบปฏิบัติการนั้นมีสถาปัตยกรรมการออกแบบได้ 2 แบบ คือ สถาปัตยกรรมไมโครเคอร์เนล (microkernels) และสถาปัตยกรรมโมโนลิทิก (monolithic kernel) บางครั้งเรียกว่ามาโครเคอร์เนล (macrokernels)

สถาปัตยกรรมไมโครเคอร์เนล (Microkernels Architecture)

ไมโครเคอเนลมาจากแนวคิดในการออกแบบระบบปฏิบัติการแบบโมดูล โดยลดขนาดของส่วนการทำงานหลักของระบบปฏิบัติการให้มีขนาดเล็กเพื่อให้มีการทำงานที่รวดเร็วและลดความซับซ้อน ทำให้มีข้อผิดพลาดน้อยลง โดยระบบปฏิบัติการแรกที่พัฒนาในรูปของไมโครเคอเนล คือ Mach และได้รับความนิยมเรื่อยมาในระบบปฏิบัติการรุ่นใหม่ ๆ เช่น Windows2000 ซึ่งนอกจากจะทำให้ระบบปฏิบัติการมีความยืดหยุ่นมากขึ้นแล้วยังมีความเป็นไปได้ในการพอร์ทระบบปฏิบัติการไปใช้บนแพลทฟอร์มอื่นมากขึ้นด้วย

ระบบปฏิบัติการแบบเดิมมีโครงสร้างแบบระดับชั้น (layered operating system) โดยแต่ละระดับจะทำหน้าที่ที่แตกต่างกันและมีการประมวลผลในโหมดเคอร์เนล ซึ่งทำให้เกิดปัญหาตามมาเมื่อมีการเปลี่ยนแปลงของบริการบางอย่าง ซึ่งอาจจะเกิดผลกระทบกับระดับอื่น การหาข้อผิดพลาดในการทำงานและเป็นเรื่องที่ยากในการพัฒนาระบบรักษาความปลอดภัย

แนวคิดเรื่อง microkernel คือ การนำส่วนบริการบางอย่างออกจากเคอร์เนล ให้เหลือเพียงงานหลักๆ ของระบบปฏิบัติการ โดยไมโครเคอรเนลจะทำงานในโหมดเคอนัลและส่วนบริการอื่นๆ จะทำงานในระดับผู้ใช้

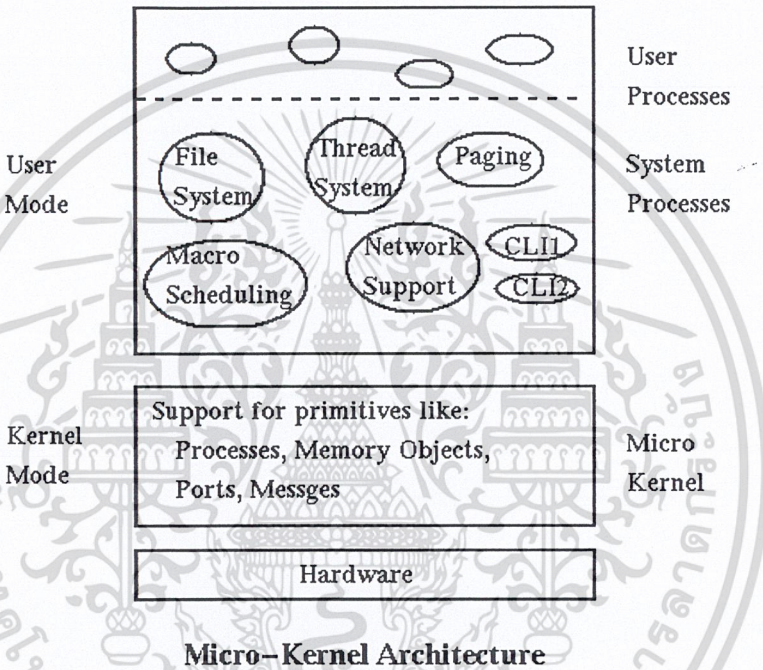
การติดต่อระหว่างโปรเซสในยูสเซอร์โหมด (user mode) จะทำผ่านไมโครเคอร์เนลโดยการส่งเมสเสจ เช่น ถ้าแอพพลิเคชั่นต้องการเปิดไฟล์ จะส่งเมสเสจไปยังไฟล์เซอร์เวอร์ (file server) ผ่านไมโครเคอร์เนล หรือถ้าต้องการสร้างโปรเซสหรือเธรดจะส่งเมสเสจไปยังโปรเซสเซอร์เวอร์ (process server) เป็นต้น ดังนั้นหน้าที่อย่างหนึ่งของไมโครเคอร์เนล คือ การส่งผ่านเมสเสจ (message exchange) ซึ่งในขั้นตอนการส่งผ่านจะสามารถตรวจสอบเมสเสจได้ว่าเป็นเมสเสจที่ถูกต้องหรือไม่

ข้อดีของไมโครเคอร์เนล คือ

- Uniform interface ในการร้องขอบริการ แอพพลิเคชันสามารถทำได้โดยการส่งเมสเสจ (message passing) โดยไม่เกิดความแตกต่างทั้งในระดับ user mode หรือ kernel mode
- Extensibility การเพิ่มเติมคุณสมบัติของระบบปฏิบัติการเพื่อรองรับอุปกรณ์ฮาร์ดแวร์ใหม่ ๆ หรือรูปแบบการทำงานของซอร์ฟแวร์ใหม่ ๆ สามารถทำได้ง่ายขึ้นและมีผลกระทบกับส่วนอื่นน้อยลงรวมทั้งสามารถมีรูปแบบของการบริการที่แตกต่างกันในบริการประเภทเดียวกันได้ เช่น การรองรับระบบไฟล์หลาย ๆ แบบ นอกจากนี้การอัพเดทฟังก์ชันบางอย่างของระบบปฏิบัติการจะทำได้ง่ายขึ้น โดยจะทำการอัพเดทเฉพาะโมดูลที่มีการแก้ไขเท่านั้น ซึ่งถ้าเป็นระบบปฏิบัติการแบบเก่า อาจจะต้องทำการติดตั้งระบบใหม่เลยก็ได้
- Flexibility นอกจากการเพิ่มคุณสมบัติใหม่ ๆ เข้าไปในระบบปฏิบัติการแล้ว คุณสมบัติเดิมยังสามารถปรับปรุงให้มีขนาดเล็กลงและมีความเร็วมากขึ้นด้วยหรือการที่ผู้ใช้สามารถเลือกคุณสมบัติต่าง ๆ ให้เหมาะสมกับการใช้งาน โดยไม่จำเป็นต้องติดตั้งทุก ๆ คุณสมบัติของระบบปฏิบัติการลงไปทั้งหมด
- Portability ด้วยสถาปัตยกรรมแบบไมโครเคอร์เนล ที่มีเฉพาะส่วนของไมโครเคอร์เนลเท่านั้นที่ทำการติดต่อกับฮาร์ดแวร์ ดังนั้นส่วนที่เป็นคำสั่งเฉพาะของโปรเซสเซอร์จะอยู่เฉพาะในไมโครเคอรเนลเท่านั้น การย้ายระบบปฏิบัติการไปทำงานยังแพล็ตฟอร์มอื่นจะเป็นการเปลี่ยนแปลงเฉพาะส่วนของไมโครเคอร์เนลเท่านั้น ส่วนอื่น ๆ ยังคงสามารถใช้งานได้ โดยอาจจะมีการเปลี่ยนแปลงไม่มากนัก
- Reliability ซอฟต์แวร์ที่มีขนาดใหญ่หรือที่มีความซับซ้อนมากย่อมมีโอกาสที่จะเกิดข้อผิดพลาดได้มากตามไปด้วย แต่การที่ไมโครเคอร์เนลมีขนาดที่เล็กจึงสามารถทดสอบข้อผิดพลาดต่าง ๆ ได้ง่าย แม้ว่าบริการอื่น ๆ ของระบบปฏิบัติการอาจจะเกิดข้อผิดพลาด แต่ส่วนของไมโครเคอร์เนลจะเป็นส่วนที่มีเสถียรภาพมากที่สุด นอกจากนี้ในไมโครเคอร์เนลจะมีฟังก์ชัน API ที่ใช้ในการติดต่อไม่มากนัก ทำให้สามารถควบคุมข้อผิดพลาดที่อาจจะเกิดขึ้นได้ดียิ่งขึ้น
- Distributed system support สถาปัตยกรรมแบบไมโครเคอร์เนลสามารถพัฒนาให้อยู่ในรูปแบบของระบบการทำงานแบบกระจาย รวมทั้งการควบคุมคลัสเตอร์โดยระบบปฏิบัติการ

แบบกระจายได้ เมื่อเมสเสจถูกส่งจากไคลเอนต์หนึ่งไปยังเซอร์เวอร์โปรเซส เมจเสจนั้นจะต้องระบุ identifier ของเซอร์วิสที่ต้องการ และไมโครเคอร์เนลจะเป็นตัวกลางที่จะส่งแมสเสจนั้นไปยังเครื่องที่มีเซอร์วิสนั้นให้บริการอีกทีหนึ่ง

- Support for object-oriented operating system (OOOS) สถาปัตยกรรมแบบไมโครเคอร์เนลจะรองรับการออกแบบในรูปแบบเชิงวัตถุ โดยองค์ประกอบของระบบปฏิบัติการจะเรียกว่าคอมโนเนนท์ (ออปเจคท์) ซึ่งมีการกำหนดอินเตอร์เฟซที่ใช้ในการติดต่อกับคอมโพเนนท์อื่นและการสื่อสารระหว่างคอมโพเนนท์ทำได้โดยการส่งผ่านแมสเสจผ่านไมโครเคอร์เนล



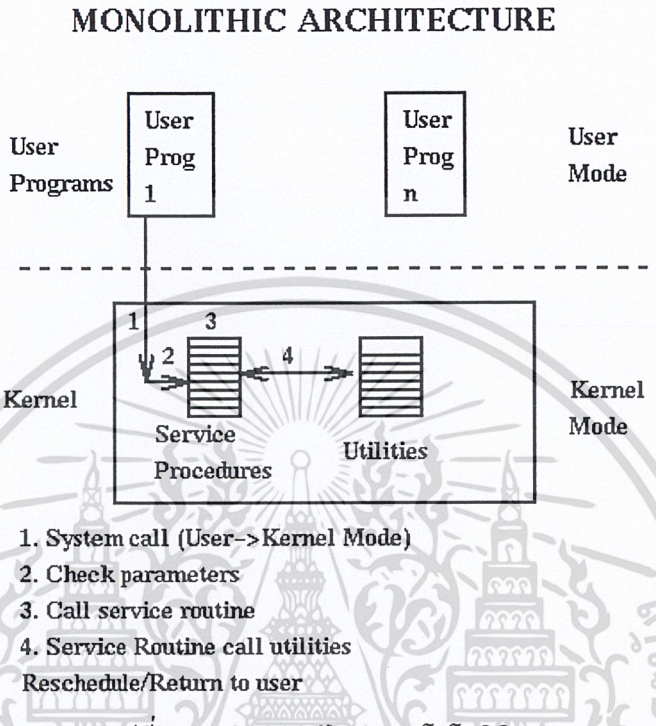
Micro–Kernel Architecture

รูปที่ 2-2 แสดงสถาปัตยกรรมไมโครเคอร์เนล

สถาปัตยกรรมโมโนลิทิก (Monolithic Kelnel Architecture)

สถาปัตยกรรมโมโนลิทิก (Monolithic kernel) เปรียบเสมือนโปรเซสใหญ่ ๆ หนึ่งโปรเซสโดยที่มันอาจจะแบ่งย่อยเป็นโมดูล (หรือเลเยอร์) แต่ในขณะที่รันโมดูลทั้งหมดจะรวมกันเป็นสิ่งเดียวและมีขนาดใหญ่โดยที่โมดูลดังกล่าวจะติดต่อกันโดยตรงโดยการเรียกฟังก์ชันระหว่างกัน

ระบบปฏิบัติการลีนุกซ์เป็นระบบปฏิบัติการชนิดสถาปัตยกรรมโมโนลิทิก(Monolithic Architecture) ซึ่งมีลักษณะเป็นกลุ่มก้อนอันเดียวก้อนเดียว สถาปัตยกรรมการออกแบบชนิดนี้เป็นชนิดที่เก่าแก่ที่สุดของระบบปฏิบัติการ มีข้อดี คือ มี คุณภาพ (performance) มากเพราะมีลักษณะเป็นกลุ่มก้อนทำให้มีโอเวอร์เฮด (overhead) น้อยกว่าสถาปัตยกรรมของเคอร์เนลแบบอื่น



รูปที่ 2- 3 แสดงสถาปัตยกรรมโมโนลิทิก

2.6 ซอร์ดโค้ดที่สำคัญในระบบปฏิบัติการลีนุกซ์ (The Linux Kernel Source)

หัวข้อนี้อธิบายถึงซอร์ดโค้ดในไดเรคทอรี่ต่างว่าแต่ละไฟล์ทำงานเกี่ยวกับอะไรทำให้มองลีนุกซ์ในภาพรวมได้กว้างขึ้น

เมื่อเข้ามายังไดเรคทอรี่ /usr/src/linux (ตำแหน่งไดเรกทอรี่นี้เป็นพาเรนท์ไดเรกทอรี่ของเคอร์เนล) ทำให้เห็นไดเรคทอรี่ต่าง ๆ ดัง ต่อไปนี้

arch ไดเรกทอรี่นี้จะมีไฟล์ที่เกี่ยวกับสถาปัตยกรรมที่ support กับเคอร์เนล ซึ่งจะประกอบด้วยสถาปัตยกรรมมากมาย เช่น สถาปัตยกรรม i386 , alpha เป็นต้น

include ไดเรกทอรี่นี้เป็นที่สำหรับเก็บเฮดเดอร์ไฟล์

doc ไดเรกทอรี่นี้เป็นที่เก็บข้อมูลต่าง ๆ ของเคอร์เนล

ไฟล์ที่สำคัญที่สุด คือ ไฟล์ sched.c เป็นไฟล์หลักของเคอร์เนล ควบคุมในเรื่องของเวลาบนระบบการกำหนดให้โปรเซสพักหรือเริ่มทำงานอีกครั้ง (กำหนดให้โปรเซสอยู่ในสถานะรอหรือเริ่มรันโปรเซสอีกครั้งหนึ่ง) เป็นไฟล์ที่เกี่ยวกับการจัดการเวลาของลีนุกซ์และมีโค้ดที่เกี่ยวกับการหยุดการรันโปรเซสหรือรันโปรเซส รวมทั้งลำดับความสำคัญของโปรเซส และฟังก์ชันสำหรับแอ็คเซสหรือกำหนดข้อมูลระดับของผู้ใช้งาน

นอกจากนี้ยังมีไฟล์อื่นที่ทำหน้าที่เกี่ยวกับการควบคุมโปรเซส เช่น ฟังก์ชันระบบ fork หรือ exit หรือฟังก์ชันควบคุมการส่งสัญญาณอินเตอร์รัพ , signal.c , ไฟล์ module.c และ ksym.c จะควบคุมกลไกการเพิ่มหรือลบโมดุลออกจากเคอร์เนล

นอกจากนี้ยังมีไฟล์อื่นที่ทำหน้าที่เกี่ยวกับการควบคุมโปรเซส เช่น ฟังก์ชันระบบ fork หรือ exit หรือฟังก์ชันควบคุมการส่งสัญญาณอินเตอร์รัพ , signal.c , ไฟล์ module.c และ ksym.c จะควบคุมกลไกการเพิ่มหรือลบโมดุลออกจากเคอร์เนล

ซอร์ฟไฟล์ที่เหลือในไดเรกทอรีของเคอร์เนล จะทำหน้าที่เกี่ยวกับการติดต่อกับโปรแกรมของผู้ใช้งาน เช่น read.c อ่านข้อมูลจากโปรแกรมของผู้ใช้งานและเขียนข้อมูลลงในโครงสร้างข้อมูลของเคอร์เนล , resource.c จะร้องขอหรือยกเลิกการจองพอร์ต I/O ของระบบ , dma.c จะร้องขอหรือยกเลิกการจองช่องทางในการใช้งาน DMA , printk.c จะควบคุมการแสดงข้อความต่าง ๆ ของเคอร์เนล , sys.c จะควบคุมการกำหนดคอนฟิกกูเรชั่นของระบบ รวมทั้งฟังก์ชั่นที่เกี่ยวกับการควบคุมระดับการใช้งาน เช่น uname หรือ setid เป็นต้น

ไดเร็กทอรี่ fs

ระบบไฟล์แต่ละชนิดที่มีใช้งานบนลีนุกซ์จะแม็พผ่าน VFS(Virtual File System) อย่างไรก็ตาม รายละเอียดการทำงานของแต่ละไฟล์ในไดเรกทอรี่นี้เป็นสิ่งสำคัญ ดังมีรายละเอียดต่อไปนี้

ไฟล์ exec

ฟังก์ชันระบบที่สำคัญที่สุดบนยูนิกซ์ก็คือ exec ถึงแม้ว่าจะมีรูปแบบให้ใช้งาน 6 รูปแบบด้วยกัน แต่ที่จริงแล้วจะมีเพียงฟังก์ชันเดียว คือ execve โดยฟังก์ชันจะใช้ตารางเก็บข้อมูลรูปแบบของไฟล์ Binary ต่าง ๆ ที่ลงทะเบียนไว้เพื่อค้นหาไฟล์ที่จะรัน

รูปแบบข้อมูลของไฟล์ Binary ที่รันโดยฟังก์ชัน exec แต่ละชนิดจะเก็บไว้ในโครงสร้างข้อมูลและกำหนดการทำงานกับข้อมูล 3 รูปแบบต่อไปนี้ คือ การโหลดไฟล์ไบนารีหรือไฟล์ไลบรารีรวมทั้งการโหลดไฟล์ส่วนประกอบภายในของเคอร์เนล

โครงสร้างข้อมูลรูปแบบไฟล์ไบนารี่จะเก็บไว้ในไฟล์ <linux/binfmts.h> อย่างไรก็ตามรูปแบบของไฟล์ไบนารีส่วนใหญ่จะสนับสนุนเฉพาะการโหลดไฟล์เท่านั้น

ไฟล์ devices.c และ block_dev.c

ไฟล์ devices.c จะทำหน้าที่เพิ่มและลบไฟล์ดีไวซ์ออกจากระบบ นอกจากนี้ยังกำหนดฟังก์ชันที่จะใช้เปิดและปิดไฟล์ดีไวซ์ สนับสนุนการโหลดโมดุลที่เคอร์เนลต้องการโดยอัตโนมัติ ส่วนไฟล์ block_dev.c จะเก็บฟังก์ชันในการอ่านหรือเขียนข้อมูลไปยังอุปกรณ์ ฟังก์ชันดังกล่าวจะทำงานที่เกี่ยวข้องทุกอย่าง ยกเว้นการควบคุมการส่งข้อมูลจริง ๆ จะมีฟังก์ชันทำงานต่างหาก

VFS

เป็นส่วนหนึ่งของซอร์ฟแวร์ในการจัดการกับระบบไฟล์ของลีนุกซ์ ทุก ๆ ระบบไฟล์จะทำงานผ่านซอร์ฟแวร์นี้เพื่อให้การแอ็คเซสข้อมูลบนแต่ละระบบไฟล์เป็นไปในรูปแบบเดียวกัน โดยใช้โครงสร้างข้อมูล file_operations (ซึ่งเป็นส่วนหนึ่งของ VFS) , เคอร์เนลจะแอ็กเซสข้อมูลบนระบบไฟล์ต่าง ๆ โดยใช้การ mount เพื่อแอ็คเซสโครงสร้างข้อมูลของ superblock (บล็อกแรกบนดิสก์) ไฟล์ super.c จะรวมซอร์สโค้ดสำหรับทำงานกับ superblock เช่น อ่าน เมาท์หรือยกเลิกการเมาท์ นอกจากนี้ไฟล์ยังทำหน้าที่ส่งข้อมูลกลับเมื่อมีการใช้คำสั่ง /proc/mounts และ /proc/filesystems

ฟังก์ชัน register_filesystem และ unregister_filesystem จะทำหน้าที่เพิ่มและลบระบบไฟล์ออกจากเคอร์เนลก็กำหนดไว้ในไฟล์ super.c

inode

ลีนุกซ์จะใช้ inode เก็บตำแหน่งของข้อมูลบนดิสก์ โดยแต่ละ inode จะใช้หมายเลขที่ไม่เหมือนกันเป็นคีย์เมื่อผู้ใช้งานส่งตำแหน่งของไฟล์ที่ต้องการเข้าไป

เคอร์เนลจะแปลเป็นหมายเลข inode โดยจะใช้หน่วยความจำแคชเก็บตารางแม็พข้อมูลดังกล่าว ซึ่งจะมีอยู่ 2 ส่วนคือ หน่วยความจำแคชเก็บโครงสร้างข้อมูล inode รวมทั้งฟังก์ชันในการอ่านและเขียนข้อมูล inode จะกำหนดไว้ในไฟล์ inode.c และหน่วยความจำแคชเก็บชื่อไฟล์ของ inode นั้น ไฟล์ dcache.c จะกำหนดกลไกจัดการหน่วยความจำแคช

ฟังก์ชั่นระบบที่ใช้งานหน่วยความจำแคช เช่น sys_mkdir, sys_symlink, sys_rename จะกำหนดไว้ในไฟล์ name.c

หน่วยความจำแคชที่มีความจุมากที่สุด จะกำหนดขึ้นมาโดยไฟล์ buffer.c สำหรับเก็บข้อมูลที่ใช้งานบ่อยๆ นอกจากนี้ยังมีไฟล์ file_table.c จะกำหนดหรือยกเลิกโครงสร้างข้อมูล file

open.c

ไฟล์ในไดเร็กทอรี่ fs ส่วนใหญ่แล้วจะทำงานกับไฟล์, ไฟล์ open.c ก็เช่นกันจะทำงานโดยใช้ฟังก์ชั่นระดับต่ำหลายฟังก์ชั่น เช่น open.c, do_open.c หรือ sys_close นอกจากนี้ยังมีฟังก์ชั่นที่ใช้สำหรับแก้ไขข้อมูลของ inode เช่น chown, chmod, chroot, chdir หรือ fcdir และฟังก์ชั้นที่เกี่ยวกับเวลา เช่น utime และ utimes ก็กำหนดไว้ที่ไฟล์นี้เช่นกัน

read_write.c และ readdir.c

ไฟล์ read_write.c จะประกอบด้วยฟังก์ชั่น read และ write สำหรับอ่านและเขียนข้อมูลบนไฟล์อย่างง่ายๆ อย่างไรก็ตามการอ่านและเขียนข้อมูลจริงจะเกิดขึ้นโดยผ่านฟังก์ชั่นที่กำหนดใน flip -> f_op นอกจากนี้ read_write.c ยังมีฟังก์ lseek และ llseek เพื่อค้นหาตำแหน่งที่ต้องการ สำหรับ llseek จะใช้ตัวแปร loff_t(long long) กำหนดตำแหน่งเพื่อเพิ่มความสามารถ lseek ให้สามารถแอ็คเซสตำแหน่งได้สูงขึ้น

การอ่านข้อมูลในไดเร็กทอรี่จะใช่ฟังก์ชั่นระบบ readdir อ่านข้อมูลทีละไดเร็กทอรีหรือ getdents ซึ่งสามารถจะอ่านข้อมูลได้ทีละหลายไดเร็กทอรี แต่ก็สนับสนุนเมื่อใช้ libc-5.2 ขึ้นไป

select.c

เป็นโค้ดสำหรับฟังก์ชั่น select อย่างไรก็ตามโค้ดค่อนข้างจะยากต่อการทำความเข้าใจเนื่องจากโครงสร้างข้อมูลค่อนข้างซับซ้อน แต่ก็เป็นจุดเริ่มต้นที่ดีสำหรับการศึกษาซอร์สโค้ดเนื่องจากโค้ดจะไม่เกี่ยวข้องกับฟังก์ชั่นอื่นมากนัก

pipes และ fifos

โค้ดที่สร้างช่องทางในการเชื่อมต่อระหว่างโปรเซส (pipe และ fifos) จะคล้ายกับแคแรกเตอร์ไดรเวอร์ โดยทั่วไปทั้งสองแบบข้างต้นจะใช้โค้ดเดียวกันยกเว้นการเปิด Pipe และ fifos จะแตกต่างกันสำหรับ fifo ทุกฟังก์ชั่นยังมีโครงสร้างข้อมูล inode_operations และ file_operation กำหนดไว้ที่ไฟล์ fifo.c โดยเฉพาะ

สิ่งหนึ่งที่น่าสนใจของ Pipe และ fifo ก็คือ Pipe.c จะกำหนดโครงสร้างข้อมูล file_operation ไว้ 2 โครงสร้างข้อมูลสำหรับอ่านและเขียนข้อมูล

ฟังก์ชั่นควบคุมการทำงานของไฟล์

ฟังก์ชั่นควบคุมการทำงานของไฟล์ดีไวซ์หรือ socket จะสร้างขึ้นโดยซอร์สโค้ด fcntl.c หรือ iotl.c, ไฟล์ fcntl.c จะกำหนดคำสั่งที่เกี่ยวข้องไว้ล่วงหน้าเกือบทั้งหมด นอกจากนี้ยังมีโค้ดของฟังก์ชั่น dup และ kill_fasync รวมอยู่ด้วย

ฟังก์ชั่นสำหรับล็อกไฟล์

ไฟล์ lock.c จะสร้างคำสั่งของฟังก์ชั่น fcntl รวมทั้งฟังก์ชั่นระบบ flock และจักการกับออพชันต่างๆ

นอกจากนี้แล้วในไดเร็กทอรียังมีไฟล์ stat.c ที่กำหนดฟังก์ชันระบบ stat, lstat, readlink ขึ้นมา ไฟล์ในไดเร็กทอรีนี้จะขึ้นกับแพล็ตฟอร์มที่ใช้งานประกอบด้วยฟังก์ชันสำหรับจัดการกับหน่วยความจำ (Paging) จองหรือยกเลิกการจองหน่วยความจำ

ฟังก์ชันการสลับเปลี่ยนข้อมูลระหว่างพื้นที่หน่วยความจำและดิสก์

swap.c จะทำงานเมื่อมีการระบุออพชั่น swap= และ Buff= ค่าเหล่านี้สามารถจะปรับได้โดยใช้ฟังก์ชัน sysctl

swap_state.c จะทำงานเกี่ยวกับหน่วยความจำแคช เป็นไฟล์ที่ทำความเข้าใจได้ยากที่สุดในไดเร็กทอรีนี้

swapfile.c จัดการระบบสับเปลี่ยนพื้นที่หน่วยความจำดิสก์ นอกจากนี้ยังมีการสร้างฟังก์ชัน swapon และ swapoff เพื่ออนุญาต หรือห้ามการสับเปลี่ยนพื้นที่ดังกล่าว

vmscan.c สร้างระบบหน้าของหน่วยความจำ (paging policies) เช่น ฟังก์ชันตรวจสอบพื้นที่หน่วยความจำที่จะนำไปเก็บยังดิสก์

page_io.c เป็นโค้ดระดับต่ำควบคุมการส่งผ่านข้อมูลระหว่างพื้นที่หน่วยความจำและดิสก์

การจองและยกเลิกการจองหน่วยความจำ

การจองและยกเลิกการจองหน่วยความจำจะใช้ฟังก์ชันในไดเร็กทอรีนี้

kmalloc.c จะจองและยกเลิกการจองหน่วยความจำ พื้นที่จองได้จะประกอบด้วยพื้นที่ส่วนย่อย ๆ ที่เท่ากันหลายส่วน เป็นหน้าที่ของ kmalloc.c จะต้องจัดการกับลิงค์ลิสต์ข้อมูลเหล่านี้

ฟังก์ชัน page_alloc.c จะกำหนดฟังก์ชันเพื่อทำงานต่าง ๆ ที่เกี่ยวข้อง เช่น ฟังก์ชัน __get_free_page จะกำหนดพื้นที่หน่วยความจำที่ยังไม่มีการใช้งาน หรือ try_to_free_pages เพื่อยกเลิกการจองหน่วยความจำที่ยังมีการใช้งานอยู่

ฟังก์ชัน vmalloc.c จะกำหนดฟังก์ชัน vmalloc , vremap และ vfree เพื่อทำงานกับพื้นที่หน่วยความจำเสมือน ส่วนใหญ่จะใช้กับบัฟเฟอร์ของ PCI ที่ตำแหน่งแอดเดรสสูง ๆ

ไฟล์อื่น ๆ

ส่วนที่สำคัญที่สุดในระบบจัดการหน่วยความจำของลีนุกซ์ จะอยู่ที่ไฟล์ memory.c ซึ่งจะควบคุมฮาร์ดแวร์ในการทำงานเกี่ยวกับการสับเปลี่ยนข้อมูล ในฟังก์ชันนี้จะกำหนดหลาย ๆ ฟังก์ชันขึ้นมา เช่น verify_area หรือ remap_page_range ส่วนฟังก์ชันนี้ที่เหลือจะเป็นฟังก์ชันระดับต่ำทำงานเกี่ยวกับการจัดการตารางแม็พข้อมูล โดยใช้โครงสร้างข้อมูล struct vm_operations นอกจากนี้แล้วยังสามารถกำหนดให้อ่านข้อมูลล่วงหน้าได้อีกด้วยที่โครงสร้างข้อมูล file , มีไฟล์ mmap.c ทำหน้าที่แม็พแอดเดรสระหว่างแอดเดรสจริงและแอดเดรสเสมือน

นอกจากนี้ยังมีฟังก์ชันระบบ 4 ฟังก์ชันสำหรับล็อกและยกเลิกการล็อกพื้นที่หน่วยความจำ จะกำหนดไว้ในไฟล์ mlock.c , ฟังก์ชัน mprotect.c ทำหน้าที่ป้องกันการละเมิดการแอ็คเซสหน่วยความจำโดยไม่มีเพอร์มิสชั่นที่ถูกต้อง ทั้งสองฟังก์ชันข้างต้นจะทำงานโดยกำหนดแฟลกพื้นที่หน่วยความจำหน้าที่ต้องการ

บทที่ 3
การคอมไพล์เคอร์เนลและการดีบักด้วย GDB

3.1 การคอมไพล์เคอร์เนล

เมื่อเรา download ไฟล์เคอร์เนลตัวใหม่จากเว็บ www.kernel.org มาแล้ว (โดยทั่วไปแล้วไฟล์ที่ download มาเป็นไฟล์ที่มีนามสกุล .tar.gz หรือ tgz คือตัวเดียวกัน) ทำการแตกไฟล์โดยใช้คำสั่ง

```
# tar  -xzvf  linux-x.x.xx.tar.gz
```

เมื่อทำการแตกไฟล์แล้ว คำสั่งจะสร้างไดเร็กทอรี่ที่ชื่อว่า linux ขึ้นมา และทำการแตกไฟล์ต่างๆ ที่ใช้สำหรับการคอมไพล์ให้ไปอยู่ภายใต้ไดเรกทอรี่ linux ดังรูป

```
[root@localhost adul]# cd linux
[root@localhost linux]# ls
arch          Documentation  include  kernel        Makefile  README           scripts
COPYING       drivers        init     lib           mm        REPORTING-BUGS
CREDITS       fs             ipc      MAINTAINERS   net       Rules.make
[root@localhost linux]#
```

รูปที่ 3-1 แสดงไฟล์หลังจากการขยายไฟล์ของเคอร์เนล

ขั้นตอนในการคอมไพล์เคอร์เนล

make mrproper

คำสั่งนี้เป็นการเตรียมความพร้อมก่อนการคอมไพล์ โดยจะทำการตรวจสอบว่ามี .o ค้างอยู่ภายใต้ไดเร็กทอรี่นี้อยู่บ้างหรือไม่ ถ้ามีอยู่จะถูกลบทิ้ง นอกจากนั้นยังลบไฟล์ .config ด้วย แสดงการใช้คำสั่ง

```
#make mrproper
```

make config

คำสั่งนี้เป็นคำสั่งให้เราเลือกรายการที่คอมไพล์ เช่น ใช้ซีพียูตัวใด มีอะไรบ้างในสิ่งแวดล้อมภายในระบบ ซึ่งเราสามารถเปลี่ยนจาก config เป็นคำสั่งอื่น ๆ ได้ แสดงการใช้คำสั่งดังต่อไปนี้

#make config	ใช้ในการเลือกรายการที่ต้องการคอมไพล์ แต่ละตัวโดยละเอียด
#make menuconfig	จะมีลักษณะการเลือกรายการ เหมือนกับ make config แต่จะเป็นเมนูให้เลือก
#make xconfig	จะเลือกรายการที่ภายใต้ Xwindows
#make oldconfig	เหมือนกับ make config แต่ใช้ค่าปกติ (default) ในการเลือกรายการที่ต้องการคอมไพล์

เพื่อความสะดวกและความปลอดภัยในการคอนฟิกเคอร์เนลในขั้นตอนนี้ถ้าคอนฟิกเคอร์เนลไม่ดีอาจทำให้การคอมไพล์เคอร์เนลมีปัญหาได้ดังนั้นควรจะทำการก็อบปี้คอนฟิกของเคอร์เนลที่มั่นใจว่าคอมไพล์ผ่านแล้วไปไว้ที่ไดเรกทอรี่ /{HOME}/ linux/.config แล้วทำการสั่งคำสั่ง Make oldconfig ก็จะได้การคอนฟิกเคอร์เนลตามที่เราต้องการ เช่น

Cp /{HOME}/linux/configs/kernel-2.4.2-i686.config /{HOME}/linux/.config

{HOME} หมายถึง ไดเร็กทอรี่ของซอร์ดโค้ดที่ทำการรีคอมไพล์เคอร์เนล

make dep

คำสั่งนี้เป็นคำสั่งที่เริ่มคอมไพล์จริง

#make bzImage

คำสั่งนี้เป็นคำสั่งที่สร้างไฟล์ System.map , vmlinux และ bzImage สำหรับที่นำไปใส่ในตัว os loader เช่น lilo เพื่อที่สามารถเลือกเคอร์เนลในการโหลดตอนต้นได้

ไฟล์ bzImage อยู่ที่ /linux/arch/i386/boot

ไฟล์ System.map และ vmlinux อยู่ที่ /linux

#make modules

คำสั่งนี้เป็นคำสั่งในการคอมไพล์โมดูลต่าง ๆ ที่เลือกไว้

#make modules_install หรือ #sudo make modules_install

คำสั่งนี้เป็นคำสั่งในการติดตั้งโมดูลแต่ถ้าเพิ่ม sudo เข้าไปหมายถึงการติดตั้งโมดูลจะไม่ไปทับกับเคอร์เนลตัวเก่า

เมื่อทำการคอมไพล์ตามขั้นตอนต่าง ๆ ครบทั้ง 6 ขั้นตอนแล้วซึ่งจะได้ไฟล์ที่สำคัญ 3 ไฟล์ คือ System.map , vmlinux และ bzImage ซึ่งต้องก็อปปี้ไฟล์ทั้ง 3 ไฟล์นี้ไปเก็บไว้ที่ไดเรกทอรี/boot

cp คือ คำสั่งในการก๊อปปี้ไฟล์ คือ เช่น cp System.map /boot

เข้าไปแก้ไขคำสั่งต่าง ๆ ในไฟล์ชื่อ lilo.conf (อยู่ที่ /etc/lilo.conf) แล้วทำการพิมพ์คำสั่ง lilo อีกครั้งหนึ่งเพื่อเป็นการติดตั้ง lilo ตัวใหม่โดยมีบรรทัดที่ต้องแก้ไขดังนี้ คือ

1. เพิ่มบรรทัด Label = linux2 (เป็นชื่ออะไรก็ได้แล้วแต่ที่เราต้องการตั้ง) เพื่อใช้เป็นเมนูตอนที่เราบูตเครื่องขึ้นมาสำหรับเป็นเมนูเคอร์เนลตัวใหม่

2. เพิ่มบรรทัด boot = /boot/bzImage ซึ่งทำให้เคอร์เนลใช้ bzImage ตัวใหม่

* ตัวอย่างการเข้าไปแก้ไขไฟล์ lilo.conf โดยเพิ่ม 3 บรรทัดนี้เข้าไปต่อจากเมนูตัวเดิม

label = linux2.4.5

boot = /boot/bzImage

read_only

เมื่อบูตเครื่องขึ้นมาจะมีเมนูให้เลือก OS เพิ่มขึ้นอีกหนึ่งเมนู คือ linux2.4.5 เพื่อเข้ามาใช้เคอร์เนลตัวใหม่ สังเกตุได้ว่าขณะที่ทำการบูตก่อนที่เข้ามายังเมนู root นั้นเวอร์ชันของเคอร์เนลเปลี่ยนไปจากเดิมโดยเป็นตัวเลขของเคอร์เนลที่ทำการคอมไพล์

การรีคอมไพล์เคอร์เนลถ้าไม่เข้าใจวิธีการทำงานของแต่ละขั้นตอนอาจทำให้เคอร์เนลพังได้บ่อยครั้งต้องลงโปรแกรมใหม่เพราะการรีคอมไพล์เคอร์เนลที่ผิดพลาด

3.2 การใช้ GCC บนระบบปฏิบัติการลีนุกซ์

ในส่วนนี้กล่าวถึงการใช้คอมไพเลอร์ (Compiler) เพื่อที่นำมาคอมไพล์บนระบบปฏิบัติการลีนุกซ์ วิธีการในการคอมไพล์ รวมถึงโปรแกรมช่วยในการคอมไพล์และตรวจสอบ (debug) การทำงานของโปรแกรม

GCC ย่อมาจาก GNU C Compiler เป็นโปรแกรมที่สำคัญและจะขาดไม่ได้เลย หากต้องการพัฒนาโปรแกรมด้วยภาษาซีเพราะโปรแกรม GCC จะช่วยแปล (Compile) ภาษาซีให้เป็นภาษาเครื่อง (Machine Language) และทำการลิงค์เพื่อให้เกิดไฟล์ที่สามารถนำมาทำงานได้ (Executable File) ที่เรียกกันว่า ตัวแปลภาษาหรือคอมไพล์เลอร์

หน้าที่ของ GCC

GCC จะมีหน้าที่รับไฟล์ที่ส่งเข้ามาแล้วทำตาม 4 ขั้นตอนดังนี้

1. ปรีโปรเซสซิ่ง (Preprocessing)

เป็นกระบวนการที่รับภาษาซี (Source File) เข้ามา แล้วทำงานดังนี้

- นำเอาคำสั่งหมายเหตุ (Comment) ออกจากโปรแกรม
- ทำงานตามปรีโปรเซสเซอร์ไดเรกทีฟ (Preprocessor Directives) บรรทัดที่ขึ้นต้นด้วย # ซึ่งมีหลายคำสั่ง เช่น #include, #define

2. คอมไพล์เลชั่น (Compilation)

เป็นกระบวนการแปลภาษาซี (Source File) ที่ได้จากกระบวนการปรีโปรเซสซิ่งเป็นภาษาแอสเซมบลี (Assembly Code)

3. แอสเซมบลี (Assembly)

เป็นกระบวนการแปลภาษาแอสเซมบลีเป็นออบเจ็คโค้ดในระบบยูนิกซ์ซึ่งจะเป็นภาษาเครื่อง แต่ยังไม่สามารถนำมาทำงานได้ เพราะยังไม่ได้ทำการลิงค์เข้ากับไลบรารี

สำนักหอสมุดกลาง พระจอมเกล้าลาดกระบัง

4. ลิงค์กิ้ง (Linking)

เป็นกระบวนการที่นำเอาอ็อบเจ็คโค้ดมาลิงค์เข้ากับไลบรารี่จะทำให้เกิดไฟล์ที่สามารถทำงานได้ คือ เอ็กซิคิวเทเบิลไฟล์

GCC จะไม่แสดงผลใดๆ ออกทางหน้าจอหากการคอมไพล์ไม่มีข้อผิดพลาดใดๆ

รูปแบบการใช้งาน GCC

GCC มีรูปแบบการใช้งาน ดังนี้

gcc [option | filename] …

ซึ่งออพชั่นนั้นมีมากมายให้เลือก แต่จะขอกล่าวถึงออพชั่นที่ใช้กันบ่อยในหัวข้อถัดไป

ออพชั่นของ GCC ที่ควรทราบดังนี้

-c

เป็นการคอมไพล์ซอร์ดไฟล์แต่ไม่ทำการลิงค์จะทำให้เกิดออบเจ็คไฟล์ที่มีชื่อเหมือนกับซอร์ดไฟล์ที่ถูกคอมไพล์ เช่น หาก ซอร์ดไฟล์ชื่อ test.c ใช้คำสั่ง gcc -c test.c จะได้ไฟล์ ชื่อ test.o หากผลการคอมไพล์นั้นถูกต้อง

-o file

เป็นการกำหนดชื่อเอาท์พุทให้เป็นไปตามชื่อไฟล์ที่ระบุหลัง -o ในกรณีการคอมไพล์ภาษาซีโดยปกติเอาท์พุตไฟล์ที่ได้จะเป็น เอ็กซิคิวเทเบิลไฟล์ยกตัวอย่างเช่น การใช้คำสั่ง gcc -o test test.c คือ การสั่งแปลซอร์ดไฟล์ภาษาซีชื่อ test.c หากผลการคอมไพล์ถูกต้องจะได้เอ็กซิคิวเทเบิลไฟล์ชื่อ test

3.3 การดีบักโปรแกรมด้วย GDB

นอกจากการเรียกคอมไพล์โปรแกรมด้วย gcc แล้ว เรายังสามารถดีบัก (debug) โปรแกรมได้ด้วย โดยเรียกใช้โปรแกรม gdb มาช่วยในการดีบัก ซึ่ง gdb จะช่วยให้เราสามารถดีบักการทำงาน ดังแสดงต่อไปนี้

- ตรวจสอบค่าของตัวแปรทั้งหมดในโปรแกรม
- สามารถกำหนดจุด breakpoint เพื่อหยุดการทำงานในบรรทัดใด ๆ ตามต้องการได้
- สามารถตรวจสอบการทำงานไปทีละบรรทัดได้

นอกจากนี้ gdb ยังทำให้เราสามารถตรวจดูโครงสร้างภายในหรือหน่วยความจำที่ใช้ในขณะโปรแกรมทำงานได้ด้วย

gdb เป็น ดีบักโปรแกรม(โปรแกรมที่ใช้หาข้อผิดพลาดในซอร์ดโค้ดที่จะนำมาคอมไพล์เป็นโปรแกรม) โดยที่ผู้ใช้สามารถ ตั้ง breakpoint เป็นเลขบรรทัดใน source code และ ชื่อฟังก์ชันผู้ใช้สามารถดูข้อมูลภายในตัวแปรได้

เมื่อตัดสินใจใช้ gdb ในการหาข้อผิดพลาดในโปรแกรม สิ่งที่จำเป็นต้องรู้ คือ ชื่อของตัวแปร , หมายเลขบรรทัด และที่อยู่ของซอร์ดไฟล์

46180

จุดประสงค์ของการหาข้อผิดพลาดของ GDB คือ ทำให้ผู้ใช้สามารถเข้าไปข้างในโปรแกรมได้ขณะที่กำลังทำงานอยู่ หรือขณะที่โปรแกรมทำงานผิดพลาด

GDB สามารถทำเรื่องหลักๆ ได้ 4 เรื่อง เพื่อช่วยให้ผู้ใช้แก้ไขโปรแกรมดังต่อไปนี้

1. เริ่มการใช้งานโปรแกรมเจาะจงไปในที่ที่อาจจะเกิดข้อผิดพลาด
2. ทำให้โปรแกรมหยุดตามเงื่อนไขที่เจาะจงไว้
3. ตรวจสอบสิ่งที่เกิดขึ้นเมื่อโปรแกรมหยุด
4. เปลี่ยนบางอย่างในโปรแกรม ดังนั้นผู้ใช้ก็จะสามารถแก้ไขข้อผิดพลาดได้ทีละจุดไปเรื่อยๆ

ผู้ใช้สามารถใช้ GDB ในการหาข้อผิดพลาดของโปรแกรมที่เขียนขึ้นในภาษาซี

GDB ถูกอ้างอิงด้วยคำสั่งของ Shell คือ gdb เมื่อเริ่มมันจะอ่านคำสั่งจาก Terminal จนกระทั่งสั่งให้ออก ด้วยคำสั่ง Quit เราสามารถอ่านรายละเอียดของ gdb โดยใช้คำสั่ง help ภายใน ตัว gdb

เราสามารถเรียกใช้ gdb โดยไม่มี argument หรือ option ได้ แต่ทางที่ดีที่สุดในการเรียกใช้ควรจะตามด้วยชื่อที่สามารถเรียกใช้ทำงานได้ของโปรแกรม เช่น gdb program

ลักษณะของคำสั่งจะแบ่งเป็นกลุ่มๆ ดังนี้

aliases	ชื่ออื่นๆ ของคำสั่งอื่นๆ
breakpoint	ทำให้โปรแกรมหยุดที่จุดที่ต้องการ
data	ตรวจสอบข้อมูล
files	เจาะจงและตรวจสอบแฟ้มข้อมูล
internals	คำสั่ง maintenance
obscure	ลักษณะของ obscure
running	ทำให้โปรแกรมทำงาน
stack	ตรวจสอบ stack
status	ถามถึงสภาวะ
support	รองรับสิ่งอำนวยความสะดวก
tracepoints	trace โปรแกรมที่ทำงานอยู่โดยไม่หยุดที่ใดๆ
user-defined	คำสั่งที่ผู้ใช้ตั้งเอง

โดยที่รายละเอียดของคำสั่งในกลุ่มต่างๆ เหล่านี้ ผู้ใช้สามารถอ่านได้จากการ พิมพ์ help ที่หลัง prompt ของ gdb แล้วตามด้วยกลุ่มของคำสั่ง หรือ ตามด้วยชื่อคำสั่ง เช่น

(gdb) help break

จะปรากฏผลดังนี้

(gdb) help break

Set breakpoint at specified line or function.

Argument may be line number, function name, or "*" and an address.

If line number is specified, break at start of code for that line.

If function is specified, break at start of code for that function.

If an address is specified, break at that exact address.

With no arg, uses current execution address of selected stack frame.

This is useful for breaking on return to a stack frame.

Multiple breakpoints at one place are permitted, and useful if conditional.

Do "help breakpoints" for info on other commands dealing with breakpoints.

(gdb)

OPTION ต่างๆในการเรียกใช้มีดังนี้

-b BAUDRATE	ตั้ง Serial Port สำหรับการค้นหาข้อผิดพลาดปลายทาง
--batch	ออกหลังจากกระบวนการนั้นๆ
--cd = DIR	เปลี่ยนไดเร็กทอรีปัจจุบันเป็นไดเรกทอรีDIR
--command = FILE	ดำเนินการตามคำสั่ง gdb จากแฟ้มข้อมูลชื่อ file
--core = COREFILE	วิเคราะห์ core dump จากแฟ้มข้อมูลชื่อ COREFILE
--dbx	DBX Compatibility Mode
--directiry = DIR	ค้นหาแฟ้มต้นกำเนิดในไดเรกทอรีDIR
--epoch	แสดงข้อมูลโดย Epoch Emacs – GDB
--exce = EXECFILE	ใช้แฟ้มข้อมูลชื่อ EXECFILE ในการดำเนินการ
--fullname	แสดงข้อมูลโดย Emacs GDB
--help	แสดงข้อมูลนี้
--mapped	ใช้แฟ้มข้อมูล mapped symbol ถ้าใช้ในระบบนี้ได้
--nw	ห้ามใช้ Window interface
--nx	ห้ามอ่านแฟ้มข้อมูล .gdninit
--quiet	ไม่อนุญาตให้แสดงหมายเลขของรุ่นตอน Startup
--readnow	อ่านแฟ้มข้อมูลสัญลักษณ์ทั้งหมดในการเข้าถึงครั้งแรก
--se = FILE	ใช้แฟ้มข้อมูลชื่อ FILE เป็นแฟ้มข้อมูลสัญลักษณ์ที่ทำงานได้
--symbols = SYMFILE	อ่านสัญลักษณ์จากแฟ้มข้อมูลชื่อ SYMFILE
--tty = TTY	ใช้ TTY สำหรับรับและแสดงข้อมูล เมื่อโปรแกรมกำลังหาข้อผิดพลาดอยู่
--version	แสดงข้อมูลของรุ่น แล้วออกจากการทำงาน
--w	ใช้ Window Interface

--write ตั้งการเขียนเป็นแบบที่สามารถทำงานได้ และเป็นแฟ้มข้อมูลแบบ Core

--xdb XDB compatibility Mode

ดังที่กล่าวมาแล้วข้างต้นว่า option ต่างๆนี้อาจจะไม่จำเป็น สำหรับการใช้งานเท่าใดนัก

การใช้ GDB ดีบักเคอร์เนล

<u>การคอมไพล์เคอร์เนลเมื่อต้องการทำการดีบักเคอร์เนล</u>

ก่อนที่เราทำการใช้ GDB ดีบักเคอร์เนลเราต้องทำการรีคอมไพล์เคอร์เนลเพื่อนำเอาไฟล์ Vmlinux มาทำการดีบักซึ่งในไฟล์ Vmlinux ประกอบไปด้วยไฟล์ต่างที่ถูกคำสั่งใน Makefile ทำการคอมไพล์ และออพชั่นในการคอมไพล์นั้นต้องเป็นออพชั่นที่อนุญาติการดีบักได้ด้วยซึ่งเราทำการเปลี่ยนแฟล็กของ CFLAGS จาก

CFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -fno-strength-reduce

(ข้างบนเป็นการกำหนดใน Make file ดั้งเดิม)

เราต้องทำการเปลี่ยนเป็น

CFLAGS = -g -Wall -Wstrict-prototypes -O2 -fno-strength-reduce

เมื่อทำการเปลี่ยนแฟล็กแล้วถัดนั้นมาเราก็คอมไพล์เคอร์เนลตามปกติซึ่งเมื่อเราทำขั้นตอน Make bzImage เราก็ได้ไฟล์ Vmlinux ที่สามารถนำมาทำการดีบักได้ ซึ่งไฟล์นี้สร้างขึ้นมาที่ไดเรกทอรี่ /{HOME}/linux ซึ่งการดีบักนั้นจำเป็นต้องมีหน่วยความจำเพียงพอสำหรับการดีบัก

{HOME} หมายถึง ไดเรกทอรี่ของซอร์ดโค้ดที่ทำการรีคอมไพล์เคอร์เนล

<u>เริ่มต้นคำสั่งการดีบักเคอร์เนล</u>

เมื่อเราทำการบูตเคอร์เนลด้วย bzImage (คือ ไฟล์บูตของ Vmlinux ซึ่งได้มาพร้อมกับ Vmlinux ตอนสั่งคำสั่ง Make bzImage) ตัวใหม่แล้ว GDB สามารถดูค่าของโครงสร้างข้อมูลตัวแปรโกลบอลได้ โดยทำการสั่งคำสั่งต่อไปนี้

GDB /linux/vmlinux /proc/kcore

เนื่องจากเคอร์เนลไม่ได้เป็นโปรเซสของส่วนของยูสเซอร์โหมดเราจึงไม่สามารถดูตัวแปรและทำการแก้ไขในส่วนต่าง ๆ ต่อไปนี้ได้

- แก้ไขโครงสร้างข้อมูลบางค่า (Modify data structures)
- เรียกฟังก์ชันบางอย่างของเคอร์เนล (Call kernel functions)
- ทดสอบค่าของโครงสร้างข้อมูลแบบโลคอล (Examine local data structures)
- ถึงแม้ว่ามีการจำกัดการเข้าถึงบางอย่างแต่เราก็ยังสามารถดูเหตุการณ์ต่าง ๆ ได้อีกมากมาย

3.4 การทดลองที่ 1 การโมดิฟายซอร์ดโค้ด , การรีคอมไพล์เคอร์เนล , การดีบักเคอร์เนล (Modify Source Code , Recompile Kernel , Debug Kernel)

วัตถุประสงค์

1. เพื่อศึกษาถึงการทำงานของซอร์ดโค้ดต่าง ๆ ของระบบปฏิบัติการลีนุกซ์และเข้าใจถึงหน้าที่การทำงานของไฟล์แต่ละไฟล์
2. เพื่อศึกษาถึงโมดูลต่าง ๆ ที่สำคัญของระบบปฏิบัติการลีนุกซ์
3. เพื่อศึกษาและสามารถทำการรีคอมไพล์เคอร์เนลได้
4. เพื่อศึกษาการดีบักโปรแกรมด้วย GDB

ทฤษฎี คำสั่งในการรีคอมไพล์เคอร์เนล

1.make mrproper

2..make menuconfig

3.make bzImage

4.make module

5.make module_install

การคอมไพล์ต้องคอมไพล์ในไดเรกทอรีที่มี Makefile เช่น ในไดเรกทอรี/{HOME}/ linux สำหรับการคอมไพล์นั้นมีหลายคำสั่งที่นิยมใช้กันนอกเหนือจากคำสั่งข้างต้นเพิ่มเข้าไป เช่น make clean , make oldconfig , make xconfig , make zImage , sudo make install , sudo make modules_install ขึ้นอยู่กับลักษณะการใช้งาน

การดีบักโปรมแกรมด้วย GDB มี 2 ลักษณะ คือ การดีบักไฟล์ที่ไม่ได้เกี่ยวข้องกับเคอร์เนลซึ่งเป็นไฟล์ที่สร้างขึ้นมาใหม่กับการดีบักไฟล์ Vmlinux ซึ่งเป็นไฟล์ที่เกี่ยวกับเคอร์เนลดังนั้นก่อนที่ทำการดีบักต้องทำการคอมไพล์ให้แตกต่างจากเดิม

ปฏิบัติ

1. ให้นักศึกษาทำการดาวน์โหลดเคอร์เนลเวอร์ชันล่าสุดจาก www.kernel.org เพื่อทำการรีคอมไพล์เคอร์เนลตัวใหม่
2. ให้นักศึกษาทดลองใช้งานเคอร์เนลเวอร์ชันใหม่หลังจากการรีคอมไพล์แล้วศึกษาถึงความแตกต่างระหว่างเวอร์ชันใหม่กับเวอร์ชันเก่า

คำถามท้ายการทดลอง

1. เพราะเหตุใดจึงต้องทำการรีคอมไพล์เคอร์เนลเวอร์ชันใหม่อยู่เสมอ
2. ให้นักศึกษาอธิบายการทำงานของไฟล์หลัก ๆ 3 ไฟล์ คือ ไฟล์ bzImage , vmlinuz , System.map ว่าแต่ละไฟล์ทำหน้าที่ต่างกันอย่างไร
3. คำสั่งคอนฟิกแต่ละคำสั่งทำงานต่างกันอย่างไร
4. ทดลองดีบักโปรแกรมธรรมดาจากไฟล์ปกติกับดีบักเคอร์เนลและตอบให้ได้ว่าการดีบักทั้งสองแบบต่างกันอย่างไร
5. ก่อนทำการดีบักเคอร์เนลต้องคอมไพล์เคอร์เนลต่างกับการคอมไพล์เคอร์เนลแบบปกติอย่างไร

การดีบักโปรแกรม

ในขั้นแรกนั้นต้องทำการดีบักโปรแกรมง่ายๆ ก่อน เพื่อทำให้เกิดความเข้าใจ ยกตัวอย่างโปรแกรมอย่างง่ายๆ
ในการดีบักโปรแกรม

```
  UW PICO(tm) 4.0                File: a.c

#include <stdio.h>
main()
{
  int x = 1;
  for (x==1;x<=10;x++)
    printf("%d \n",x);
}

                        [ Read 7 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

รูปที่ 3-2 แสดงโปรแกรมที่ใช้ในการดีบัก

ในการคอมไพล์เพื่อให้สามารถดีบักโปรแกรมได้นั้นจะต้องเพิ่มออพชัน –g เข้าไปด้วย ดังรูป เพื่อให้สามารถดีบักโปรแกรมได้

```
download@venus142:~/Adul$ gcc -g -o a a.c
download@venus142:~/Adul$
```

รูปที่ 3-3 แสดงการคอมไพล์โปรแกรมด้วย gcc โดยเพิ่มออพชัน –g เข้าไปด้วย

การดีบักโปรแกรมโดยใช้ gbd แสดงดังรูป

```
download@venus142:~/Adul$ gdb a
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb)
```

รูปที่ 3-4 แสดงการเรียกใช้โปรแกรม gdb

การเบรกโปรแกรมโดยใช้ "b" ตามด้วยฟังก์ชัน หรือบรรทัด

```
(gdb) b main
Breakpoint 1 at 0x80483f6: file a.c, line 4.
(gdb) b 7
Breakpoint 2 at 0x8048421: file a.c, line 7.
(gdb)
```

รูปที่ 3-5 แสดงการใช้คำสั่งเบรกใน gdb

การรันโปรแกรมโดยใช้ "r" การรันโปรแกรมจะไปหยุดเมื่อมีเบรก ในกรณีมีการสั่งให้เบรกในฟังก์ชัน main

```
(gdb) r
Starting program: /home/download/Adul/a

Breakpoint 1, main () at a.c:4
4         int x = 1;
(gdb)
```

รูปที่ 3-6 แสดงการรันโปรแกรมภายใต้ gdb

การใช้คำสั่ง disp จะเป็นการเพิ่มตัวแปรเข้าไปใน ลิสต์ของตัวแปร จะถูกแสดงค่าทุกครั้งเมื่อมีการทำงาน

```
(gdb) disp x
1: x = -1073743416
(gdb)
```

รูปที่ 3-7 แสดงการใช้ disp แสดงค่าตัวแปรที่ต้องการ

คำสั่ง "s" เป็นการรันคำสั่งทีละบรรทัดโดยเข้าไปในฟังก์ชันย่อยด้วย

```
(gdb) s
5         for (x==1;x<=10;x++)
1: x = 1
(gdb) s
6           printf("%d \n",x);
1: x = 1
(gdb)
```

รูปที่ 3-8 แสดงการทำงานทีละบรรทัดโดยเข้าไปในฟังก์ชันย่อยด้วย

คำสั่ง "n" เป็นการรันคำสั่งทีละบรรทัดเหมือนกันแต่ไม่เข้าไปในฟังก์ชันย่อย

```
(gdb) n
1
5         for (x==1;x<=10;x++)
1: x = 1
(gdb)
```

รูปที่ 3-9 แสดงการทำงานทีละบรรทัดแต่ไม่เข้าไปในฟังก์ชันย่อย

การดีบักเคอร์เนล

สำหรับการดีบักเคอร์เนลนั้น โปรแกรมต่างๆ ที่ต้องการดีบักจะรันอยู่บนยูสเซอร์โหมด แต่ที่ผ่านมาเราคอมไพลเคอร์เนลเพื่อให้ทำงานอยู่ในบนเคอร์เนลโหมด เราไม่สามารถที่จะไปดีบักตัวเคอร์เนลที่รันอยู่ได้ เราจึงต้องทำการคอมไพลเคอร์เนลให้รันอยู่บนยูสเซอร์โหมดเสมือนเป็นโปรแกรมหนึ่งตัวโปรแกรมโดยทั่วไปที่ต้อง การดีบักจะรันอยู่ภายใต้ gdb ก็จะเหมือนกับเคอร์เนลที่เราจะต้องรันมันให้อยู่ภายใต้ gdb เพื่อที่จะทำให้เราสามารถดูค่าต่างๆ ได้ ตามการใช้งานของ gdb

ในการที่ดีบักตัวเคอร์เนลได้นั้นก็จะมีลักษณะเช่นเดียวกับการดีบักโปรแกรม คือ ในขั้นตอนการคอมไพล์นั้นจะต้องมีการเพิ่ม – g เข้าไปด้วย แต่สำหรับโปรแกรมโดยทั่วไปนั้นจะมีไฟล์แค่ไฟล์เดียวตอนคอมไพล์จึงทำให้ไม่ยุ่งยาก แต่สำหรับตัวเคอร์เนลนั้นมีไฟล์ต่างๆ หลายไฟล์จึงต้องมีการเขียน Makefile เพื่อรวมเอา .o ต่างๆ มารวมกัน ดังนั้นจึงต้องมีการแก้ไข Makefile โดยการเพิ่ม –g เข้าไปเพื่อที่จะทำให้สามารถดีบักได้

แต่นอกวิธีนี้แล้วก็ยังมีอีกวิธีซึ่งมีลักษณะใกล้เคียงกัน ซึ่งจะเป็น Patch สำหรับUML (User Mode Linux) โดยเราต้องเพิ่ม Patch นี้เข้าไปในตัวเคอร์เนลปกติ ซึ่งตัวเคอร์เนลปกตินั้นหาได้ที่ www.kernel.org จากนั้นนำมาแตกไฟล์ ซึ่งได้กล่าวมาแล้วข้างต้น หลังจากนั้นให้ทำการดาวน์โหลด Patch UML ให้ตรงกับเคอร์เนลเวอร์ชันที่ดาวน์โหลมหาได้ที่ http://prdownloads.sourceforge.net/user-mode-linux/ จากนั้นทำการติดตั้ง Patch ดังกล่าวโดยใช้คำสั่ง

#cd /uml/linux

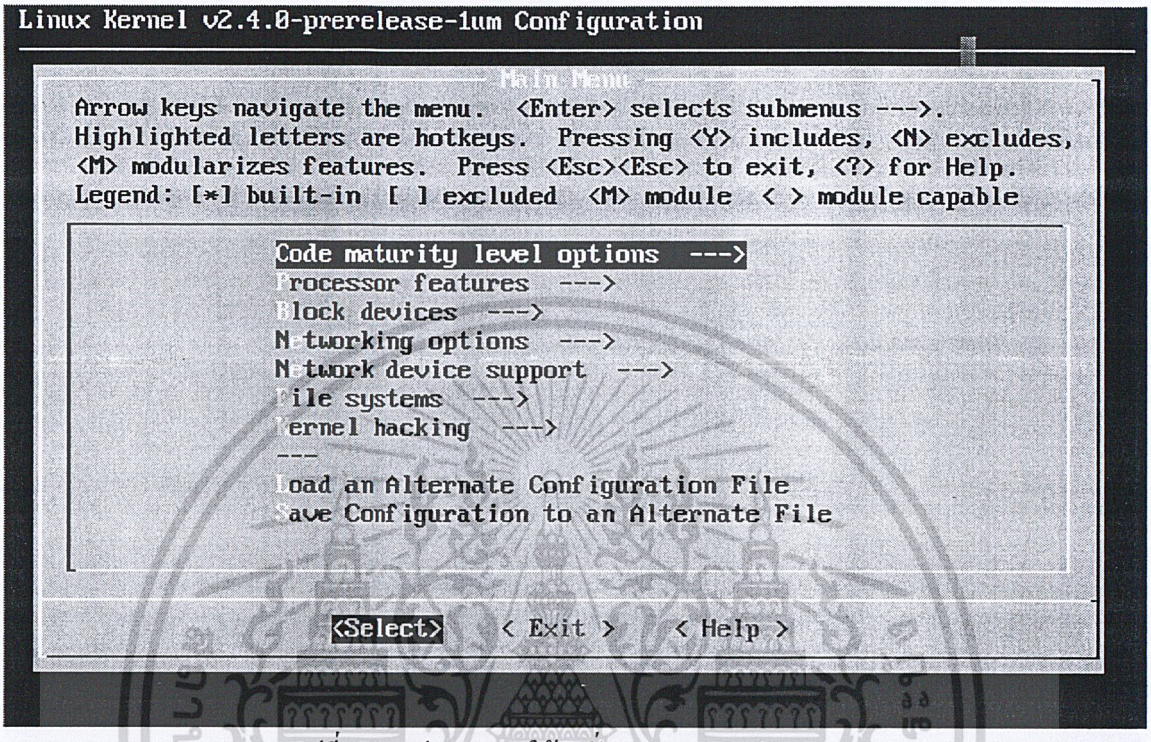
#cat uml-patch-x.x.xx bz2 | bunzip2 - | patch –p1

หลังจากนั้นทำทำการ make config โดยใช้คำสั่ง

make config ARCH=um หรือ

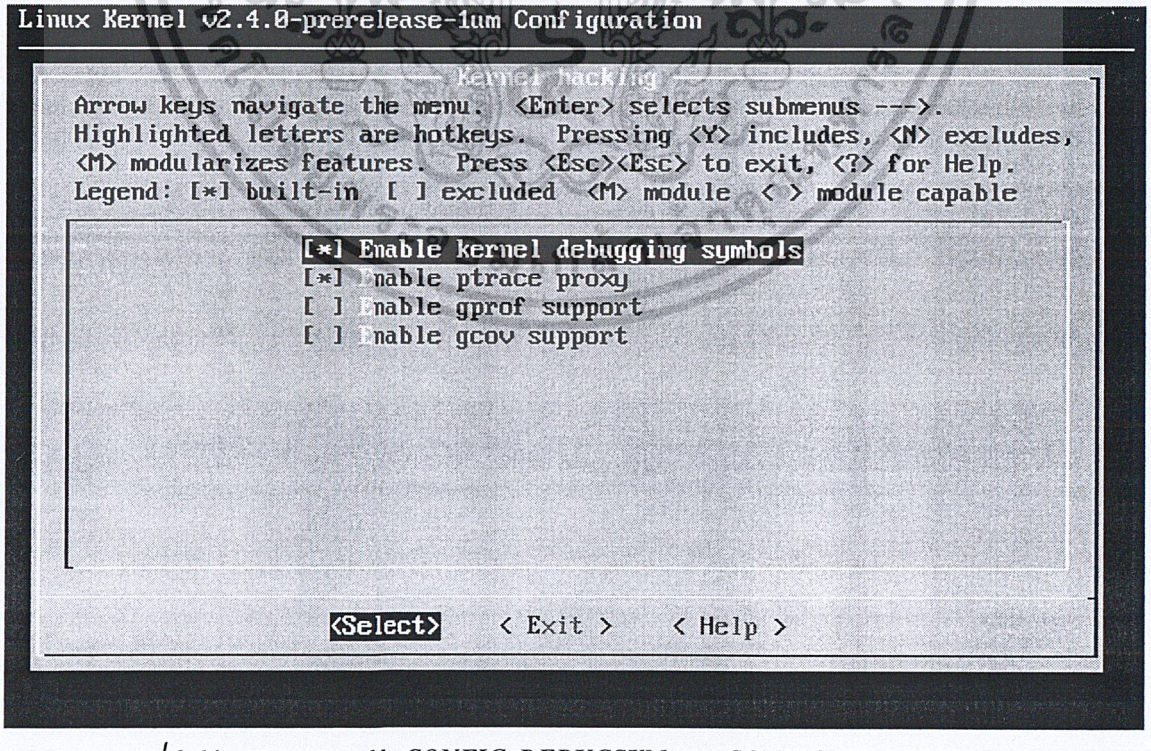
make xconfig ARCH=um หรือ

make menuconfig ARCH=um แสดงดังรูป 3-10



รูปที่ 3-10 รูปแสดงการใช้คำสั่ง make menuconfig

ในการคอนฟิกนั้นต้อง enable CONFIG_DEBUGSYM และ CONFIG_PT_PROXY ด้วย



รูป 3-11 แสดงการenable CONFIG_DEBUGSYM และ CONFIG_PT_PROXY

ขั้นตอนสุดท้ายใช้คำสั่ง

make linux ARCH=um

ซึ่งผลลัพธ์ที่ออกมาจะได้ไฟล์ Linux อยู่ในไดเรกทอรีบนสุด เพื่อนำไปใช้ในการดีบัก

```
[root@localhost linux]# ls
arch            fs        lib          net              System.map
COPYING         include   linux        README           uml-patch-2.4.2.bz2
CREDITS         init      MAINTAINERS  REPORTING-BUGS   vmlinux.o
Documentation   ipc       Makefile     Rules.make
drivers         kernel    mm           scripts
[root@localhost linux]#
```

รูปที่ 3-12 แสดงไฟล์ linux ที่ได้หลังจากการคอมไพล์

ทำการดีบักลีนุกซ์ภายใต้ gdb

```
[root@localhost linux]# gdb linux
GNU gdb 5.0rh-5 Red Hat Linux 7.1
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

รูปที่ 3-13 แสดงการดีบักลีนุกซ์

ใช้ออปชัน "b" ใส่เบรกพอยนต์ไปที่ฟังก์ชัน main ของเคอร์เนล

```
(gdb) b main
Breakpoint 1 at 0x10008109: file /home/adul/linux/arch/um/main.c, line 32.
(gdb)
```

รูปที่ 3-14 แสดงการใช้เบรกพอยนต์ในเคอร์เนล

ใช้ออปชัน "r" รันลีนุกซ์ จะหยุดเมื่อถึงเบรกพอยนต์

```
(gdb) r
Starting program: /home/adul/linux/linux

Breakpoint 1, main (argc=1, argv=0xbffffbc4, envp=0xbffffbcc)
    at /home/adul/linux/arch/um/main.c:32
32              for(i=1;i<argc;i++) n += strlen(argv[i]);
(gdb)
```

รูปแสดง 3-15 แสดงการรันไฟล์ที่ทำการดีบัก

ใช้ออปชัน “s” เป็นการรันคำสั่งทีละบรรทัดโดยเข้าไปในฟังก์ชันย่อยด้วย การใช้คำสั่ง disp จะเป็นการเพิ่มตัวแปรเข้าไปใน ลิสต์ของตัวแปร จะถูกแสดงค่าทุกครั้งเมื่อมีการทำงาน

```
(gdb) s
25      {
(gdb) s
32              for(i=1;i<argc;i++) n += strlen(argv[i]);
(gdb) s
31              n = 0;
(gdb) s
32              for(i=1;i<argc;i++) n += strlen(argv[i]);
(gdb) s
33              if(n < 128){
(gdb) disp argc
1: argc = 1
(gdb) disp n
2: n = 0
(gdb)
```

รูปที่ 3-16 แสดงการรันทีละสเต็ป

แสดงการใช้ออปชัน info break เพื่อแสดงตำแหน่งที่ใส่เบรกพอยนต์

```
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x10008109 in main
                                       at /home/adul/linux/arch/um/main.c:32
        breakpoint already hit 2 times
2   breakpoint     keep y   0x10008642 in schedule at sched.c:508
3   breakpoint     keep y   0x10009f35 in do_fork at fork.c:562
(gdb)
```

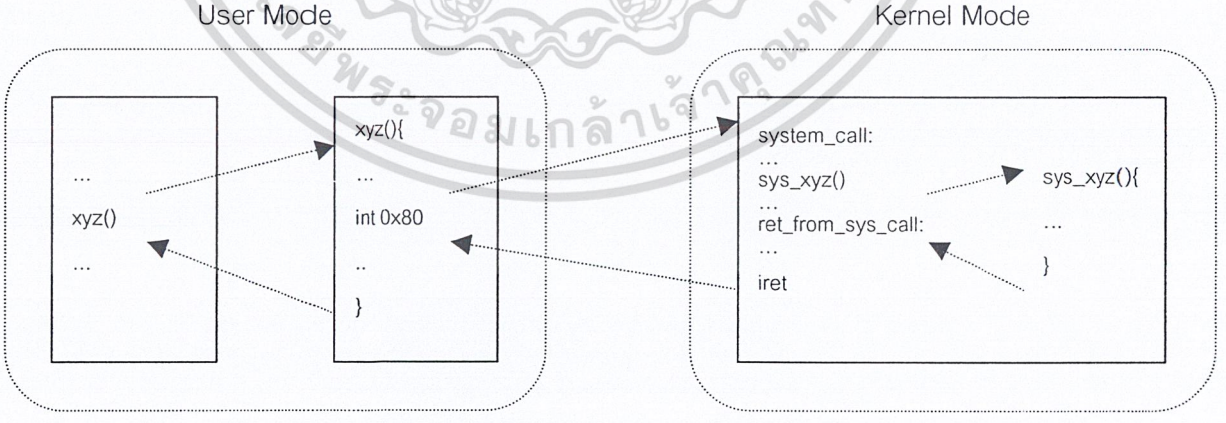
รูปที่ 3-17 แสดงตำแหน่งเบรกพอยนต์

บทที่ 4

ซิสเต็มคอล (System call)

4.1 การทำงานของซิสเต็มคอล

ระบบปฏิบัติการลีนุกซ์ได้ออกแบบให้โปรเซสปกติทำงานในส่วนของยูสเซอร์โหมดเมื่อต้องการติดต่อกับส่วนต่าง อาทิ ส่วนของการติดต่อกับฮาร์ดแวร์ต่าง ๆ เช่น ซีพียู , ดิสก์ , พรินเตอร์ และอื่น ๆ ซึ่งการติดต่อกับส่วนต่าง ๆ บางส่วนไม่สามารถติดต่อได้โดยตรงต้องทำโดยผ่านเคอร์เนลโดยระบบปฏิบัติการได้ใส่ส่วนของเลเยอร์พิเศษไว้ระหว่างแอพพลิเคชั่นกับฮาร์ดแวร์เพื่อใช้ติดต่อกับฟังก์ชันของเคอร์เนลโหมดซึ่งมีมากมายซึ่งเป็นฟังก์ชันที่ส่วนของยูสเซอร์โหมดไม่สามารถใช้ฟังก์ชันบางฟังก์ชันจากส่วนของเคอร์เนลโหมดได้ ซึ่งเป็นการออกแบบให้เคอร์เนลมีความปลอดภัยเพราะการทำงานในส่วนของยูสเซอร์โหมดไม่ได้ติดต่อและเรียกใช้ฟังก์ชันต่าง ๆ เหล่านั้นเข้ามาโดยตรง ซึ่งถ้าโปรแกรมเมอร์ต้องการเรียกใช้ฟังก์ชันในส่วนของเคอร์เนลโหมดก็ต้องเขียนโปรแกรมขึ้นมาเรียกใช้โดยผ่านทางซิสเต็มคอลแล้วทำการรีคอมไพล์เคอร์เนลใหม่ก็จะสามารถเรียกใช้งานฟังก์ชันนั้นได้ ซึ่งการทำงานนั้นส่วนของเคอร์เนลจะทำการตรวจสอบก่อนว่าฟังก์ชันที่เรียกใช้งานนั้นถูกเรียกโดยวิธีการที่ถูกต้องหรือไม่ ซึ่งหลังจากการรีคอมไพล์เคอร์เนลและเริ่มรันเคอร์เนลใหม่ก็จะสามารถใช้งานฟังก์ชันของเคอร์เนลนั้นได้ ซึ่งในระบบปฏิบัติการลีนุกซ์นั้นเรียกสัญญาณที่ใช้ติดต่อระหว่างโปรเซสในส่วนของยูสเซอร์โหมดกับอุปกรณ์ฮาร์ดแวร์หรือฟังก์ชันต่าง ๆ ในส่วนของเคอร์เนลโหมดนั้นว่าซิสเต็มคอล (System Call) แสดงภาพการติดต่อระหว่างยูสเซอร์โหมดและเคอร์เนลโหมด



รูปที่ 4-1 แสดงการทำงานของซิสเต็มคอลที่ User Mode ติดต่อไปยัง Kernel Mode

การทดลองที่ 2 ซิสเต็มคอล (System Call)

วัตถุประสงค์

1. เพื่อศึกษาการทำงานของซิสเต็มคอล (System Call)
2. แก้ไขเคอร์เนล (Kernel) เพื่อสร้างซิสเต็มคอลขึ้นมาใหม่ได้

ทฤษฎี

4.2 การติดต่อของซิสเต็มคอล (System call linkage)

โปรแกรมใน User Space จะทำการเรียก System call Stub ที่ซึ่งบรรจุคำสั่งที่ใช้ทำงาน สำหรับผลลัพท์ของมัน CPU จะสลับการทำงานให้อยู่ใน Mode ของ Supervisor Mode และเริ่มทำการ execute ในตำแหน่งที่อยู่ใน Kernel Space

ใน Kernel Version 2.4.2 จะบรรจุ System Call table ซึ่งจะอยู่ใน File arch/i386/kernel/entry.S แสดง Code File ดังกล่าว ข้างล่าง

```
.data
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall)      /* 0  -  old "setup()" system call*/
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
        .long SYMBOL_NAME(sys_read)
        .long SYMBOL_NAME(sys_write)
        .long SYMBOL_NAME(sys_open)            /* 5 */
        ......
        .long SYMBOL_NAME(sys_getdents64)      /* 220 */
        .long SYMBOL_NAME(sys_fcntl64)
        .long SYMBOL_NAME(sys_ni_syscall)      /* reserved for TUX */

.rept NR_syscalls-221
                .long SYMBOL_NAME(sys_ni_syscall)
        .endr
```

Entry 1 จะบรรจุตำแหน่งที่ตั้งของ exit() System Call (ใน Kernel ฟังก์ชั่นที่ชื่อ Sys_exit)

4.3 การกำหนดเลขของซิสเต็มคอล (Defining the System Call Number)

เราสามารถที่จะกำหนดซิสเต็มคอลเพิ่มเข้าไปได้ใน sys_call_table โดยเข้าไปแก้ไขตาราง ในไฟล์ที่ชื่อว่า arch/i386/kernel/entry.S ดังกล่าวข้างล่าง

```
.data
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall)       /* 0  -  old "setup()" system call*/
        .long SYMBOL_NAME(sys_exit)
        …
        .long SYMBOL_NAME(sys_getdents64)       /* 220 */
        .long SYMBOL_NAME(sys_fcntl64)
        .long SYMBOL_NAME(sys_ni_syscall)       /* reserved for TUX */
        .long SYMBOL_NAME(sys_my_new_call)      /* 223 */
        .endr.
```

4.4 การสร้างซิสเต็มคอลสตับ (Generating a System Call Stub)

System call stub นั้นจะสร้างโดยใช้ marcro เรียกจากโปรแกรมใน User Space macro สำหรับที่จะสร้าง stub ขึ้นมานั้นมีตัวแปรได้ 0 – 5 ตัวแปร เราสามารถสร้าง stub โดยการเรียก macro ข้างล่างนี้ ใน User โปรแกรม

```
#include <linux/unistd.h>
…
        _syscall2(int , foo, char *, baz, double, bar);
…
```

และในบางครั้งเราสามารถเรียก function kernel สมมุติถ้า index ใน sys_call_table คือ 223 จะได้เรียกได้ดังนี้

```
#Include <sys/syscall.h>
…
        syscall(223,&baz_arg,bar_arg);
…
```

4.5 เคอร์เนลฟังก์ชั่นออเกไนซ์เซชัน (Kernel Function Organization)

ฟังก์ชั่นในเคอร์เนล(Kernel Function) คือ ส่วนที่ทำงานในโหมดของซุปเปอร์ไวเซอร์ (Supervisor mode) ในการทำงานนั้น เราจะลองมาพิจารณาฟังก์ชันอย่างง่ายๆ ดูก่อน เป็นฟังก์ชันที่ไม่มีการส่งค่าเข้าไปและคืนค่ากลับมา

```
asmlinkage void sys_foo(void){
        /*Write a value to the console*/
}
```

ถัดมาเป็นฟังก์ชันที่ไม่มีการส่งค่าเข้าไป แต่มีการคืนค่ากลับมา ยกตัวอย่างเช่น

```
asmlinkage int sys_close(void) {
        Return current->uid;
}
```

ปฏิบัติ

1. ให้เพิ่มซิสเต็มคอลเข้าไปในเคอร์เนล โดยออกแบบฟังก์ชันเคอร์เนลง่ายๆ ในการทดสอบ
2. เขียนโปรแกรมอย่างง่ายๆ เพื่อใช้ในการทดสอบซิสเต็มคอล โดยเรียกใช้ฟังก์ในเคอร์เนลที่เพิ่มเข้าไป เพื่อให้รู้ว่าสามารถไปทำในโค้ดของเคอร์เนล

ผลการทดลอง

ทำการเพิ่มซิสเต็มคอลที่ 191 และ 192 เพิ่มเข้าไปที่ไฟ /arch/i386/kernel/entry.S และ /include/asm/unistd.h โดยซิสเต็มคอลที่ 191 จะไม่มีการส่งค่าเข้าไปและคืนค่ากลับมา ส่วน 192 จะมีการส่งค่าเข้าไปแต่จะไม่มีการคืนค่ากลับมา

```
 UW PICO(tm) 3.7                File: entry.S

        .long SYMBOL_NAME(sys_getcwd)
        .long SYMBOL_NAME(sys_capget)
        .long SYMBOL_NAME(sys_capset)           /* 185 */
        .long SYMBOL_NAME(sys_sigaltstack)
        .long SYMBOL_NAME(sys_sendfile)
        .long SYMBOL_NAME(sys_ni_syscall)               /* streams1 */
        .long SYMBOL_NAME(sys_ni_syscall)               /* streams2 */
        .long SYMBOL_NAME(sys_vfork)            /* 190 */
        .long SYMBOL_NAME(sys_newcall)          /* 191 */
        .long SYMBOL_NAME(sys_newcallreceive)   /* 192 */

       /*
        * NOTE!! This doesn't have to be exact - we just have
        * to make sure we have _enough_ of the "sys_ni_syscall"
        * entries. Don't panic if you notice that this hasn't
        * been shrunk every time we add a new system call.
        */
       .rept NR_syscalls-190
               .long SYMBOL_NAME(sys_ni_syscall)
       .endr

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

รูปที่ 4-2 แสดงการแก้ไขไฟล์ entry.S

```
 UW PICO(tm) 3.7                File: unistd.h

#define __NR_capget             184
#define __NR_capset             185
#define __NR_sigaltstack        186
#define __NR_sendfile           187
#define __NR_getpmsg            188     /* some people actually want streams */
#define __NR_putpmsg            189     /* some people actually want streams */
#define __NR_vfork              190
#define __NR_newcall            191     /* Add new */
#define __NR_newcallreceive     192     /* Add new */

/* user-visible error numbers are in the range -1 - -122: see <asm-i386/errno.h$

#define __syscall_return(type, res) \
do { \
        if ((unsigned long)(res) >= (unsigned long)(-125)) { \
                errno = -(res); \
                res = -1; \
        } \
        return (type) (res); \
} while (0)

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

รูปที่ 4-3 แสดงการแก้ไขไฟล์ unistd.h

จากนั้นเข้าไปเพิ่มฟังก์ชันในเคอร์เนลในที่นี้จะเข้าไปเพิ่มที่ไฟล์ time.c ดังรูป

```
  UW PICO(tm) 3.7                File: time.c                    Modified

void get_fast_time(struct timeval * t)
{
        do_get_fast_time(t);
}

asmlinkage void sys_newcall(void) /* Addnew */
{
        printk("<0>Linux Kernel Man! \n");
}

asmlinkage void sys_newcallreceive(char * fd) /* Addnew */
{
        printk("<0>Linux Internal Group : %s \n",fd);
}_

/* The xtime_lock is not only serializing the xtime read/writes but it's also
   serializing all accesses to the global NTP variables.
   NOTE NOTE: We really need a spinlock here as the global irq locking
   only protect us against the timer irq and not against other time-related
   syscall running under us. */

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

รูปที่ 4-4 แสดงการแก้ไขไฟล์ time.c

เมื่อแก้ทั้งสามไฟล์เสร็จแล้วก็จะต้องทำการคอมไพล์เคอร์เนลใหม่อีกครั้ง จากนั้นเขียนโปรแกรมเพื่อเรียกฟังก์การทำงานที่อยู่ในเคอร์เนล

โปรแกรมที่ใช้ทดสอบ เรียกซิสเต็มคอล ที่รันอยู่ในยูสเซอร์โหมดจะมีอยู่ 2 กรณีคือในกรณีที่เรียกโดยตรง

```
  UW PICO(tm) 3.7                File: a.c                       Modified

#include <sys/syscall.h>

main()
{
  syscall(191);
  syscall(192,"Hello");
}_

                        [ Read 8 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

รูปที่ 4-5 แสดงการเขียนโปรแกรมเรียกใช้ฟังก์ชันในเคอร์เนลโดยเรียกโดยตรง

ผลจากการรันโปรแกรม

```
root@Olala07V:~# gcc -o syscalldirect a.c
root@Olala07V:~# ./syscalldirect
Linux Kernel Man!
Linux Internal Group : Hello
root@Olala07V:~# _
```

รูปที่ 4-6 แสดงการทดสอบการรันโปรแกรม

หรือเรียกโดยการสร้างสคับเขียนโปรแกรมดังรูป

```
  UW PICO(tm) 3.7                File: a.c

#include <linux/unistd.h>

_syscall0(int,newcall);
_syscall1(int,newcallreceive,char *,fd);

main()
{
  newcall();
  newcallreceive("Hello");
}

                    [ Read 10 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

รูปที่ 4-7 แสดงการเขียนโปรแกรมเรียกใช้ฟังก์ชันในเคอร์เนลโดยเรียกโดยผ่านสคับ

ผลจากการรันโปรแกรม

```
root@Olala07V:~# gcc -o syscallstub a.c
root@Olala07V:~# ./syscallstub
Linux Kernel Man!
Linux Internal Group : Hello
root@Olala07V:~# _
```

รูปที่ 4-8 แสดงการทดสอบการรันโปรแกรม

บทที่ 5
โปรเซสและตารางการจัดการโปรเซส

5.1 ความมายของโปรเซส (PROCESSES)

หลักการของ process คือ เพื่อการทำงานบน operating system ในลักษณะที่เป็นการทำงานแบบหลายโปรแกรมในขณะเวลาเดียวกัน ประโยชน์ของมันคือ การรันโปรแกรม execution เช่น ในระบบ Linux เมื่อผู้ใช้ทำการรัน vi พร้อมกัน 16 ผู้ใช้ การทำงานของ process นั้นแยกจากกันโดยอิสระ 16 process (ถึงแม้ว่ามีการ share ทรัพยากรต่าง ๆ ร่วมกันก็ตาม) ใน Linux นั้นเราเรียกโปรเซสต่าง ๆ ที่ทำการรันว่า "task" ซึ่งสามารถทำการสร้างและทำลายมันได้ ในลีนุกซ์นั้นสามารถทำงานแบบ multithread ได้และมีการทำงานในลักษณะ LWP (lightweight processes) คือ การที่มีหลายโปรเซสทำงานบนพื้นที่ address space เดียวกันในลักษณะโปรเซสแบบ parent และ child

การที่ทำการอธิบายได้ว่าโปรเซสนั้นกำลังทำงานอยู่บนซีพียูหรือว่าโปรเซสถูกห้ามทำการบางอย่างหรือไม่นั้นถูกกำหนดโดยการอนุญาติการทำงานบนพื้นที่บน address space โดยที่ adress space นั้นทำการกำหนดพื้นที่ให้กับโปรเซสต่าง ๆ ซึ่งเรียกว่า process descriptor (ประกอบด้วยส่วนต่าง ๆ อาทิ เช่น งานที่ทำการรันก่อนหน้านี้ งานที่ต้องทำการรันต่อไป) โดยที่ process descriptor นั้นทำหน้าที่ในการอธิบายการทำงานของ process ต่าง ๆ

โดยเคอร์เนลควบคุมโปรเซสโดยสามารถสร้าง ลบ หรือทำลายโปรเซสออกจากระบบปฏิบัติการได้งานทุกงานในลีนุกซ์ถูกเอ็กซิคิวท์โดยโปรเซส โดยโปรเซสก็คือโปรแกรมที่คอมพิวเตอร์ทำการรันและโปรแกรมก็คือกลุ่มของคำสั่งที่อยู่บนพื้นที่หน่วยความจำ

โปรเซส หมายถึง งานหรือโปรแกรมที่อยู่ในหน่วยความจำที่ทำการรันอยู่ในขณะนั้นและการรันนั้นมีกระทบกับทรัพยากรของระบบส่วนใดส่วนหนึ่งบางครั้งคำว่า process และ task อาจใช้ในความหมายเดียวกัน คือ งานที่ทำการรันอยู่ในขณะนั้นโดยทั่วไปจะต่างกันตรงที่ process คือ โปรแกรมที่ทำการรันบนส่วนของ user mode ส่วน task คือ โปรแกรมที่ทำการรันบนส่วนของ kerenel mode

5.2 โปรเซสสเตท (Process State)

State ต่าง ๆ ของโปรเซสทำหน้าที่ในการอธิบายสิ่งที่สามารถเกิดขึ้นได้กับ process ต่าง ๆ มันประกอบไปด้วย array of flag ซึ่งทำการชี้ไปยัง process ที่สามารถเกิดขึ้นได้ ใน Linux รุ่นใหม่นี้มีลักษณะการทำงานของ state แบบ mutually exclusive โดยที่ flag มีการเซ็ทการทำงานให้กับ state เพียงหนึ่ง flag ส่วนที่เหลือนั้นต้องการทำการ clear flag สำหรับ state ที่สามารถเกิดขึ้นได้มีดังต่อไปนี้

TASK_RUNNING

State นี้มีการทำงานโดยแบ่งออกเป็น state ย่อย 2 state คือ เอ็กซิคิว และขณะรอการทำงานเพื่อการเอ็กซิคิวและพร้อมที่จะทำการรัน

#define TASK_RUNNING 0

TASK_INTERRUPTIBLE (Waiting)

State นี้คือ state ที่เมื่อโปรเซสอยู่ในสถานะ suspend หรือหลับอยู่สามารถที่ทำการอินเตอร์รัพได้ เช่น เมื่อมีการทำการสั่งอินเตอร์รัพฮาร์ดแวร์หรือมีการส่งสัญญาณเพื่อทำให้โปรเซส wake up เป็นต้น เมื่อมีการอินเตอร์รัพแล้วถ้าโปรเซสอยู่ในสถานะ TASK_RUNNING ต้องกลับไปทำงานยังโหมดของ TASK_RUNNING ต่อไป

#define TASK_INTERRUPTIBLE 1

TASK_UNINTERRUPTIBLE (Waiting)

State นี้คือ state ที่โปรเซสไม่ยอมให้มีการอินเตอร์รัพเกิดขึ้น เช่น กรณีที่โปรเซสเปิด device file และและทำการ probing เพื่อทำการติดต่อกับ hardware device ดังนั้น device driver ต้องไม่ยอมให้มีการอินเตอร์รัพจนกระทั่งทำการ probing จนเสร็จ

#define TASK_UNINTERRUPTIBLE 2

TASK_STOPPED

Stateนี้คือstateที่โปรเซสเอ็กซิคิวหยุดซึ่งหยุดหลังจากที่ได้รับสัญญาณSIGSTOP, SIGTSTP,SIGTTIN หรือสัญญาณ SIGTTOU

#define TASK_STOPPED 8

TASK_ZOMBIE

State นี้คือ state ที่โปรเซสอยู่ในสถานะที่ไม่สามารถทำงานต่อได้เรียกว่าโปรเซส terminated ซึ่งอาจเรียกตามลักษณะการทำงานว่าโปรเซสกำพร้าเนื่องจากโปรเซสไม่ทำงานต่อแต่เคอรเนลก็ยังไม่สามารถที่ทำการละทิ้งโปรเซสนี้ได้เนื่องจากอาจยังมีโปรเซสพาเร้นท์ที่ยังต้องการใช้โปรเซสนี้อยู่แต่โปรเซสพาเร้นท์ยังไม่ส่งสัญญาณออกมาทำให้โปรเซสนี้เป็นโปรเซสกำพร้า

#define TASK_ZOMBIE 4

5.3 โครงสร้างข้อมูลของโปรเซส ในลีนุกซ์ (Process Descriptor)

เคอร์เนลใช้โครงสร้างข้อมูลมากมายในการควบคุมโปรเซสโดยโครงสร้างข้อมูลทำหน้าที่คอยทำการติดต่อกันระหว่างโปรเซสซึ่งโครงสร้างของโปรเซสอธิบายในรูปซึ่งจากรูปเป็นการอธิบายถึงการจัดการของโครงสร้างข้อมูลในการจัดการกับโปรเซสต่าง ๆ

ส่วนของ source code ต่อไปนี้ คือ การอธิบายการทำงานของโปรเซสเมื่อมีการสร้างโปรเซสขึ้นมาใหม่ซึ่งการสร้างโปรเซสใหม่แต่ละครั้งก็ต้องมีการสร้างโครงสร้างข้อมูลของโปรเซสขึ้นมาใหม่ให้กับโปรเซสนั้น

```
struct task_struct {
        /*
         * offsets of these are hardcoded elsewhere - touch with care
         */
        volatile long state;        /* -1 unrunnable, 0 runnable, >0 stopped */
        unsigned long flags;        /* per process flags, defined below */
        int sigpending;
        mm_segment_t addr_limit;/* thread address space:
                                                0-0xBFFFFFFF for user-thead
                                                0-0xFFFFFFFF for kernel-thread
                                            */
        struct exec_domain *exec_domain;
        volatile long need_resched;
        unsigned long ptrace;

        int lock_depth;             /* Lock depth */

/*
 * offset 32 begins here on 32-bit platforms. We keep
 * all fields in a single cacheline that are needed for
 * the goodness() loop in schedule().
 */
        long counter;
        long nice;
        unsigned long policy;
        struct mm_struct *mm;
        int has_cpu, processor;
        unsigned long cpus_allowed;
        /*
         * (only the 'next' pointer fits into the cacheline, but
         * that's just fine.)
         */
        struct list_head run_list;
        unsigned long sleep_time;
```

```
        struct task_struct *next_task, *prev_task;
        struct mm_struct *active_mm;

/* task state */
        struct linux_binfmt *binfmt;
        int exit_code, exit_signal;
        int pdeath_signal;  /*  The signal sent when the parent dies  */
        /* ??? */
        unsigned long personality;
        int dumpable:1;
        int did_exec:1;
        pid_t pid;
        pid_t pgrp;
        pid_t tty_old_pgrp;
        pid_t session;
        pid_t tgid;
        /* boolean value for session group leader */
        int leader;
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->p_pptr->pid)
         */
        struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
        struct list_head thread_group;

        /* PID hash table linkage. */
        struct task_struct *pidhash_next;
        struct task_struct **pidhash_pprev;

        wait_queue_head_t wait_chldexit;   /* for wait4() */
        struct semaphore *vfork_sem;                /* for vfork() */
        unsigned long rt_priority;
```

```
        unsigned long it_real_value, it_prof_value, it_virt_value;
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
        struct timer_list real_timer;
        struct tms times;
        unsigned long start_time;
        long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
        unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
        int swappable:1;
/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        int ngroups;
        gid_t     groups[NGROUPS];
        kernel_cap_t  cap_effective, cap_inheritable, cap_permitted;
        int keep_capabilities:1;
        struct user_struct *user;
/* limits */
        struct rlimit rlim[RLIM_NLIMITS];
        unsigned short used_math;
        char comm[16];
/* file system info */
        int link_count;
        struct tty_struct *tty; /* NULL if no tty */
        unsigned int locks; /* How many file locks are being held */
/* ipc stuff */
        struct sem_undo *semundo;
        struct sem_queue *semsleeping;
/* CPU-specific state of this task */
        struct thread_struct thread;
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
```

```
/* signal handlers */
        spinlock_t sigmask_lock;  /* Protects signal and blocked */
        struct signal_struct *sig;

        sigset_t blocked;
        struct sigpending pending;

        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;

/* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty */
        spinlock_t alloc_lock;
};

/*
 * Per process flags
 */
#define PF_ALIGNWARN        0x00000001      /* Print alignment warning msgs */
                                            /* Not implemented yet, only for 486*/
#define PF_STARTING   0x00000002      /* being created */
#define PF_EXITING    0x00000004      /* getting shut down */
#define PF_FORKNOEXEC       0x00000040      /* forked but didn't exec */
#define PF_SUPERPRIV  0x00000100      /* used super-user privileges */
#define PF_DUMPCORE  0x00000200      /* dumped core */
#define PF_SIGNALED   0x00000400      /* killed by a signal */
#define PF_MEMALLOC 0x00000800      /* Allocating memory */
#define PF_VFORK      0x00001000      /* Wake up parent in mm_release */
```

```
#define PF_USEDFPU      0x00100000      /* task used FPU this quantum (SMP) */

/*
 * Ptrace flags
 */

#define PT_PTRACED      0x00000001
#define PT_TRACESYS     0x00000002
#define PT_DTRACE       0x00000004      /* delayed trace (used on m68k, i386) */
#define PT_TRACESYSGOOD     0x00000008

/*
 * Limit the stack by to some sane default: root can always
 * increase this limit if needed..  8MB seems reasonable.
 */
#define _STK_LIM        (8*1024*1024)

#define DEF_COUNTER  (10*HZ/100)      /* 100 ms time slice */
#define MAX_COUNTER(20*HZ/100)
#define DEF_NICE        (0)

/*
 *  INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x1fffff (=2MB)
 */
#define INIT_TASK(tsk)  \
{                                                       \
    state:          0,                                  \
    flags:          0,                                  \
    sigpending:             0,                          \
    addr_limit:             KERNEL_DS,                  \
    exec_domain:  &default_exec_domain,                 \
    lock_depth:             -1,                         \
    counter:                DEF_COUNTER,                \
```

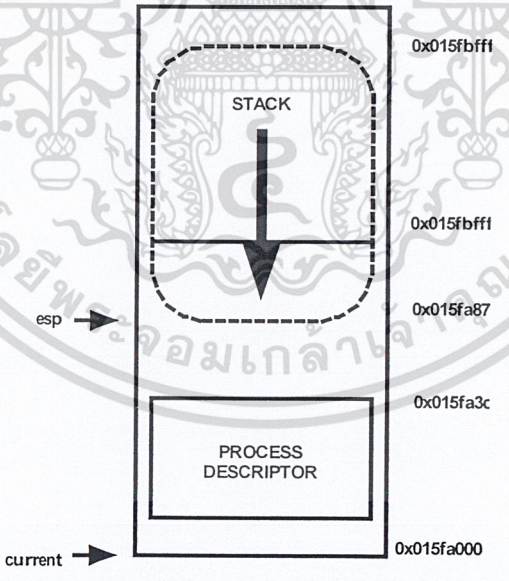
```
    nice:           DEF_NICE,                                       \
    policy:             SCHED_OTHER,                                \
    mm:                 NULL,                                       \
    active_mm:          &init_mm,                                   \
    cpus_allowed: -1,                                               \
    run_list:           LIST_HEAD_INIT(tsk.run_list),               \
    next_task:          &tsk,                                       \
    prev_task:          &tsk,                                       \
    p_opptr:            &tsk,                                       \
    p_pptr:             &tsk,                                       \
    thread_group:   LIST_HEAD_INIT(tsk.thread_group),               \
    wait_chldexit:  __WAIT_QUEUE_HEAD_INITIALIZER(tsk.wait_chldexit),\
    real_timer:         {                                           \
        function:       it_real_fn                                  \
    },                                                              \
    cap_effective:  CAP_INIT_EFF_SET,                               \
    cap_inheritable:    CAP_INIT_INH_SET,                           \
    cap_permitted: CAP_FULL_SET,                                    \
    keep_capabilities:  0,                                          \
    rlim:           INIT_RLIMITS,                                   \
    user:           INIT_USER,                                      \
    comm:               "swapper",                                  \
    thread:             INIT_THREAD,                                \
    fs:                 &init_fs,                                   \
    files:          &init_files,                                    \
    sigmask_lock:   SPIN_LOCK_UNLOCKED,                             \
    sig:            &init_signals,                                  \
    pending:            { NULL, &tsk.pending.head, {{0}}},          \
    blocked:            {{0}},                                      \
    alloc_lock:         SPIN_LOCK_UNLOCKED                          \
}
```

โครงสร้างข้อมูลของโปรเซสหรือ process descriptor นั้นในเคอร์เนลสามารถแสดงโดยกำหนดโครงสร้าง struct task_struct ซึ่งทุกครั้งที่สร้างโปรเซสขึ้นมาใหม่ก็มีการ allocate โครงสร้างข้อมูลสำหรับ process ทุกครั้ง

5.4 โปรเซสในเคอร์เนลโหมด (Storing a process descriptor)

Task array ประกอบด้วยพอยเตอร์ที่ชี้ไปยัง process descriptor เท่านั้นไม่ได้มีพื้นที่ของโครงสร้างข้อมูลด้วยตัวมันเอง เพราะการ allocate หน่วยความจำให้กับ process descriptor เป็นในลักษณะ dynamic การ allocate หน่วยความจำให้กับ process แต่ละครั้งไม่ได้อยู่พื้นที่เดียวกันทุกครั้งซึ่งต่างกับส่วนของเคอร์เนลที่เก็บข้อมูลในหน่วยความจำในลักษณะถาวร ระบบปฏิบัติการลีนุกซ์เก็บสองโครงสร้างข้อมูลของแต่ละโปรเซสในหน่วยความจำขนาด 8 KB ซึ่งสองโครงสร้างข้อมูลนั้น คือ ส่วนของ process descriptor และส่วนของ kernel mode process stack

การทำงานของโปรเซสในส่วนของเคอร์เนลโหมดการเข้าถึง stack ที่อยู่ใน kernel data segment แตกต่างกับการใช้ stack โดยโปรเซสที่อยู่ในส่วนของ user โหมด ซึ่งส่วนของการควบคุมโดยเคอร์เนลใช้พื้นที่ของ stack น้อยเพียง 2000-3000 ไบต์เท่านั้น สำหรับความต้องการของเคอร์เนล stack จากที่มีพื้นที่ทั้งหมด 8 KB สำหรับส่วนของ kernel stack และ process descriptor โดยการทำงานของหน่วยความจำจะมีลักษณะโดย process descriptor ใช้พื้นที่ของ stack โดยเริ่มจากจุด start ของพื้นที่หน่วยความจำส่วนของ stack นั้นเริ่มนับจากด้านท้ายของพื้นที่หน่วยความจำ



รูปที่ 5-1 การจัดเก็บโครงสร้างข้อมูลของโปรเซสในเคอร์เนลสแต็ก

รีจิสเตอร์ esp คือ stack pointer ของซีพียู ซึ่งอยู่ที่ address สูงสุดของ stack ในระบบอินเทลนั้น stack จะเริ่มนับตั้งแต่จุดสุดท้ายของพื้นที่หน่วยความจำและนับถัดไปเรื่อย ๆ โดยเข้าหาจุดเริ่มต้นของพื้น

ที่หน่วยความจำเมื่อทำการเปลี่ยนโหมดจาก user โหมดไปยังเคอร์เนลโหมดโปรเซสก็ทำงานในส่วนของเคอร์เนล stack ทันทีโดยที่มีรีจิสเตอร์ esp ชี้ไปยังพื้นที่ที่ใช้งานในหน่วยความจำนั้น

การแสดงถึงการรูปแบบของการเก็บข้อมูลลงใน stack สามารถเขียนด้วยภาษาซีในการอธิบาย stack ได้ดังนี้ (ประกาศโครงสร้างข้อมูลแบบ union)

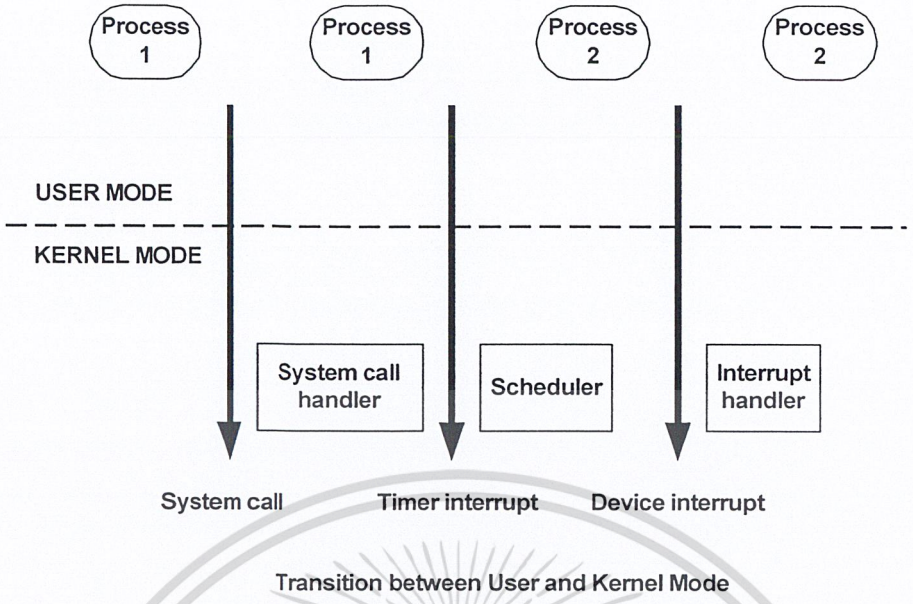
```
union task_union {
        struct task_struck task;
        unsigned long stack[2048];
}
```

หลังจากเปลี่ยนโหมดจาก user mode เป็น kernel mode รีจิวเตอร์ esp อยู่ที่ address 0x015fc000 , process descriptor เริ่มที่ address 0x15fa000 โดยที่ค่าของ esp จะลดลงเท่ากับข้อมุลที่เขียนลงใน stack โดยที่ process descriptor มีพื้นที่น้อยกว่า 1000 ไบต์ ดังนั้นเคอร์เนล stack จึงสามารถขยายได้ถึง 7200 ไบต์

การทำงานของโปรเซสในการติดต่อกับส่วนของเคอร์เนลโหมด (The Process / Kernel Model)

ซีพียูโดยทั่วไปแบ่งการทำงานออกเป็น 2 โหมด คือ ในส่วนของ user mode และส่วนของ kernel mode ถึงแม้ว่าซีพียูบางชนิดอาจแบ่งการทำงานออกเป็น 4 state ก็ตามแต่ว่าในลีนุกซ์ก็จะใช้เพียง 2 mode เท่านั่น คือ user mode และ kernel mode ดังที่ได้กล่าวไปแล้ว

เมื่อโปรแกรมทำการเอ็กซิคิวใน user mode มันก็ไม่สามารถเข้าถึงส่วนของโครงสร้างข้อมูลของเคอร์เนลหรือส่วนของเคอร์เนลโปรแกรมได้ ซ๊พียูได้เตรียมอินสตรัคชั่นพิเศษไว้เพื่อทำการสวิทช์จาก User Mode ไปยัง Kernel Mode โดยที่โปรแกรมที่ทำการเอ็กซิคิวบน User Mode นั้นจะทำการสวิทช์มายัง Kernel Mode ได้ก็ต่อเมื่อมีการร้องขอเซอร์วิสโดยเคอร์เนลเท่านั้น เมื่อเคอร์เนลรันโปรแกรมได้ตามต้องการแล้วก็ส่งโปรแกรมกลับมายังส่วนของ User Mode เหมือนเดิม

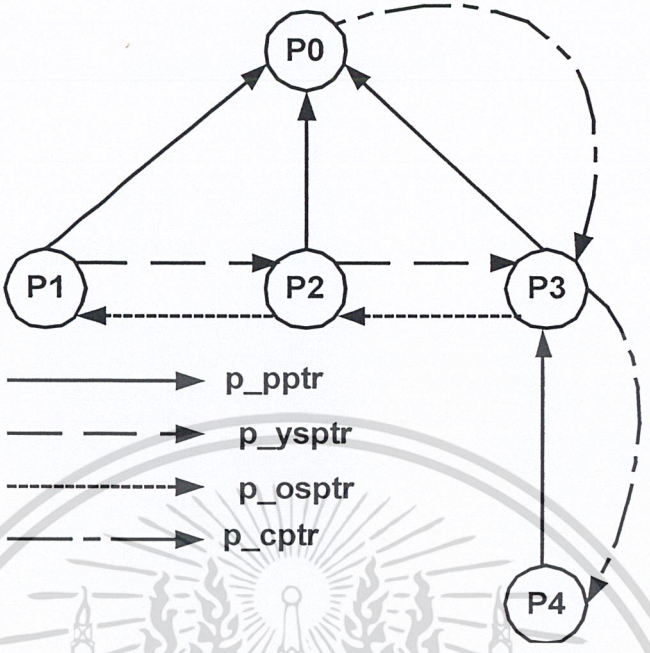


รูปที่ 5-2 แสดงการทำงานของโปรเซสในการติดต่อกับส่วนของเคอร์เนลโหมด

5.5 ความสัมพันธ์ระหว่างโปรเซส (Parenthood Relationship Among Process)

โปรเซสสร้างโดยโปรแกรมที่มีความสัมพันธ์ระหว่างโปรเซสพาเรนและโปรเซสไชล์เพราะโปรเซสสามารถสร้างโปรเซส children ได้อีกเป็นจำนวนมากและในโปรเซส children ยังมีความสัมพันธ์กันแบบ sibling ระหว่างโปรเซส children ด้วยกันโดยที่ทุก ๆ ฟิลด์ต้องมีส่วนแนะนำใน process descriptor เพื่อแสดงถึงความสัมพันธ์ระหว่างโปรเซส โปรเซส 0 และ 1 สร้างโดยเคอร์เนลโดยที่มีโปรเซส 1 (init) เป็นบรรพบุรุษของทุกโปรเซส โครงสร้างของโปรเซส P ประกอบด้วยฟิลด์ต่าง ๆ ดังต่อไปนี้

- p_opptr (original paremt)
  ชี้ไปยังโปรเซส original parent เป็นโปรเซสที่เกิดขึ้นก่อนครั้งแรก
- p_pptr (parent)
  ชี้ไปยังโปรเซส parent มีค่าเท่ากับโปรเซสที่เป็น original parent
- p_cptr (child)
  ชี้ไปยังโปรเซสลูกที่ถูกสร้างขึ้นโดยโปรเซสที่เป็นโปรเซส parent
- p_ysptr (younger sibling)
  ชี้ไปยังโปรเซสที่เป็นโปรเซสลูกด้วยกันแต่ว่ามีอายุน้อยกว่าโปรเซสนั้นหรือว่าเกิดทีหลังโปรเซสนั้นนั่นเองแต่ว่ามีโปรเซสพาเรนท์เดียวกัน
- p_osptr (older sibling)
  ชี้ไปยังโปรเซสที่เป็นโปรเซสลูกด้วยกันแต่ว่ามีอายุมากกว่าโปรเซสนั้นหรือว่าเกิดก่อนโปรเซสนั้นนั่นเองแต่ว่ามีโปรเซสพาเรนท์เดียวกัน



รูปที่ 5-3 แสดงความสัมพันธ์ระหว่างโปรเซส

สำหรับความสัมพันธ์ระหว่างโปรเซสในลีนุกซ์นั้นโปรเซสทั้งหมดนั้นลิงค์ถึงกันหมดโดยใช้คุณสมบัติที่เรียกว่า process list โดยที่โปรเซสทุกโปรเซสเป็นลูกของโปรเซส 0 ทั้งหมดซึ่งเมื่อมีการสร้างโปรเซสใหม่โปรเซสก็ทำการ copy โปรเซสพ่อไปยังโปรเซสลูกโดยใช้คุณสมบัติที่เรียกว่า Copy On Write ซึ่งเมื่อทำการ fork โปรเซสขึ้นมาใหม่เคอร์เนลทำการ copy โปรเซสพ่อไปยังโปรเซสลูกดังนี้

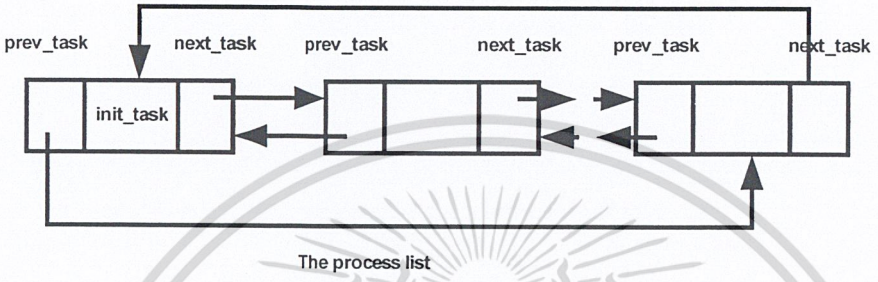
- ทำการอัลโลเคด page frame สำหรับ page tables ของโปรเซสลูก
- ทำการอัลโลเคด page frame สำหรับ page ของโปรเซสลูก
- เซ็ทค่า page tables ของโปรเซสลูก
- copy page ของโปรเซสพ่อไปยัง page ของโปรเซสลูก

ซึ่งเมื่อสร้างโปรเซสขึ้นมาใหม่แล้วโปรเซสพ่อจะลิงค์กับโปรเซสลูกจนกระทั่งโปรเซสลูกทำการรันจนหมดโค้ดเมื่อโปรเซสลูกทำการรันจนหมดโค้ด โปรเซสพ่อก็ทราบโดยโค้ดนั้นซึ่งหลังจากนั้นโปรเซสพ่อก็จะปล่อยโปรเซสลูกโดยใช้ฟังก์ชัน release

5.6 ลิสต์ของโปรเซส (Process list)

คือ ลิสต์ของโปรเซสต่าง ๆ ซึ่งต่อกันในลักษณะเป็นลิสต์แบบวงกลมโดยที่แต่ละลิสต์ประกอบไปด้วยพอยเตอร์ที่ชี้ไปยัง process descriptor ของแต่ละโปรเซส สำหรับโปรเซสแต่ละโปรเซสนั้นในภาษาซีจะเขียนเป็นโครงสร้างข้อมูล task_struct

จากรูปลิสต์สองทางที่ต่อกันในลักษณะวงกลมทำหน้าที่ลิงค์โปรเซสแต่ละโปรเซสเข้าด้วยกันซึ่งเรียกลิสต์รูปนี้ว่า process list โดยที่ใน process descriptor ประกอบด้วยฟิลด์ที่สำคัญสองฟิลด์ คือ prev_task และ next_task โดยที่ prev_task ทำหน้าที่ ชี้ไปยังโปรเซสก่อนหน้านี้ส่วน next_task ทำหน้าที่ในการชี้ไปยังโปรเซสถัดไป โดยที่ส่วนหัวของลิสต์ คือ init_task ซึ่งคือ โปรเซสแรกสุดนั่นเอง โดยฟิลด์ prev_task ของ init_task ชี้ไปยัง process descriptor สุดท้ายที่แทรกเข้าไปในลิสต์

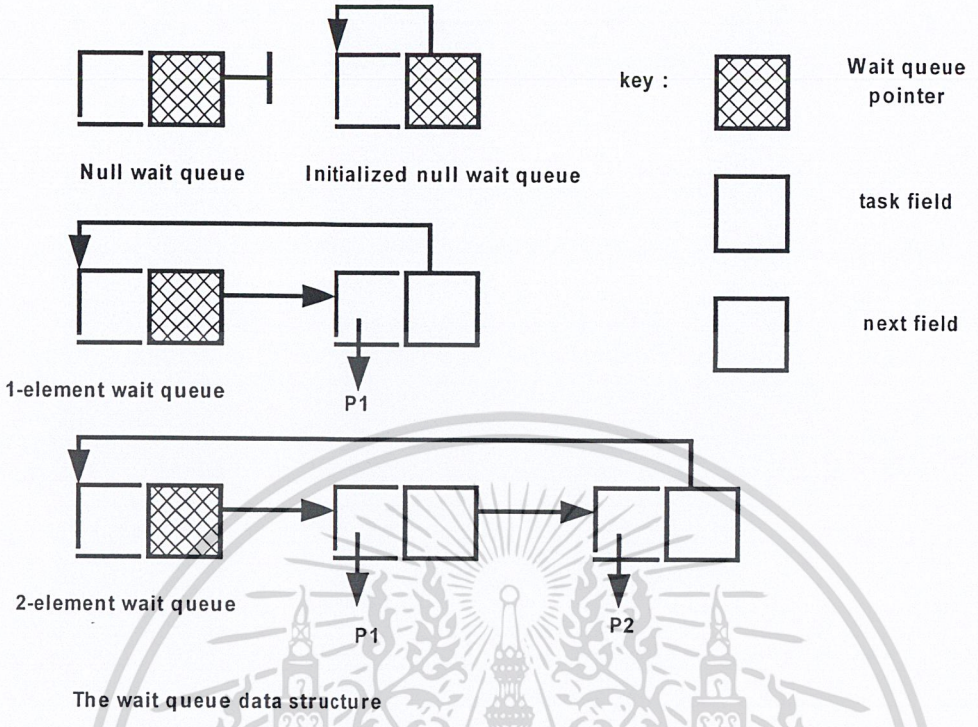


รูปที่ 5-4 แสดงโปรเซสลิส (Process List)

5.7 ลิสต์ของเวทคิว (Wait Queues)

ทำงานต่างจากคิวแบบ runqueue เพราะ runqueue เป็นคิวที่เก็บโปรเซสที่อยู่ในสถานะ TASK_RUNNING เข้าด้วยกัน ส่วน waitqueue เก็บโปรเซสที่รอเข้าด้วยกันซึ่งต่อไปนี้เป็นการอธิบายว่าโปรเซสสถานะใดอยู่ในคิวแบบใด

- TASK_RUNNING โปรเซสที่อยู่ในสถานะนี้อยู่ใน runqueue
- โปรเซสใน TASK_STOPPED หรือ TASK_ZOMBIE ไม่จำเป็นต้องมีการติดต่อกันระหว่างลิสต์ภายในคิวดังนั้นจึงไม่จำเป็นต้องมีกลุ่มในคิวเพราะโปรเซสที่อยู่ในสองสถานะนี้อยู่ในสถานะ terminate แล้ว แต่อาจยังถูกใช้โดยโปรเซส parent อยู่โปรเซสที่อยู่ในสถานะนี้จึงไม่จำเป็นต้องมีคิว
- โปรเซสใน TASK_INTERRUPTIBLE หรือ TASK_UNINTERRUPTIBLE เป็นโปรเซสเฉพาะเหตุการณ์และโปรเซสเหล่านี้ไม่มีข้อมูลพอที่จะให้โปรเซสทำงานต่อไปได้โดยโปรเซสทำการ sleep หรือ suspend ดังนั้นโปรเซสเหล่านี้จึงต้องรอและรอในคิวชนิดที่เรียกว่า waitqueue ซึ่งคิวนี้ทำหน้าที่เก็บลิสต์ของโปรเซสที่รอเหตุการณบางอย่างโดยปกติแล้ว waitqueue นั้นทำงานในส่วนของเคอร์เนลโหมด



รูปที่ 5-6 แสดง เวทคิว (Wait Queue)

5.8 การทำงานของโปรเซสและตารางการจัดการโปรเซส

แบ่งออกเป็นส่วนที่สำคัญดังต่อไปนี้

5.8.1 การสร้างโปรเซส

การจัดการโปรเซสในลีนุกซ์แบ่งการทำงานเป็น 2 ลักษณะ คือ การสร้างโปรเซสและการเอ็กซิคิวท์โปรเซสซึ่งใช้ฟังก์ชันระดับต่ำเพื่อช่วยการทำงาน

- ฟังก์ชัน fork ทำหน้าที่ ในการสร้างโปรเซสขึ้นมาใหม่
- ฟังก์ชัน execve ทำหน้าที่ ในการเอ็กซิคิวท์โปรเซสหลังจากสร้างโปรเซสขึ้นมา

ระบบลีนุกซ์โปรเซสต้องมีข้อมูลทั้งหมดที่เกี่ยวกับระบบปฏิบัติการเพื่อที่ทำการเอ็กซิคิวท์โปรแกรม คุณสมบัติของโปรเซสต้องประกอบด้วยสิ่งต่าง ๆ ดังนี้ คือ process identity , environment , และ context

The fork() system call ทำหน้าที่ ในการสร้างโปรเซสขึ้นมาใหม่ โดยมีขั้นตอนการทำงาน ดังนี้

1. allocate โครงสร้างข้อมูลโปรเซส descriptor โดยสร้างโครงสร้างข้อมูล struct task_struct ขึ้นมาใหม่และทำการลิงค์โปรเซสเข้าไปยัง task list

2. สร้างพื้นที่ของ kernel stack ใหม่สำหรับการทำงานในกรณีที่โปรเซสต้องการที่จะทำงานในส่วนของการเอ็กซิคิวบนเคอร์เนลโหมด

3. ก๊อปปี้ข้อมูลแต่ละฟิลด์จากโครงสร้างข้อมูลของ parent process descriptor ลงไปยังโครงสร้างข้อมูลของ child process descriptor

4. ทำการเปลี่ยนแปลงฟิลด์ต่าง ๆ ของ child descriptor เพื่อเป็นการแสดงว่าโปรเซสนี้เป็นโปรเซสที่สร้างขึ้นมาใหม่ โดย

- เซ็ทค่า process identifier (PID) ใหม่ เพื่อเป็นการแสดงว่าโปรเซสนั้นมี process identifier (PID) เป็นของตนเอง
- สร้างลิงค์เพื่อชี้ไปยังโปรเซส parent และ โปรเซส siblings
- initial โปรเซสในเรื่องของเวลา เช่น creat time , เซ็ทเวลาใน time slice (ช่วงเวลาของโปรเซสที่เข้าจองซีพียู) เป็นต้น

5.8.2 โปรเซสไอดี (PID)

- Process ID (PID) เป็นการแสดงถึงความแตกต่างของแต่ละโปรเซสซึ่งแต่ละโปรเซสก็ต้องมี PID เป็นของตัวเองเพื่อแสดงตัวว่าเป็นโปรเซสต่อระบบปฏิบัติการมีประโยชน์สำหรับการทำติดต่อกับสัญญาณ system call ต่าง ๆ เพื่อทำการเปลี่ยนแปลงโปรเซสหรือเพื่อรอโปรเซสอื่น ๆ เป็นต้น สำหรับ PID คือ ตัวเลขธรรมดาทั่วไป เช่น 0 , 1 , 2 โดยหมายเลข 0 หมายถึงโปรเซสบรรพบุรุษโดยที่โปรเซสทุกโปรเซสต้องมีโปรเซสนี้เป็นบรรพบุรุษ โปรเซสที่สร้างขึ้นมาทีหลังต้องมีข้อมูลของโปรเซสนี้รวมอยู่ด้วย เช่น ข้อมูลของ root directory โดยที่ตัวเลขเหล่านี้ไม่ใช่ index ที่ชี้ไปยัง task vector ในที่นี้ค่าของ ProcessID มีค่าเป็น interger ตั้งแต่ 0 – 32,767 หมายเลข 0 และ 1 เป็นหมายเลขพิเศษของโปรเซสของระบบส่วนโปรเซสที่เหลือเรียกโปรเซสนั้นว่า “normal processes”

- Credentials ทุกโปรเซสต้องมี user ID เป็นของตนเองและ แต่ละโปรเซสต้องมีการรวมกลุ่ม user ID กันซึ่งเรียกว่า group ID เป็นของตนเองด้วยโดยที่มีหนึ่งหรือมากกว่านั่นก็ได้เพื่อที่จะนำ user ID และ group ID เหล่านั้นมาใช้ทำการเข้าถึงทรัพยากรและอุปกรณ์ device ต่าง ๆ และไฟล์ เป็นต้น

5.8.3 ไพออริตี้ของโปรเซสในลีนุกซ์

ค่า Priority คือ ค่าของตัวเลขที่แสดงถึงน้ำหนักในการที่จะให้ซีพียูเลือกว่าจะเลือกโปรเซสใดเข้ามาทำงานถ้าโปรเซสใดมีค่า Priority ที่สูงกว่าซีพียูก็สามารถเลือกโปรเซสนั้นเข้ามาทำงานได้ทันที Nonrealtime processes มีค่า Priority อยู่ 2 ชนิด คือ ชนิด a static priority และ dynamic priority ส่วน Realtime processes ได้เพิ่มชนิดของ priority เข้าไปอีกซึ่งใช้ใน Realtime processesเรียกว่า realtime priority

- Static priority เรียก Static เพราะว่าค่าของ priority ไม่สามารถเปลี่ยนแปลงได้ในช่วงเวลานั้น ซึ่งสามารถเข้ามาแก้ไขได้โดยผู้ใช้เท่านั้นมีค่าตั้งแต่ 1-99 ตารางการจัดาการโปรเซสไม่สามารถเปลี่ยนได้จะเปลี่ยนได้โดยยูสเซอร์เท่านั้น
- Dynamic priority ไพออริตี้ชนิดนี้ใช้กับโปรเซสชนิด Conventional process เท่านั้นโดยมันมีค่าผลรวมของ base time quantum (อาจเรียกได้ว่าเป็น base priority ของโปรเซส) และมีส่วนสัมพันธ์กับจำนวนคลีอกของซีพียูไทม์ของแต่ละโปรเซสก่อนที่ค่า time quantum ของแต่ละ epoch หมดไป
- Realtime priority เป็นไพออริตี้ของโปรเซสที่ทำการตอบสนองอย่างทันทีทันใด

5.8.4 การเก็บงานไว้รันภายหลังเมื่อมีเหตุการณ์บางอย่างเกิดขึ้น (อินเตอร์รัพ)

การทำงานบางอย่างเมื่อเราไม่ทราบว่าต้องใช้เวลาในการทำงานเท่าไรการใช้วิธีถ่วงเวลาจะไม่เหมาะสมหรือถ้าต้องการกำหนดเวลาในการรันโปรแกรมบางอย่างล่วงหน้า , ลีนุกซ์สนับสนุนการทำงานดังกล่าวอยู่ 2 วิธีด้วยกัน คือ การใช้ Task Queues และ Kernel Timers

โครงสร้างข้อมูล **Task Queues**

โครงสร้างข้อมูลจะเป็นลิสต์ของงานที่จะต้องทำโดยเคอร์เนลเมื่อรันฟังก์ชันบางอย่าง โดยปรกติจะเป็นงานที่จะต้องทำต่อหลังจากการอินเตอร์รัพหรือเก็บงานไว้รันจนกระทั่งเกิดการอินเตอร์รัพ (return from sleep mode) และโมดูลจะต้องควบคุมการทำงานในส่วนนี้ทั้งหมด โครงสร้างข้อมูล Task Queues จะกำหนดไว้ในไฟล์ <linux/tqueue.h> ดังต่อไปนี้

```
struct  tq_struct {
        struct  tq_struct  next;    /* linked list of active bh 's */
        int     sync;               /*must be initialized to zero */
        void    (*routine);         /*function to call */
        void    (*data);            /* argument to function */
}
```

5.8.5 โปรเซสสวิทชิ่ง (Process Switching)

ในการควบคุมการเอ็กซิคิวท์ของโปรเซส เคอร์เนลต้องสามารถ suspend โปรเซสในขณะที่โปรเซสเอ็กซิคิวท์ได้และสามารถทำให้โปรเซสอื่นรันต่อหลังจากที่โปรเซสนั้น suspend อยู่ ซึ่งการที่เคอร์เนลสามารถควบคุมโปรเซสสามารถสั่งให้โปรเซสเอ็กซิคิวท์และ suspend เรียกการทำงานนี้ว่า process switching , task switching หรือ context switching ซึ่งองค์ประกอบของ process switching ในลีนุกซ์ประกอบด้วยส่วนต่าง ๆ ดังนี้ คือ

- Hardware Context
- Hardware Support

- Linux Code
- Saving the floating point registers

5.8.6 ฮาร์ดแวร์คอนเทก (Hardware Context)

ขณะที่โปรเซสแต่ละโปรเซสมีพื้นที่บน address space เป็นของตนเองทุกโปรเซสก็ต้องมีการแชร์ซีพียูรีจิสเตอร์ร่วมกัน ดังนั้นก่อนที่มีการเอ็กซิคิวท์โปรเซส เคอร์เนลต้องโหลดค่าต่าง ๆ เข้าไปในรีจิสเตอร์ ซึ่งกลุ่มของข้อมูลที่โหลดเข้าไปในรีจิสเตอร์ก่อนที่โปรเซสทำการเอ็กซิคิวท์บนซีพียู เรียกว่า Hardware Context ซึ่ง Hardware Context นี้เป็นส่วนย่อยมาจาก Process Execution Context ซึ่งต้องใช้ข้อมูลทั้งหมดในการทำ Process Execution ในลีนุกซ์ ส่วนของ Hardware Context ของโปรเซสจะเก็บไว้ใน TSS Segment ขณะที่ส่วนที่เหลือของการทำ Process Execution ถูกเก็บไว้ใน Kernel Mode Stack ซึ่ง TSS Segment ก็คือส่วนของ TSS field ของ process descriptor

5.8.7 เทรด (Thread)

ระบบลีนุกซ์นั้นสนับสนุนการทำงานในลักษณะ process และ threads โดย threads นั้นหมายถึงโปรเซสที่สร้างขึ้นมาใหม่โดยที่มีการแชร์พื้นที่บน address space ที่เดียวกับ โปรเซสพาเรนท์โดยที่พื้นที่ address space นั้นแบ่งการทำงานเป็นส่วน ๆ ในลักษณะ threads หลาย ๆ threads

การสร้าง threads ใหม่ขึ้นมานั้นสามารถทำได้โดยใช้ clone system call

- fork นั้นสร้างโปรเซสขึ้นมาใหม่โดยที่โปรเซสนั้นมีพื้นที่ address space เป็นของตัวเอง
- clone นั้นสร้างโปรเซสขึ้นมาใหม่โดยที่โปรเซสนั้นใช้พื้นที่บน address space ของโปรเซสพาเรนท์ มีการแชร์โครงสร้างข้อมูลกับโปรเซสพาเรนท์ แต่ว่าก็ยังมี identity ของโปรเซสที่เป็นของตนเอง เช่น Process ID และ Group ID เป็นต้น
- นอกจากนั้น clone ยังสามารถส่ง application ให้กันระหว่าง threads ซึ่งแชร์ข้อมูลระหว่างกันได้ด้วย

โปรเซสบนระบบลีนุกซ์สามารถจะติดต่อกันได้โดยการส่งข้อความเพื่ออินเตอร์รัพการทำงานของอีกโปรเซสหนึ่งหรือแม้แต่การใช้งานหน่วยความจำร่วมกัน , แต่ไม่สามารถจะใช้ตัวแปรร่วมกันได้

ในระบบลีนุกซ์มีออบเจ็คชนิดหนึ่งเรียกว่า thread ซึ่งมีการทำงานต่างจากโปรเซส คือ สามารถแยกการรันโปรเซสออกเป็นหลายส่วนด้วยกัน เช่น การรันส่วนของโปรแกรมไปพร้อมกัน 2 ส่วนในเวลาเดียวกัน เช่น การที่ word processor ตรวจสอบความถูกต้องของคำและจัดเรียงหน้าใหม่ในเวลาเดียวกัน การทำงานทั้งหมดเกิดขึ้นภายในโปรเซสเดียวกันและทำงานบนตัวแปรเดียวกัน

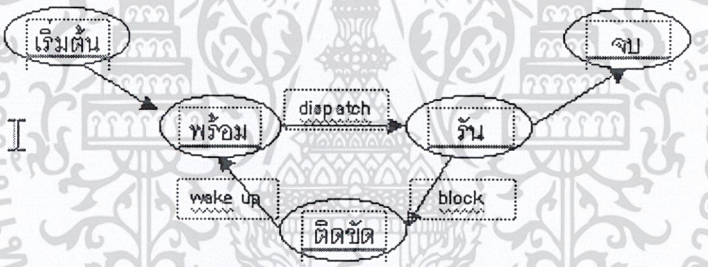
การเขียนโปรแกรมจัดการกับ thread จะมีประโยชน์ในงาน application บางอย่าง เช่น multithread database server ซึ่งเป็นเรื่องที่ซับซ้อน

thread ได้รับการกำหนดมาตรฐานกลไกการทำงานไว้ใน POSIX.4 ซึ่งเหมาะสำหรับ application แบบ real-time ซึ่งพบการใช้งาน thread ในระบบปฏิบัติการอื่นเช่นกัน เช่น วินโดวส์ NT , วินโดส์ 95

5.8.8 ตารางการจัดการโปรเซส (Process Scheduling)

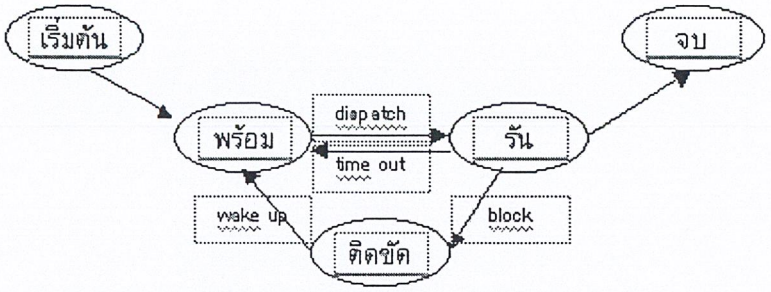
การคัดเลือกหาโปรเซสที่จะเข้าไปอยู่ในสถานะรันเมื่อซีพียูว่างนั้น ถือได้ว่าเป็นการจัดสรรซีพียูให้กับโปรเซสต่าง ๆ ซึ่งเป็นหน้าที่ของระบบปฏิบัติการส่วนของระบบปฏิบัติการที่มีหน้าที่จัดสรรซีพียูเรียกว่า "ตัวจัดคิวในระยะสั้น" (Short-term Scheduler) หน้าที่ของตัวจัดคิวระยะสั้นคือ คัดเลือกโปรเซสซึ่งรออยู่ในสถานะพร้อมที่เหมาะสมที่สุดให้เข้าไปอยู่ในสถานะรัน(ครอบครองซีพียู) กลไกคัดเลือกมีหลายวิธี แล้วแต่ผู้ออกแบบ ระบบปฏิบัติการจะเลือกใช้วิธีใด ซึ่งแต่และวิธีก็มีข้อดีข้อเสียในตัวมันเอง วิธีการนี้ได้แก่

1. การจัดคิวแบบ FCFS (first-come-first-served) เป็นวิธีที่ง่ายที่สุด คือ โปรเซสไหนเข้ามารอในคิวก่อนก็จะให้ได้ครอบครองซีพียูก่อน พูดง่าย ๆ คือ "มาก่อนได้ก่อน" ไม่มีการใช้ระยะเวลาควอนตัม ไม่มีการเปลี่ยนสถานะของโปรเซสจากสถานะรันมายังสถานะพร้อมโดยตรง แต่ถ้าโปรเซสใดต้องการใช้อุปกรณ์อินพุต-เอาพุต โปรเซสนั้นต้องปลดปล่อยซีพียู ข้อดีคือ การจัดคิวทำได้ง่ายไม่ยุ่งยากซับซ้อน ข้อเสียคือ โปรเซสอื่นเสียเวลารอนานมาก



รูปที่ 5-7 แสดงการจัดการโปรเซสแบบมาก่อนได้ก่อน (FCFS)

2. การจัดคิวแบบ RR (round - robin) เป็นแบบ FCFS "มาก่อนได้ก่อน" แต่ต่างกันตรงที่การครอบครองซีพียูจะต้องถูกควบคุมโดยระยะเวลาควอนตัม ข้อดีคือ เป็นการยุติธรรมแก่โปรเซสที่ต้องการครอบครองซีพียู โปรเซสอื่นไม่เสียเวลาการรอที่นาน ข้อเสียคือ ถ้าโปรเซสใดต้องการระยะเวลาในการทำงานนาน ต้องวนกลับมาอยู่ในคิวสถานะพร้อมหลายรอบ ถ้าคิวมีโปรเซสมาก เวลาที่ใช้ในการรอก็นาน การกำหนดให้ระยะเวลาควอนตัมน้อย ๆ ทำให้โปรเซสต่าง ๆ ได้ครอบครองซีพียูได้เร็วขึ้น เป็นจุดเริ่มต้นของระบบการแบ่งกันใช้เวลา หรือไทม์แชริ่ง (Timesharing system) เป็นการแบ่งเวลาของซีพียูให้กับทุก ๆ โปรเซสได้ใช้ทั่วกัน



รูปที่ 5-8 แสดงการจัดการโปรเซสแบบวนรอบ (Round-Robin)

3. การจัดคิวแบบลำดับความสำคัญ (Priority queue) จะแตกต่างจากคิวแบบธรรมดา ภายในคิวจะมีการจัดเรียงลำดับของโปรเซสต่าง ๆ ตามลำดับความสำคัญ แม้โปรเซสที่เข้ามาในคิวทีหลังแต่มีความสำคัญสูงกว่าก็จะได้ไปครอบครองซีพียูก่อน วิธีการพิจารณาความสำคัญของโปรเซส พิจารณาได้จากองค์ประกอบต่างๆเช่น

1. ผู้ใช้เป็นเจ้าของ เช่น โปรเซสของผู้ใช้ธรรมดาจะมีความสำคัญต่ำกว่าผู้ใช้ที่เป็นผู้ควบคุมดูแลระบบ

2. ประเภทโปรเซส เช่น ถ้าโปรเซสในโหมดแบต (Batch mode) จะมีความสำคัญต่ำกว่าโหมดโต้ตอบ (Interactive mode) เพราะโหมดโต้ตอบต้องสนองให้กับผู้ใช้ให้รวดเร็วที่สุดเท่าที่จะทำได้

3.ระยะเวลาที่โปรเซสเข้ามาอยู่ในระบบ โปรเซสที่อยู่ในระบบนานก็จะได้รับความสำคัญก่อน เพราะโปรเซสนั้นทำงานอยู่นานแล้ว ควรจะจบออกไปเสียที

4. การจัดคิวแบบ SJN (Shortest job next) จะคัดเลือกเอาโปรเซสที่ต้องการเวลาในการทำงานน้อยที่สุด มีความสำคัญมากกว่า เพื่อให้โปรเซสนั้นจบออกไปได้เร็วขึ้น ข้อเสียนี้มีผลต่อโปรเซสที่ต้องการเวลาในการทำงานนาน การจัดการโปรเซสในลักษณะแบบนี้ระบบปฏิบัติการลีนุกซ์ไม่ใช้เพราะไม่ทราบเวลาการทำงานที่แน่นอนของโปรเซส

5. การจัดคิวแบบ **SRT** (Shortest remaining time) วิธีนี้คล้าย SJN แต่จะเอาโปรเซสที่ "เหลือเวลาในการทำงานน้อยที่สุด" เข้าไปครอบครองซีพียู ซึ่งทำให้โปรเซสที่ใช้เวลานานในการทำงานสามารถจบออกไปได้เร็วขึ้น แต่โปรเซสที่ต้องการเวลาน้อยก็จบได้เร็วเช่นกัน การจัดคิวแบบ SJN และ SRT จะต้องมีการประมาณเวลาในการทำงานของโปรเซสไว้ล่วงหน้า แต่วิธีแบบ SRT ต้องมีการบันทึกเวลาที่โปรเซสทำงานไปแล้วไว้ด้วย เพื่อจะได้ทราบเวลาที่เหลือ การจัดการโปรเซสในลักษณะแบบนี้ระบบปฏิบัติการลีนุกซ์ไม่ใช้เช่นกันเพราะไม่ทราบเวลาการทำงานที่แน่นอนของโปรเซสทำให้ไม่ทราบถึงเวลาที่เหลือในการทำงานของโปรเซสด้วย

6. การจัดคิวแบบหลายระดับ (Multi-level queue)การจัดคิวแบบต่าง ๆ ที่กล่าวมาทั้งหมดเป็นการคัดเลือกโปรเซสในสถานะพร้อม แต่ไม่ว่าจะคัดเลือกด้วยวิธีใดก็ตามล้วนแล้วแต่เป็นการจัดคิวภายในคิวเพียง 1 คิวเท่านั้น เรียกว่าเป็นการจัดคิวแบบ 1 ระยะดับ เพื่อให้การจัดมีประสิทธิภาพ จึงมีการจัดคิวแบบหลายระดับ การจัดไม่จำเป็นต้องเป็นประเภทเดียวกัน เช่น ในคิวที่ 1 อาจเป็นคิวแบบลำดับความสำคัญ คิวที่ 2 อาจเป็นแบบ FCFS เป็นต้น

5.9 การทดลองที่ 3 โปรเซสและการจัดการโปรเซส
(Process And Process Scheduling)

วัตถุประสงค์
1. เพื่อศึกษาถึงสถานะ (state) ของโปรเซสและหลักการทำงานของโปรเซส
2. เพื่อศึกษาถึงอัลกอริทึม (Algorithm) ในการจัดการโปรเซสของระบบปฏิบัติการลีนุกซ์
3. เพื่อศึกษาและวิเคราะห์ถึงความเป็นเป็นได้เกี่ยวกับการจัดการโปรเซสชนิดอื่น ๆ ที่สามารถนำมาทดแทนตารางการจัดการโปรเซสปัจจุบัน

ทฤษฎี

ตารางการจัดการโปรเซส **(Linux Process Scheduling)**

ตารางการจัดการโปรเซส คือ ส่วนที่สำคัญของระบบปฏิบัติการเพราะเป็นส่วนที่ทำหน้าที่ในการคัดเลือกโปรเซสที่เหมาะสมที่สุดในการเข้ามาครอบครองซีพียูเพื่อที่ให้ซีพียูทำการรันโปรเซส ณ ช่วงระยะเวลาคาบเวลาหนึ่ง

อัลกอริทึมการจัดการโปรเซสของระบบปฏิบัติการลีนุกซ์

1. FIFO (First In First Out) หรือ FCFS (First Come First Serve)

คิวมีส่วนสำคัญมากในตารางการจัดการโปรเซสชนิดนี้ โดยระบบปฏิบัติการรันโปรเซสในส่วนที่อยู่ส่วนต้นของคิว เมื่อมีโปรเซสใหม่เข้ามาจะเข้าไปอยู่ส่วนท้ายของคิว โดยโปรเซสไม่ปล่อยซีพียูจนกว่าทำการรันโปรเซสนั้นเสร็จหรือมีการปฏิบัติการกับส่วนของ I/O ต่าง ๆ โปรเซสไหนมาก่อนมีส่วนเข้าครอบครองซีพียูก่อน

2. RR (Round-Robin) การจัดการโปรเซสแบบ RR อาจเรียกว่าเป็นการจัดการโปรเซสแบบเวียนเทียนหรือวนรอบ ลักษณะการคัดเลือกโปรเซสในคิวจะเป็นแบบ "มาก่อนได้ก่อน" แต่ต่างกันตรงที่การครอบครองซีพียูของโปรเซสในสถานะรัน จะถูกจำกัดเวลาได้ด้วยระยะเวลาควอนตัมหรือเป็นแบบตัดตอนได้ ทำให้โปรเซสที่ต้องการเวลาในการทำงานนานจะต้องเปลี่ยนสถานะหมุนวนระหว่างสถานะพร้อมและสถานะรัน

ไพออริตี้ของโปรเซสในลีนุกซ์

ค่า Priority คือ ค่าของตัวเลขที่แสดงถึงน้ำหนักในการที่จะให้ซีพียูเลือกว่าจะเลือกโปรเซสใดเข้ามาทำงานถ้าโปรเซสใดมีค่า Priority ที่สูงกว่าซีพียูก็สามารถเลือกโปรเซสนั้นเข้ามาทำงานได้ทันที Nonrealtime processes มีค่า Priority อยู่ 2 ชนิด คือ ชนิด a static priority และ dynamic priority

- Static priority เรียก Static เพราะว่าค่าของ priority ไม่สามารถเปลี่ยนแปลงได้ในช่วงเวลานั้น ซึ่งสามารถเข้ามาแก้ไขได้โดยผู้ใช้เท่านั้นมีค่าตั้งแต่ 1-99 ตารางการจัดาการโปรเซสไม่สามารถเปลี่ยนได้จะเปลี่ยนได้โดยยูสเซอร์เท่านั้น
- Dynamic priority ไพออริตี้ชนิดนี้ใช้กับโปรเซสชนิด Conventional process เท่านั้นโดยมันมีค่าผลรวมของ base time quantum (อาจเรียกได้ว่าเป็น base priority ของโปรเซส) และมีส่วนสัมพันธ์กับจำนวนคล็อกของซีพียูไทม์ของแต่ละโปรเซสก่อนที่ค่า time quantum ของแต่ละ epoch หมดไป

Static priority ของเรียลไทม์โปรเซสมีความสำคัญมากกว่า Dynamic priority ของโปรเซสชนิด Conventional process แต่ว่าการจัดการโปรเซสนั้นจะเริ่มด้วย Conventional process เท่านั้นเมื่อไม่มีโปรเซสเรียลไทม์ใดที่อยู่ในสถานะ TASK_RUNNING สเตท

โปรเซสในลีนุกซ์แบ่งเป็น 3 ระดับ

Interactive processes คือ โปรเซสที่ทำการตอบสนองกับยูสเซอร์โดยรอเหตุการณ์บางอย่าง เช่น คีย์บอร์ดหรือเม้าส์ เป็นต้น

Batch processes คือ โปรเซสที่ไม่จำเป็นต้องมีความสัมพันธ์กับยูสเซอร์เนื่องจากเป็นโปรเซสที่รันอยู่เบื้องหลัง เพราะเป็นโปรเซสที่ไม่จำเป็นต้องมีการตอบรับ โปรเซสชนิดนี้ถูกจัดการโดยตารางการจัดการโปรเซส

Realtime processes คือ โปรเซสที่แข็งแกร่งที่สุดที่ตารางการจัดการโปรเซสต้องการ ดังนั้นโปรเซสไม่ถูกบล็อกโดยโปรเซสที่มีค่าไพออริตี้ที่มัน พวกมันเป็นโปรเซสที่มีอัตราการตอบสนองต่ำและเป็นโปรเซสที่มีระดับความสำคัญมากที่สุด มีความแปรผันเวลาของโปรเซสที่ต่ำ ชนิดของโปรแกรมที่เป็นโปรเซสแบบเรียลไทม์ เช่น แอพพลิเคชั่นประเภทวีดีโอและซาวน์ การควบคุมหุ่นยนต์และโปรแกรมที่ใช้ในงานประเภท physical sensor เป็นต้น

โครงสร้างข้อมูลที่สำคัญของตารางการจัดการโปรเซส

โปรเซสแต่ละโปรเซสเชื่อมต่อกันโดยใช้ลิงค์ลิสต์ในการเชื่อมต่อโครงสร้างข้อมูลของแต่ละโปรเซสเข้าด้วยกัน โดยขณะที่มีรันคิวลิสต์ทำหน้าที่เชื่อมต่อโปรเซสที่ทำการรันนั้นเข้าด้วยกัน นั่นคือ โปร

เซสที่อยู่ในสถานะ TASK_RUNNING สเตท โดยที่มีโครงสร้างข้อมูลของโปรเซส init_task อยู่บนต้นหัวของลิสต์

แต่ละโครงสร้างข้อมูลของโปรเซสประกอบด้วยฟิลด์ต่าง ๆ ที่สำคัญที่เกี่ยวกับตารางการจัดการโปรเซส ดังนี้

need_reched

เป็นแฟล็กที่ทำการตรวจสอบโดยฟังก์ชัน ret_from_intr() เป็นตัวตัดสินใจว่าจะกลับมาเรียกฟังก์ชัน schedule() หรือไม่ ซึ่งเมื่อภายในรันคิวลิสต์ไม่มีโปรเซสรันแล้วก็จำการ rescheduling ใหม่เพื่อค้นหาโปรเซสเข้ามาทำงานต่อไป

policy

ประกอบด้วยค่าต่าง ๆ ดังต่อไปนี้

SCHED_FIFO เป็นการจัดการโปรเซสของเรียลไทม์โปรเซสแบบมาก่อนได้ก่อน เมื่อซีพียูทำการเลือกโปรเซสเพื่อที่ทำการรัน โปรเซสนั้นจะถูกเก็บมาไว้ที่ส่วนต้นหัวของรันคิวลิสต์ โดยที่โปรเซสทำการครอบครองซีพียูไปเรื่อย ๆ จนกระทั่งมีโปรเซสที่มีค่าไพออริตี้ที่สูงกว่าถูกเลือกเข้ามา แต่ถ้าไม่มีโปรเซสที่มีค่าไพออริตี้ที่สูงกว่าซีพียูจะทำการรันโปรเซสที่อยู่ต้นหัวของคิวไปเรื่อย ๆ

SCHED_RR เป็นการจัดการโปรเซสของเรียลไทม์โปรเซสแบบราวน์โรบินเมื่อซีพียูทำการเลือกโปรเซสนั้นขึ้นมาเพื่อทำการรัน ตารางการจัดการโปรเซสจะทำการย้ายโปรเซสปัจจุบันไปไว้ที่ส่วนท้ายของลิสต์คิว เป็นลักษณะวนรอบสลับกันครอบครองซีพียู การจัดการโปรเซสแบบนี้ยุติธรรมสำหรับทุกโปรเซสเปรียบเสมือนว่าทุกโปรเซสมีค่าไพออริตี้เท่ากันแต่ว่าสลับกันเข้ามาครอบครองซีพียูแบบเวียนเทียนเท่านั้น

SCHED_OTHER เป็นการจัดการโปรเซสของโปรเซสชนิด Conventional process โดยที่มีการแบ่งเวลาให้กับ Conventional process นั้น ๆ

rt_priority

ไพออริตี้ชนิดนี้ใช้กับโปรเซสชนิดเรียลไทม์โปรเซส , Conventional processไม่สามารถใช้กับโปรเซสชนิดนี้ได้

priority

ไพออริตี้ชนิดนี้เป็นค่า base time quantum หรือค่าไพออริตี้พื้นฐานของโปรเซส

counter

คือ จำนวนคล็อกของซีพียูของโปรเซสก่อนที่จะหมด time quantum โดยมันจะลดลงทีละ 1 จนค่า time quantum หมด เมื่อ epoch เริ่มขึ้นค่า counter จะทำการเก็บ time quantum ของแต่ละโปรเซสโดยมีฟังก์ชัน update_process_time ทำการลดค่า counter ทีละหนึ่งเมื่อคล็อกเข้ามา 1 ลูก

หน้าที่ที่สำคัญของฟังก์ชัน goodness()

หัวใจของอัลกอริทึมตารางการจัดการโปรเซส คือ การแบ่งแยกความแตกต่างของแต่ละโปรเซส เพื่อทำให้โปรเซสที่สมควรที่สุดถูกเลือกมาให้ซีพียูรัน ซึ่งการแบ่งชนิดของแต่ละโปรเซสทำโดยผ่านฟังก์ชัน goodness() โดยฟังก์ชันนี้ทำการรับพารามิเตอร์มา 2 ค่า คือ prev (โครงสร้างข้อมูลของโปรเซสที่ทำการรันถัดไป) และ p (โครงสร้างข้อมูลของโปรเซสที่ถูกนำมาคำนวณ) ค่าของ C ที่รีเทรินนั้นรีเทรินโดยฟังก์ชัน goodness() โดยที่ค่าที่ออกมาจากฟังก์ชัน goodness() ของ p สามารถวัดได้ ดังนี้

C = -1000

P ต้องไม่ถูกเลือกโดยตารางการจัดการโปรเซสค่านี้รีเทรินเมื่อรันคิวลิสต์ประกอบด้วย init_task เท่านั้น

C = 0

P นั้นหมดค่า time quantum เว้นแต่ P คือ โปรเซสแรกในรันคิวลิสต์และทุกโปรเซสที่ทำการรันนั้นหมดค่า time quantum โปรเซสนี้ไม่ถูกเลือกในการเอ็กซิคิวชั่น

0<C<1000

P คือ Conventional process ที่ไม่หมดค่า time quantum ค่า C ที่สูงกว่าแสดงว่ามีระดับ level ของฟังก์ชัน goodness() ที่สูงกว่า

C >= 1000

P คือ เรียลไทม์โปรเซสค่า C ที่สูงกว่าแสดงว่ามีระดับ level ของฟังก์ชัน goodness() ที่สูงกว่า

หน้าที่ของฟังก์ชัน **schedule()**

ฟังก์ชัน schedule ทำหน้าที่ในการจัดการโปรเซส โดยมีวัตถุประสงค์หลัก คือ ทำหน้าที่ในการค้นหาโปรเซสเพื่อนำโปรเซสไปเข้าในรันคิวลิสต์และส่งต่อให้ซีพียูรัน

เมื่อโปรเซสนั้นอยู่ในรันคิวลิสต์โปรเซสต้องมีการคำนวนค่า goodness() ตลอดเวลาเมื่อหมดค่า Time quantum โปรเซสก็จะทำการสวิทชิ่งไปยังโปรเซสถัดไปซึ่งและฟังก์ชัน schedule() ก็จะทำการค้นหาโปรเซสเข้ามายังคิวลิสต์อยู่เรื่อยๆ โดยมีโปรเซสเคอร์เรนเป็นโปรเซส

อัลกอริทึมในการจัดการโปรเซส

การจัดการโปรเซสในลีนุกซ์นั้นทำงานโดยแบ่งเวลาของซีพียูออกมาเรียกว่า epoch ในแต่ละ epoch ทุก ๆ โปรเซสจะมีค่า time quantum ของแต่ละโปรเซสซึ่งค่าของเวลานั้นจะถูกคำนวนเมื่อเริ่มการทำ epoch ปกติแล้วโปรเซสที่ทำงานแตกต่างกันจะมีค่า time quantum ที่แตกต่างกัน สำหรับค่า time quantum นั้นค่าเวลาที่มากที่สุดที่ซีพียูแบ่งให้กับแต่ละโปรเซสใน epoch นั้น เมื่อโปรเซสใช้เวลาของ time quantime หมดแล้วโปรเซสทำการสลับการทำงานไปให้โปรเซสอื่นโดยจะถูกแทนที่การทำงานโดยโปร

เซสอื่นที่อยู่ในสถานะรันเนเบิล แต่อย่างไรก็ตามโปรเซสอาจจะถูกเลือกโดยการจัดการโปรเซสได้หลาย time quantum ภายใน epoch เดียว ซึ่งในกรณีนี้ถ้าขณะที่โปรเซสทำงานโดยยังไม่หมด time quantum เช่น ถ้ามันปิดตัวเองลงโดยมันรอ I/O มันจะทำการรักษาค่าของ time quantum ของมันไว้และมันอาจจะถูกเรียกอีกครั้งใน epoch เดิมเพราะมีหลาย time quantumg ค่า epoch จะจบลงก็ต่อเมื่อทุกโปรเซสในสถานะรันเนเบิลหมดค่า time quantum ทั้งหมด ซึ่งถ้าเกิดกรณีนี้การจัดการโปรเซสจะทำการคำนวณค่า time quantum ของทุกโปรเซสใหม่แล้วเริ่ม epoch ใหม่อีกครั้ง

ปฏิบัติ 1. ให้นักศึกษาทำการเปลี่ยนค่า time quantum แล้วทำการวัดค่า TURN AROUND TIME ของแต่ละโปรเซสและดูความแตกต่างของการทรัพยากรโดยใช้คำสั่ง TOP และดูสถานะของโปรเซสโดยใช้คำสั่ง PS

2. ให้นักศึกษาทำการเปลี่ยนตารางการจัดการโปรเซสจาก FIFO และ Round Robin เป็นตารางการจัดการโปรเซสชนิดอื่น เช่น Shortest Job First (SJF) , Multilevel Queue Scheduling หรือเปลี่ยนอัลกอริทึมการจัดการโปรเซสโดยการออกแบบด้วยตัวเองแล้วทำการวัดค่า TURN AROUND TIME ว่าอัลกอริทึมการจัดการโปรเซสแบบใดมีประสิทธิภาพ (Performance) ในงานลักษณะใดและหาความเหมาะสมระหว่างอัลกอริทึมการจัดการโปรเซสกับงานแต่ละงานนั้น

คำถามท้ายการทดลอง

1. อธิบายถึงสถานะ (state) ของโปรเซสแต่ละสถานะเพื่อทำความเข้าใจสถานะของโปรเซสแต่ละสถานะนั้น ๆ
2. ทำไมระบบปฏิบัติการลีนุกซ์จึงใช้อัลกอริทึมการจัดการโปรเซสแบบ FIFO และ Round Robin
3. จงอธิบายถึงข้อดีและข้อเสียของอัลกอริทึมการจัดการโปรเซสในแต่ละชนิด
4. มีอัลกอริทึมใดบ้างที่สามารถนำมาทดแทนตารางการจัดการโปรเซสปัจจุบันได้
5. จงอธิบายถึงลักษณะการใช้งานที่เหมาะสมเมื่อนำอัลกอริทึมในข้อ 4 มาทดแทนอัลกอริทึมการจัดการโปรเซสดั้งเดิม

5.8.10 การเปลี่ยนอัลกอริทึมการจัดการโปรเซสจากคิวเดี่ยวเป็นหลายคิว (Modification of Linux process scheduling algorithm using multi priority queues)

ในโปรเจคนี้ทำการทดสอบตารางการจัดการโปรเซสในแต่ละอัลกอริทึมโดยทำการเปรียบเทียบระหว่างตารางการจัดการโปรเซสอัลกอริทึมใหม่กับอัลกอริทึมดั้งเดิมโดยใช้ส่วนของเรื่องเกี่ยวกับคิวของไพออริตี้เข้ามาเกี่ยวข้องโดยอัลกอริทึมที่ทำขึ้นมาใหม่เป็นชนิดคิวหลายคิว (multil-queue) โดยที่หนึ่งในคิวนั้นทำหน้าที่เซ็ทระดับของค่าไพออริตี้ ซึ่งปัญหาของซิงเกิลคิวนั้นคือมีปัญหาเรื่องของคอขวด (bottleneck)

ส่วนที่สำคัญในการแก้ไขซอร์ดโค้ด

ค่าต่าง ๆ ของไพออริตี้คิว

การแสดงการกำหนดกลุ่มของไพออริตี้คิวในไฟล์ Sched.c มีทั้งหมด 33 คิว

```
#define NUM_PRIO_QUEUES (32)
#define REALTIME_QID    (NUM_PRIO_QUEUES)

static struct list_head prio_queues[NUM_PRIO_QUEUES+1];
int topqid = -1;
```

การกำหนดไพออริตี้คิว (Establishing priority queues)

ตารางการจัดการโปรเซสทำการเลือกโปรเซสที่อยู่ในรันคิวลิสต์ที่มีค่า goodness() สูงที่สุด โดยที่โปรเซสที่มีค่า goodness() สูงที่สุด คือ โปรเซสชนิดเรียลไทม์โปรเซสโดยที่โปรเซสนั้นถูกสั่งโดยฟิลด์ที่ชื่อ rt_priority ทุก ๆ ช่วงเวลาของสัญญาณอินเตอร์รัพท์ค่าเคาน์เตอร์ลดลงทีละหนึ่งจนกระทั่งถึง 0 ก็จะมีการ reschedule เกิดขึ้น งานนั้นมีค่าของเคาว์เตอร์เป็น 0

ค่าของ goodness() สำหรับงานที่ไม่ใช่แบบเรียลไทม์โปรเซสในสถาปัตยกรรมแบบ X86 นั้นค่า goodness() มีค่าอยู่ในช่วงตั้งแต่ 0-64 สำหรับ assignment ของงานนี้กล่าวถึงไพออริตี้คิวซึ่งโปรเซสต่าง ๆ ที่อยู่ในคิวนั้นมีค่าไพออริตี้แยกจากกันของใครของมันแต่มีเรนจ์ที่อยู่ในช่วงเดียวกันโดยที่หมายเลขของไพออริตี้คิวที่เลือกนั้นต้องอยู่ในเรนจ์ของค่า goodness() โดยที่มีฟังก์ชัน function(task_to_qid(struct task_struct *p) ทำหน้าที่ ในการแม็พจากงานแต่ละโปรเซสที่เป็นอินเดกซ์ออกมาให้เป็นกลุ่มของไพออริตี้คิว สำหรับไพออริตี้คิวนั้นมีอยู่ด้วยกัน 2 คิวที่เป็นชนิดพิเศษ

1. qid = REALTIME_QID เป็นไพออริตี้คิวที่มีค่าสูงสุดซึ่งประกอบไปด้วยกลุ่มของโปรเซสที่เป็ยชนิดของเรียลไทม์โปรเซส
2. qid = 0 เป็นไพออริตี้คิวที่มีค่าน้อยที่สุด คือ กลุ่มของโปรเซสที่มีค่าเคาน์เตอร์เป็น 0 และโปรเซสนั้นไม่เคยถูกจัดการโดยตารางการจัดการโปรเซส

เนื่องจากเรามีคิว 32 ระดับเท่านั้นเราต้องกระจายคิวให้ครบทุกระดับ เนื่องจากทุกโปรเซสมีค่าของไพออริตี้ประมาณเรนจ์ 16-32 แต่จำนวนคิวที่มีอยู่นั้นมีทั้งหมด 64 คิว ดังนั้นเราจึงต้องทำการแม็พค่าของ na_goodness ต่อไปนี้ไปที่เรนจ์ดังกล่าวโดย 0 – 0 (ratio), 01..16 – 1..8 (2:1), 17..32 – 9..24 (1:1), 33..63 – 25..31 (4:1) โดยเราใช้คิวเดี่ยวนั้นซึ่งแต่ละโปรเซสต้องผ่านฟังก์ชัน na_goodness เป็นตัวกำหนดเรนจ์ของไพออริตี้คิวต่าง ๆ

ต่อไปนี้เป็นการอิมพลีเมนฟังก์ชัน task_to_qid() ดังต่อไปนี้

```
static int  task_to_qid(struct task_struct *p) {
    int qid;

    // give sufficiently high boost to realtime tasks
    if ((p->policy & ~SCHED_YIELD) != SCHED_OTHER) {
        qid = REALTIME_QID;
        goto out;
    }

    /*na_goodness is based on the number of ticks left.

    *if time slice is over, no other calculation is done.

    */
    qid = p->counter;
    if (!qid)
        goto out;
    qid += 20 - p->nice;

    /* Now translate the na_goodness value to a queue index qid
     *  00     =>  0       ratio
     *  01..16  =>  1..8    2:1
     *  17..32  =>  9..24   1:1
     *  33..63  =>  25..31  4:1
     */

  if (qid < 17) {
      qid = (qid == 0) ? 0 : ((qid > 0) ? ((qid+1)>>1) : 1);
      goto out;
  }
  if (qid > 32) {
      qid = 24 + ((qid & 31) >> 2);
      goto out;
  }
  qid = qid - 8;
```

```
out:
    return qid;
}
```

Schedule()

ส่วนที่ต้องทำการแก้ไขส่วนถัดไป คือ ส่วนของตารางการจัดการโปรเซส , ส่วนที่สำคัญในตารางการจัดการโปรเซสคือส่วนของลาเบล search_range เราต้องกำหนดเรนจ์ของไพออริตี้คิวเพื่อทำการค้นหางานของโปรเซสโดยพิจารณาจากค่าของ goodness() ซึ่งคือ ค่า Cนั่นเอง โดยส่วนเริ่มแรกที่ทำการแก้ไขนั้นเราต้องทำการอัพเดทอินเดกซ์ของ topqid ลงในไพออริตี้คิวที่ถูกกำหนดเป็นไพออริตี้คิวที่มีค่าสูงที่สุด ซึ่งคิวนี้ไม่มีทางที่จะว่างเปล่าเพราะว่าคิวนี้มีโปรเซส current อยู่ ถัดนั้นมาเราต้องกำหนดไพออริตี้คิวที่มีค่าของไพออริตี้คิวที่มีค่าต่ำที่สุดโดยใช้อินเดกซ์ของ botqid เพื่อใช้ในการค้นหา โดยเราต้องกำหนดค่า default ของ botqid ต้องมากกว่า 0

```
still_running_back:

  qid = topqid;
  while (qid > 0) {
      if (! list_empty(&prio_queues[qid])) break;
      qid--;
  }
  topqid = qid;

  botqid = qid-PROC_CHANGE_PENALTY-1;
  if (botqid <= 0) botqid = 1;
```

เนื่องจากงานที่อยู่ในแต่ละไพออริตี้คิวยังไม่ถูกเรียกใช้งาน เราจึงต้องทำการนำไพออริตี้คิวที่อยู่ในเรนจ์ของ [topqid..botqid] ของโปรเซสไปเปรียบเทียบกับค่าของ goodness() ของโปรเซสเคอร์เรนที่อยู่ในตางรางการจัดการโปรเซส ณ. ปัจจุบัน

```
search_range:

while (qid >= botqid) {
  int found_local_task = 0;
  list_for_each(tmp, &prio_queues[qid]) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p, this_cpu)) {
```

```
            int weight;
            weight = goodness(p, this_cpu, prev->active_mm);
            if (weight == 0) {
               tmp = tmp->prev;
               list_del(&p->run_list);
               list_add_tail(&p->run_list,&prio_queues[0]);
               continue;
            }
            if (p->processor == this_cpu)
               found_local_task = 1;
            if (weight > c)
               c = weight, next = p;
         }
      }
      if (found_local_task)
         goto found_local_task_cont;
      qid--;
   }
```

เมื่อเราทำการค้นหาค่าของไพออริตี้คิวในเรนจ์ของ [topqid..botqid] แล้วและไม่พบงานที่นำมารันบนซีพียูเนื่องจากโปรเซสเหล่านี้มีค่า counter = 0 ดังนั้นต้องนำโปรเซส ณ. ปัจจุบันซึ่งเป็นโปรเซส current ไปทำงานแทนและต่อจากนั้นเราต้องทำการค้นหาโปรเซสที่นำมารันใหม่โดยการค้นหาจากคิวล่าง (search down) ลงมาเพื่อที่นำโปรเซสไปให้ซีพียูรันซึ่งในช่วงเวลาปกติเรากำหนดค่าของ search down เป็นค่า qid = 1 สำหรับงานที่มีคุณภาพเราต้องสามารถหาโปรเซสที่เหมาะสมที่สุดนำมาให้ตารางการจัดการโปรเซสนำมาจัดการได้จากขั้นตอนนี้เราทำการกำหนด botqid ใหม่ โดยเราเลือกส่วนของ current path เพื่อทำการลดความยุ่งยากของโค้ดลง ถ้าเราค้นหาไม่พบโปรเซสในครั้งแรกเราต้องทำการค้นหาเป็นครั้งที่สอง ถ้าเราทำการค้นหาครั้งที่สองในช่วงเรนจ์ที่สองแล้วและยังไม่สามารถหาโปรเซสที่เหมาะสมมาให้ซีพียูรันได้ เราต้องมีส่วนที่ทำการตรวจสอบค่าของ counter ของงานทั้งหมดในระบบ ในตารางการจัดการโปรเซสต้องมีส่วนที่จัดการค่าของcounter นั้น เมื่อได้ค่าของ goodness() ในงานทีทำการรันเป็น 0 เช่น เมื่อเราได้ค่าของ counter เป็น 0 สำหรับในการอิมพลีเมนท์นั้นค่าของ counter ของโปรเซสที่เป็น 0 ต้องถูกจัดให้อยู่ภายในคิวของคิวที่มีค่าของไพออริตี้คิวที่ต่ำที่สุด เมื่อโปรเซสทั้งหมดนั้นมาอยู่ในคิวที่มีค่าของไพออริตี้คิวที่ต่ำที่สุดแล้วถัดนั้นมาเราต้องนำโปรเซสเข้าที่ลาเบล recalculat ซึ่งขั้นนี้เป็นขั้นตอนในการทดสอบว่าจะนำมาเข้าโค้ดดังต่อไปนี้หรือไม่

```
if (c < 0) {
   /* we didn't find any suitable task, did we search all the queues ?   */
   if (!tried_all_queues && (botqid > 1)) {
      tried_all_queues = 1;
      botqid = 1;
      goto search_range;
   }
   /* we searched all queues and still didn't find any task to run.
    * now check whether we have to run recalculate
    * if the 0-bucket is non empty then c would be 0
    */
   if (!list_empty(&prio_queues[0]))
      goto recalculate;
}
/* Do we need to re-calculate counters? */
if (!c)
   goto recalculate;

/* lable to continue when search range found task */
found_local_task_cont:
```

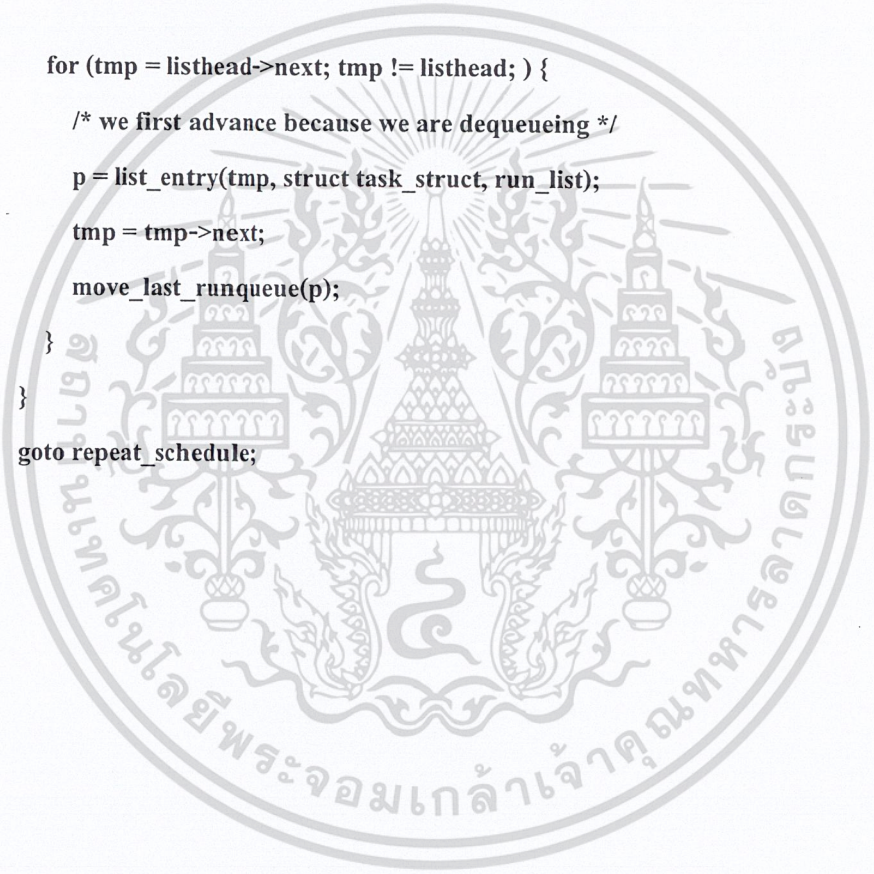
หน้าที่ของลาเบล recalculate นั้นทำการอัพเดทค่าของ counter ของโปรเซสต่าง ๆ โดยที่ไม่มีการทำการล็อกรันคิว สำหรับการล็อกรันคิวกระทำเมื่องานก่อนหน้านี้มีค่า counter เป็น 0 ซึ่งโปรเซสนั้นยังคงอยู่ในคิวของโปรเซสที่มีค่าของไพออริตี้ที่ต่ำสุดต่อไปและโปรเซสนี้ไม่ถูกเรียกโดยตารางการจัดการโปรเซส เราทำการรีคิวทุกโปรเซสในไพออริตี้คิวที่ต่ำที่สุดซึ่งในโปรเซสนั้นจะมีค่าของไพออริตี้คิวใหม่โดยการอัพเดทค่าของ counter และทำการ repeat_schedule ต่อไป

```
recalculate:
   {
      struct task_struct *p;
      struct list_head *listhead = &prio_queues[0];
      spin_unlock_irq(&runqueue_lock);
      read_lock(&tasklist_lock);
```

```
    for_each_task(p)
        p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
    read_unlock(&tasklist_lock);
    spin_lock_irq(&runqueue_lock);

    /* now we hold the lock again...
     * all tasks should have been in the 0-bucket.
     * after reassignment no task should be in the 0-bucket
     * so we run through it and assign a new bucket for each task.
     */

    for (tmp = listhead->next; tmp != listhead; ) {
        /* we first advance because we are dequeueing */
        p = list_entry(tmp, struct task_struct, run_list);
        tmp = tmp->next;
        move_last_runqueue(p);
    }
}
goto repeat_schedule;
```



บทที่ 6
การจัดการหน่วยความจำ

6.1 ความต้องการของหน่วยความจำ (Memory Management Requirement)

รีโลเคชัน (Relocation)

ในระบบ Multiprogramming หน่วยความจำของระบบจะถูกใช้โดยโปรเซสต่างๆ ซึ่งโดยทั่วไปจะไม่ทราบถึงตำแหน่งของหน่วยความจำที่กำหนดโปรเซสนั้นในขณะที่สร้าง Process image หรือการทำการประมวลผล และเมื่อมีการ Swap โปรเซสเข้าและออกจากหน่วยความจำหลัก ตำแหน่งของโปรเซสในหน่วยความจำหลักจะมีการเปลี่ยนแปลง ซึ่งระบบปฏิบัติการจำเป็นต้องทำการ Relocate เพื่อให้การอ้างอิงหน่วยความจำเป็นไปอย่างถูกต้อง

โปรเทคชั่น (Protection)

แต่ละโปรเซสควรจะต้องมีการป้องกันการเข้าถึงจากโปรเซสอื่น ไม่ว่าจะเกิดจากข้อผิดพลาดหรือจงใจให้เกิดขึ้นก็ตามและโปรเซสต่างๆ ที่ทำงานอยู่ในระบบไม่ควรมีการอ้างถึงตำแหน่งหน่วยความจำที่ไม่ใช่ของตนเองโดยไม่ได้รับอนุญาต ซึ่งจากการที่โปรเซสมีการเปลี่ยนตำแหน่งที่อยู่ในหน่วยความจำซึ่งเกิดจากการ Relocation ทำให้การอ้างอิงหน่วยความจำแต่ละครั้งของโปรเซสต้องมีการคำนวณหาตำแหน่งที่ถูกต้องใน Physical Memory เสมอ

โดยทั่วไป โปรเซสจะไม่ได้รับอนุญาตให้เข้าถึงส่วนใดๆ ของระบบปฏิบัติการไม่ว่าจะเป็นโปรแกรมหรือข้อมูล (การติดต่อผ่านระบบปฏิบัติการจะทำผ่าน System Call เท่านั้น) และโปรเซสจะไม่สามารถใช้งานส่วนคำสั่งของโปรเซสอื่นได้ นอกจากนี้ถ้าไม่มีการกำหนดโดยเฉพาะ โปรเซสจะไม่สามารถเข้าถึงข้อมูลของโปรเซสอื่นได้เช่นกัน

การป้องกันหน่วยความจำจะเป็นกลไกในระดับโปรเซสเซอร์ เนื่องจากการอ้างอิงหน่วยความจำจะเกิดขึ้นในขั้นตอนการประมวลผล โดยระบบปฏิบัติการไม่สามารถทราบได้ถึงการอ้างอิงทั้งหมดของโปรเซส

แชร์ริ่ง (Sharing)

กลไกในการป้องกันใดๆ จะต้องมีข้อยืดหยุ่นเพื่อให้โปรเซสสามารถเข้าถึงพื้นที่หน่วยความจำเดียวกันให้ได้ เช่นในกรณีที่โปรเซสหลายๆ มาจากโปรแกรมเดียวกัน ถ้าโปรเซสสามารถใช้ชุดคำสั่งเดียวกันได้ จะทำให้ประหยัดหน่วยความจำมากขึ้น หรือโปรเซสหลายๆ โปรเซสที่ร่วมกันทำงานในงานเดียวกัน อาจจะต้องการเข้าถึงข้อมูลเดียวกัน ระบบจัดการหน่วยความจำจะต้องยอมให้เกิดการใช้งานหน่วยความจำร่วมกันได้ ภายใต้การควบคุมที่กำหนดขึ้น

6.2 การแบ่งหน่วยความจำ (Memory Partitioning)

ในการจัดการหน่วยความจำ การย้ายข้อมูลระหว่างหน่วยความจำมีความจำเป็นต้องแบ่งหน่วยความจำออกเป็นส่วนๆ เพื่อให้การย้ายสามารถทำได้อย่างรวดเร็วตามความจำเป็น แล้วการใช้ใช้หน่วยความจำเป็นไปอย่างมีประสิทธิภาพ

การแบ่งหน่วยความจำแบบคงที่ (Fixed Partitioning)

ทุกๆ รูปแบบของการจัดการหน่วยความจำ จะกำหนดให้ระบบปฏิบัติการอยู่ในพื้นที่หน่วยความจำส่วนหนึ่งคงที่ และพื้นที่ส่วนที่เหลือจะถูกใช้โดยโปรเซสต่างๆ รูปแบบในการจัดการที่ง่ายที่สุดของการจัดแบ่งหน่วยความจำคือการกำหนดพื้นที่ส่วนต่างๆ ของหน่วยความจำให้มีขนาดคงที่ โดยอาจจะเท่ากันหรือไม่เท่ากันก็ได้

ในกรณีที่มีขนาดของพาร์ทิชันเท่ากัน โปรเซสที่มีขนาดเล็กกว่าหรือเท่ากับขนาดของพาร์ทิชันจะสามารถโหลดเข้าสู่พาทิชันที่ว่างได้ ถ้าไม่มีพาร์ทิชันว่างและไม่มีโปรเซสใดที่อยู่ในอยู่ในสถานะ Ready หรือ Running ระบบสามารถ Swap โปรเซส ใดโปรเซสหนึ่งออกจากพาร์ทิชันได้ เพื่อที่จะสร้างที่ว่างให้โปรเซสอื่นที่พร้อมจะประมวลผล สามารถโหลดเข้าสู่หน่วยความจำหลัก และทำการประมวลผลต่อไป

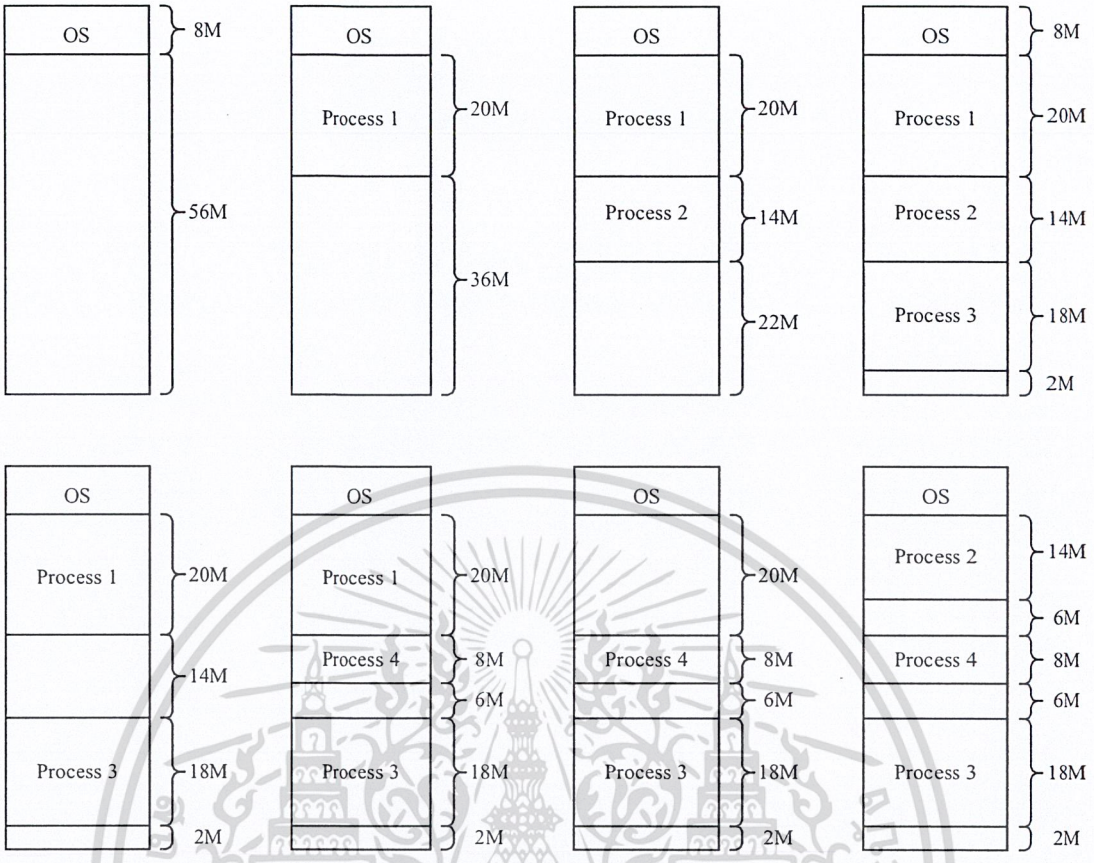
รูปแบบการแบ่งพาร์ทิชันแบบขนาดเท่ากันมีข้อเสียคือ

- โปรแกรมที่มีขนาดใหญ่กว่าพาร์ทิชันจะต้องถูกแบ่งโมดูลของโปรแกรมบางส่วนออกเป็นส่วนย่อยๆ ซึ่งจะทำการโหลดเข้าสู่หน่วยความจำหลักเมื่อต้องการใช้งาน
- ในแต่ละพาร์ติชันจะถูกใช้งานโดยโปรแกรมเดียวเท่านั้น ซึ่งถ้าขนาดของโปรแกรมเล็กกว่าขนาดของพาร์ทิชัน จะทำให้เกิดที่ว่างที่ไม่ถูกใช้งานในพาร์ทิชัน (Internal Fragmentation) ซึ่งถือว่าเป็นการใช้งานหน่วยความจำที่ไม่มีประสิทธิภาพ

การแบ่งหน่วยความจำแบบปรับเปลี่ยนได้ (Dynamic Partitioning)

การจัดการแบ่งพาร์ทิชันแบบไดนามิคได้มีการคิดค้นขึ้นเพื่อแก้ไขข้อบกพร่องของการจัดแบ่งพาร์ทิชั่นแบบคงที่ โดยขนาดของพาร์ทิชั่นจะขึ้นอยู่กับขนาดของโปรเซสที่ถูกโหลดเข้าสู่หน่วยความจำซึ่งสามารถเปลี่ยนแปลงได้ตามโปรเซสที่เข้ามาทำงานในระบบ ซึ่งจะทำให้การใช้งานหน่วยความจำมีประสิทธิมากยิ่งขึ้น

เช่นในระบบหนึ่งมีขนาดหน่วยความจำหลัก 64 เมกกะไบต์ และมีโปรเซส 4 โปรเซสทำงานอยู่ในระบบโดย Process 1 มีขนาด 20 Mbyte, Process 2 มีขนาด 14 Mbyte, Process 3 มีขนาด 18 Mbtye และ Process 4 ขนาด 8 Mbyte มีการสลับเข้ามายังหน่วยความจำดังรูป



รูปที่ 6-1 แสดงตัวอย่างการจัดสรรเมมโมรี่ของแต่ละโปรเซส

จากโปรเซสข้างต้น จะสังเกตว่าเมื่อเกิดการสลับโปรเซสเข้าออกระหว่างหน่วยความจำหลักกับหน่วยความจำสำรองไปเรื่อยๆ จะเกิดพาทิชันขนาดเล็กจำนวนมาก ซึ่งไม่สามารถใช้ในการบรรจุโปรเซสได้ เรียกว่า Externel Fragmentation ซึ่งในการแก้ปัญหานี้ จำเป็นที่จะต้องมีการรวมพาทิชันเล็กๆ เหล่านี้เข้าด้วยกัน เรียกว่าการ Compaction โดยทำการย้ายโปรเซสที่อยู่ในหน่วยความจำหลักใหม่ ให้อยู่ต่อเนื่องกัน ซึ่งจะทำให้เกิดพาทิชันว่างพาทิชันเดียว

6.3 ตำแหน่งของหน่วยความจำ (Memory Addresses)

ในอินเทล 80x86 ไมโครโปรเซสเซอร์จะแบ่งแอดเดรสออกได้เป็น 3 ชนิด

ลอจอคอลแอดเดรส (Logical address)

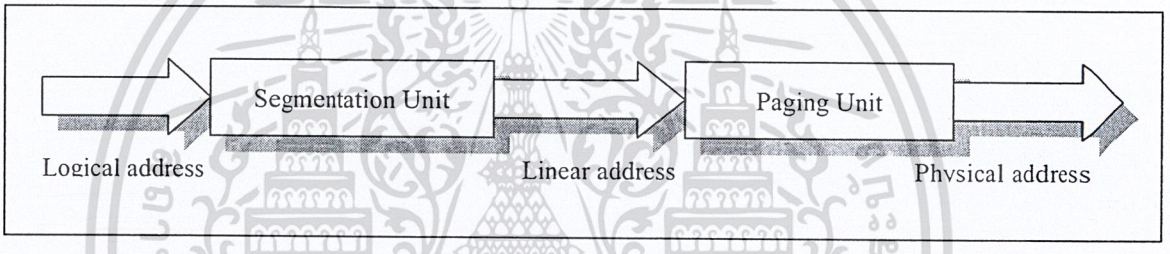
เป็นตำแหน่งของ ตัวแปรหรือคำสั่ง จะถูกแบ่งออกเป็นสองส่วน คือ Segment Identifier และ ออฟเซต (Offset) หรือเรียกอีกอย่างว่า Displacement ซึ่งจะแสดงระยะทางจากจุดเริ่มต้นของ Segment ถึงตำแหน่งที่แท้จริง

ลิเนียร์แอดเดรส (Linear address)

จะมีขนาด 32 bit โดยไม่คิดเครื่องหมาย ซึ่งจะสามารถใช้ตำแหน่งได้มากถึง 4 กิกะไบต์ ลิเนียร์แอดเดรส (Linear Address) โดยปกติจะแสดงในรูปของเลขฐาน 16 ดังนั้นจะมีค่าระหว่างจะ 0x00000000 ถึง 0xffffffff

ฟิสิคอลแอดเดรส (Physical address)

จะมีขนาด 32 bit โดยไม่คิดเครื่องหมาย จะเป็นตำแหน่งที่ใช้อ้างไปในหน่วยความจำจริงๆ ที่มีอยู่



รูปที่ 6-2 แสดง การเปลี่ยนลอจิคอลแอดเดรส เป็นฟิซิคอลแอดเดรส

ซีพียูจะเป็นตัวควบคุม การเปลี่ยนจากลอจิคอลแอดเดรส (logical address) ไปยังลิเนียร์แอดเดรส (linear address) โดยใช้ Hardware circuit ซึ่งเรียกว่า Segmentation Unit และในส่วนถัดมา คือ Paging Unit จะเป็นตัวเปลี่ยนจาก Linear Address เป็น Physical Address

6.4 เซกต์เมนในฮาร์ดแวร์ (Segment in Hardware)

รูปแบบของ 80386 อินเทลไมโครโปรเซสเซอร์ แสดงดังต่อไปนี้

เซกเมนต์รีจีสเตอร์ (Segmentation Registers)

จะอ้างถึง Logical Address ซึ่งแบ่งออกเป็นสองส่วน คือ ส่วนของ Segment Indentifiier ซึ่งมีขนาด 16 บิต ส่วนนี้จะเรียกว่า Segment Selector ส่วนต่อมาคือส่วนของ Offset จะมีขนาด 32 bit

มันจะทำให้ง่ายและเร็ว ในการนำ Segment Selector กลับมา Processor จึงจัดให้มี Segment register สำหรับ Segment Selector แต่ละตัว ซึ่ง Register เหล่านี้ เรียกว่า cs, ss, ds, es, fs และ gs แม้จะมีถึง 6 ตัว แต่ก็ต่างวัตถุประสงค์กัน 3 จาก 6 มีรายละเอียดต่อไปนี้

CS (code segment register)	จะชี้ไปยัง Segment ที่ประกอบไปด้วย คำสั่งของ Program
SS (stack segment register)	จะชี้ไปยัง Segment ที่ประกอบไปด้วย Stack ของ Program ในขณะนั้น
DS (data segment register)	จะชี้ไปยัง Segment ที่ประกอบไปด้วย ข้อมูลต่างๆ

CS register จะมีอยู่ 2 bit ที่ใช้กำหนดสิทธิพิเศษของ CPU ค่า 0 จะแสดงถึงความพิเศษมากสุด และค่า 3 จะแสดงถึงความพิเศษน้องสุดลีนุกซ์จะใช้ 0 และ 3 เป็นตัวกำหนด Kernel mode หรือ User mode ตามลำดับ

เซกเมนต์เดสคลิบเตอร์ (Segment Descriptors)

จะมีขนาด 8 byte หรือเท่ากับ 64 bit ซึ่งจะเป็นตัวอธิบายคุณสมบัติของแต่ละ Segment แต่ละ Segment Descriptors นี้จะถูกเก็บอยู่ใน Global Descriptor Table (GDT) หรือ Local Descriptor Table (LDT) โดยปกติ GDT จะถูกกำหนดขึ้นมาอันเดียว แต่ละ Process ก็จะมี LDT เป็นของตัวเอง ซึ่ง GDT นี้จะบรรจุอยู่ในรีจิสเตอร์ gdtr ส่วนของ LDT นี้จะเก็บอยู่ในรีจิสเตอร์ ltdr

Data Segment Descriptor

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

BASE (24 - 31)	G	B	0	AVL	Limit (16-19)	1	dpl	s=1	Type	BASE (16 - 23)	
BASE (0 - 15)						LIMIT (0 -15)					

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Code Segment Descriptor

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

BASE (24 - 31)	G	B	0	AVL	Limit (16-19)	1	dpl	s=1	Type	BASE (16 - 23)	
BASE (0 - 15)						LIMIT (0 -15)					

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

System Segment Descriptor

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

BASE (24 - 31)	G		0		Limit (16-19)	1	dpl	s=1	Type	BASE (16 - 23)
BASE (0 - 15)						LIMIT (0 -15)				

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Segment Descriptor จะประกอบไปด้วยส่วนต่างๆ ดังนี้

- จะมีขนาด 32 bit จะบรรจุลิเนียร์แอดเดรส (linear address) ของตำแหน่งแรก ของเซกเมนต์
- Flag G ถ้าเคลียร์จะแสดงว่าขนาดของเซกเมนต์มีขนาด 1 ไบต์แต่ถ้าไม่ใช่ จะมีขนาด 4096 ไบต์
- 20 bit Limit field จะแสดงระยะของแต่ละเซกเมนต์ถ้าแฟลก G เท่ากับ 0 จะมีขนาด 1 ไบต์ ถึง 1 MB แต่ถ้าไม่จะมีขนาด 4 KB ถึง 4 GB
- S flag ถ้า clear จะเป็น System segment แต่ถ้าไม่จะเป็น Normal code หรือ Data segment
- 4 bit Type Field จะแสดงลักษณะ ของ Segment

  Code Segment Descriptor

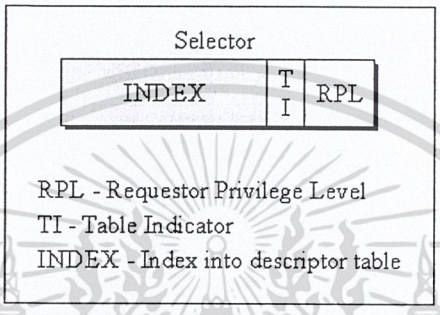
  Data Segment Descriptor

  Task State Segment Descriptor (TSSD)

  Local Descriptor Table Descriptor (LDTD)
- DPL 2 bit (Descriptor Private Level) จะเป็นบิตที่ใช้สำหรับจำกัดการเข้าถึง Segment
- Segment-Present flag แฟลกนี้ถ้าถูกเซตเป็น 0 หมายถึง segment ไม่ได้อยู่ใน Main Memory ลีนุกซ์จะเซตส่วนนี้เป็น 1 เสมอ ถ้ามันไม่เคยถูก Swap ลง Disk
- Additional Flag จะเรียก D หรือ G ขึ้นอยู่กับว่า Segment จะบรรจุโค้ดหรือข้อมูลถ้าเซต จะใช้ 32 bit Long แต่ถ้า Clear จะใช้ 16 bit Long
- Bit 53 จะสงวนไว้ โดยปกติจะเซตเป็น 0
- AVL Flag จะถูกใช้ใน Operating System แต่จะไม่ใช้ในลีนุกซ์

เซกเมนต์ซีเลกเตอร์ (Segment Selectors)

เพื่อความเร็วในการเปลี่ยนจาก Nonprogrammable Register ซึ่งรีจิสเตอร์ ตัวนี้ไม่สามารถเซตได้โดยโปรแกรมเมอร์ เมื่อ Segment Selector logical address ไปยัง linear address ทาง Intel Processor จึงเพิ่มรีจิสเตอร์เข้าไปอีกตัวหนึ่ง ซึ่งเรียกว่า ถูกโหลดลงใน Segment Registor ซึ่งจะสอดคล้องกับที่ Segment Descriptor โหลดจาก Memory ไปเทียบกับ Nonprogrammable Register ถ้าตรงกัน การเปลี่ยนจาก logical Address เป็น linear address สามารถอ้างไปยัง Segment ที่ต้องการได้ โดยปราศจากการเรียกไปยัง GDT หรือ LDT ที่เก็บอยู่ใน Main Memory

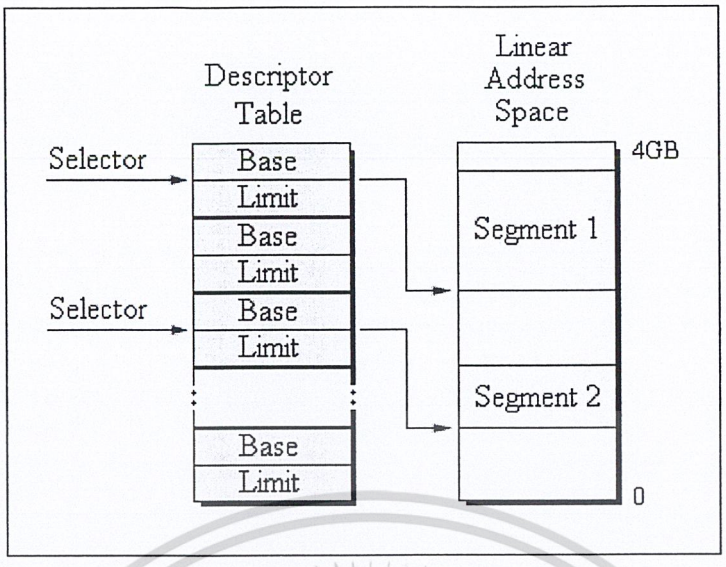


รูปที่ 6-3 แสดง รายละเอียดยดของเซกเมนต์รีจิสเตอร์ (Segment Registor)

- 13 bit index ตำแหน่งของ Descriptor table
- Ti (Table Indicator) flag รายละเอียดของ Segment Descriptor ว่าอยู่ใน GDT (TI = 0) หรือ ใน LDT (TI = 1)
- RPL (Requestor Privilege Level) จะเป็นตัวกำหนดระดับสิทธิพิเศษของซีพียู (CPU) มันจะสอดคล้องกับ Segment Selector ที่โหลดเข้ามาใน CS รีจิสเตอร์

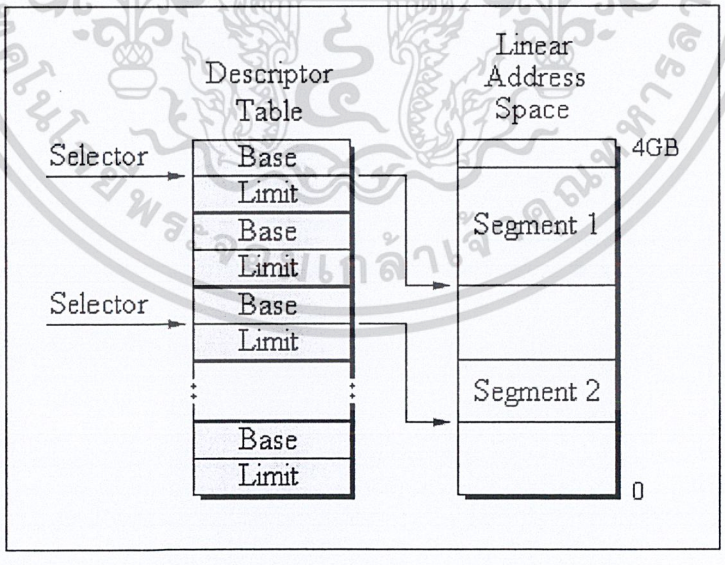
6.5 เซกเมนต์ในลีนุกส์ (Segment in Linux)

แต่ละ Segment Selector จะเป็นตัวชี้ไปยัง Descriptor table ซึ่ง Descriptor จะเป็นตัวบอกตำแหน่งที่ตั้งและขนาดของ Segment ที่อยู่ใน Linear Address Space ความสัมพันธ์ระหว่าง Segment Selector กับ Segment จะมีความสัมพันธ์แบบหนึ่งต่อหนึ่ง ดังรูป



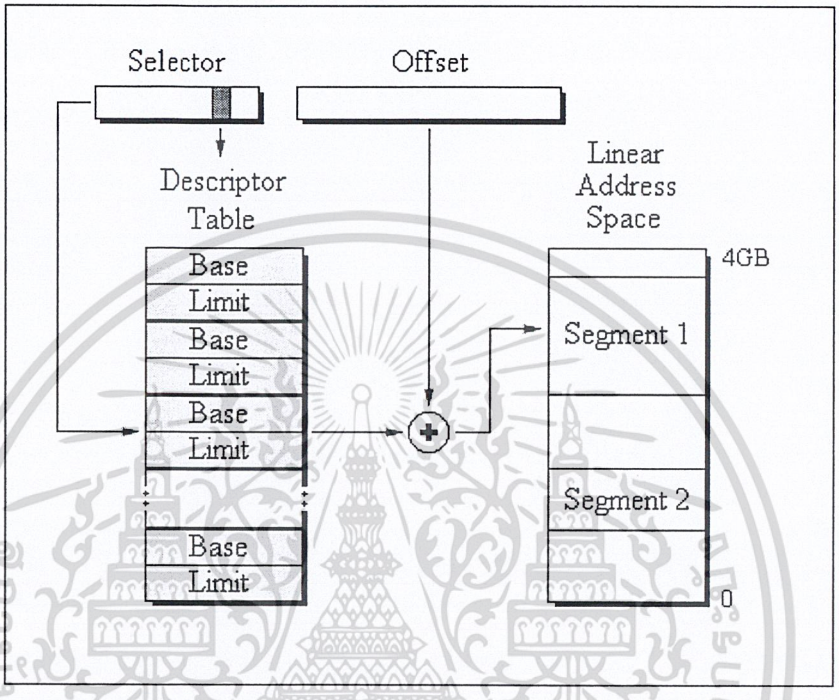
รูปที่ 6-4 แสดงการใช้ Selector ชี้ไปยัง Segment

Linear Address คำนวณได้โดย จะใช้ Segment Selector เป็นตัวชี้ไปยัง Descriptor Table เพื่อรับที่จะรับ ตำแหน่งที่ตั้งของ Segment และจะบวกกับ Offset ก็จะได้ตำแหน่งของ Linear Address ที่ต้องการ ดังรูป



รูปที่ 6-5 แสดงการหาตำแหน่งใน Segment โดยบวกกับค่า Offset

สำหรับการกำหนด Descriptor Table นั้น ส่วนของ Addressing ใน Processor จะใช้ บิต TI เป็นตัวตัดสินใจว่าจะเลือกใช้ Descriptor Table ตัวไหน ระหว่าง Global Descriptor Table (GDT) หรือ Local Descriptor Table (LDT) ซึ่งจะมี Register อยู่สองตัวซึ่งเป็นตัวเก็บตำแหน่งที่ตั้งของ Descriptor Table โดย GDTR จะเป็นตัวเก็บตำแหน่งที่ตั้งของ Global Descriptor Table และ LDTR จะเป็นตำแหน่งที่ตั้งของ Local Descriptor Table ของ Linear Address Space ดังรูป



รูปที่ 6-6 แสดงการใช้ Ti ในการเลือก Descriptor Table

ต่อไปคือ Segment ที่ใช้ใน Linux

- Kernel Code Segment จะเป็นส่วนของ Segment Descriptor ที่อยู่ใน Golbal Descriptor Table (GDT) จะมีค่าดังต่อไปนี้
  - Base = 0x00000000
  - Limit = 0xfffff
  - G (Granularity Flag) = 1
  - S (System Flag) = 1
  - Type = 0xa , สำหรับ Code Segment สามารถอ่านและ Executed
  - DPL (Descriptor Privilege Level) =0 สำหรับ Kernel mode
  - D/B (32 bit Address Flag) = 1

  Segment Selector จะถูกกำหนดด้วย Macro ที่ชื่อว่า __KERNEL_CS

- Kernel Data Segment จะเป็นส่วนของ Segment Descriptor ที่อยู่ใน Golbal Descriptor Table (GDT) จะมีค่าดังต่อไปนี้
  - Base = 0x00000000
  - Limit = 0xfffff
  - G (Granularity Flag) = 1
  - S (System Flag) = 1
  - Type = 2 , สำหรับ Data Segment สามารถอ่านและเขียน
  - DPL (Descriptor Privilege Level) =0 สำหรับ Kernel mode
  - D/B (32 bit Address Flag) = 1

  Segment Selector จะถูกกำหนดด้วย Macro ที่ชื่อว่า __KERNEL_DS
- User Code Segment จะเป็นส่วนของ Segment Descriptor ที่อยู่ใน Golbal Descriptor Table (GDT) จะมีค่าดังต่อไปนี้
  - Base = 0x00000000
  - Limit = 0xfffff
  - G (Granularity Flag) = 1
  - S (System Flag) = 1
  - Type = 0xa , สำหรับ Code Segment สามารถอ่านและExecuted
  - DPL (Descriptor Privilege Level) =3 สำหรับ User mode
  - D/B (32 bit Address Flag) = 1

  Segment Selector จะถูกกำหนดด้วย Macro ที่ชื่อว่า __USER_CS
- User Data Segment จะเป็นส่วนของ Segment Descriptor ที่อยู่ใน Golbal Descriptor Table (GDT) จะมีค่าดังต่อไปนี้
  - Base = 0x00000000
  - Limit = 0xfffff
  - G (Granularity Flag) = 1
  - S (System Flag) = 1
  - Type = 2 , สำหรับ Data Segment สามารถอ่านและเขียน
  - DPL (Descriptor Privilege Level) =3 สำหรับ User mode
  - D/B (32 bit Address Flag) = 1

Segment Selector จะถูกกำหนดด้วย Macro ที่ชื่อว่า __USER_DS

- Task State Segment (TSS) จะเป็น Segment สำหรับแต่ละโปรเซส Segment Descriptor ของ Segment เหล่านี้จะเก็บอยู่ใน GDT ที่ตั้งของ TSS Descriptor จะสัมพันธ์กันกับแต่ละ Process ซึ่งประกอบด้วย Address ของ TSS ซึ่งสอดคล้องกับ Process Descriptor Flag G จะถูกเซตให้ Clear

และ Limit จะถูกเซตให้เป็น 0xeb ดังนั้น TSS Segment จะมีความยาว 236 bytes ส่วน Type จะถูกกำหนดเป็น 9 หรือ 11 และ DPL จะถูกเซตให้เป็น 0 ดังนั้น Process ที่อยู่ใน User Mode จะไม่สามารถเข้าใช้ใช้งานได้

- Default LDT Segment โดยปกติจะถูก Share โดยทุกโปรเซส ซึ่งแต่ละโปรเซสจะมี LTD Segment Descriptor เป็นของตัวเอง ที่ซึ่งจะเป็นตัวบอกไปยัง Default LTD Segment ดังนั้นแต่ละโปรเซส GDT จะประกอบด้วย 2 Segment Descriptor ที่ต่างกัน อันแรกคือ TSS Segment และ อีกอันคือ LDT Segment TSS และ LDT Descriptor ของแต่ละโปรเซสจะถูกเพิ่มเข้าไปใน GDT เมื่อโปรเซสนั้นถูกสร้างขึ้นมา

ทรานซะเลชันลุกอะไซด์บัฟเฟอร์ (Translation Lookaside Buffers)

ใน Intel 80x86 processors จะมีแคช (cache) อื่นอีกตัวหนึ่งซึ่งเรียกว่า Translation Lookaside buffers หรือ TLB เป็นตัวที่เพิ่มความเร็วในการ Linear address ไปเป็น Physical Address หลักการคือ เมื่อ Linear Address ถูกใช้เป็นครั้งแรก ในการคำนวณที่จะอ้างไปถึง Page Table ที่อยู่ใน Ram ก็จะช้ามาก จึงมีการเก็บ Physical Address ไว้ใน TLB ดังนั้นเมื่อมีการอ้างถึง Linear Address ตำแหน่งเดิม ก็จะสามารถอ้างได้อย่างรวดเร็วขึ้น

เรกกูล่าร์เพจจิ่ง (Regular Paging)

ใน i80386 ส่วนของ Paging Unit ของ Intel Processors จะกำหนดให้แต่ละ Page มีขนาดเท่ากับ 4 KB จาก 32 bit ของ Linear Address จะถูกแบ่งออกเป็น 3 ส่วนดังนี้

- Page Directory มีขนาด 10 bit
- Page Table มีขนาด 10 bit
- Offset มีขนาด 12 bit

การเปลี่ยน Linear Address เป็น Physical Address สามารถทำได้โดย หาตำแหน่งเริ่มต้นของ Page Directory จะเก็บอยู่ใน Register cr3 Page Directory จะถูกคำนวณเพื่อหาตำแหน่งเริ่มต้นของ Page Table Page Table จะประกอบไปด้วย Physical Address ของ Page Frame Offset จะเป็นตัวคำนวณหาตำแหน่งที่อยู่ใน Page Frame

ทรีเลเวลเพจจิ่ง (Three-Level Paging)

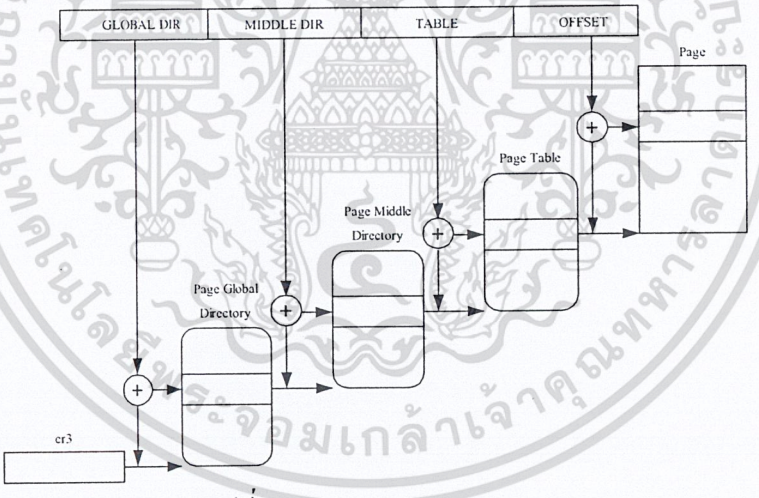
Two-Level Paging จะใช้โดย Microprocessors ขนาด 32 บิต แต่ในปัจจุบัน Processor ได้พัฒนามาเป็น 64 บิต จึงทำให้ Two-Level Paging มีความยาวไม่พอจึงจำเป็นที่จะต้องขยายมาเป็น Three-Level Paging

เพจจิ่งในลีนุกส์ (Paging in Linux)

Linux จะนำรูปแบบของ Three-Level Paging มาใช้บนสถาปัตยกรรมแบบ 64 บิต ซึ่งรูปแบบของ Paging Table จะกำหนดเป็น 3 ชนิดต่อไปนี้

- Page Global Directory
- Page Middle Directory
- Page Table

Page Global Directory จะประกอบไปด้วย Page Middle Directory หลายอัน และ Page Table Directory ก็จะประกอบไปด้วย Page Table หลายอันเช่นกัน แต่ละ Page Table ก็จะอ้างไปยัง Page Frame ที่มันชี้อยู่ Linear Address จะถูกแบ่งออกเป็น 4 ส่วน จากรูปจะไม่แสดงขนาดเพราะขึ้นอยู่กับสถาปัตยกรรมของ Computer



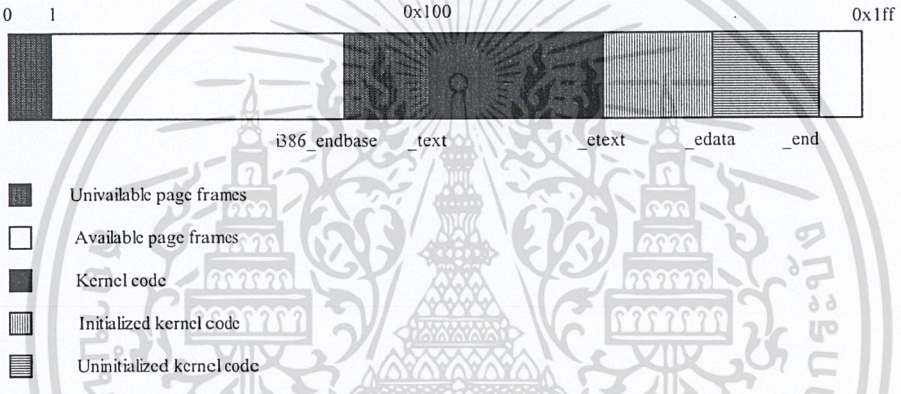
รูปที่ 6-7 Linux Paging Model

รีเวอร์สเพจแฟรม (Reserved Page Frame)

ในส่วนของเคอร์เนลโค้ดและโครงสร้างข้อมูลจะถูกจองไว้ ใน Page Frame ตามกฎโดยทั่วไป ลีนุกซ์เคอร์เนลจะถูกติดตั้งไว้ในแรม เริ่มจาก 0x00100000 ของ Physical Address จากจำนวน 2 M จำนวน Page Frame ที่ต้องการจะขึ้นอยู่กับเคอร์เนลเป็นตัวจัดการ

เคอร์เนลไม่สามารถโหลดลงไปเริ่มที่เมกกะไบต์แรกของแรมได้เพราะ

- Page Frame ที่ 0 จะใช้โดย BIOS ซึ่งจะเป็นตัวเก็บการปรับแต่งของฮาร์ดแวร์
- Physical Address ในช่วงระหว่าง 0x000a0000 ถึง 0x000fffff จะจองโดย Bios และจะเป็นตัว map ให้เข้ากับหน่วยความจำภายในของ การ์ดแสดงผล ISA
- ในเมกกะไบต์แรกบางที่ก็จะถูกจองไว้ โดยรูปแบบของ Computer



รูปที่ 6-8 แสดงการจองหน่วยความจำของเคอร์เนล

บทที่ 7
บทสรุปและวิจารณ์

7.1 บทสรุปและวิจารณ์

การเข้าไปเปลี่ยนแปลงแก้ไขซอร์ดโค้ดในระบบปฏิบัติการลีนุกซ์นั้นเป็นเรื่องที่ควรทำสำหรับผู้ที่ต้องการที่ศึกษาในเรื่องของระบบปฏิบัติการ เพราะการที่เราต้องศึกษาระบบปฏิบัติการอย่างแท้จริงนั้นจำเป็นต้องเข้าใจถึงโครงสร้างภายในซึ่งภายในวิทยานิพนธ์เล่มนี้นั้นประกอบไปด้วยเนื้อหาหลัก ๆ เพียงแค่น้อยนิดเท่านั้นถ้าเทียบกับส่วนของระบบปฏิบัติการลีนุกซ์ทั้งหมด ส่วนที่สำคัญที่สุดของวิทยานิพนธ์เล่มนี้ คือ การเข้าไปแก้ไขไฟล์ที่สำคัญที่สุดของระบบปฏิบัติการลีนุกซ์ คือ ไฟล์ sched.c ซึ่งเป็นไฟล์ที่รวมโครงสร้างภายในของระบบปฏิบัติการลีนุกซ์ การเข้าไปแก้ไขไฟล์นี้จำเป็นต้องมีความรู้ในโครงสร้างของไฟล์เป็นอย่างดี เพราะซอร์ดโค้ดทุกบรรทัดในระบบปฏิบัติการลีนุกซ์มีความสำคัญเป็นอย่างยิ่ง เพราะไม่ว่าเราจะเข้าไปแก้ซอร์ดโค้ดส่วนใดย่อมมีผลกับระบบปฏิบัติการลีนุกซ์เวอร์ชันใหม่ที่เราแก้ทั้งสิ้น ซึ่งการที่เราจะเข้าไปแก้ไขตามวิทยานิพนธ์เล่มนี้จำเป็นต้องมีความรู้ในส่วนต่าง ๆ แยกตามหัวข้อดังต่อไปนี้

1. **การคอมไพล์เคอร์เนลและการดีบักโปรแกรมด้วย GDB**

การที่เราเข้าไปแก้ไขซอร์ดโค้ดในระบบปฏิบัติการลีนุกซ์จุดประสงค์ของเรา คือ เพื่อเข้าไปแก้ไขอัลกอริทึมต่าง ๆ ที่เราไม่ต้องการให้เหมือนเดิมเพื่อให้เหมาะกับการใช้งานตามที่เราต้องการ ดังนั้นการคอมไพล์เคอร์เนลจึงมีความจำเป็น โดยระบบปฏิบัติการลีนุกซ์แต่ละบริษัทได้สร้างไฟล์ Make file ไว้ให้แล้ว

2. **ซิสเต็มคอล (Systam Call)**

การที่เราเข้าไปทำความเข้าใจในส่วนนี้ เพื่อที่จะนำค่าต่างๆ ที่ต้องการที่อยู่ภายในเคอร์เนลออกมาใช้งาน ซิสเต็มคอลเป็นหนทางที่จะทำให้เราสามารถทราบค่าต่างๆ ได้ วิทยานิพนธ์เล่มนี้เป็นทดลองเพิ่มซิสเต็มคอลขึ้นมาใช้งาน

3. **การเปลี่ยนอัลกอริทึมการจัดการโปรเซส**

อัลกอริทึมการจัดการโปรเซสนั้นมีหลายอัลกอรทึม วิทยานิพนธ์เล่มนี้ได้เปลี่ยนอัลกอริทึมจากของเดิมที่ใช้คิวเดี่ยวเป็นการใช้คิวหลายคิว เนื่องจากข้อเสียของคิวเดิม คือ การเกิดปัญหาคอขวด บางครั้งโปรเซสรอนานเกินไป เมื่อใช้คิวแบบหลายคิวทำให้โปรเซสสามารถรันในส่วนของยูสเซอร์โหมดได้มากกว่าแบบคิวเดี่ยวซึ่งแสดงถึงประสิทธิภาพที่ดีกว่าของคิวเดี่ยว

7.2 ปัญหาที่เกิดขึ้น

1. ข้อมูลที่เกี่ยวโครงสร้างระบบปฏิบัติการลีนุกซ์นั้นมีน้อยโดยส่วนใหญ่เป็นวิธีการใช้งาน ส่วนในระดับของเคอร์เนลมีน้อย นอกจากนั้นข้อมูลที่เกี่ยวกับการคอมไพล์เคอร์เนลก็ยังมีน้อย ทำให้มีปัญหาเรื่องของเคอร์เนลพังบ่อย

2. การศึกษาในส่วนของซอร์ดโค้ดทำได้ยากเพราะมีมากมายผู้ที่เข้าไปแก้ไขจำเป็นต้องทราบถึงหน้าที่การทำงานของแต่ละไฟล์และในขณะที่คอมไพล์นั้นมีการลิงค์ระหว่างไฟล์และมีการทำอ็อบเจ็คไฟล์ทำให้การดีบักนั้นทำได้ยากเพราะไม่สามารถเข้าถึงตัวแปรที่สำคัญของไฟล์ที่ต้องการทำการแก้ไขได้ นอกจากนั้นยังมีปัญหาของเรื่องการใช้งานเวอร์ชันของเคอร์เนลของลีนุกซ์ซึ่งแต่ละเวอร์ชันก็เปลี่ยนแปลงอยู่ตลอดเวลาซึ่ง ณ. ปัจจุบัน

3. ปัญหาในการคอนฟิกเคอร์เนล การคอมไพล์เคอร์เนลทุกครั้งต้องมีการคอนฟิกเคอร์เนลเพื่อที่ระบบปฏิบัติการลีนุกซ์นั้นสามารถใช้อุปกรณ์ต่าง ๆ ได้อย่างถูกต้องแต่ถ้ามีการเลือกอุปกรณ์ผิดซึ่งเกิดในกรณีผู้ที่ไม่เคยทำการคอมไพล์เคอร์เนล ซึ่งถ้าเกิดกรณีเลือกผิดขึ้นมาเคอร์เนลที่เราทำการคอมไพล์จะไม่สามารถใช้งานได้ตามที่เราต้องการ

7.3 วิธีการแก้ไข

การแก้ไขสามารถแยกเป็นส่วน ๆ ตามเรื่องที่เกิดปัญหาได้ดังนี้

1. ข้อมูลในประเทศไทยมีน้อยต้องหาข้อมูลจากเว็บไซต์ต่างประเทศ
2. ก่อนที่ทำการแก้ไขซอร์ดโค้ดควรศึกษาสภาพแวดล้อมของไฟล์นั้นก่อน
3. ทำการก็อบปี้คอนฟิกตัวเก่าไปไว้ในไดเรกทอรี่ที่ทำการคอมไพล์เคอร์เนลตัวใหม่

7.4 แนวทางในการพัฒนา

1.เนื่องจากในระบบปฏิบัติการลีนุกซ์มีส่วนที่ต้องทำการแก้ไขอยู่มากมายซึ่งในปัจจุบันโปรแกรมเมอร์ทั่วโลกร่วมมือกันแก้ไขระบบปฏิบัติการลีนุกซ์ให้มีประสิทธิภาพมากขึ้น ซึ่งวิทยานิพนธ์เล่มนี้ได้เข้าไปแก้ส่วนของซิสเต็มคอล (Syatam Call) กับส่วนของตารางการจัดการโปรเซสเท่านั้นยังมีส่วนที่ต้องทำการแก้ไขอีกมาก เช่น ส่วนของดีไวซ์ไดรเวอร์ต่าง ๆ

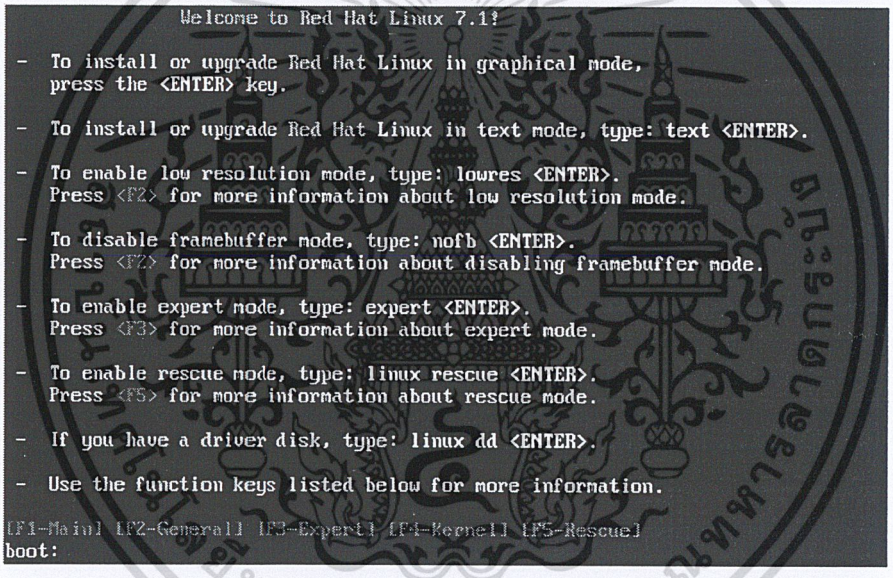
2. ในส่วนของตารางการจัดการโปรเซสนั้นมีส่วนที่สามารถเข้าไปแก้ไขได้อีกเพราะวิทยานิพนธ์เล่มนี้ได้ทำการแก้ไขเพียงแค่ส่วนของไพออริตี้คิวเท่านั้น ซึ่งความเหมาะสมของตารางการจัดการโปรเซสนั้นไม่มีโปรเซสใดที่ดีที่สุดทุกอัลกอริทึมมีข้อดีข้อเสียแตกต่างกันไป โปรแกรมเมอร์สามารถเข้ามาแก้ไขให้ตรงกับลักษณะการใช้งานได้

ภาคผนวก ก.
วิธีการในการติดตั้งลีนุกซ์ Redhat 7.1

การติดตั้งลีนุกซ์ Redhat 7.1

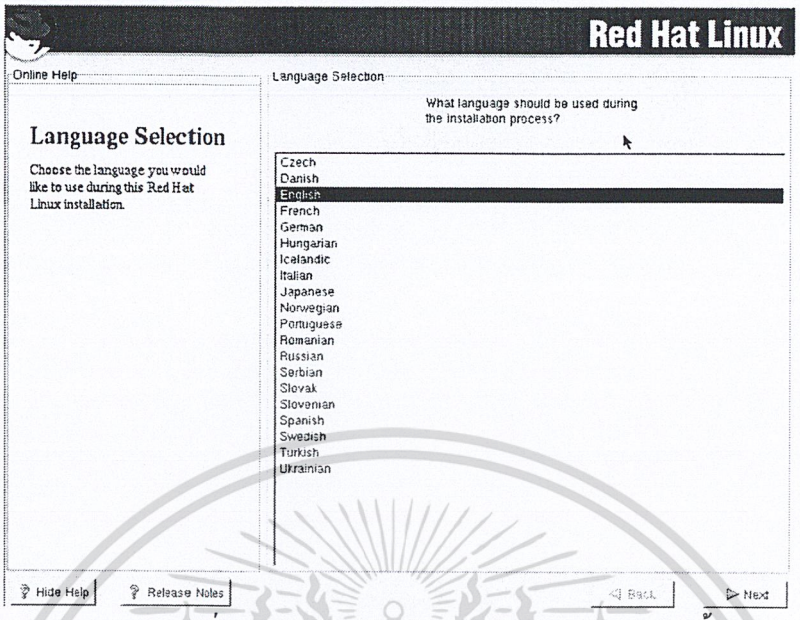
เมื่อบูตเครื่องขึ้นมาหลังจากใส่แผ่นลีนุกซ์ Redhat 7.1 จะแสดงเมนูดังรูป ถ้าไม่แสดงรูปดังกล่าวขึ้นมาแสดงว่า ในไบออส ไม่ได้ตั้งให้บูตด้วยซีดีรอม

แสดงหัวข้อที่ใช้ในการติดตั้ง ซึ่งที่ใช้กันส่วนมากจะใช้ในหัวข้อแรกและหัวข้อที่สอง ในหัวข้อแรกจะใช้ในการติดตั้งแบบกราฟฟิกโหมด ส่วนในหัวที่จะเป็นในลักษณะเท็กซ์โหมด ซึ่งทั้งสองหัวข้อนี้จะมีลักษณะคล้ายกัน แต่ในส่วนของเท็กซ์โหมดจะมีรายละเอียดมากกว่า เช่น สามารถเลือกแพกเกจในการติดตั้งได้ตามต้องการ



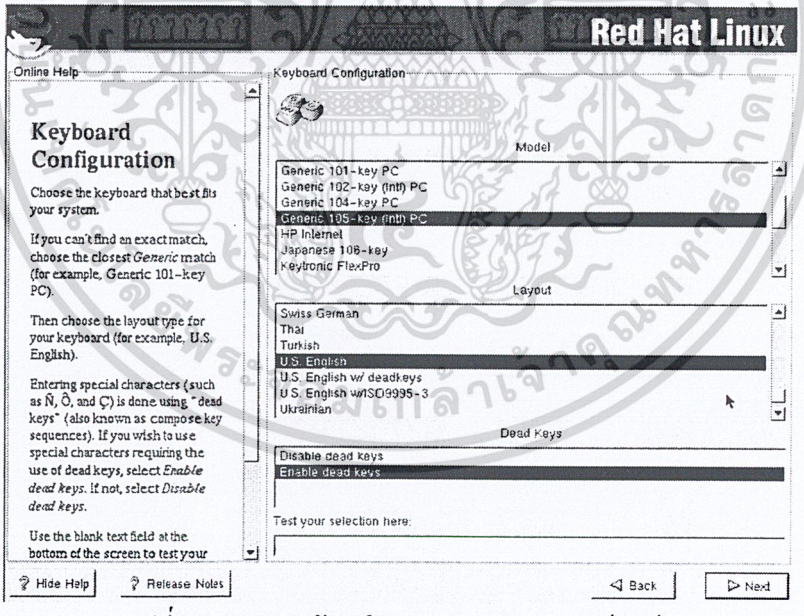
รูปที่ ก-1 แสดง หน้าจอในการติดตั้ง Linux Red Hat

รูปถัดมาเป็นการเลือกภาษาที่ใช้ในการติดตั้ง



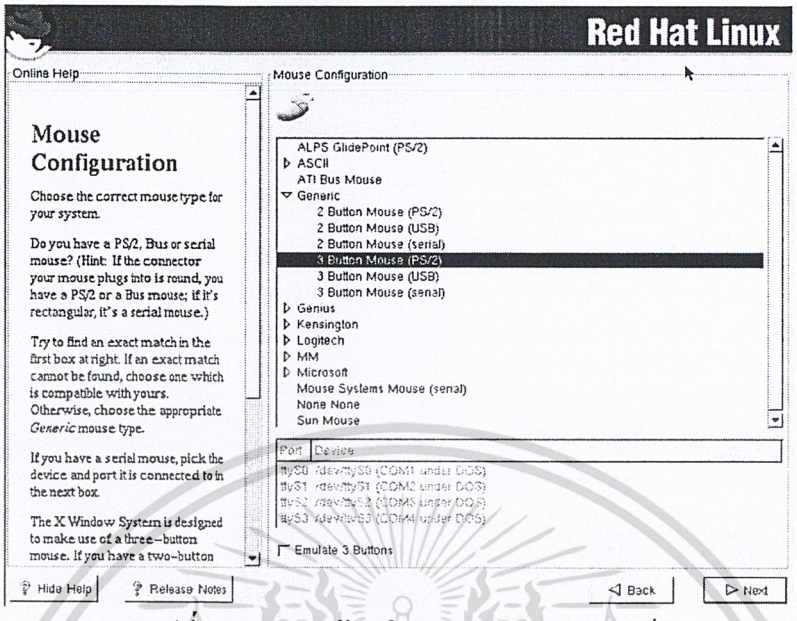
รูปที่ ก-2 แสดง หน้าจอในการเลือกภาษาในการติดตั้ง

แสดงการเลือกคีย์บอร์ดที่ต้องการใช้งาน และภาษาที่ต้องการใช้งาน



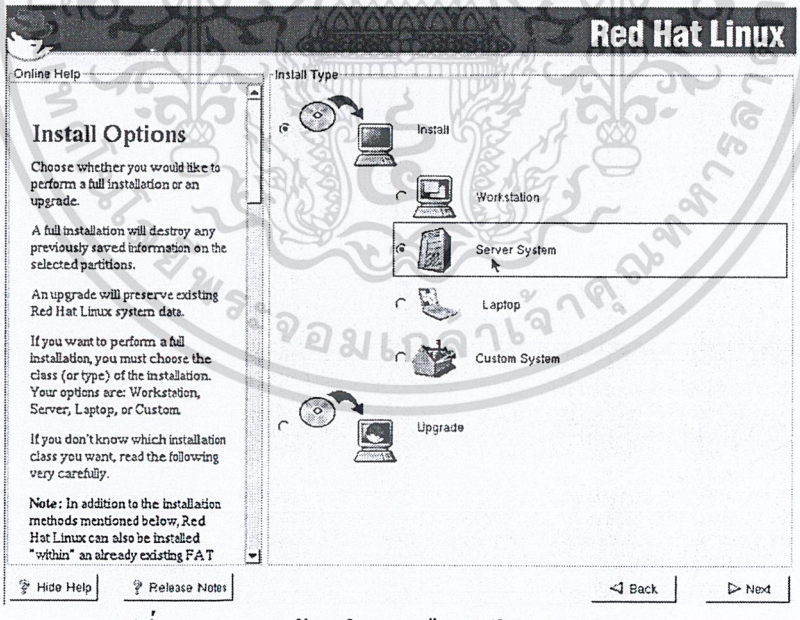
รูปที่ ก-3 แสดง หน้าจอในการคอนฟิกูเรชัน คีย์บอร์ด

แสดงการเลือกชนิดของเมาส์ที่ต้องใช้งาน ว่าเป็นชนิดไหน



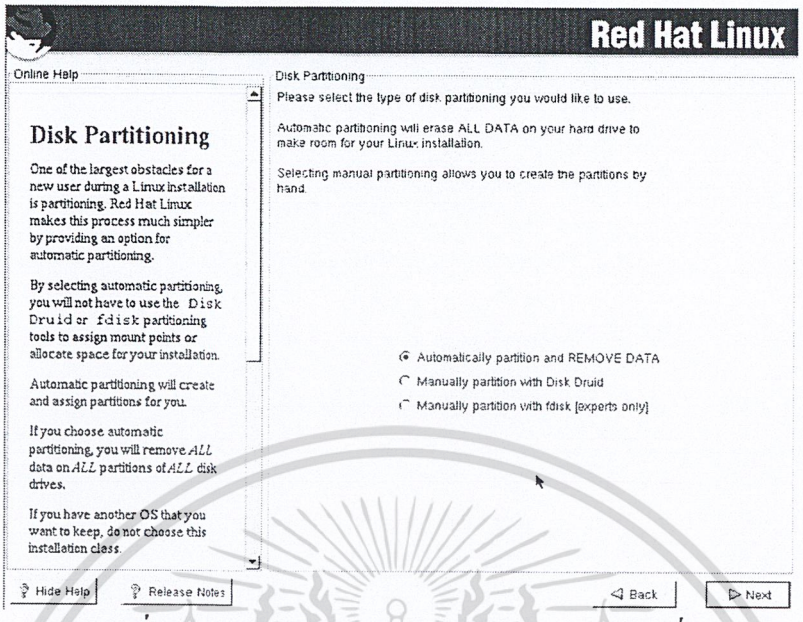
รูปที่ ก-4 แสดง หน้าจอในการคอนฟิกูเรชัน เมาส์

เลือกการการติดตั้งว่าจะเป็นการติดตั้งใหม่ หรือ อัพเกรด โดยการติดตั้งใหม่นั้นจะเป็นการติดแบบไหนตามเมนู



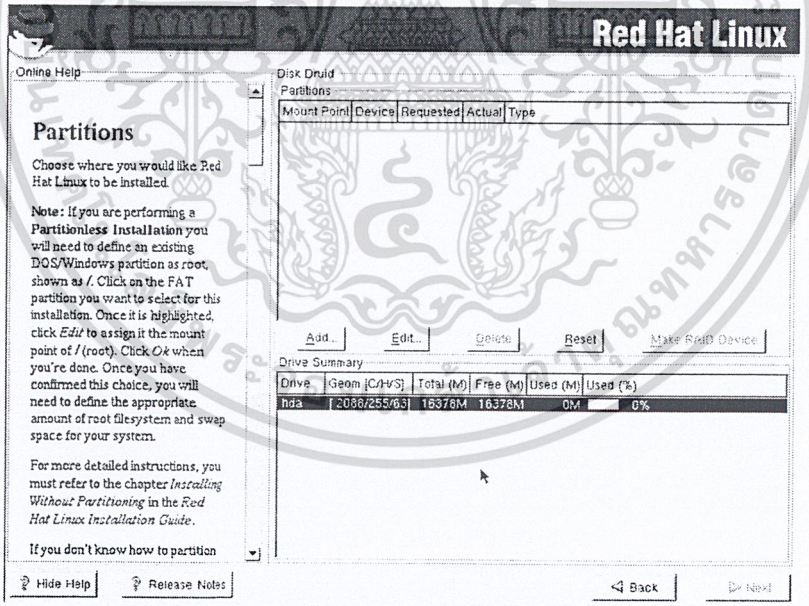
รูปที่ ก-5 แสดง หน้าจอในการเลือกวิธีในการลงโปรแกรม

แสดงการจัดการเกี่ยวกับการแบ่งพาติชันของดิสก์



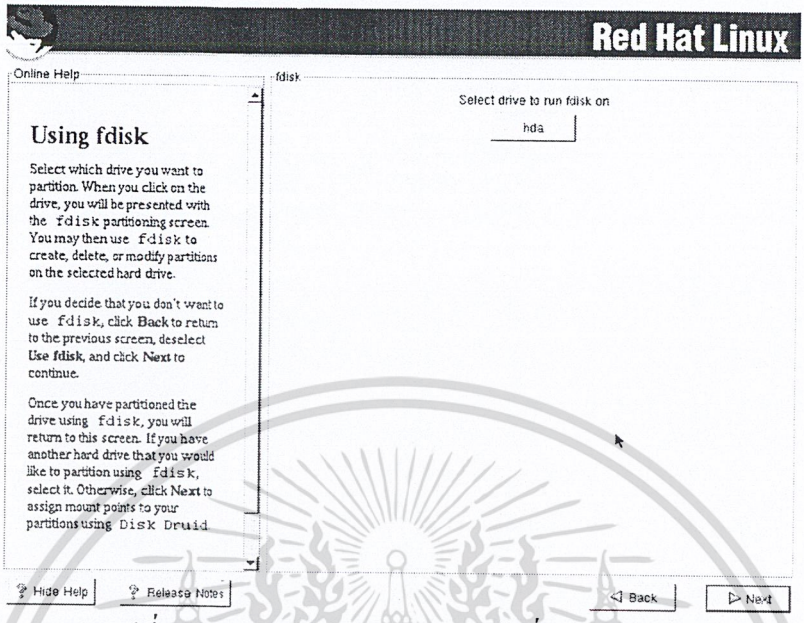
รูปที่ ก-6 แสดง หน้าจอในการเลือกวิธีในการแบ่งพาร์ทิชั่น

แสดงการแบ่งพาติชั่นโดย Disk Druid



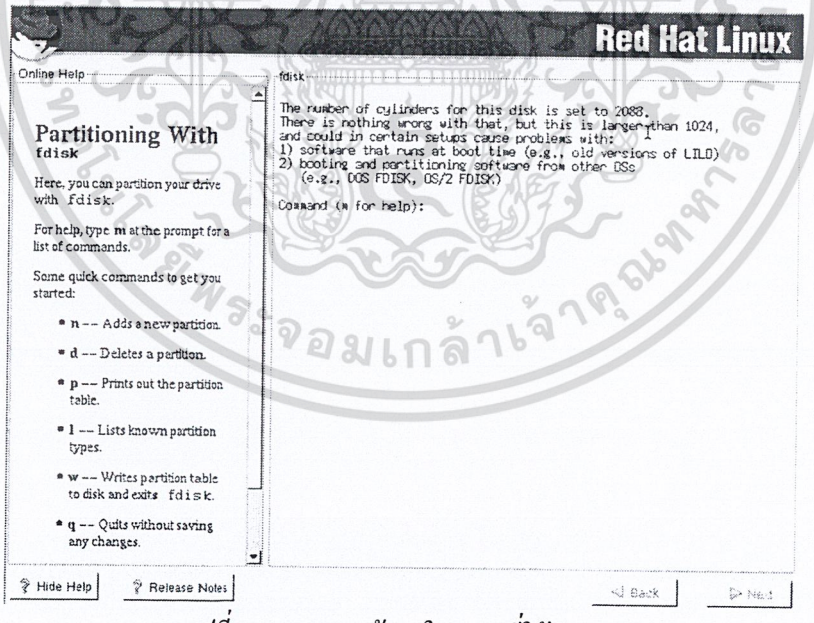
รูปที่ ก-7 แสดง หน้าจอในการแบ่งพาร์ทิชั่นโดยใช้ Disk Druid

แสดงการจัดการแบ่งพาติชันของดิสก์แบบใช้ fdisk เป็นการเลือกว่าจะ fdisk ตัวไหน ในกรณีที่มีฮาร์ดดิสก์หลายก้อน



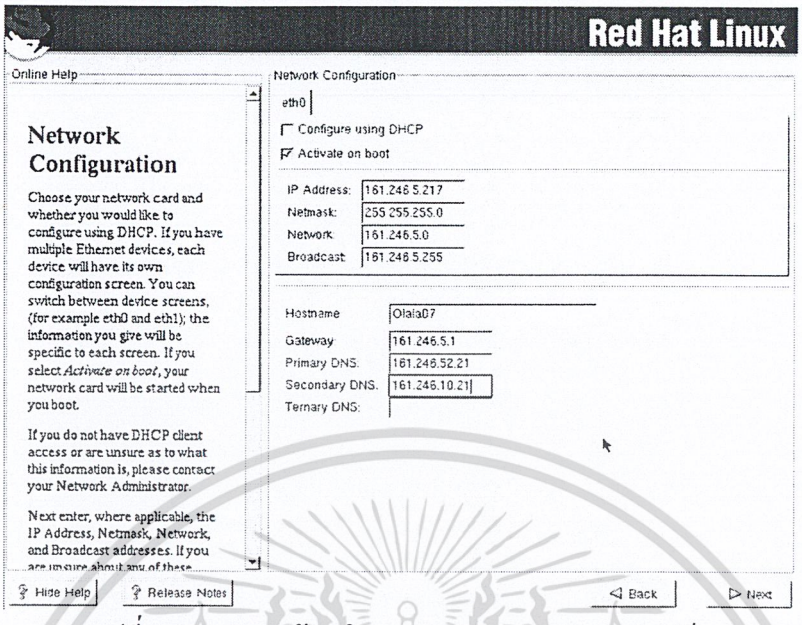
รูปที่ ก-8 แสดง หน้าจอในการแบ่งพาร์ทิชั่นโดยใช้ Fdisk

แสดงการใช้ fdisk



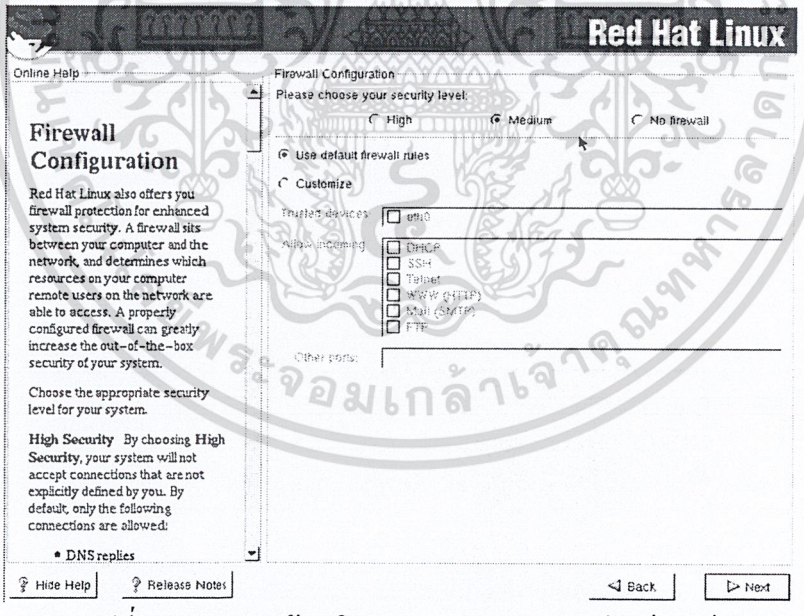
รูปที่ ก-9 แสดง หน้าจอในขณะที่ใช้ Fdisk

เป็นการกำหนดค่าเกี่ยวกับเน็ตเวิร์ก



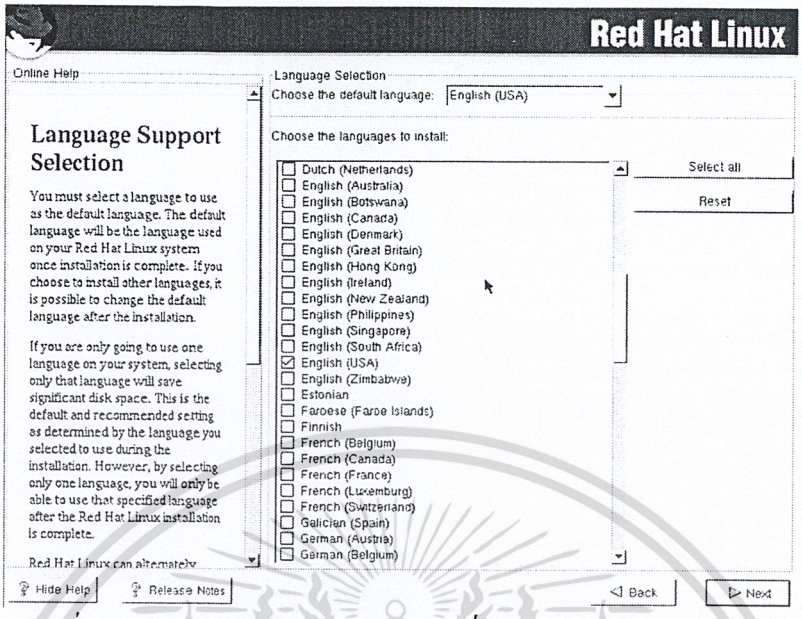
รูปที่ ก-10 แสดง หน้าจอในการคอนฟิกูเรชันระบบเน็ตเวิร์ก

กำหนดค่าความปลอดภัยของเครื่อง และต้องการเปิดบริการอะไรบ้างในเครื่องที่ใช้งาน



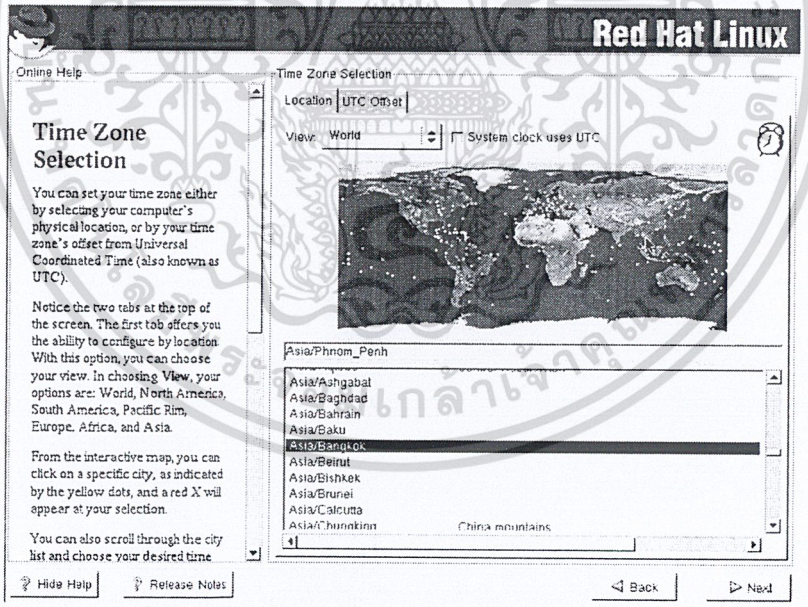
รูปที่ ก-11 แสดง หน้าจอในการคอนฟิกูเรชันระบบไฟร์วอลล์

เลือกภาษาที่ต้องการใช้



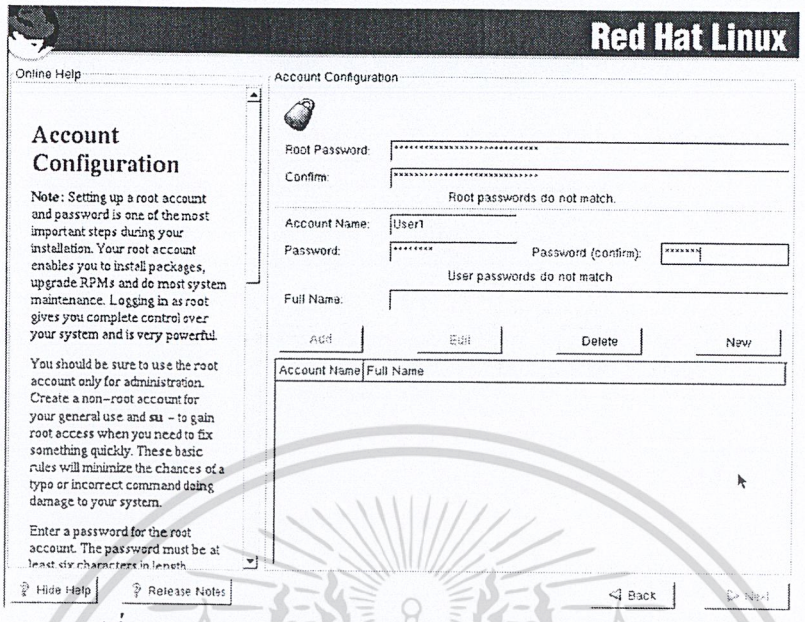
รูปที่ ก-12 แสดง หน้าจอในการเลือกภาษาที่ใช้ในระบบปฏิบัติการลีนุกซ์

กำหนดโซนของเวลา



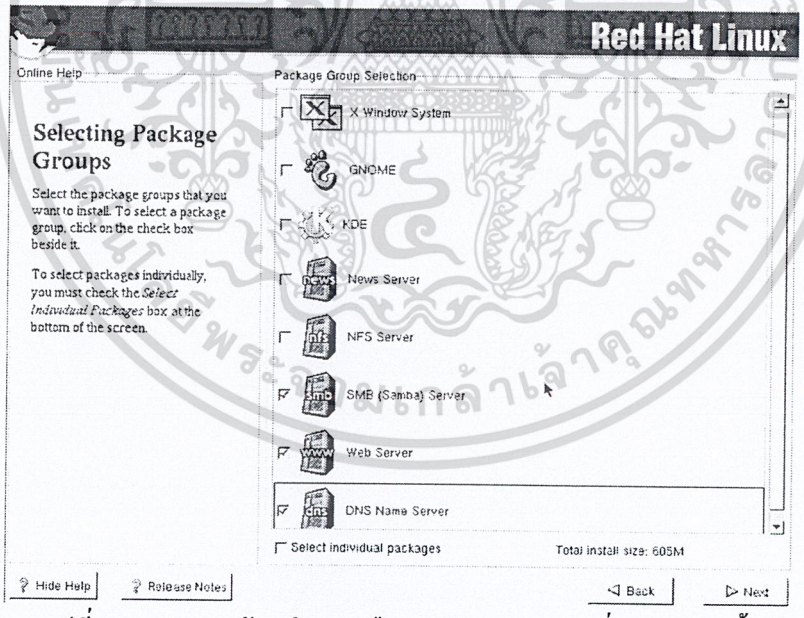
รูปที่ ก-13 แสดง หน้าจอในการเลือกโซนของเวลา

ใส่ค่าพาสเวิร์ดของ Root และเพิ่มยูสเซอร์ที่ต้องการใช้เครื่อง



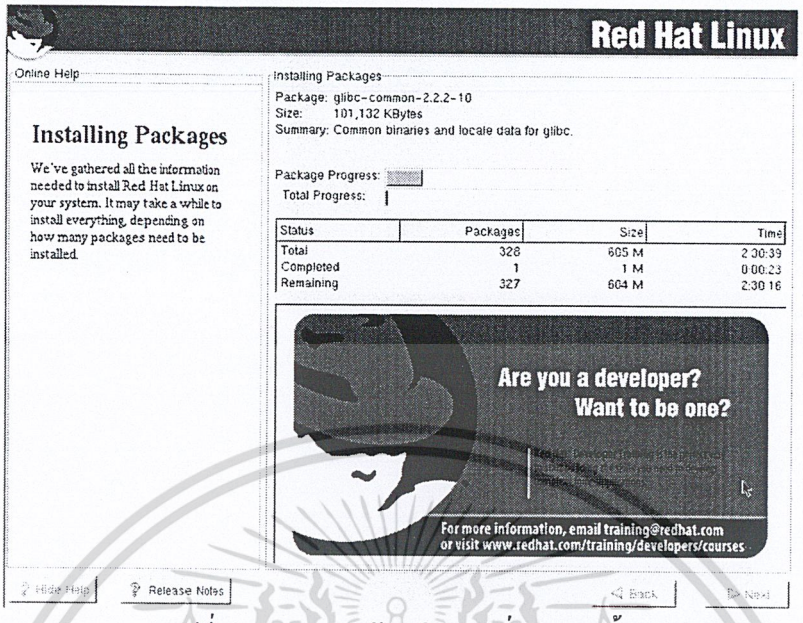
รูปที่ ก-14 แสดง หน้าจอในการเลือกการคอนฟิกูเรชันแอคเคาท์

เลือกกลุ่มของแพกเกจที่ต้องการติดตั้ง



รูปที่ ก-15 แสดง หน้าจอในการเลือกกลุ่มของแพกเกจที่ต้องการติดตั้ง

เริ่มการติดตั้งแพกเกจที่เลือกไว้



รูปที่ ก-16 แสดง หน้าจอในการเริ่มการติดตั้ง

ภาคผนวก ข.

ตอบคำถามท้ายการทดลอง

คำถามท้ายการทดลองบทที่ 3 การคอมไพล์เคอร์เนลและการดีบักด้วย GDB

1. เพราะเหตุใดจึงต้องทำการรีคอมไพล์เคอร์เนลเวอร์ชันใหม่อยู่เสมอ
2. ให้นักศึกษาอธิบายการทำงานของไฟล์หลัก ๆ 3 ไฟล์ คือ ไฟล์ bzImage , vmlinuz , System.map ว่าแต่ละไฟล์ทำหน้าที่ต่างกันอย่างไร
3. คำสั่งคอนฟิกแต่ละคำสั่งทำงานต่างกันอย่างไร
4. ทดลองดีบักโปรแกรมธรรมดาจากไฟล์ปกติกับดีบักเคอร์เนลและตอบให้ได้ว่าการดีบักทั้งสองแบบต่างกันอย่างไร
5. ก่อนทำการดีบักเคอร์เนลต้องคอมไพล์เคอร์เนลต่างกับการคอมไพล์เคอร์เนลแบบปกติอย่างไร

คำตอบท้ายการทดลองบทที่ 3 การคอมไพล์เคอร์เนลและการดีบักด้วย GDB

(1) การคอมไพล์เคอร์เนลเปรียบเสมือนเราติดตั้งระบบปฏิบัติการใหม่ซึ่งระบบปฏิบัติการที่ดีต้องรองรับฮาร์ดใหม่ ๆ ได้อยู่เสมอและมีประสิทธิภาพมากกว่าระบบปฏิบัติการเดิม ระบบปฏิบัติการลีนุกซ์ได้ออกแบบให้ผู้ใช้สามารถทำการแก้ไขเคอร์เนลได้ด้วยตัวเองซึ่งทำให้ผู้ใช้สามารถกำหนดการทำงานของเคอร์เนลให้เข้ากับสภาพแวดล้อมที่ผู้ใช้ต้องการใช้งานในปัจจุบันเคอร์เนลรุ่นล่าสุด คือ เวอร์ชั่น 2.5.6

(2) ไฟล์ที่สำคัญที่ได้หลังจากการรีคอมไพล์เคอร์เนล คือ bzIamge เป็นไฟล์ที่สำหรับผู้ใช้ที่ทำการรีคอมไพล์เคอร์เนลนำไปใช้ในการรันเคอร์เนลตัวใหม่ เป็นไฟล์ที่มีลักษณะเดียวกับ vmlinuz ใช้ในการบูตได้เหมือนกันเพื่อไม่ให้ผลที่ได้จากการบูตไปซ้ำกับเคอร์เนลตัวเดิมจึงสร้าง bzImage ขึ้นมาอีกตัวหนึ่งเพื่อง่ายต่อการใช้งาน ส่วนอีกไฟล์หนึ่ง คือ Vmlinux เป็นไฟล์ลักษณะเดียวกันกับ vmlinuz และ bzImage แต่ต่างกันที่การบีบอัดไฟล์สามารถดีบักได้เพราะ vmlinux มีขนาดที่ใหญ่กว่าส่วนไฟล์ System.map คือไฟล์ที่เก็บชื่อและแอดเดรสของรูทีนในระดับเคอร์เนลเพื่อให้โมดูลเรียกใช้งาน

(3) คำสั่งคอนฟิกที่ใช้งานโดยทั่วไปมีอยู่ด้วยกัน 4 คำสั่ง คือ

1. Make config เป็นการคอนฟิกเคอร์เนลในส่วนของเท็กโหมด
2. Make menuconfig เป็นการคอนฟิกเคอร์เนลในส่วนของกราฟฟิกโหมด
3. Make xconfig เป็นการคอนฟิกเคอร์เนลบนเอ็กวินโดว์
4. Make oldconfig เป็นการคอนฟิกเคอร์เนลจากไฟล์ {HOME}/linux/.config โดยขึ้นอยู่กับไฟล์ .config ซึ่งเป็นไฟล์ซ่อนต้องสั่งคำสั่ง ls –al จึงจะเห็นไฟล์นี้

ซึ่งเมื่อสั่งคำสั่งใน 4 คำสั่งนี้คำสั่งใดก็ได้จะสร้างไฟล์ .config ขึ้นมาให้

(4) การดีบักโปรแกรมธรรมดาสามารถดีบักได้เลยบนยูสเซอร์โหมดแต่สำหรับการดีบักเคอร์เนลนั้นมีบางไฟล์ที่ไม่สามารถเข้าไปดูค่าต่างๆ ได้ ทำได้แค่รู้ตอนนี้ดีบักอยู่ไฟล์นั้นเท่านั้น เช่น ไฟล์ lib.c ซึ่งเก็บ ไลบราลี่ต่างๆ

(5) ถ้าเราจะทำการดีบักเคอร์เนลการคอมไพล์เคอร์เนลจะต่างไปจากเดิม คือเราต้องเข้าไปแก้ไฟล์ใน Makefile ด้วยโดยเปลี่ยนแฟล็กของ CFLAGS จาก

CFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -fno-strength-reduce

(ข้างบนเป็นการกำหนดใน Make file ดั้งเดิม)

เราต้องทำการเปลี่ยนเป็น

CFLAGS = -g -Wall -Wstrict-prototypes -O2 -fno-strength-reduce

เมื่อทำการเปลี่ยนแฟล็กแล้วถัดนั้นมาเราก็คอมไพล์เคอร์เนลตามปกติซึ่งเมื่อเราทำขั้นตอน Make bzImage เราก็ได้ไฟล์ Vmlinux ที่สามารถนำมาทำการดีบักได้ ซึ่งไฟล์นี้สร้างขึ้นมาที่ไดเรกทอรี่ /{HOME}/linux ซึ่งการดีบักนั้นจำเป็นต้องมีหน่วยความจำเพียงพอสำหรับการดีบัก

{HOME} หมายถึง ไดเรกทอรี่ของซอร์ดโค้ดที่ทำการรีคอมไพล์เคอร์เนล

คำถามท้ายการทดลองบทที่ 4 ซิสเต็มคอล

1. ทำไมถึงต้องมีซิสเต็มคอล
2. การติดต่อไปยังฟังก์ชั่นในเคอร์เนลมีกี่แบบ อะไรบ้างจงอธิบาย

คำตอบท้ายการทดลองบทที่ 4 ซิสเต็มคอล

(1) การที่ต้องมีซิสเต็มคอลนั้น เพื่อเป็นตัวกลางในการติดต่อระหว่างโปรแกรมที่รันอยู่บนยูสเซอร์โหมด เพื่อใช้ในการติดต่อกับฮาร์ดแวร์ โดยตัวโปรแกรมที่รันอยู่บนยูสเซอร์โหมดนั้นไม่จำเป็นต้องเขียนโปรแกรมติดต่อกับฮาร์ดแวร์โดยตรง ซึ่งจะทำให้เกิดความยุ่งยาก

(2) มี 2 แบบ คือ การติดต่อโดยตรง กับการติดต่อโดยการสร้างสตับ สำหรับการติดต่อโดยตรง โดยการเรียกไปยังซิสเต็มคอลเบอร์นั้นเลย

```
#Include <sys/syscall.h>
…
        syscall(223,&baz_arg,bar_arg);
…
```

กับการติดต่อโดยผ่านสตับ โดยการสร้างสตับขึ้นมาก่อนแล้วเรียกผ่านโดยใช้ ฟังก์ชันนั้น

```
#include <linux/unistd.h>
...
        _syscall2(int , foo, char *, baz, double, bar);
...
```

คำถามท้ายการทดลองบทที่ 5 โปรเซสและตารางการจัดการโปรเซส

1. อธิบายถึงสถานะ (state) ของโปรเซสแต่ละสถานะเพื่อทำความเข้าใจสถานะของโปรเซสแต่ละสถานะนั้น ๆ
2. ทำไมระบบปฏิบัติการลีนุกซ์จึงใช้อัลกอริทึมการจัดการโปรเซสแบบ FIFO และ Round Robin ร่วมกัน
3. มีอัลกอริทึมใดบ้างที่สามารถนำมาทดแทนตารางการจัดการโปรเซสปัจจุบันได้
4. จงเลือกอัลกอรทึมในข้อ 3 มา 1 อัลกอริทึมอธิบายถึงลักษณะการใช้งานที่เหมาะสมเมื่อนำอัลกอริทึมในข้อ 3 มาทดแทนอัลกอริทึมการจัดการโปรเซสดั้งเดิม

คำตอบท้ายการทดลองบทที่ 5 โปรเซสและตารางการจัดการโปรเซส

(1) สถานะของโปรเซสแบ่งออกเป็น 5 สถานะ ดังนี้

1. TASK_RUNNING

State นี้มีการทำงานโดยแบ่งออกเป็น state ย่อย 2 state คือ เอ็กซิคิว และขณะรอการทำงานเพื่อการเอ็กซิคิวและพร้อมที่จะทำการรัน

2. TASK_INTERRUPTIBLE (Waiting)

State นี้คือ state ที่เมื่อโปรเซสอยู่ในสถานะ suspend หรือหลับอยู่สามารถที่ทำการอินเตอร์รัพได้ เช่น เมื่อมีการทำการสั่งอินเตอร์รัพฮาร์ดแวร์หรือมีการส่งสัญญาณเพื่อทำให้โปรเซส wake up เป็นต้น เมื่อมีการอินเตอร์รัพแล้วถ้าโปรเซสอยู่ในสถานะ TASK_RUNNING ต้องกลับไปทำงานยังโหมดของ TASK_RUNNING ต่อไป

3. TASK_UNINTERRUPTIBLE (Waiting)

State นี้คือ state ที่โปรเซสไม่ยอมให้มีการอินเตอร์รัพเกิดขึ้น เช่น กรณีที่โปรเซสเปิด device file และและทำการ probing เพื่อทำการติดต่อกับ hardware device ดังนั้น device driver ต้องไม่ยอมให้มีการอินเตอร์รัพจนกระทั่งทำการ probing จนเสร็จ

4. TASK_STOPPED

Stateนี้คือstateที่โปรเซสเอ็กซิคิวหยุดซึ่งหยุดหลังจากที่ได้รับสัญญาณSIGSTOP, SIGTSTP,SIGTTIN หรือสัญญาณ SIGTTOU

5. TASK_ZOMBIE

State นี้คือ state ที่โปรเซสอยู่ในสถานะที่ไม่สามารถทำงานต่อได้เรียกว่าโปรเซส terminated ซึ่งอาจเรียกตามลักษณะการทำงานว่าโปรเซสกำพร้าเนื่องจากโปรเซสไม่ทำงานต่อแต่เคอรเนลก็ยังไม่สามารถที่ทำการละทิ้งโปรเซสนี้ได้เนื่องจากอาจยังมีโปรเซสพาเร้นท์ที่ยังต้องการใช้โปรเซสนี้อยู่แต่โปรเซสพาเร้นท์ยังไม่ส่งสัญญาณออกมาทำให้โปรเซสนี้เป็นโปรเซสกำพร้า

(2) สาเหตุที่ระบบปฏิบัติการลีนุกซ์ใช้อัลกอริทึมแบบ FIFO กับ Round-Robin ร่วมกันเพราะว่าอัลกอริทึมแบบ FIFO เป็นอัลกอริทึมที่ง่ายและมีความยุติธรรมสำหรับการจัดการโปรเซส คือ โปรเซสไหนมาก่อนก็จะได้เข้าครอบครองซีพียูก่อนแต่ข้อเสียของอัลกอริทึมแบบ FIFO คือ มีบางโปรเซสที่สำคัญรอนานเกินไปถ้าเป็นโปรเซสที่สำคัญเมื่อมาทีหลังต้องรอโปรเซสที่มาก่อนแต่ว่าโปรเซสที่มาก่อนใช้เวลานานมาก ดังนั้นเพื่อเป็นการป้องกันปัญหาโปรเซสรอนานเกินไปจึงต้องมีการกำหนดค่า time quantum ขึ้นมาเพื่อช่วยในการลดเวลาในการรอซึ่งเป็นหลักการทำงานของอัลกอริทึมแบบ Round-Robin โดยที่ถ้ามีโปรเซสใดใช้เวลานานเกินค่า time quantum ก็จะสวิทช์ไปยังโปรเซสอื่นทันที

(3) การจัดการคิวหลายระดับ (Multi-level queue scheduling)
การจัดการคิวหลายระดับแบบป้อนกลับ (Multi-level feedback queue scheduling)
การจัดการโปรเซสแบบมาทีหลังได้ก่อน (Last in first out).....

(4) การจัดการคิวหลายระดับ (Multi-level queue scheduling)

การจัดการโปรเซสของระบบปฏิบัติการลีนุกซ์แบบดั้งเดิมนั้นเป็นแบบ FIFO ทำงานร่วมกับ Round-Robin ซึ่งได้ออกแบบให้ทำงานบนคิวเดี่ยว (Single queue) ซึ่งข้อเสียของการใช้ Single queue คืออาจทำให้เกิดปัญหาคอขวด (Bottle neck) ได้เมื่อรันด้วยจำนวนโปรเซสที่มากเพราะทำให้โปรเซสรอนานเกินไปซึ่งทางแก้ไข คือ ต้องใช้การจัดการโปรเซสแบบ Multi-level queue มาช่วยซึ่งอัลกอริทึมแบบ Muli-level queue จะช่วยให้การทำงานแบบ distributed system มีประสิทธิภาพเหมาะสำหรับระบบที่มีโปรเซสหลาย ๆ โปรเซสทำงานร่วมกันซึ่งจะทำงานได้อย่างมีประสิทธิภาพมากกว่าแบบคิวเดี่ยว

บรรณานุกรม

[1] Daniel P.Bovet , Macro Cesati (2001) : "Understanding the Linux Kernel" , Published by O'Reilly&Associates , United States of America , 2001.

[2] Scott Andrew Maxwell (2001) : "Linux Core Kernel Commentary 2nd Edition" , Published by Steve Sayre , Unired States of America , 2001

[3] สันติ ศรีลาศักดิ์ , วรวุฒิ เที่ยงธรรม (2542) : "เจาะประเด็นงานเขียนโปรแกรมบนลีนุกซ์" , ซีเอ็ดยูเคชั่น , กรุงเทพ ฯ , 2542

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง