บทที่ 3
ระบบปฏิบัติการ RT-Linux

โดยแนวคิดของ Victor Yodaiken และนิสิตปริญญาโท Michael Barabanov ที่ต้องการพัฒนาระบบปฏิบัติการที่สนับสนุนการทำงานเวลาจริงแบบเข้มงวด (hard real-time system) ซึ่งมีความสามารถทางด้านเครือข่าย (Networking), มีโปรแกรมสำเร็จรูปและไลบรารีที่ช่วยในการพัฒนา เช่น ระบบแสดงผลแบบกราฟฟิกบน X-windows และภาวะการใช้งานที่เอื้ออำนวยต่อการพัฒนาโปรแกรมเช่น มีตัวแปลโปรแกรม (compiler) และตัวตรวจสอบโปรแกรม (debugger) เป็นต้น รวมทั้งความสามารถทางด้านอื่นๆ เช่นเดียวกับระบบปฏิบัติการชั้นนำปกติ โดยทั่วไปมีแนวทางในการปฏิบัติ [4] คือ 1) เพิ่มความสามารถด้านต่างๆ ที่ต้องการให้กับระบบปฏิบัติการที่สนับสนุนการทำงานแบบเวลาจริงอยู่แล้ว ซึ่งอาจจะต้องพัฒนาขึ้นใหม่ทั้งหมด เช่น VX-Works, QNX และ OS9 เป็นต้น หรือ 2) แก้ไขระบบปฏิบัติการที่มีความสามารถทางด้านต่างๆ ตามต้องการ ได้แก่ Linux, Solaris, AIX, HP-Unix, SCO-Unix ให้เป็นระบบปฏิบัติการที่สนับสนุนการทำงานแบบเวลาจริง เช่นการแก้ไขโปรแกรมตามมาตรฐาน POSIX.1b ในกรณีของ Linux [8], VAX และ VMS เป็นต้น

แต่แนวทางที่ใช้ในที่นี้แตกต่างออกไปคือ เป็นการนำเอาระบบปฏิบัติการ Linux ซึ่งไม่สนับสนุนการทำงานเวลาจริงแบบเข้มงวด (hard real-time) แต่มีความสามารถทางด้านอื่น ๆ ที่ต้องการอย่างสมบูรณ์ มาแทรกชั้นของเครื่องเสมือน (Virtual Machine) ไว้ภายใต้ชั้นการทำงานของใจกลางระบบปฏิบัติการ (kernel) อีกต่อหนึ่ง โดยในชั้นของเครื่องเสมือนนี้จะคอยจัดลำดับการทำงานของงานแบบเวลาจริงตามลำดับความสำคัญ รวมทั้งรองรับรูทีนสำหรับอินเทอร์รัพที่ต้องการการตอบสนองแบบเวลาจริง และถือว่างานของใจกลางระบบปฏิบัติการ (kernel) เดิมเป็นงานแบบเวลาจริงที่มีความสำคัญน้อยที่สุด ระบบปฏิบัติการที่พัฒนาขึ้นนี้มีชื่อว่า RT-Linux

ตอนต้นบทนี้จะกล่าวถึงระบบปฏิบัติการ Linux [13] ซึ่งใช้กันอย่างแพร่หลาย ในด้านทั่วไปและความสามารถในการทำงานแบบเวลาจริง หลังจากนั้นจะกล่าวถึงในส่วนของ RT-Linux [14] ในแง่ของโครงสร้างของเครื่องเสมือน การจัดอินเทอร์รัพแบบเวลาจริงรวมถึงคุณสมบัติเด่นที่เป็นประโยชน์ต่อการใช้งานอีกด้วย

3.1 ระบบปฏิบัติการ Linux

ระบบปฏิบัติการ Linux เป็นระบบปฏิบัติการที่มีลักษณะคล้าย Unix สามารถคัดลอกมาใช้ได้ฟรี เขียนขึ้นโดย Linus Torvalds และโดยการช่วยเหลือของนักเขียนโปรแกรมมากมายจากทุกมุมโลกผ่านทางเครือข่ายอินเทอร์เน็ต Linux มุ่งพัฒนาเพื่อให้สอดคล้องกับมาตรฐาน POSIX อีกทั้งยังรวมคุณสมบัติเด่นของ Unix ไว้อย่างครบครัน เช่น การทำงานแบบหลายภาระกิจ

(multi-tasking) การจัดการหน่วยความจำ (memory management) และการสร้างหน่วยความจำเสมือน (virtual memory) ไลบรารีรวมของระบบ (shared library) การทำงานแบบหลายผู้ใช้ (multi-user) ระบบเครือข่าย (networking) และระบบแสดงผลแบบกราฟฟิก (Graphic User Interface) เป็นต้น

ระบบปฏิบัติการ Linux เริ่มพัฒนาบนเครื่องคอมพิวเตอร์ส่วนบุคคลที่ใช้หน่วยประมวลผลแบบ 386 และเรื่อยมาสำหรับ 486 และ 586 ในปัจจุบันมีการพัฒนาเพื่อใช้งานได้สำหรับเครื่องคอมพิวเตอร์ที่มีโครงสร้างแบบอื่น ๆ [6] ได้แก่ 68000-series, MIPS, ARM, SPARC, Alpha เป็นต้น เนื่องจากการที่ Linux เป็นระบบเปิด สามารถศึกษาดัดแปลงจากต้นฉบับโปรแกรมได้ และสามารถคัดลอกมาใช้ได้ฟรี ทำให้มีการใช้งานอย่างแพร่หลาย รวมทั้งมีผู้ร่วมในการพัฒนาและนำระบบปฏิบัติการไปใช้เป็นการทดสอบจำนวนมากผ่านทางเครือข่ายอินเทอร์เน็ต

Linux มีซอฟท์แวร์ที่เป็นประโยชน์อยู่มากเช่น ตัวแปลโปรแกรม (compiler) และตัวทดสอบโปรแกรม (debugger) ในภาษาต่าง ๆ (C, Pascal, Fortran, Perl เป็นต้น) โปรแกรม Emacs, TeX, LaTeX ซึ่งใช้ในการทำงานเอกสาร โปรแกรม X-Windows รวมถึงโปรแกรมต่าง ๆ ที่ใช้งานบน X-Windows ซึ่งเป็นระบบติดต่อผู้ใช้แบบกราฟฟิก โปรแกรมเกี่ยวกับเครือข่าย (การรับ-ส่งจดหมายอิเลคทรอนิกส์ การใช้ไฟล์ร่วมกันผ่านทางเครือข่าย – Network File System การใช้ข้อมูลร่วมกันผ่านทางเครือข่าย - Network Information System เป็นต้น) และอีกหลาย ๆ โปรแกรมซึ่งจะถูกพัฒนาขึ้นอย่างต่อเนื่องโดยผู้ใช้จากทั่วทุกมุมโลก

นอกจากนี้ยังมีการพัฒนารูปแบบการใช้งานเป็นภาษาไทยบน Linux โดยบุคคลหลายกลุ่มที่สำคัญ เช่น ในประเทศญี่ปุ่นมีการพัฒนาตัวอักษรภาษาไทย [20] การใช้งานโปรแกรม LaTeX ภาษาไทยของกลุ่มนักศึกษาไทยที่ Tokyo Institute of Technology [21] รวมทั้งในประเทศไทย [22] เองด้วย

แต่การที่จะนำระบบปฏิบัติการแบบหลายภาระกิจมาใช้ในการควบคุมจำเป็นจะต้องคำนึงความสามารถในการทำงานแบบเวลาจริงด้วย ซึ่งจะได้กล่าวในหัวข้อต่อไป

3.2 ความสามารถในการทำงานแบบเวลาจริงของ Linux

การสนับสนุนการทำงานแบบเวลาจริงเป็นผลมาจากการแก้ไขส่วนต่าง ๆ ของระบบปฏิบัติการ รวมถึงไลบรารีที่มีอยู่ให้สอดคล้องกับมาตรฐาน POSIX.1b ซึ่งเป็นข้อกำหนดเกี่ยวกับส่วนต่าง ๆ ที่สนับสนุนการทำงานแบบเวลาจริง เช่น การล็อกส่วนของโปรแกรมที่กำลังทำงานให้อยู่ในหน่วยความจำตลอดเวลา, การเพิ่มความละเอียดของหน่วยเวลาและการเพิ่มโปรแกรมจัดลำดับงาน (scheduler) แบบความสำคัญคงที่ (static priority) เป็นต้น รายละเอียดของการแก้ไขตามมาตรฐาน POSIX.1b [8] มีดังต่อไปนี้

3.2.1 สัญญาณ (signal)

รายละเอียดของการกำหนดเพิ่มเติมในเรื่องของสัญญาณมีดังนี้คือ

- เพิ่มสัญญาณสำหรับใช้งานแบบกำหนดเองอีกสองสัญญาณคือ SIGUSR1 และ SIGUSR2
- สัญญาณที่เพิ่มขึ้นใหม่สามารถส่งข้อมูลเป็นพอยท์เตอร์หรือจำนวนเต็มไปยังรูทีนรองรับสัญญาณ (signal handler) ได้ ทำให้สามารถส่งข้อมูลซึ่งอาจแสดงสาเหตุของการเกิดสัญญาณนั้น ๆ ได้
- สัญญาณใหม่นี้จะถูกเก็บลงในคิวในกรณีที่มีสัญญาณเกิดขึ้นในเวลาเดียวกัน ทำให้สัญญาณที่เกิดขึ้นพร้อม ๆ กันไม่สูญหายไป
- มีการกำหนดความสำคัญของสัญญาณแต่ละสัญญาณ และสัญญาณที่สำคัญกว่าจะทำงานก่อน
- กำหนดฟังก์ชันเพื่อรอการเกิดของสัญญาณ และจะไม่ทำงานต่อจนกว่าจะมีสัญญาณที่รออยู่เกิดขึ้น

3.2.2 การติดต่อระหว่างโปรแกรม (Inter Process Communication – IPC)

มาตรฐาน POSIX.1b ได้กำหนดแก้ไขเกี่ยวกับ IPC ซึ่งหมายความรวมถึง การใช้หน่วยความจำร่วมกัน (shared memory) คิวข้อมูล (messages queue) และ เซมาฟอร์ (semaphore) ไว้ดังนี้คือ

- ให้มีการใช้สายอักขระในการอ้างถึง IPC แทนการใช้เลขจำนวนเต็มซึ่งทำให้เกิดการอ้างถึง IPC ที่ซ้ำกันได้ง่าย และมีการแสดงการปรากฏของ IPC ในแบบเดียวกับ /proc
- เซมาฟอร์จะถูกกำหนดขึ้นใน 2 ลักษณะคือ 1) แบบอาศัยเคอร์เนลล์ (kernel) และ 2) แบบอาศัยไลบรารี เดิมจะมีเซมาฟอร์ในแบบที่ 1 เพียงอย่างเดียว ซึ่งการจะใช้งานเซมาฟอร์จะต้องเรียกผ่าน system call ทำให้งานที่มีการใช้เซมาฟอร์หลาย ๆ ครั้งติดต่อกันจะช้า และจะมีผลไปถึงงานอื่น ๆ ในระบบด้วย จึงเกิดเซมาฟอร์ในแบบที่ 2 ซึ่งจะเรียกใช้งานผ่านไลบรารีร่วม (shared library) ของระบบ ทำให้เร็วกว่ามาก แต่อย่างไรก็ตามในงานบางอย่างยังจำเป็นที่จะต้องใช้เซมาฟอร์ในแบบที่ 1 เนื่องจากเหตุผลในด้านความปลอดภัย
- กำหนดให้การซ้อนไฟล์ลงในหน่วยความจำ (memory mapped files) และการใช้งานหน่วยความจำร่วม (shared memory) ทำได้ด้วยฟังก์ชันเดียวกัน

3.2.3 การล็อกหน่วยความจำ (Memory locking)

มีการกำหนดฟังก์ชันเพื่อสั่งไม่ให้มีการ paging สำหรับหน่วยความจำช่วงที่ต้องการ

(หน่วยความจำที่โปรแกรมใช้งาน เช่น code, stack, data, shared memory, memory mapped files, shared libraries) เพื่อที่จะลดเวลาในการเข้าถึงหน่วยความจำนั้น ๆ

3.2.4 Synchronous I/O

ในงานบางอย่างเช่น ระบบฐานข้อมูล ระบบจดหมายอิเลคทรอนิกส์ ระบบบันทึกการทำงาน การบันทึกข้อมูลจะต้องถูกเก็บลงสู่ฮาร์ดดิสก์ทันที (ปกติจะถูกเขียนลงในหน่วยความจำก่อน จนกว่าหน่วยความจำที่ใช้เก็บจะเต็มหรือถึงกำหนดเวลา จึงจะทำการเก็บลงสู่ฮาร์ดดิสก์ ทั้งนี้เพราะการเก็บข้อมูลลงในหน่วยความจำทำได้เร็วกว่าเก็บลงสู่ฮาร์ดดิสก์นั่นเอง) เพื่อประกันไม่ให้ความเสียหายเกิดขึ้นในกรณีที่ระบบเกิดพังหรือไฟฟ้าดับ เป็นต้น

นอกจากนี้ยังมีฟังก์ชันที่ทำหน้าที่นำข้อมูลที่อยู่ในหน่วยความจำเก็บลงสู่ฮาร์ดดิสก์โดยเลือกเฉพาะในส่วนที่มีการเปลี่ยนแปลงเท่านั้น (จากเดิมจะไม่ตรวจสอบว่าข้อมูลมีการเปลี่ยนแปลงหรือไม่) ซึ่งจะช่วยเพิ่มความเร็วขึ้นอีกหลายเท่าตัว

3.2.5 เวลา (Timers)

ในเรื่องของเวลา มีการกำหนดฟังก์ชันเกี่ยวกับเวลาขึ้นใหม่ โดยมีความละเอียดในระดับ 1 ใน พันล้านวินาที และกำหนดฟังก์ชันในการตั้งเวลาให้สามารถตั้งเวลาได้อย่างน้อย 32 ค่าต่อโปรแกรม

3.2.6 การจัดลำดับงาน (Scheduling)

การจัดลำดับงานใน Linux เป็นแบบที่คำนึงถึงประสิทธิภาพโดยรวมของระบบ โดยพยายามจัดลำดับงานเพื่อให้ผู้ใช้ ใช้งานได้เท่าเทียมกัน ทำให้โปรแกรมบางอย่างที่ต้องการการทำงานแบบเวลาจริงในการติดต่อกับอุปกรณ์ภายนอก การคำนวณค่าภายในกรอบเวลาที่กำหนด ซึ่งสามารถโปรแกรมให้ทำงานได้อย่างมีประสิทธิภาพในระบบปฏิบัติการง่าย ๆ เช่น DOS ไม่สามารถทำงานตามต้องการได้ใน Linux รวมทั้งวิธีการในการจัดการงานแบบนี้จะกินเวลาในการสับเปลี่ยนงานมากกว่าแบบอื่น (อาจกินเวลามากกว่า 100 มิลลิวินาที) ในการแก้ปัญหานี้ POSIX.1b กำหนดวิธีการจัดลำดับงานไว้ 3 แบบ ซึ่งแต่ละแบบจะมีค่าความสำคัญคงที่ ดังนี้คือ

3.2.6.1 SCHED_FIFO

เป็นการจัดลำดับงานแบบมีการกำหนดค่าความสำคัญและสามารถขัดจังหวะโดยงานที่สำคัญกว่าได้ ในการจัดลำดับงานแบบนี้ งานที่กำลังทำอยู่ในขณะใด ๆ จะทำต่อไปเรื่อยจนกว่า 1) งานนั้นหยุดทำงานเองด้วยเหตุผลต่าง ๆ เช่น หยุดรอการใช้งานอุปกรณ์ภายนอก หยุดรอการหน่วงเวลา หยุดรอการทำงานในเวลาที่กำหนด เป็นต้น และ 2) ไม่มีการอินเทอร์รัพซึ่งทำให้เกิดงานที่มีความสำคัญมากกว่า มีการกำหนดคิว

สำหรับแต่ละระดับของความสำคัญ และแต่ละงานที่เปลี่ยนมาอยู่ในสถานะพร้อมทำงาน จะถูกเพิ่มเข้าไปต่อท้ายงานที่มีอยู่เดิมในคิว วิธีนี้เป็นวิธีที่นิยมใช้สำหรับระบบปฏิบัติการแบบเวลาจริงมากที่สุด

3.2.6.2 SCHED_RR

เป็นการจัดลำดับการทำงานตามค่าความสำคัญแบบ round-robin และงานที่สำคัญกว่าจะสามารถขัดจังหวะการทำงานของงานที่สำคัญน้อยกว่าได้ วิธีนี้คล้ายกับแบบ SCHED_FIFO มากแต่ต่างกันตรงที่ถ้ามีงานที่มีความสำคัญเท่ากันรออยู่ในคิว จะสามารถขัดจังหวะการทำงานในขณะนั้นได้เมื่อถึงเวลาที่กำหนดค่า ๆ หนึ่ง

3.2.6.3 SCHED_OTHER

เป็นวิธีการจัดลำดับงานแบบใดก็ได้ที่ต่างไปจาก 2 แบบแรก ใน Linux จะเป็นการจัดลำดับงานที่เรียกว่า nice level งานที่เป็น SCHED_OTHER ทั้งหมดจะถือว่ามีค่าความสำคัญต่ำสุด

โดยปกติในระบบปฏิบัติการแบบหลายผู้ใช้มักจะกำหนดให้ผู้ใช้เพียงหนึ่งคนหรือจำนวนไม่มาก สามารถกำหนดวิธีการจัดลำดับงานของงานต่าง ๆ ได้ เพื่อป้องกันความสับสน โดยปกติงานที่ต้องการทำแบบเวลาจริงจะถูกจัดลำดับแบบ SCHED_FIFO และกำหนดค่าความสำคัญไว้สูง ๆ นอกจากนี้ควรจะล็อกหน่วยความจำที่ใช้อยู่เพื่อไม่ให้มีการ paging อีกด้วย

3.2.7. Asynchronous I/O

POSIX.1b กำหนดฟังก์ชันสำหรับการอ่านและหรือเขียนได้พร้อมกันในการติดต่ออุปกรณ์อินพุตและเอาท์พุตในครั้งเดียว และจะมีการส่งสัญญาณมายังโปรแกรมเมื่อทำงานที่ต้องการเสร็จหมดแล้ว ภายในฟังก์ชันจะมีการคำนวณวิธีหรือลำดับในการอ่านและเขียนตามที่ได้รับมาให้ทำงานได้เร็วที่สุด การกำหนดดังนี้จะช่วยให้งานบางอย่างเช่น ระบบฐานข้อมูลทำงานได้เร็วขึ้นอย่างมาก และทำให้โปรแกรมแสดงภาพหรือเสียง ทำงานได้โดยไม่สะดุดเวลาอ่านข้อมูล นอกจากนี้ยังมีการกำหนดค่าความสำคัญให้กับการทำ asynchronous I/O ด้วย เพื่อช่วยในกรณีที่ทำงานที่มีการติดต่อกับอุปกรณ์ชนิดเดียวกันพร้อมกัน เช่นการชมภาพยนตร์จากในฮาร์ดดิสก์พร้อมกับการแปลโปรแกรม ก็จะกำหนดให้การติดต่อกับฮาร์ดดิสก์ของโปรแกรมชมภาพยนตร์มีความสำคัญกว่า เป็นต้น

แต่อย่างไรก็ตามถึงแม้ว่าจะมีการแก้ไขตามมาตรฐาน POSIX.1b ได้ครบถ้วนทุกประการ ก็ยังปัญหาในการที่จะใช้งาน Linux แบบเวลาจริงอยู่ 2 ข้อใหญ่ ๆ คือ

(1) เวลาในการตอบสนองต่อการอินเทอร์รัพ

ตัวอย่างเช่น งานที่รอการติดต่อกับอุปกรณ์ภายนอกจะเปลี่ยนเป็นสถานะพร้อมทำงานเมื่อได้รับการอินเทอร์รัพจากอุปกรณ์นั้น ใน Linux อินเทอร์รัพจะได้รับการตอบสนอง 2 แบบคือ แบบช้าและแบบเร็ว ในแบบช้าเมื่อเกิดการอินเทอร์รัพขึ้น รูทีนสำหรับรองรับอินเทอร์รัพนั้นจะทำงานและจบด้วยการเรียกตัวจัดการงานทุกครั้ง นั่นคือถ้างานที่รออยู่มีค่าความสำคัญสูงที่สุดในขณะนั้น งานจะถูกทำในทันที แต่ถ้าเป็นในแบบเร็วรูทีนรองรับการอินเทอร์รัพจะจบด้วยการเซ็ตค่าแฟลกเก็บไว้เพื่อแสดงว่ารูทีนรองรับอินเทอร์รัพนั้นได้ทำงานแล้ว และจะกลับไปทำงานเดิมที่ทำก่อนที่จะถูกอินเทอร์รัพต่อไปจนกว่าจะถึงรอบเวลาของตัวจัดลำดับงาน (ใน Linux ปกติจะมีค่าเป็น 10 มิลลิวินาที) ซึ่งแบบนี้จะทำให้งานที่รออยู่ถึงแม้ว่าจะมีค่าความสำคัญสูงสุดก็ตาม อาจจะต้องรอนานเท่ากับ 1 รอบของตัวจัดลำดับงาน

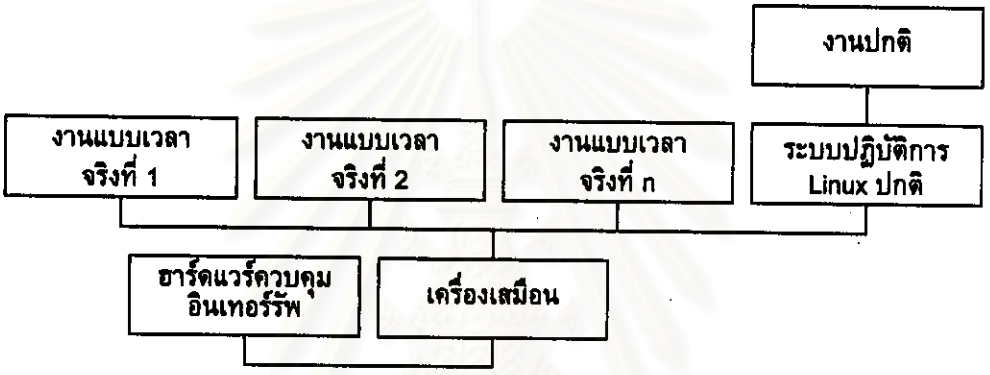
(2) ความละเอียดและถูกต้องของเวลา

ใน Linux ทั่วไป สัญญาณนาฬิกาจะอินเทอร์รัพทุก ๆ 10 มิลลิวินาที (ยกเว้นในเครื่องแบบ Alpha ซึ่งสัญญาณนาฬิกาจะถูกตั้งให้อินเทอร์รัพทุก ๆ 1 มิลลิวินาที) ดังนั้นฟังก์ชันที่เกี่ยวกับเวลาต่าง ๆ เช่นฟังก์ชันตั้งเวลาและฟังก์ชันหน่วงเวลา ซึ่งจะเช็คทุกครั้งที่มีอินเทอร์รัพจากสัญญาณนาฬิกาว่าถึงเวลาที่กำหนดหรือยัง อาจจะทำงานผิดพลาดไปถึง 2 เท่าของคาบสัญญาณนาฬิกา หรือ 20 มิลลิวินาที ทั้งที่อาจจะต้องการจริงเพียง 1 นาโนวินาที

จะเห็นได้ว่าการทำตามมาตรฐาน POSIX.1b ดังได้อธิบายไปแล้วข้างต้น ไม่เพียงพอสำหรับการทำงานเวลาจริงในแบบเข้มงวด (hard real-time) ซึ่งต้องการการทำนายได้ (predictability) และการตอบสนองเร็ว (low latency) เพราะงานในแบบเวลาจริงอาจถูกขัดจังหวะการทำงานจากอินเทอร์รัพที่ไม่มีความสำคัญได้ และเวลาในการเริ่มต้นทำงานแบบเวลาจริงหรือการเปลี่ยนการทำงานจากงานที่มีความสำคัญน้อยกว่าไปสู่งานที่ความสำคัญมากกว่า (Pre-emption) อาจไม่ทันท่วงทีและไม่ถูกต้องถึงระดับความละเอียดของเวลาที่ต้องการ เนื่องจากการจัดลำดับงาน (scheduling) จะทำตามการขัดจังหวะของสัญญาณนาฬิกาทุก ๆ 10 มิลลิวินาที (ในกรณีของ Linux) นอกจากนี้การที่ระบบปฏิบัติการไม่รับการอินเทอร์รัพในบางเวลาที่เป็นการทำงานในช่วงวิกฤตทำให้การตอบสนองต่อการขัดจังหวะไม่เป็นแบบเวลาจริง RT-Linux สามารถแก้ปัญหาดังกล่าวได้ โดยหัวข้อถัดไปจะอธิบายถึงโครงสร้างของ RT-Linux ในส่วนที่เพิ่มเติมจาก Linux ปกติเพื่อแก้ปัญหาดังกล่าว

3.3.โครงสร้างทั่วไปของ RT-Linux

RT-Linux มีโครงสร้างแสดงได้ดังรูปที่ 3.1 [3] คือการแทรกชั้นของเครื่องเสมือน (Virtual Machine) ไว้ภายใต้การทำงานของระบบปฏิบัติการปกติ เครื่องเสมือนจะคอยจัดลำดับให้งานแบบเวลาจริงทำงานตามลำดับความสำคัญซึ่งถูกกำหนดไว้เป็นค่าคงที่ จนไม่มีงานแบบเวลาจริง เครื่องเสมือนจึงจะสับเปลี่ยนให้ระบบปฏิบัติการปกติทำงาน อาจกล่าวได้ว่าระบบปฏิบัติการ Linux เดิมเปรียบเสมือนงานแบบเวลาจริงที่มีความสำคัญน้อยที่สุดนั่นเอง



รูปที่ 3.1 แสดงโครงสร้างของ RT-Linux

การจัดการเกี่ยวกับอินเทอร์รัพ เพื่อลดความล่าช้าและให้สามารถขัดจังหวะการทำงานของระบบปฏิบัติการได้ตลอดเวลา อินเทอร์รัพจะถูกจัดการโดยเครื่องเสมือน โดยอินเทอร์รัพที่ต้องการการทำงานแบบเวลาจริงจะถูกจัดลำดับให้ทำงานทันที ส่วนอินเทอร์รัพที่ไม่ต้องการการทำงานแบบเวลาจริงจะถูกส่งต่อให้กับระบบปฏิบัติการปกติเมื่อไม่มีงานแบบเวลาจริงทำงานอยู่ในขณะนั้น และระบบปฏิบัติการปกติจะควบคุมอินเทอร์รัพผ่านทางตัวซอฟท์แวร์ควบคุมอินเทอร์รัพจำลองซึ่งถูกสมมุติขึ้นในเครื่องเสมือน นั่นคือเมื่อระบบปฏิบัติการปกติสั่งห้ามการอินเทอร์รัพ ตัวซอฟท์แวร์จำลองก็เพียงแต่รับอินเทอร์รัพที่ไม่ต้องการการทำงานแบบเวลาจริงมาเข้าคิวรอไว้ เมื่อระบบปฏิบัติการปกติทำงานอีกครั้งและสั่งอนุญาตการอินเทอร์รัพ ตัวซอฟท์แวร์จำลองก็จะส่งผ่านอินเทอร์รัพที่อยู่ในคิวไปให้กับระบบปฏิบัติการปกติจัดการ ซึ่งระบบปฏิบัติการปกติจะเสมือนว่าได้รับการอินเทอร์รัพจากฮาร์ดแวร์จริง ๆ การจัดการอินเทอร์รัพในวิธีนี้ ทำให้สามารถสนองตอบการอินเทอร์รัพได้เร็วและตลอดเวลา

ในเรื่องความละเอียดและถูกต้องของเวลาในการจัดลำดับการทำงาน RT-Linux ใช้การขัดจังหวะสัญญาณนาฬิกาแบบตั้งเวลา (one-shot timer) ร่วมกับแบบคาบเวลา (periodic timer) ซึ่งทำให้การสับเปลี่ยนการทำงานและฟังก์ชันเกี่ยวกับเวลาต่าง ๆ มีความละเอียดและถูกต้องสูง อีกทั้งยังใช้งานหน่วยความจำในแบบคงที่ เพื่อให้เวลาในการสับเปลี่ยนงานลดลงอีกด้วย

นอกจากนี้ ในซอฟท์แวร์ระบบควบคุมหนึ่งๆ จะมีทั้งงานแบบเวลาจริง เช่นในส่วนของการควบคุม และงานแบบปกติเช่นการจัดเก็บข้อมูล การแสดงผล เป็นต้น งานในทั้งสองส่วนนี้สามารถส่งผ่านข้อมูลกันได้สองทาง คือ ผ่านทางคิวที่ไม่มีการล็อก (lock-free queue) หรือการใช้หน่วยความจำร่วมกัน (share memory) ซึ่งการใช้งานเป็นไปตามมาตรฐาน POSIX รวมทั้งมีการใช้เซมาฟอร์เพื่อป้องกันความสับสนในการใช้ทรัพยากรร่วมกัน

ในการใช้งานจริง RT-Linux สามารถลดความล่าช้าในการอินเทอร์รัพลงเหลือเพียงไม่เกิน 30 ไมโครวินาที [1] และสามารถทำงานแบบเวลาจริงได้ในคาบเวลาเพียง 50 ไมโครวินาที ในสภาวะที่มีภาระการทำงานแบบปกติจำนวนมาก เช่นการอ่านข้อมูลจากฮาร์ดดิสก์, การใช้งาน X-windows, การเรียกโปรแกรม browser สำหรับ WWW เป็นต้น

3.4. การจัดการอินเทอร์รัพใน RT-Linux

โดยปกติการควบคุมการอินเทอร์รัพในเครื่องคอมพิวเตอร์รุ่น x86 จะมีแฟลกที่บอกสถานะของการเปิดรับอินเทอร์รัพ เรียกว่า แฟลกยอมรับอินเทอร์รัพ (Interrupt Enable Flag) และมีคำสั่งที่เกี่ยวข้อง 3 คำสั่งคือ 1) *cli* ใช้ในการปิดรับการอินเทอร์รัพโดยตั้งค่าให้อินเทอร์รัพแฟลกมีค่าเป็น 0 2) *sti* ใช้ในการเปิดรับการอินเทอร์รัพโดยตั้งค่าให้อินเทอร์รัพแฟลกมีค่าเป็น 1 และ 3) *iret* เป็นคำสั่งที่ทำหน้าที่เก็บค่าและคืนค่าระหว่างการเรียกรูทีนรองรับอินเทอร์รัพ ทั้ง 3 คำสั่งเป็นคำสั่งเครื่อง (machine instruction)

ใน RT-Linux การควบคุมอินเทอร์รัพของ Linux ปกติจะถูกแทนที่ด้วยซอฟท์แวร์ควบคุมการอินเทอร์รัพซึ่งเลียนแบบการทำงานของฮาร์ดแวร์ประกอบด้วย 1) SFIF ตัวแปรสำหรับใช้แทนอินเทอร์รัพแฟลก 2) S_CLI คำสั่งที่ใช้ในการปิดรับการอินเทอร์รัพ 3) S_STI คำสั่งในการเปิดรับการอินเทอร์รัพ และ 4) S_IRET เลียนแบบการทำงานของคำสั่ง *iret* เมื่อมีอินเทอร์รัพเกิดขึ้น การทำงานจะสวิทช์ไปยังรูทีนรองรับภายในเครื่องเสมือน ซึ่งจะทำการตรวจสอบว่าเป็นอินเทอร์รัพที่ต้องการการทำงานแบบเวลาจริงหรือไม่ อินเทอร์รัพแบบเวลาจริงจะทำงานทันที ในขณะที่อินเทอร์รัพปกติจะถูกบันทึกไว้ในตัวแปร SFREQ เมื่อหมดความต้องการในงานแบบเวลาจริง Linux ปกติจะเริ่มทำงานอีกครั้งและถ้า SFIF มีค่าเป็น 1 Linux จะทำการจัดการกับอินเทอร์รัพที่ถูกบันทึกไว้ โดยเริ่มจากอินเทอร์รัพที่มีความสำคัญมากที่สุดก่อนจนกว่าจะครบทุกอินเทอร์รัพที่ถูกบันทึกไว้

การใช้ซอฟท์แวร์เลียนแบบควบคุมการอินเทอร์รัพ การทำงานของ Linux ปกติจะไม่สามารถปิดรับการอินเทอร์รัพได้ ทำให้อินเทอร์รัพที่ต้องการการทำงานแบบเวลาจริง ได้รับการตอบสนองอย่างทันท่วงที

3.5. การใช้งาน RT-Linux

RT-Linux เริ่มต้นพัฒนาโดย Victor Yodaiken และ Michael Barabanov ตั้งแต่ปี ค.ศ. 1995 จนในปัจจุบันเริ่มมีการใช้งานแพร่หลายมากขึ้นและมีผู้ร่วมทำการพัฒนาผ่านทางเครือข่ายอินเทอร์เน็ต [16] เป็นจำนวนมาก อาทิเช่น มอดูลสำหรับใช้งานเชมาฟอร์ [15], มอดูลสำหรับแสดงสถานะของงานเวลาจริง [17] เป็นต้น

RT-Linux ในส่วนที่พัฒนาในช่วงแรกนั้น นอกจากจะแทรกชั้นของเครื่องเสมือนไว้ภายใต้การทำงานของ Linux ปกติแล้ว ยังประกอบด้วยมอดูลอีก 2 มอดูลที่จะต้องโหลดเข้าสู่ระบบคือ มอดูลตัวจัดการงานแบบเวลาจริง (rt_prio_sched.o) และมอดูลสำหรับการใช้งานคิว (rt_fifo_new.o) โดยในหัวข้อนี้จะแสดงรายละเอียดของฟังก์ชันการใช้งานต่าง ๆ ของแต่ละส่วน รวมทั้งตัวอย่างการเขียนโปรแกรมสำหรับงานแบบเวลาจริงแบบง่าย ๆ และในตอนท้ายจะกล่าวถึงปัญหาในการใช้ของ RT-Linux และบทสรุป

3.5.1 เครื่องเสมือน

ในชั้นของเครื่องเสมือนประกอบด้วยฟังก์ชันการใช้งาน ดังนี้

- int request_RTirq(unsigned int irq, void (*handler)(void));
  ขอใช้อินเทอร์รัพ irq แบบเวลาจริง โดยมี handler เป็นรูทีนรองรับ ฟังก์ชันคืนค่าเป็นจำนวนจริงบอกความสำเร็จในการทำงาน
- void free_RTirq(unsigned int irq);
  คืนค่าอินเทอร์รัพ irq ที่ใช้กลับสู่ระบบ ฟังก์ชั่นไม่ส่งค่ากลับ
- #include <asm/rt_irq.h>
  ในกรณีที่มีการใช้ฟังก์ชัน request_RTirq และฟังก์ชัน free_RTirq จะต้องใส่ไฟล์ include นี้ด้วย
- typedef long long RTIME;
  ตัวแปรสำหรับเก็บค่าเวลาเป็นหน่วยของสัญญาณอินเทอร์รัพของนาฬิกา
- #define RT_TICKS_PER_SEC 1193180LL
  กำหนดค่าสัญญาณอินเทอร์รัพของนาฬิกาต่อวินาที
- RTIME rt_get_time(void);
  แสดงเวลาปัจจุบันของระบบในหน่วยของสัญญาณอินเทอร์รัพของนาฬิกา

- int rt_request_timer(void (*fn)(void));

  ขอใช้งานฟังก์ชัน fn ตามสัญญาณนาฬิกาปลุก

- void rt_free_timer(void);

  ลบฟังก์ชันที่ทำงานตามสัญญาณนาฬิกาปลุก

- void rt_set_timer(RTIME time);

  กำหนดเวลา time ในหน่วยของสัญญาณอินเทอร์รัพจากนาฬิกา ที่ต้องการให้เกิดสัญญาณนาฬิกาปลุก

- void rt_no_timer(void);

  ลบค่าเวลาที่ตั้งไว้ของนาฬิกาปลุก

- #include <asm/rt_time.h>

  ในกรณีที่มีการใช้ค่า RTIME, RT_TICKS_PER_SEC, ฟังก์ชัน rt_get_time, rt_request_timer, rt_free_timer, rt_set_timer, rt_no_timer จะต้องใส่ไฟล์ include นี้ด้วย

3.5.2 มอดูลตัวจัดการงานแบบเวลาจริง

ประกอบด้วยฟังก์ชันและการกำหนดค่าต่าง ๆ ที่เกี่ยวกับการจัดการงาน คือ

- #define RT_LOWEST_PRIORITY 1000000

  กำหนดค่าความสำคัญต่ำที่สุดของงานแบบเวลาจริง

- typedef struct rt_task_struct RT_TASK;

  สำหรับงานเวลาจริงแต่ละงานจะต้องมี RT_TASK ซึ่งเป็น structure เก็บสถานะ ความสำคัญ และอื่น ๆ และไม่สามารถแก้ไขโดยตรงได้

- int rt_task_init(RT_TASK *task, void (*fn)(int data), int data, int stack_size, int priority);

  สร้างงานแบบเวลาจริง task ขึ้นใหม่ โดย fn จะเป็นฟังก์ชันการทำงานของงานนี้ data เป็นค่าจำนวนเต็มที่ส่งผ่านให้ฟังก์ชัน fn ในตอนเริ่มต้นงาน stack_size เป็นขนาดของ stack และ priority เป็นค่าความสำคัญซึ่งค่าสูงสุดเท่ากับหนึ่งและค่าต่ำสุดมีค่าเท่ากับ RT_LOWEST_PRIORITY

- int rt_task_make_periodic(RT_TASK *task, RTIME start_time, RTIME period);

  กำหนดให้งานแบบเวลาจริง task เป็นแบบคาบเวลา (periodic) เริ่มต้นทำงานที่เวลา start_time และทำทุกคาบเวลา period โดยหน่วยของเวลาเป็นสัญญาณอินเทอร์รัพของสัญญาณนาฬิกาทั้งคู่

- int rt_task_delete(RT_TASK *task);

  ลบงานแบบเวลาจริง task ออกจากระบบ

- int rt_task_wait(void);

  หยุดการทำงานของงานเวลาจริงแบบคาบเวลาที่เรียกใช้ฟังก์ชันนี้ จนกว่าจะถึงคาบเวลาต่อไป

- int rt_task_suspend(RT_TASK *task);

  เป็นการหยุดการทำงานของงานแบบเวลาจริง task

- int rt_task_wakeup(RT_TASK *task);

  กำหนดสถานะพร้อมทำงานให้กับงานแบบเวลาจริง task หลังจากถูกสร้างจาก rt_task_init หรือหลังจากถูกหยุดการทำงานจาก rt_task_suspended

- void rt_use_fp(int allow);

  ถ้า allow □ 0 จะอนุญาตให้ใช้การคำนวณทศนิยม (floating-point) ในงานแบบเวลาจริงได้

- #include <linux/rt_sched.h>

  ในกรณีที่มีการใช้ค่ากำหนดและฟังก์ชันต่าง ๆ ในมอดูลนี้จะต้องใส่ไฟล์ include นี้ด้วย

3.5.3 มอดูลสำหรับการใช้งานคิว

ประกอบด้วยฟังก์ชันสำหรับใช้งานคิวดังนี้

- #define RTF_NO 64

  กำหนดจำนวนทั้งหมดของคิว ค่าปกติจะเป็น 64

- int rtf_create(unsigned int fifo, int size);

  สร้างคิว fifo ขึ้นและจองหน่วยความจำ size ไบต์ ถ้าทำสำเร็จฟังก์ชันจะส่งค่า size กลับ แต่ถ้าทำไม่สำเร็จจะคืนค่า -1

- int rtf_destroy(unsigned int fifo);

  ลบคิว fifo ออกจากระบบ ถ้าทำสำเร็จฟังก์ชันจะคืนค่า 0

- int rtf_resize(unsigned int minor, int size);

  เปลี่ยนขนาดของคิว minor เป็น size โดยข้อมูลที่อยู่ในคิวจะถูกลบไปด้วย ฟังก์ชันคืนค่า size ในกรณีที่ทำสำเร็จ ถ้าทำไม่สำเร็จจะคืนค่าเป็นลบ

- int rt_fifo_put(unsigned int fifo, char * buf, int count);

  เขียนข้อมูลที่ชี้โดยพอยท์เตอร์ buf ลงสู่คิว fifo เป็นจำนวน count ไบต์ คืนค่า –1 ถ้าที่ว่างในคิวเหลือน้อยกว่า count ในกรณีที่ทำสำเร็จคืนค่า count

- int rt_fifo_get(unsigned int fifo, char * buf, int count);

  อ่านข้อมูลจากคิว fifo เป็นจำนวน count ไบต์เก็บลงสู่ buf คืนค่า –1 ถ้ามีข้อมูลน้อยกว่า count ไบต์ในคิว ในกรณีที่ทำสำเร็จคืนค่า count

- int rtf_create_handler(unsigned int fifo, int (*handler)(unsigned int fifo))

  กำหนดให้รูทีน handler เป็นรูทีนรองรับคิว fifo ซึ่งจะถูกเรียกใช้งานทุกครั้งที่มีการอ่านหรือเขียนคิว

- #include <linux/rtf.h>

  ในกรณีที่มีการใช้ค่ากำหนดและฟังก์ชันต่าง ๆ ในมอดูลนี้จะต้องใส่ไฟล์ include นี้ด้วย

3.5.4 ตัวอย่างการโปรแกรมงานแบบเวลาจริง

งานแบบเวลาจริงที่ต้องการจะถูกเขียนขึ้นในมอดูลใหม่ ในลักษณะของฟังก์ชัน อาจเป็นการใช้งานอินเทอร์รัพแบบเวลาจริงหรืองานเวลาจริงแบบคาบเวลาก็ได้ การใช้งานทำได้โดยทำคำสั่งบน shell ของ Linux เพื่อบรรจุมอดูลเข้าสู่ระบบ (insmod) และทำคำสั่ง shell เพื่อถอดมอดูลออกจากระบบ (rmmod) เมื่อไม่ต้องการใช้งาน นอกจากนี้ถ้ามีการใช้งานคิวระหว่างงานแบบเวลาจริงและงานปกติจะต้องทำการสร้าง device รองรับด้วยคำสั่ง mknod ซึ่งรายละเอียดของคำสั่งทั้ง 3 มีดังนี้

- # insmod modulename

  เป็นการสั่งบรรจุมอดูล modulename เข้าสู่ระบบ เช่นก่อนการใช้งานตัวจัดการงานแบบเวลาจริงจะต้อง insmod rt_prio_sched.o และก่อนการใช้งานคิวจะต้อง insmod rt_fifo_new.o เป็นต้น

- # rmmod modulename

  เป็นคำสั่งในการถอดมอดูลออกจากระบบ

- # mknod name type major minor

  เป็นคำสั่งที่ต้องใช้ในกรณีที่โปรแกรมปกติต้องการใช้งานคิว เช่นต้องการใช้คิวที่ 1 จะต้องทำคำสั่ง mknod /dev/rtf1 c 63 1 ที่บรรทัดรับคำสั่งก่อนการใช้งานโปรแกรม สำหรับคิวที่ ?? มีรูปแบบดังนี้ mknod /dev/rtf?? c 63 ??

ตัวอย่างของการใช้งานแบบง่าย ๆ ประกอบด้วยงานแบบเวลาจริง 2 งานคือ fun_p เป็นงานแบบคาบเวลาและ fun_i เป็นงานแบบอินเทอร์รัพ โดย fun_p จะทำหน้าที่ตั้งค่าตัวแปร count = 0 และเมื่อครบคาบเวลา (100 ไมโครวินาที) ก็จะเก็บค่า count ลงคิวที่ 1 และ fun_i ทำหน้าที่นับจำนวนการเกิดอินเทอร์รัพ 12 โดยการเพิ่มค่า count ทีละหนึ่ง นอกจากนี้ในมอดูลจะต้องประกอบด้วยฟังก์ชัน init_module ซึ่งถูกเรียกใช้งานตอนถูกบรรจุเข้าสู่ระบบ ถ้าฟังก์ชันนี้ไม่คืนค่า 0 มอดูลจะถูกถอดออกจากระบบทันทีและ cleanup_module ซึ่งจะถูกเรียกใช้งานตอนถูกถอดออกจากระบบในกรณีปกติ

โปรแกรม simple.c

```
/* ไฟล์ include สำหรับโปรแกรมที่เขียนขึ้นเป็นมอดูล */
#define __KERNEL__
#define __RT__
#define MODULE
#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/module.h>
/* ไฟล์ include สำหรับฟังก์ชันการใช้งานของ RT-Linux */
#include <linux/rt_sched.h>        /* เนื่องจากใช้งานฟังก์ชันของตัวจัดลำดับงาน */
#include <linux/rtf.h>             /* เนื่องจากมีการใช้งานคิว */
#include <asm/rt_irq.h>            /* เนื่องจากมีการใช้งานอินเทอร์รัพ */
/* กำหนดให้ task1 เป็นงานแบบเวลาจริง */
RT_TASK task1;
int count;
/*
 *  ฟังก์ชันสำหรับการทำงานเวลาจริงแบบคาบเวลา
 */
void fun_p(int a) {
    int count;
    while(1) {
        count = a;
        rt_task_wait();       /* รอการทำงานในรอบต่อไป */
        rtf_put( 1, &count, sizeof(count));  /* เก็บค่า count ลงคิวที่ 1 */
    }
}
/*
 *  ฟังก์ชันสำหรับการทำงานเวลาจริงแบบอินเทอร์รัพ
 */
void fun_i(void) {
    count++;        /* เพิ่มค่า count ทุกครั้งที่มีการอินเทอร์รัพ */
}
```

```
/*
 *  ฟังก์ชันเริ่มต้นของมอดูล
 */
int init_module(void) {
    /* กำหนด stime และ period เป็นตัวแปรเวลาในหน่วยสัญญาณอินเทอร์รัพนาฬิกา */
    RTIME stime, period;
    /* กำหนดงานแบบเวลาจริง task1 ด้วยฟังก์ชัน fun_p ส่งผ่านค่า 0 ให้แก่ฟังก์ชัน มีค่า
       stacksize = 1000 และมีความสำคัญเท่ากับ 10 (1 สูงสุด และ 1000000 ต่ำสุด) */
    rt_task_init( &task1, fun_p(data), 0, 1000, 10);
    /* stime = เวลาในอีก 1 มิลลิวินาทีข้างหน้า */
    stime = rt_get_time + RT_TICKS_PER_SEC/1000;
    /* period = เวลา 100 ไมโครวินาที */
    period = RT_TICKS_PER_SEC/10000;
    /* กำหนดให้งาน task1 เริ่มต้นทำงานเวลา stime และมีคาบเวลาเท่ากับ period */
    rt_task_make_periodic( &task1, stime, period);
    /* ขอใช้งานอินเทอร์รัพ 12 โดยมีฟังก์ชัน fun_i เป็นรูทีนรองรับ*/
    request_Rtirq(12, fun_i);
    rtf_creat( 1, 40);  /* สร้างคิว 1 ขนาด 40 ไบต์ */
    return 0;          /* ถ้าไม่คืนค่า 0 มอดูลจะถูกลบออกจากระบบ */
}
/*
 *  ฟังก์ชันที่ทำงานเมื่อใช้คำสั่ง rmmod ที่ shell ของ Linux
 */
void cleanup_module(void) {
    rtf_destroy(1);  /* ลบคิว 1 ออกจากระบบ */
    free_RTirq(12);  /* คืนอินเทอร์รัพ 12 ให้กับระบบ */
}
/* จบโปรแกรม */
```

เมื่อต้องการใช้งานโปรแกรม simple.c จะต้องแปลโดยตัวแปลโปรแกรม ดังนี้

```
    #  gcc -I{RTLINUX_PATH}/include -c simple.c
```

และบรรจุมอดูลด้วยคำสั่ง insmod simple.o

3.5.5 ปัญหาในการใช้งานและการแก้ไข

จากการทดลองใช้งานและการติดตามรายการจดหมายอิเลคทรอนิค [16]พบว่า มีปัญหาในการใช้งานพอสมควรเช่นเดียวกับระบบปฏิบัติการโดยทั่วไป โดยส่วนใหญ่จะมีการเสนอวิธีการแก้ปัญหาผ่านทางอินเทอร์เน็ต ปัญหาที่เด่นชัดที่สุดที่พบคือ การใช้งานหน่วยคำนวณทศนิยมแบบพลวัต (Floating Point Unit–FPU) ภายในงานแบบเวลาจริง สาเหตุของปัญหาเกิดจากการเก็บค่าและคืนค่าสถานะของ FPU ในการสับเปลี่ยนงาน (context switching) ระหว่างงานแบบเวลาจริงกับงานปกติ

การแก้ปัญหาดังกล่าวมีผู้เสนอแนวทางการแก้ไข 3 แนวทางใหญ่ ๆ คือ 1) การปรับปรุงการเก็บค่าและคืนค่าสถานะของ FPU ใหม่ [18] 2) การใช้งานไลบรารีสำหรับคำนวณทศนิยมแบบจุดคงที่ (fixed point library) [19] และ 3) การใช้งานตัวจำลอง FPU (Floating point Emulation) โดยทั้ง 3 วิธีมีข้อเปรียบเทียบกันดังนี้คือ วิธีที่ 1 เป็นการแก้ปัญหาที่ต้นเหตุ ทำให้ไม่ต้องเปลี่ยนแปลงมากนัก และสามารถใช้งานได้ดี อย่างไรก็ตามก็ยังมีปัญหาอยู่บ้างในบางกรณีแต่น้อยมาก วิธีที่ 2 และ 3 เป็นการหลีกเลี่ยงการใช้งาน FPU ซึ่งเป็นการแก้ปัญหาที่ปลายเหตุ ในวิธีที่ 3 ไม่ต้องแก้ไขโปรแกรมมากนัก แต่ไม่เป็นที่นิยมเพราะการคำนวณจะช้าลงมาก และสำหรับในวิธีที่ 2 การคำนวณจะทำได้เร็วกว่าแบบที่ 3 เพราะตำแหน่งของจุดทศนิยมคงที่ แต่จะช้ากว่าแบบแรก และยังต้องทำการแก้ไขโปรแกรมในส่วนที่คำนวณทศนิยมทั้งหมดอีกด้วย วิทยานิพนธ์นี้เลือกใช้วิธีที่ 1

จากการพิจารณาและทดสอบในเบื้องต้น สรุปได้ว่า RT-Linux สามารถทำงานแบบเวลาจริงได้ โดยสามารถแก้ปัญหาของ Linux (แสดงในหัวข้อ 3.2) ดังต่อไปนี้คือ ลดความล่าช้าในการตอบสนองต่อการอินเทอร์รัพแบบเวลาจริงจากวิธีการเครื่องเสมือน เพิ่มความละเอียดและความถูกต้องของเวลาโดยใช้การขัดจังหวะสัญญาณนาฬิกาแบบตั้งเวลา (one-shot timer) รวมทั้งการแยกการจัดลำดับงานแบบเวลาจริงไว้ภายในเครื่องเสมือน ทำให้สามารถสับเปลี่ยนงานแบบเวลาจริงได้อย่างรวดเร็ว และในขณะที่ไม่มีงานแบบเวลาจริงอยู่ในระบบจะมีผลกระทบต่องานปกตินอ้ยมาก สำหรับในบทต่อไปจะกล่าวถึงการพัฒนาซอร์ฟแวร์ตัวควบคุมโดยตรง ซึ่งทำงานบน RT-Linux ต่อไป