การออกแบบและพัฒนาอินเตอร์พรีตเตอร์ภาษา CL

THE DESIGN AND DEVELOPMENT OF A CL INTERPRETER

สุรชัย ล้อเจริญ
SURACHAI LOCHAROEN

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์
บัณฑิตวิทยาลัย
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ. 2559

การออกแบบและพัฒนาอินเตอร์พรีตเตอร์ภาษา CL

THE DESIGN AND DEVELOPMENT OF A CL INTERPRETER

สุรชัย ล้อเจริญ

SURACHAI LOCHAROEN

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์

บัณฑิตวิทยาลัย

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2550

THE DESIGN AND DEVELOPMENT OF A CL INTERPRETER

SURACHAI LOCHAROEN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF ENGINEERING IN COMPUTER ENGINEERING

SCHOOL OF GRADUATE STUDIES

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2007

หัวข้อวิทยานิพนธ์	การออกแบบและพัฒนาอินเตอร์พรีตเตอร์ภาษา CL
นักศึกษา	นาย สุรชัย ล้อเจริญ
รหัสนักศึกษา	45061034
ปริญญา	วิศวกรรมศาสตร์มหาบัณฑิต
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
พ.ศ.	2550
อาจารย์ผู้ควบคุมวิทยานิพนธ์	ผศ. ดร. วิศิษฏ์ หิรัญกิตติ

บทคัดย่อ

CL (Communication Language) เป็นภาษาสำหรับสั่งงานคอมพิวเตอร์ให้ทำงานตามกำหนดเวลาและตามการเกิดของเหตุการณ์ ภาษา CL ที่ได้ออกแบบไว้ตาม [1,2,3] นั้นยังมีข้อจำกัดสำคัญคือไม่สามารถรองรับคำสั่งตามกำหนดเวลาที่แปรเปลี่ยนไปตามค่าตัวแปรได้ ซึ่งอาจเป็นลักษณะของการสั่งงานให้คอมพิวเตอร์ทำงานเป็นคาบๆ ซ้ำๆ หรือการสั่งงานตามกำหนดเวลาที่ถูกกำหนดด้วยผลการทำคำสั่งที่มีมาก่อนหน้า นอกจากนี้อินเตอร์พรีตเตอร์ CL เดิมยังขาดการใช้ Scheduler เพื่อจัดลำดับการทำงานของคำสั่ง โดยเฉพาะกับกลุ่มคำสั่งที่ทำงานในเวลาเดียวกัน ซึ่งทำให้ไม่มีประสิทธิภาพในการประมวลผลคำสั่ง CL ดังนั้นในวิทยานิพนธ์นี้ได้เสนอวิธีการขยายขีดความสามารถของภาษา CL และปรับปรุงอินเตอร์พรีตเตอร์ ด้วยวิธีการแทนโครงสร้างข้อมูลภายในที่ใช้เก็บตารางลำดับการทำงานของของคำสั่งในโปรแกรม CL ในรูป Tree ซึ่งช่วยให้สามารถเก็บตารางการทำงานของโปรแกรมได้อย่างไม่จำกัด สามารถรองรับกำหนดเวลาในรูปแบบตัวแปรได้ สำหรับในการปรับปรุงอินเตอร์พรีตเตอร์ เราได้ออกแบบสถาปัตยกรรมและพัฒนาอินเตอร์พรีตเตอร์ที่มีการใช้ Scheduler สำหรับการจัดลำดับการทำงานและการประมวลผลคำสั่งที่เก็บอยู่ในตารางเวลา ซึ่งทำให้อินเตอร์พรีตเตอร์สามารถประมวลผลชุดคำสั่งที่เรียงลำดับและแบบทำงานพร้อมกันได้อย่างมีประสิทธิภาพ เพื่อเป็นการประเมินประสิทธิภาพของวิธีการที่นำเสนอ เราได้นำบางส่วนของวิธีการจัดเก็บและใช้งานตารางเวลาไปเปรียบเทียบกับวิธีที่ใช้โดย Timer ในลีนุกซ์ ผลการเปรียบเทียบพบว่า การจัดการตารางเวลาของ CL สามารถจัดเก็บ และค้นคืนกำหนดเวลาโดยใช้เวลาน้อยเท่าๆ กันในทุกกรณี ในขณะที่ Timer สามารถจัดเก็บกำหนดเวลาได้รวดเร็วกว่า แต่อาจต้องเสียเวลาการค้นคืนกำหนดเวลามากกว่าในภายหลัง โดยเวลาการค้นคืนจะเพิ่มขึ้นอย่างมากเมื่อกำหนดเวลาที่ต้องค้นคืนอยู่ไกลออกไปในอนาคต ด้วยเหตุนี้จึงทำให้วิธีการจัดการกำหนดเวลาของ Timer มีประสิทธิภาพสูงกว่าของ CL ในช่วงการอ้างอิงเวลาสั้นๆ ไม่เกิน 2 ชั่วโมงเท่านั้น นอกจากนี้วิธีของ CL ยังใช้จัดการกำหนดเวลาได้ทั้งในอดีต ปัจจุบัน และอนาคตอย่างไม่จำกัด ในขณะที่ของ Timer จะใช้ได้กับกำหนดเวลาปัจจุบันและอนาคตที่จำกัดเท่านั้น

Thesis Title	The Design and Development of a CL Interpreter
Student	Mr. Surachai Locharoen
Student ID.	45061034
Degree	Master of Engineering
Programm	Computer Engineering
Year	2007
Thesis Advisor	Dr. Visit Hirankitti

ABSTRACT

CL (Communication Language) is a language for programming a computer to perform instructions according to schedules and take an action in respond to an event. CL as designed and developed in [1, 2, 3] still has some limitations, two of which, rather significant ones, are addressed here. The first one is the un-support of variables to use for specifying an instruction schedule; without this CL would allow a periodic task to specify as well as a task whose schedule was determined by an execution outcome of some previous statements. The other limitation is the lack of a proper task scheduler to use by the CL interpreter for running concurrent instructions efficiently. Hence, our aim is to remove these limitations. We first proposed an internal tree-like data structure for storing a CL program with its schedule. Its merit is it can store a CL program and its schedule with an unlimited size, while instruction schedules in the program can also be specified by variables. Secondly, we improved the interpreter by redesigning its architecture to make use of a task scheduler which can efficiently access and process schedules stored in the data structure. The improved interpreter can now interpret sequential and/or concurrent instructions more efficiently. To evaluate performance of the proposed method, we selected the part of scheduler to compare with the timer in Linux. The comparison result is the following. The CL scheduler can insert and retrieve any schedule by spending only a fragment of time whilst the timer can insert a schedule faster but spend more time to retrieve it later, especially when the schedule referring a time point far beyond in the future, even more time will be needed to retrieve it. The timer out-performs the scheduler only when it handles schedules referring time points in the range within 2 hours; when beyond that range the scheduler is better. Additionally, the scheduler can handle a large number of schedules covering the past, present, and future, whilst the timer can handle a small amount of schedules covering over only the present and the future.

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

สารบัญรูป

รูปที่ หน้า

สารบัญรูป (ต่อ)

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

การที่เราสามารถโปรแกรมให้คอมพิวเตอร์ ทำกิจกรรมต่างๆ ตามกำหนดเวลา และเหตุการณ์นั้น จะเป็นประโยชน์อย่างมาก เพราะจะเป็นการลดภาระของคนในการควบคุมคอมพิวเตอร์ตลอดเวลา ให้เป็นหน้าที่ของเครื่องคอมพิวเตอร์เอง เพื่อที่เราจะได้มีเวลามากขึ้นในการทำงานที่ต้องอาศัยความคิดสร้างสรรค์ ด้วยความต้องการให้เราสามารถสั่งงานคอมพิวเตอร์ในลักษณะข้างต้น นำไปสู่การพัฒนาเว็บบราวเซอร์ที่สามารถโปรแกรมกิจกรรมการสื่อสารได้ [1, 2, 3, 4, 5] และภาษา CL [6, 7]

ภาษา CL ถูกพัฒนาขึ้น เพื่อเป็นภาษาระดับสูงสำหรับสั่งงานคอมพิวเตอร์ให้ทำงานตามกำหนดเวลาและเหตุการณ์ สามารถประยุกต์ใช้กับงานได้หลายอย่าง เช่น ใช้ควบคุมเวปบราวเซอร์ ใช้เป็นโปรแกรมจัดการเวลา (Organizer) หรือ ใช้เป็น Workflow Engine ที่ควบคุมลำดับการทำงานร่วมกันของกลุ่มคนได้

การทำงานที่กล่าวมา จะไม่สามารถเป็นจริงได้ ถ้าขาดอินเตอร์พรีตเตอร์ CL ดังนั้นการออกแบบอินเตอร์พรีตเตอร์ CL จึงเป็นปัญหาหนึ่งที่มีความน่าสนใจ

แม้ว่าอินเตอร์พรีตเตอร์ CL จะถูกออกแบบแล้วใน [6, 7] แต่ลักษณะโครงสร้างอินเตอร์พรีตเตอร์ยังทำงานไม่สมบูรณ์นัก เราพบว่ามันยังมีข้อจำกัด ดังนี้

1. ไม่สามารถจัดการเธรดได้อย่างมีประสิทธิภาพ (ใช้เธรดเป็นจำนวนมากในการประมวลผล)
2. ไม่รองรับประโยคคำสั่ง บางแบบ เช่น ประโยคคำสั่งที่มีกำหนดเวลาที่แปรเปลี่ยนไปตามค่าตัวแปร และประโยคคำสั่งที่มีการทำงานเป็นคาบเวลา

ดังนั้นเราจึงออกแบบอินเตอร์พรีตเตอร์ CL ใหม่ เพื่อแก้ไขข้อจำกัดดังกล่าว

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

จุดมุ่งหมายของวิทยานิพนธ์นี้คือออกแบบอินเตอร์พรีตเตอร์ CL ใหม่โดยใช้โครงสร้างข้อมูลภายในที่ใช้เก็บลำดับการทำงานของโปรแกรม CL ให้รองรับกำหนดเวลาของคำสั่งที่ระบุด้วยตัวแปรได้ รวมทั้งพัฒนาอินเตอร์พรีตเตอร์เพื่อประมวลผลคำสั่งที่มีอยู่ใน CL

1.3 สมมติฐานของการศึกษา

ในงานวิจัยนี้ เรามีสมมติฐานว่า

"อินเตอร์พรีตเตอร์ CL จะต้องมี โครงสร้างข้อมูลภายในสำหรับเก็บลำดับการทำงานของโปรแกรม CL ซึ่งต้องสามารถเก็บได้ทั้งการทำงานบนช่วงเวลาและเหตุการณ์บนจุดเวลา รวมทั้งสามารถแทนเส้นแกนเวลาได้ทั้งหมด"

อินเตอร์พรีตเตอร์ในปัจจุบันยังไม่สามารถรองรับการเขียนคลาสอย่างที่ปรากฏในภาษาไพธอน เนื่องจากในภาษา CL ยังไม่มีความสามารถดังกล่าว

1.4 ทฤษฎีหรือแนวคิดที่ใช้ในงานวิจัย

เราอาศัยแนวคิดการศึกษาความรู้จากธรรมชาติรอบๆ ตัวเราในชีวิตประจำวัน โดยเราสังเกตว่าคนเรามักจดจำกำหนดการนัดหมายต่างๆ ในรูปของจุดเวลา ปี เดือน วัน ชั่วโมง นาที วินาที ลดหลั่นกันไปตามลำดับ และเป็นหน่วยเวลาที่สัมพันธ์กับการเปลี่ยนแปลงของดวงดาว ซึ่งส่งผลทั้งทางตรงและทางอ้อมต่อการดำเนินชีวิตของเรา ดังนั้นในการออกแบบโครงสร้างข้อมูลสำหรับเก็บลำดับการทำงานของโปรแกรม CL เราจึงเลือกเก็บจุดเวลาที่อยู่ในรูปแบบดังกล่าว

ในบางครั้งเราสังเกตว่า เรามักมีกำหนดการการทำงานที่เป็นค่าเป็นตัวแปร หรือการทำงานที่มีกำหนดเวลาเปลี่ยนแปลงเป็นคาบเวลา ซึ่งกำหนดการของการทำงานเหล่านั้นควรถูกจัดเก็บลงไปในโครงสร้างข้อมูลได้

1.5 ขอบเขตของการวิจัย

1. อินเตอร์พรีตเตอร์ CL ใหม่ สามารถประมวลผลคำสั่งได้อย่างมีประสิทธิภาพมากกว่าอินเตอร์พรีตเตอร์ CL เดิม
2. อินเตอร์พรีตเตอร์ CL ใหม่ สามารถประมวลผลคำสั่งที่มีกำหนดเวลาเป็นคาบเวลา และเป็นตัวแปรได้
3. โครงสร้างข้อมูลสำหรับเก็บโปรแกรม CL รองรับประโยคทุกรูปแบบ

1.6 ประโยชน์ที่เกิดขึ้นจากงานวิจัย

1. ได้อินเตอร์พรีตเตอร์ที่มีความสมบูรณ์มากขึ้น
2. ได้โครงสร้างข้อมูลสำหรับเก็บตารางเวลา

1.7 ขั้นตอนของการศึกษา

การทำงานแบ่งออกเป็น 3 ส่วน ดังนี้

1. กำหนดปัญหา

1.1. ศึกษาภาษา CL และโครงสร้างอินเตอร์พรีตเตอร์ CL จากงานวิจัยเดิมที่มีมาก่อน

1.2. ใช้วิธีคิดเชิงวิจารณ์ เพื่อหาข้อจำกัดใน CL เดิม เราพบว่าเราต้องการปรับปรุงการจัดลำดับการทำงานของโปรแกรม CL ให้มีประสิทธิภาพมากขึ้น นำไปสู่การออกแบบโครงสร้างข้อมูลใหม่

2. ตั้งสมมติฐาน

2.1 ศึกษาคุณลักษณะของเวลา จุดเวลา ช่วงเวลา และระยะเวลา

2.2 สำรวจโครงสร้างข้อมูลที่มีอยู่ในปัจจุบันว่ามีโครงสร้างข้อมูลใดที่เหมาะสมในการเก็บลำดับการทำงานของโปรแกรม CL

2.3 วิเคราะห์ข้อดีข้อเสียของโครงสร้างข้อมูลแต่ละแบบ

2.4 ออกแบบโครงสร้างข้อมูลขึ้นมาใหม่ โดยรวบรวมข้อดีของโครงสร้างข้อมูลแต่ละแบบเข้าด้วยกัน และคำนึงถึงคุณสมบัติของเวลา

3. ทดสอบสมมติฐาน

3.1 ทดสอบความถูกต้องของโครงสร้างข้อมูล โดยการจัดเก็บ Statement แต่ละแบบเข้าไป

3.2 ทดสอบการประมวลผล Statement แต่ละแบบ

3.3 เปรียบเทียบประสิทธิภาพของโครงสร้างข้อมูล กับโครงสร้างข้อมูลสำหรับเก็บตารางเวลาในลีนุกซ์

1.8 ภาพรวมของเนื้อหาบทต่างๆ ในวิทยานิพนธ์

เนื้อหาในวิทยานิพนธ์นี้แบ่งเป็นบทได้ทั้งหมด 9 บท โดยเริ่มว่าด้วยเรื่องของงานวิจัยที่เกี่ยวข้อง **ในบทที่ 2** ซึ่งเกี่ยวข้องกับทฤษฎีเวลา และภาษาสำหรับ Real-time programming ต่อด้วยรายละเอียดภาษาไพธอนอันเป็นรูปแบบภาษาพื้นฐานของ CL และเป็นภาษาที่เราใช้สร้างอินเตอร์พรีตเตอร์ภาษา CL **ในบทที่ 3** หลังจากนั้นจึงกล่าวถึงภาษา CL เริ่มตั้งแต่ ไวยกรณ์ของภาษา รูปแบบของเวลา ตลอดจนความหมายของแต่ละ Statement **ในบทที่ 4** ตามมาด้วยรายละเอียดของโครงสร้างข้อมูลสำหรับเก็บลำดับการทำงานของโปรแกรม CL **ในบทที่ 5** จากนั้นเราจะอธิบายการนำโครงสร้างข้อมูลมาใช้ในอินเตอร์พรีตเตอร์ภาษา CL **ในบทที่ 6** ต่อด้วยการเปรียบเทียบอินเตอร์พรีตเตอร์ CL กับ Timer ในลีนุกซ์ **ในบทที่ 7** จากนั้นจะแสดงผลการทดลอง**ในบทที่ 8** และสุดท้ายเป็นสรุปผลการวิจัยและข้อเสนอแนะ **ในบทที่ 9**

บทที่ 2
งานวิจัยที่เกี่ยวข้อง

ภาษา CL เป็นภาษาที่นำเวลาและเหตุการณ์ มาเป็นเงื่อนไขในการประมวลผล ซึ่งเกี่ยวข้องกับงานวิจัยในเรื่องของเวลาและสถาปัตยกรรมของอินเตอร์พรีตเตอร์ โดยเราได้ทำการอ้างอิงถึงงานวิจัยที่เกี่ยวข้องกับรูปแบบเวลาและความสัมพันธ์ของเวลาในทางคณิตศาสตร์ในหัวข้อที่ 2.1 ต่อจากนั้นจะขอกล่าวถึงการคำนวณแบบเรียบไทม์ในหัวข้อที่ 2.2 ตามมาด้วยเทคนิคการจัดทำ Timer แบบต่างๆ ที่มีอยู่ในปัจจุบันในหัวข้อที่ 2.3 และสรุปในหัวข้อที่ 2.4

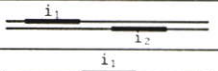
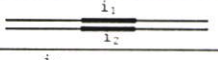
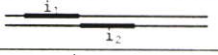
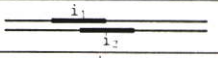
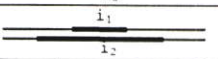
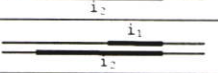
2.1 รูปแบบของเวลา และความสัมพันธ์ของเวลาแบบต่างๆ

เนื่องจากเวลาเกี่ยวข้องกับความรู้ การทำงานและเหตุการณ์ นักวิจัยหลายท่านจึงให้ความสนใจในการกำหนดรูปแบบของเวลา ในแบบคณิตศาสตร์ ซึ่งคำถามที่นักวิจัยให้ความสนใจคือ

1. ชนิดข้อมูลเวลาพื้นฐาน ของเวลา ควรเป็นเช่นไร เช่น ควรจะใช้ จุดเวลาเพียงอย่างเดียว หรือ ช่วงเวลาเพียงอย่างเดียว หรือทั้งสองอย่าง
2. ความสัมพันธ์ของเวลา

Allen [8, 9, 10] สังเกตว่าการทำงาน (Action) และเหตุการณ์ (Event) ใดๆ เกิดขึ้นบนช่วงเวลาเสมอ เช่น การรับประทานอาหารกลางวัน อาจใช้เวลาในช่วงเวลา 12:30-13:00 เหตุการณ์อ๊อดให้สัญญาณเข้าทำงานอยู่ในช่วงเวลาเล็กๆ 12:59-13:00 ดังนั้นเขาจึงเสนอให้ใช้ช่วงเวลา เป็นชนิดข้อมูลพื้นฐานสำหรับการสร้างทฤษฎีเวลา จากนั้น Allen ได้สร้างความสัมพันธ์ของช่วงเวลาขึ้น 13 ความสัมพันธ์ ดังนี้

ตารางที่ 2.1 ความสัมพันธ์ระหว่างช่วงเวลากับช่วงเวลา

ความสัมพันธ์	ลักษณะความสัมพันธ์
1. $i_1 < i_2$ (i_1 before i_2)	
2. $i_1 = i_2$ (i_1 equal i_2)	
3. $i_1\ m\ i_2$ (i_1 meets i_2)	
4. $i_1\ o\ i_2$ (i_1 overlaps i_2)	
5. $i_1\ d\ i_2$ (i_1 during i_2)	
6. $i_1\ s\ i_2$ (i_1 starts i_2)	
7. $i_1\ f\ i_2$ (i_1 finishes i_2)	
ความสัมพันธ์ที่เหลือจะเป็นความสัมพันธ์แบบตรงกันข้ามกับความสัมพันธ์ในหมายเลข 2, 3, 4, 5, 6,7	

จากความสัมพันธ์นี้ Allen ได้อธิบายพฤติกรรมของเวลาด้วย Axiom ต่อไปนี้

1. สองช่วงเวลาใดๆ จะมีความสัมพันธ์หนึ่ง ในตารางความสัมพันธ์ที่กำหนดเสมอ
2. ความสัมพันธ์มีความเป็นอิสระจากกัน (mutually exclusive) เช่น ถ้า i_1 before i_2 แล้ว i_1 ไม่สามารถมีความสัมพันธ์อื่น กับ i_2 ได้
3. ความสัมพันธ์สามารถส่งต่อได้ (Transitive) เช่น ถ้า i_1 before i_2 and i_2 meets i_3 แล้ว i_1 before i_3

แนวความคิดของ Allen ไม่ได้ให้ความสำคัญกับจุดเวลา โดยมองว่าจุดเวลาถูกแทนที่ด้วยช่วงเวลาสั้นๆได้ ซึ่งต่อมาถูก Antony Galton [11] วิจารณ์ว่า จุดเวลามีความแตกต่างจากช่วงเวลา และในหลายสถานการณ์เราใช้จุดเวลาอ้างอิงกับความรู้เช่น "อุณหภูมิของหม้อต้มน้ำมีค่า 90 องศาเซลเซียส ณ เวลา 9:00 น." ซึ่งกล่าวถึง อุณหภูมิของหม้อต้มน้ำที่จุดเวลาเดียวเท่านั้น

การอธิบายความรู้ด้วยช่วงเวลาเพียงอย่างเดียวยังก่อให้เกิดความคลุมเครืออีกด้วย เช่น ถ้าเรามีความรู้ว่า "ประตูเข้าห้องสมุดเปิดในช่วงเวลา 8:00 – 19:30" เราจะไม่สามารถรู้ได้แน่ชัดว่าประตูถูกเปิดเมื่อไร

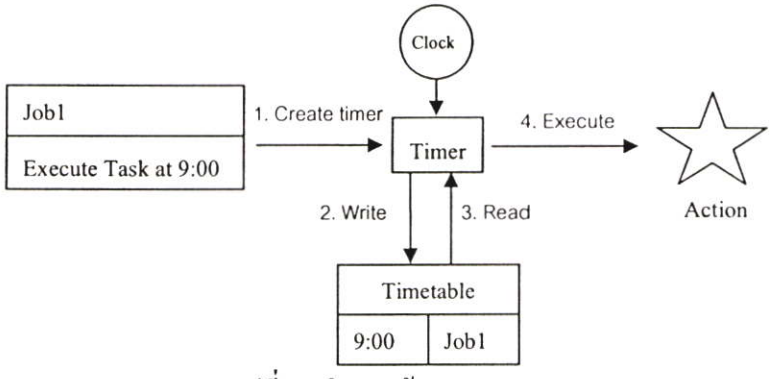
ในงานวิจัยของ Antony Galton [12] จึงได้มีการนำเสนอทฤษฎีเวลาใหม่ โดยใช้จุดเวลาและช่วงเวลา เป็นชนิดข้อมูลพื้นฐานในการสร้างทฤษฎี จุดเวลาและช่วงเวลาที่เขานำเสนอมีลักษณะคล้ายจุดและช่วงในระบบจำนวนจริง และเป็นทฤษฎีเวลาที่ใกล้เคียงกับที่ใช้ในภาษา CL มากที่สุด

2.2 การคำนวณแบบเรียลไทม์

ในทางวิทยาการคอมพิวเตอร์ Real-time computing (RTC) [13, 14] เป็นการศึกษาในเรื่องระบบฮาร์ดแวร์และระบบซอฟท์แวร์ ซึ่งอยู่ภายใต้จำกัดของเวลา กล่าวคือ การเริ่มประมวลผลการทำงาน และการสิ้นสุดทำงานต้องไม่เกินเวลาที่กำหนด ระบบที่สามารถตอบสนองอย่างรวดเร็วเพียงอย่างเดียวไม่ถือเป็นระบบ Real-time ระบบ Real-time มีความคล้ายคลึงกับ CL Interpreter

ระบบ Real-time เป็นสิ่งที่ต้องคำนึงถึงในการสร้างแอพพลิเคชั่นที่มีความสำคัญสูง เช่น ในระบบป้องกันการล็อกล้อของเบรกรถยนต์ ที่ต้องควบคุมการบังคับล้อให้หยุดในช่วงเวลาสั้นๆ เพื่อป้องกันรถเสียหลัก ระบบ real-time จะต้องทำงานให้เสร็จก่อนถึงจุดเวลาเส้นตาย แม้ว่าจะมี Load เท่าไรก็ตาม

การควบคุมเวลาเริ่มงาน และเวลาสิ้นสุดงาน ระบบจำเป็นต้องอาศัย Timer ในการตั้งเวลา โดย Timer ทำงานเป็น Background Process คอยรับคำสั่งการทำงานจากผู้ใช้ แล้วจึงบันทึกจุดเวลาสำหรับเริ่ม และสิ้นสุดการทำงานในตารางเวลา จากนั้น Timer จะอ่านเวลาจากนาฬิกามาเป็นระยะๆ เพื่อตรวจสอบลำดับการทำงานที่จัดเก็บ ว่ามีงานถึงเวลาประมวลผลแล้วหรือไม่ ถ้าพบว่ามีมันก็จะนำงานนั้นมาประมวลผล



รูปที่ 2.1 โครงสร้างของ Timer

Timetable มีลักษณะเหมือนเป็นพจนานุกรมของ tuple (จุดเวลา, งาน) ที่มีจุดเวลาเป็น key ในการค้นหาข้อมูล

โครงสร้างข้อมูลของ Timetable มีผลต่อประสิทธิภาพและวิธีการทำงานของ Timer ซึ่งเราจะได้กล่าวถึงโครงสร้างข้อมูลแบบต่างๆ ในหัวข้อต่อไป

2.3 โครงสร้างข้อมูลสำหรับเก็บตารางเวลา

ในบทความของ George Varghese และ Anthony Lauck [15] ได้จัดแบ่งประเภทของโครงสร้างข้อมูลสำหรับเก็บตารางเวลาและได้จัดลำดับไล่เรียงตามประสิทธิภาพได้ดังนี้

2.3.1 ตัวแปรนับถอยหลัง

เป็นโครงสร้างข้อมูลที่ง่ายที่สุด จุดเวลาถูกแทนด้วยตัวแปรที่มีค่าเริ่มต้นเป็นระยะเวลาหมดอายุ จัดเก็บร่วมกับฟังก์ชั่นไว้ในลิสต์

การทำงานของ Timer นี้จะเป็นแบบนับถอยหลัง โดย Timer จะคอยลดค่าของตัวแปรในลิสต์ครั้งละ 1 เป็นระยะๆ ตัวแปรที่มีค่าเป็น 0 ก็จะนำฟังก์ชั่นที่เก็บคู่กับตัวแปรนั้นมาประมวลผล

การจัดเก็บจุดเวลาไว้ในโครงสร้างเวลาเช่นนี้ มีข้อดีตรงที่ การอิมพลีเมนต์อัลกอลิธึมสำหรับเพิ่มจุดเวลา ลบจุดเวลา ค้นหาจุดเวลา ลดค่าจุดเวลา สามารถทำได้ง่าย การเพิ่มจุดเวลากระทำได้รวดเร็ว ด้วยเวลา O(1) ส่วนข้อเสียของโครงสร้างข้อมูลนี้คือ เวลาสำหรับลบจุดเวลา ค้นหาจุดเวลา ลดค่าตัวแปร ต้องใช้เวลา O(n)

2.3.2 Order List

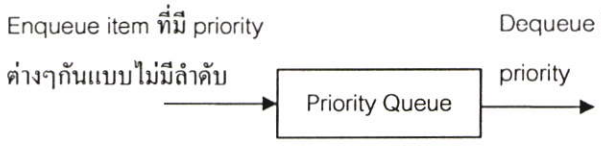
เป็นการจัดเก็บจุดเวลาแบบ Order List จุดเวลาถูกอ้างแบบสมบูรณ์ (มีจุดอ้างอิงเดียวกัน) จุดเวลาจะถูกจัดเก็บไว้ในลิสต์เรียงตามลำดับค่าของจุดเวลาจากน้อยไปหามาก

การทำงานของ Timer จะอ่านค่าจุดเวลาจากหัวของลิสต์ มาเปรียบเทียบกับค่าเวลาปัจจุบัน เป็นระยะๆ เมื่อใดก็ตามที่ค่าเวลาตรงกับเวลาปัจจุบันแล้ว Timer ก็จะตัดหัวของลิสต์มาประมวลผล

ข้อดีของโครงสร้างข้อมูลแบบนี้คือ การอ่านจุดเวลามาประมวลผล ใช้เวลาประมาณ O(1) ส่วนข้อเสียคือ การเพิ่มจุดเวลาเข้าไปในลิสต์ต้องใช้วิธีค้นหาตำแหน่งที่อยู่ของข้อมูลก่อน ซึ่งต้องใช้เวลาประมาณ O(n)

2.3.3 Priority Queue

Priority Queue เป็น Abstract Data Type แบบหนึ่งที่อนุญาตให้เรา enqueue item ด้วยค่า priority ต่างๆ กันแบบไม่เป็นลำดับ และเราสามารถ dequeue item ออกมาตามลำดับของ priority ได้ภายหลัง ดังรูป



รูปที่ 2.2 Priority Queue

เราสามารถประยุกต์ใช้ Priority Queue ในการเก็บจุดเวลาได้ โดยให้จุดเวลาเป็นค่า priority

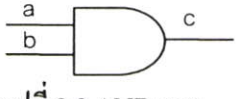
Timer สามารถอ่านค่าจุดเวลาออกมาตามลำดับ ด้วยวิธีการ dequeue เพื่อใช้ตรวจสอบกับเวลาปัจจุบันเป็นระยะ เมื่อพบว่ามีงานที่ได้เวลาแล้ว ก็ประมวลผลงานนั้น

อิมพลีเมนต์ของ Priority Queue แบบหนึ่งที่นิยมใช้กันคือใช้เป็นโครงสร้าง Tree ที่เรียงตัวแบบ Heap มีข้อดีตรงที่ การอ่านจุดเวลามาประมวลผลใช้เวลาประมาณ O(log n) และการเพิ่มจุดเวลาใช้เวลา O(log n) แต่มีข้อเสีย ตรงที่ Tree อาจจะไม่สมดุล เช่นในกรณีที่เราเพิ่มจุดเวลาเรียงลำดับกันจะทำให้ โครงสร้าง Tree มีลักษณะเป็นลิสต์ ซึ่งทำให้ประสิทธิภาพของ Tree ลดต่ำลง

2.3.4 Timing Wheel

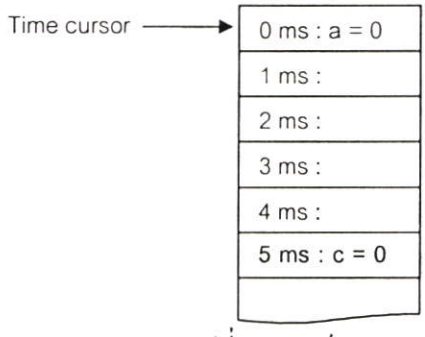
ในงาน Discrete Event Simulation โปรแกรมที่ใช้ซิมมูเลทจะต้องมีความสามารถในการสร้าง event ตามกำหนดเวลา ซึ่งโปรแกรมจะมี Timetable เช่นเดียวกันกับ Timer

ลักษณะการทำงานของโปรแกรมจะเป็นการจำลองเหตุการณ์ขึ้นมา และ Schedule เหตุการณ์จะเกิดขึ้นตามมา ไว้ในตารางเวลา ตัวอย่างเช่น การซิมมูเลท แอนเกท ดังรูป



รูปที่ 2.3 AND เกท

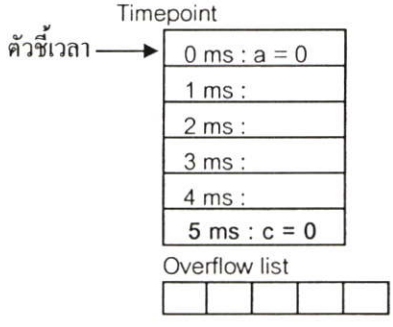
ถ้า AND เกทมีเวลาหน่วง 5 ms และมีเหตุการณ์ว่า a = 0 ที่ 0 ms โปรแกรมจะต้อง schedule เหตุการณ์ c = 0 ณ เวลา 5 ms เพื่อรอกำหนดค่าให้กับตัวแปร c ต่อไป ซึ่งโปรแกรมจะทำงานคล้ายกับการเคลื่อนตัวชี้เวลาไปบนลิสต์ของเหตุการณ์ ดังรูป



รูปที่ 2.4 ลิสต์ของเหตุการณ์

การเคลื่อนที่ของ ตัวชี้เวลาจะเคลื่อนที่ตามเวลา ถ้าหนึ่งช่องของเหตุการณ์ แทน 1 ms ตัวชี้เวลาจะเลื่อนไป 1 ช่อง ต่อ 1 มิลลิวินาที

เราสามารถใช้อาเรย์ ในการสร้างโครงสร้างข้อมูลแบบนี้ได้ แต่ขนาดของอาเรย์ต้องมีขนาดใหญ่มากๆ ดังนั้นในโปรแกรมจึงต้องปรับเปลี่ยนการเก็บจุดเวลาให้ใช้อาเรย์ที่มีขนาดคงที่และลิสต์แทน โดยจัดแบ่งเวลาออกเป็น ช่วงเวลาเท่าๆ กัน แล้วเก็บช่วงเวลาที่ครอบคลุมจุดเวลาปัจจุบันไว้ในอาร์เรย์ ส่วนจุดเวลาที่อยู่ในช่วงเวลาอื่นๆ ถัดไปจะเก็บรวมกันไว้เป็นลิสต์ (Overflow list) ตัวอย่างเช่น ในตัวอย่างก่อนหน้านี้ ถ้าเรากำหนดให้ช่วงของเวลามีขนาด 5 ms เราก็จะใช้อาเรย์ขนาด 5 ช่องสำหรับแทน 1 ช่วงเวลา และเก็บเหตุการณ์ที่อยู่ในช่วงเวลาอื่นๆ ในรูปของลิสต์ ดังรูป



รูปที่ 2.5 โครงสร้างข้อมูลแบบ Timing Wheel

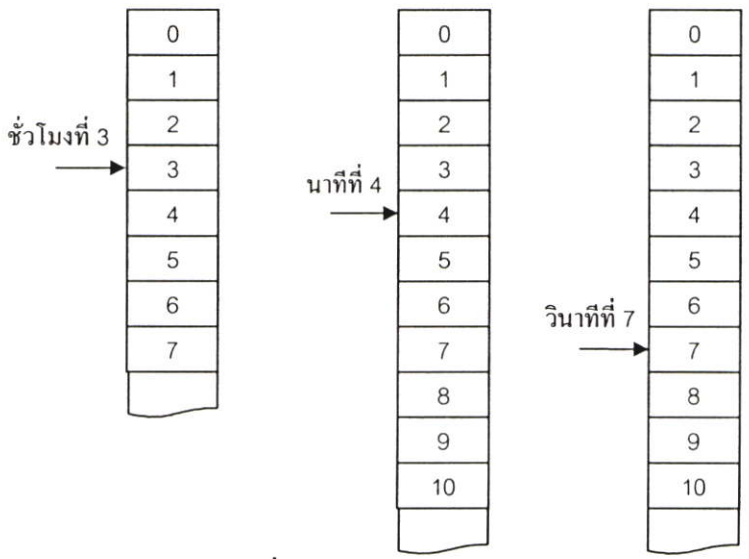
เมื่อใดก็ตามที่ ตัวชี้เวลาสแกนไปจนครบช่องในอาร์เรย์แล้ว โปรแกรมจะต้องอ่านค่าจุดเวลาในช่วงเวลาถัดไปจาก Overflow list มาเติมลงในช่องของอาเรย์ใหม่

ข้อดีโครงสร้างข้อมูลนี้คือ การใช้หน่วยความจำได้มีประสิทธิภาพ การเพิ่มจุดเวลาหรือการอ่านเวลาตามลำดับในช่วงเวลาที่อยู่ในอาร์เรย์ สามารถกระทำได้ ในเวลา O(1) ซึ่งถือว่าเป็นเวลาที่ดีที่สุด แต่เวลาดังกล่าวจะเกิดขึ้นกับกรณีที่ Schedule เหตุการณ์ที่เกิดขึ้นภาย ช่วงเวลาเดียวกันเท่านั้น ไม่เหมาะที่จะมา Schedule เหตุการณ์ที่เกิดใน จุดเวลาถัดไปไกลๆ เพราะจะทำให้จุดเวลาถูกจัดเก็บลงไปใน Overflow list และต้องเสียเวลาในการย้ายจุดเวลาภายหลัง การลบหรือการค้นหาจุดเวลาที่อยู่ใน Overflow list จะต้องค้นหาทั่วทั้งลิสต์ ซึ่งต้องใช้เวลาถึง O(n)

2.3.5 Hierarchical Timing Wheel

ในหัวข้อที่แล้ว จุดเวลาถูกจัดกลุ่มเป็นช่วงๆ ในหัวข้อนี้ได้มีการพัฒนาเพิ่มเติมให้มีการจัดกลุ่มของช่วงเป็น Hierarchy ตัวอย่างที่เห็นชัดๆ ก็ได้แก่การแบ่งช่วงเวลาออกเป็น ชั่วโมง นาที วินาที เป็นต้น

โครงสร้างข้อมูลสำหรับเก็บจุดเวลาที่มีการจัดกลุ่มของช่วงเวลาเป็น Hierarchy เราจำเป็นต้องใช้ อาร์เรย์เท่ากับจำนวนของระดับชั้นของช่วงเวลา และแต่ละลำดับของช่วงเวลาจะมีตัวชี้เวลากำกับอยู่ ตัวอย่างเช่น ถ้าเรามีการแบ่ง hierarchy ของช่วงเวลาออกเป็น ชั่วโมง นาที วินาที เราต้องใช้ อาร์เรย์สามอัน สำหรับเก็บ ลำดับช่วงเวลาของ ชั่วโมง นาที และวินาที ตามลำดับ และอาศัยตัวชี้เวลาสามอัน เพื่อชี้ที่ตำแหน่งช่วง ถ้าสมมติว่าเวลาปัจจุบันเป็น 03:04:07 เราจะได้ว่าตัวชี้เวลาจะชี้ช่องของอาเรย์ ชั่วโมงที่ 3 นาทีที่ 4 และ วินาทีที่ 7 ดังรูป



รูปที่ 2.6 Hierarchical Timing Wheel

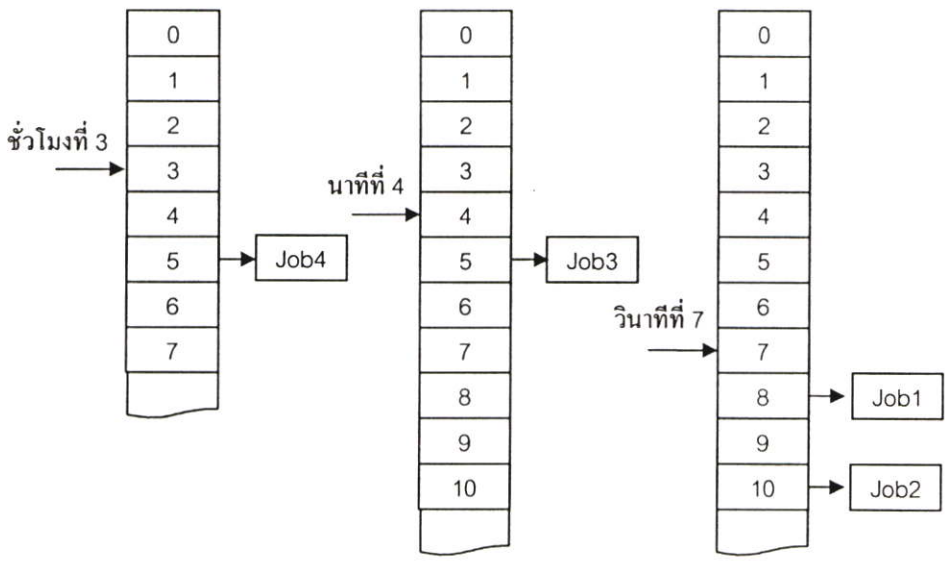
การเคลื่อนที่ของตัวชี้เวลาจะเคลื่อนที่จากช่วงเวลาเล็กไปหาช่วงเวลาใหญ่ จากตัวอย่าง ตัวชี้เวลาในระดับวินาทีจะค่อยๆเพิ่มค่าที่ละ 1 วินาที จนครบ 60 วินาที ก็จะวนกลับมาชี้ที่ 0 วินาทีอีกครั้งหนึ่ง และจะทำให้ตัวชี้เวลาในระดับนาทีเพิ่มขึ้น 1 นาที

การจัดเก็บจุดเวลา จะจัดเก็บจุดเวลาสัมพันธ์กับตัวชี้เวลา เช่นจากรูปที่ 6 ตัวชี้เวลาชี้อยู่ที่ ชั่วโมงที่ 3 นาทีที่ 4 วินาทีที่ 7 จุดเวลาที่อยู่ในช่วง นาทีที่ 4 ของชั่วโมงที่ 3 จะเก็บอยู่ใน อาร์เรย์วินาที คำถามต่อมาคือ แล้วจุดเวลาที่เกินช่วงนาทีที่ 4 ไปจะเก็บไว้ที่ใด คำตอบก็คือ เก็บไว้เป็น Overflow list ในอาเรย์นาทีในช่องที่ 5, 6, 7,...,59 ตามค่านาทีของจุดเวลานั้น ซึ่งทั้งหมดจะแทนได้หมดชั่วโมงที่ 3 พอดี และเช่นเดียวกันถ้าเราต้องการเก็บจุดเวลาที่เกินชั่วโมงที่ 3 ไปเราสามารถเก็บจุดเวลาเหล่านั้นไว้ใน Overflow list ในอาเรย์ชั่วโมงในช่องที่ 4,5,6...24 ตามค่าชั่วโมงของจุดเวลานั้น

ตัวอย่างเช่น สมมติว่าเราใช้ช่วงเวลา hierarchy แบบ ชั่วโมง นาที และ วินาที และเวลาปัจจุบันเป็น 03:04:07 น. ลองเพิ่มจุดเวลาต่อไปนี้

(03:04:08, job1), (03:04:10, job2), (03:05:09, job3), (05:05:10, job4)

จะได้โครงสร้างข้อมูลดังรูป



รูปที่ 2.7 การเติมข้อมูลเข้าไปในโครงสร้างข้อมูล Hierarchical Timing Wheel

จุดเวลาของ Job1 และ Job2 อยู่ในช่วงชั่วโมงที่ 3 นาทีที่ 4 ซึ่งจุดเวลาที่อยู่ภายในช่วงนี้ถูกโหลดเข้าไว้ในอาร์เรย์วินาทีแล้ว ดังนั้นเราจึงจัดเก็บ Job1 และ Job2 ไว้ในอาร์เรย์ ที่ช่องที่ 8 และ 10 ตามค่าวินาทีของจุดเวลานั้น

ต่อมาจุดเวลาของ Job3 มีค่า ชั่วโมงที่ 3 นาทีที่ 5 วินาทีที่ 9 ซึ่งจุดเวลานี้เกินช่วงนาทีที่ 4 มาแล้ว ดังนั้นเราจึงเก็บ Job3 ไว้เป็น Overflow list ในอาร์เรย์นาทีช่องที่ 5 ตามค่านาทีของจุดเวลานั้น

จุดเวลาของ Job4 มีค่า ชั่วโมงที่ 5 นาทีที่ 5 วินาทีที่ 10 ซึ่งจุดเวลา นี้ถือเกินช่วงชั่วโมงที่ 3 มาแล้ว ดังนั้นเราจึงจัดเก็บ Job4 ไว้เป็น Overflow list ในอาร์เรย์ชั่วโมงช่องที่ 5 ตามค่าชั่วโมงของจุดเวลานั้น

ในทุกครั้งที่มีการเคลื่อนตำแหน่งของตัวชี้เวลา Overflow list จะต้องถูกโหลดเข้าไปในอาร์เรย์ของช่วงเวลาที่เล็กลงไป เช่น ในตัวอย่าง ถ้าตัวชี้เวลานาทีที่ 4 เลื่อนมาชี้ที่นาทีที่ 5 แล้ว Job3 จะต้องถูกย้ายมาไว้ในอาร์เรย์วินาที ที่ช่องที่ 9 ตามค่าวินาทีของจุดเวลา

ข้อดีของโครงสร้างข้อมูลนี้คือ

1. เราสามารถจัดเก็บจุดเวลาได้เป็นจำนวนมาก ด้วยหน่วยความจำกัด เช่น ถ้าเราใช้อาร์เรย์เดียวในการแทนเวลาใน 1 วันเราต้องใช้ อาร์เรย์ที่มีขนาดถึง 24*60*60 แต่ถ้าเราใช้วิธีการจัดเก็บ hierarchy ของช่วงเวลา เราจะใช้อาร์เรย์เพียง 24 + 60 + 60 เท่านั้น
2. การย้ายข้อมูลใน Overflow list ทำได้รวดเร็วขึ้น เนื่องจากมีการจัดเรียงเป็น hierarchy

ข้อเสียของโครงสร้างข้อมูลนี้คือ

1. การจัดเก็บจุดเวลาที่อยู่ห่างไกล หลายๆ จุด จะทำให้จุดเวลาเหล่านั้นถูกเก็บใน Overflow list เดียวกัน ซึ่งทำให้การค้นหา การลบจุดเวลา ใช้เวลามากถึง O(n)
2. การย้ายข้อมูลจาก Overflow list ไปเก็บในอาร์เรย์ ต้องใช้เวลาเพิ่มขึ้นมา นอกเหนือจากเวลาที่ใช้ในการเคลื่อนตัวชี้เวลา

โครงสร้างข้อมูลแบบ Hierarchy Timing Wheel เป็นที่นิยมใช้ในการสร้าง Timer รวมทั้ง Timer ในลีนุกซ์ ซึ่ง Timer เหล่านั้นไม่ต้องการการจัดเก็บจุดเวลาในระยะเวลาที่ไกลมากนัก แต่โปแกรมที่เขียนด้วย CL มีการหน่วงเวลาการทำงานเป็นเวลานาน มีโอกาสจัดเก็บจุดเวลาที่กระจายอยู่ทั่วไปในทุกเวลา และต้องการจัดเก็บจุดเวลาจำนวนมาก ดังนั้นโครงสร้างข้อมูลแบบ Hierarchy Timing Wheel จึงไม่เหมาะสมอีกต่อไป เพื่อแก้ปัญหานี้เราจึงออกแบบโครงสร้างข้อมูลสำหรับจัดเก็บจุดเวลาขึ้นใหม่ ซึ่งรายละเอียดจะได้กล่าวต่อไป

2.4 สรุป

ในบทนี้เราได้รวบรวมงานวิจัยที่เกี่ยวข้อง เพื่อช่วยให้ผู้อ่านเข้าใจ วิวัฒนาการของการศึกษาเรื่องเวลา และ Timer ในบทต่อไปจะกล่าวถึง ตัวภาษา CL ในด้านไวยากรณ์ และความหมายของประโยคแต่ละแบบต่อไป

บทที่ 3
ภาษาไพธอน

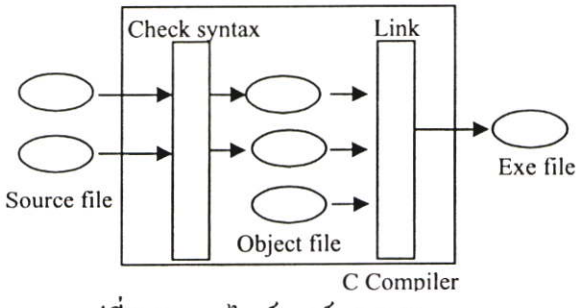
ในบทนี้จะกล่าวถึงภาษาที่ใช้สร้างภาษา CL คือ ภาษาไพธอน (Python) เหตุที่เลือกใช้ภาษาไพธอนนั้น เพราะภาษาไพธอนเป็นภาษาที่มีการทำงานแบบอินเตอร์พรีตเตอร์ (Interpreter) เหมาะสำหรับแนวคิดที่ต้องมีการตอบสนองต่อเหตุการณ์ภายนอกของภาษา CL อีกทั้งภาษาไพธอนยังมีไวยากรณ์ที่เข้าใจได้ง่าย ทำให้การพัฒนาภาษา CL เป็นไปได้สะดวกยิ่งขึ้น

เนื้อหาในบทจะเริ่มอธิบายหลักการโปรแกรมคอมพิวเตอร์ในหัวข้อที่ 3.1 หลังจากนั้นจะกล่าวถึงที่มาของภาษาไพธอนในหัวข้อ 3.2 ต่อด้วยความสามารถของภาษาไพธอนในหัวข้อที่ 3.3 จากนั้นเราจะอธิบายรายละเอียดของภาษาไพธอนในหัวข้อที่ 3.4 และสรุปในหัวข้อที่ 3.5

3.1 หลักการเขียนโปรแกรมคอมพิวเตอร์

การเขียนโปรแกรม คือการอธิบายความคาดหวังของโปรแกรมเมอร์ ออกมาเป็นข้อความคอมพิวเตอร์ที่รับข้อความนั้นมา ต้องมีความเข้าใจถึงความหมายของสัญลักษณ์ต่างๆ ในข้อความนั้นเพื่อแปลความหมายออกมาเป็นการทำงาน ซึ่งวิธีการประมวลผลโปรแกรมคอมพิวเตอร์ที่ใช้กันอยู่ในปัจจุบัน แบ่งออกได้เป็น

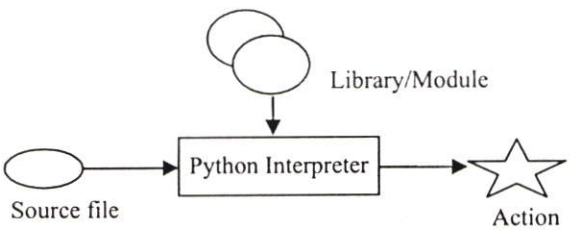
1. คอมไพเลอร์ (Compiler) เป็นวิธีการประมวลผลโปรแกรมที่แปลงโปรแกรม ให้เป็นสัญญาลักษณ์ที่เครื่องคอมพิวเตอร์เข้าใจก่อน หลังจากนั้นจึงนำสัญญาลักษณ์ที่ได้มาประมวลผล ตัวอย่างเช่น การประมวลผลโปรแกรมภาษา C, C++, Pascal คอมไฟล์เลอร์จะตรวจสอบความผิดพลาดของโค้ดคำสั่งตั้งแต่ต้นจนจบก่อน ถ้าไม่มีข้อผิดพลาดก็จะทำการแปลโค้ดคำสั่งให้เป็นไฟล์นามสกุล .obj (object file) จากนั้นก็ทำการแปลไฟล์ .obj ให้เป็นไบนารีไฟล์ .exe เพื่อทำงานต่อไป การทำงานของคอมไพเลอร์ภาษา C เป็นดังรูป



รูปที่ 3.1 คอมไพล์เลอร์ภาษา C

การใช้คอมไฟเลอร์ มักมีปัญหาในการย้ายโปรแกรมไปทำงานบนเครื่องที่แตกต่างกัน เพราะคอมไฟเลอร์จะแปลงโปรแกรมเป็นไฟล์ไบนารีที่ทำงานบนเครื่องเดียวได้เท่านั้น หากต้องการใช้บนเครื่องอื่นต้องหาไฟล์ซอร์สมาคอมไพล์ใหม่ ซึ่งยุ่งยาก

2. อินเตอร์พรีเตอร์ (Interpreter) จะเป็นวิธีการประมวลผลโปรแกรมที่ตีความหมายของประโยคในโปรแกรม ออกมาเป็นการทำงานทันที โดยไม่มีการแปลงโปแกรมให้อยู่ในรูปแบบอื่น ตัวอย่างเช่น ภาษาไพธอนที่ถูกอินเตอร์พรีตเป็นการทำงานทันที ดังแสดงในรูป



รูปที่ 3.2 อินเตอร์พรีตเตอร์ภาษาไพธอน

จากรูปตัวอย่าง ในกรณีที่มีการเรียกใช้ฟังก์ชั่นจากไลบรารี (Library) หรือโมดูล (Module) ของภาษาไพธอน อินเตอร์พรีเตอร์ไพธอนก็จะไปทำการเรียกฟังก์ชั่นเหล่านั้นให้ทำงาน

อินเตอร์พรีตเตอร์ช่วยอำนวยความสะดวกในการเขียนโปรแกรมให้กับโปรแกรมเมอร์มากกว่าคอมไพล์เลอร์ อินเตอร์พรีตเตอร์ช่วยซ้อนรายละเอียดด้านฮาร์ดแวร์ทั้งหมดไว้ ทำให้โปรแกรมเมอร์ไม่ต้องกังวลกับความเข้ากันได้ของ OS หรือ CPU

ในส่วนของประสิทธิภาพการทำงานนั้นการประมวลผลโปรแกรมแบบคอมไพเลอร์จะทำงานได้เร็วกว่าการประมวลผลโปรแกรมแบบอินเตอร์พรีเตอร์ เพราะโค้ดคำสั่งถูกคอมไพล์และลิงค์เป็นไฟล์ .exe ออกมา ซึ่งสามารถนำไปประมวลผลได้รวดเร็ว

3.2 ภาษาไพธอน

ไพธอน เป็นภาษาระดับสูงที่ประมวลผลด้วยอินเตอร์พรีตเตอร์ ถูกสร้างขึ้นในปี 1989 โดย Guido van Rossum ซึ่งภาษานี้พัฒนาขึ้นมาโดยไม่ยึดติดกับแพลตฟอร์ม กล่าวคือสามารถรันภาษาไพธอน ได้ทั้งบนระบบ Unix, Linux, Windows NT/2000/XP/2003, FreeBSD หรือแม้แต่ระบบ MAC OS X, Palm, Nokia Mobile นอกจากนี้ภาษา Python ยังเป็น Open Source ที่แจกจ่ายให้ใช้งานฟรีอีกด้วย

3.3 ความสามารถของภาษาไพธอน

ภาษาไพธอนเริ่มมีผู้ใช้มากขึ้นเรื่อยๆ เนื่องจากข้อดีหลายประการของภาษาไพธอน ซึ่งพอสรุปข้อดีของภาษาไพธอนได้ดังนี้

1. **ง่ายต่อการเรียนรู้** โดยภาษาไพธอน มีโครงสร้างของภาษาไม่ซับซ้อนเข้าใจง่าย ซึ่งโครงสร้างภาษาไพธอน จะคล้ายกับภาษา C มาก เพราะภาษาไพธอน สร้างขึ้นมาโดยใช้ภาษา C ทำให้ผู้ที่คุ้นเคยภาษา C อยู่แล้วใช้งานภาษาไพธอน ได้ไม่ยาก นอกจากนี้โดยตัวภาษาเองมีความยืดหยุ่นสูงทำให้การจัดการกับงานด้านข้อความ และ Text File ได้เป็นอย่างดี
2. **ไม่ต้องเสียค่าใช้จ่ายใดๆ ทั้งสิ้น** เพราะอินเตอร์พรีตเตอร์ภาษาไพธอน อยู่ภายใต้ลิขสิทธิ์ GNU
3. **ใช้ได้หลายแพลตฟอร์ม** ในช่วงแรกภาษาไพธอน ถูกออกแบบใช้งานกับระบบ Unix อยู่ก็จริง แต่ในปัจจุบันได้มีการพัฒนาอินเตอร์พรีตเตอร์ภาษาไพธอน ให้สามารถใช้กับระบบปฏิบัติการอื่นๆ อาทิเช่น Linux, Windows 95/98/ME, Windows NT, Windows 2000, OS/2
4. **ภาษาไพธอน ถูกสร้างขึ้นโดยได้รวบรวมเอาส่วนดีของภาษาต่างๆ เข้ามาไว้ด้วยกัน** อาทิเช่น ภาษา C, C++, Java, Perl
5. **ภาษาไพธอน สามารถนำมาพัฒนาเว็บเซอร์วิส** รวมทั้งใช้บริหารการสร้างเว็บไซต์สำเร็จรูปที่เรียกว่า Content Management Framework (CMF) ตัวอย่าง CMF ที่มีชื่อเสียงมากและเบื้องหลังทำงานด้วยไพธอน คือ Plone

3.4 ไวยากรณ์ของภาษาไพธอน [16, 17]

3.4.1 ชนิดของข้อมูล

ตัวแปรในภาษาไพธอน จะเป็น Pointer ไปยังไพธอนอ๊อบเจ็ค การกำหนดชนิดของข้อมูลให้กับตัวแปรในภาษาไพธอนจะถูกกระทำโดยอัตโนมัติ ผู้เขียนโปรแกรมสามารถเปลี่ยนแปลงตัวแปรหนึ่งให้เก็บค่าที่ต่างไปได้โดยไม่ต้องสนใจชนิดของตัวแปรที่เก็บอยู่เดิม (เนื่องจากเป็นการเปลี่ยนค่า pointer) ชนิดของข้อมูลทั่วไปได้แก่

1. ตัวเลข (Numeric) ข้อมูลชนิดตัวเลข ได้แก่

- เลขฐานสิบ (Integer) คือเลขที่ประกอบด้วยเลข 0-9 ไม่ว่าจะเป็นจำนวนบวก หรือจำนวนลบ แต่ต้องไม่มีจุดทศนิยม ตัวอย่างเช่น 1, -3, 8784 ฯลฯ
- เลขฐานแปด คือ ตัวเลขที่ขึ้นต้นด้วย 0 เช่น 0715 ฯลฯ
- เลขฐานสิบหก คือ ตัวเลขที่ขึ้นต้นด้วย 0x เช่น 0x9FAC, 0x12A ฯลฯ
- เลขทศนิยม (Float number) เช่น 3.2, -45e12 ฯลฯ

- เลขจำนวนเต็มขนาดยาว (Long integer) คือตัวเลขที่ลงท้ายด้วย L หรือ l เช่น 4890549579L, 5l ฯลฯ
- เลขจำนวนเชิงซ้อน (Complex number) เช่น 9j, 5i-4j, 45i ฯลฯ

2. จำนวนทางตรรกะ (Boolean) โดยปกติจำนวนทางตรรกะจะมีค่าเท็จ หรือจริง สำหรับภาษาไพธอนจะแทนค่าเท็จด้วย เลขศูนย์, โครงสร้างเปล่า หรือ None value เช่น 0, [], {}, (), None และแทนค่าจริงด้วย ค่าใดๆที่ไม่ใช่ศูนย์, โครงสร้างที่มีข้อมูลอยู่ เช่น 1, [5], (7, 8, 98, 347), "xyz"

3. สายอักขระ (String) เป็นข้อมูลชนิดของตัวอักษร เช่น 'This is a string.', "This is another string" สายอักขระนี้จะอยู่ภายในสัญลักษณ์ Single quoted หรือ Double quoted ก็ได้ ในข้อมูลชนิดสายอักขระนั้น จะมีอักขระพิเศษอยู่ด้วย ได้แก่

\n = Newline \' = Single quoted \b = Backspace \t = Tab

\" = Double quoted \f = Formfeed \\ = Backslash \a = Bell

\r = Carriage return \v = Vertical tab

4. ลิสต์ (Lists) มีลักษณะคล้ายกับข้อมูลชนิดอาเรย์(Array) แต่จะต่างกันตรงที่ข้อมูลที่เก็บในลิสต์ไม่จำเป็นต้องเก็บข้อมูลชนิดเดียวกัน ข้อมูลที่สามารถเก็บในลิสต์ได้แก่ข้อมูลชนิดต่างๆ ของไพธอน เช่น ตัวเลข, ตัวอักษร, สายอักขระ หรือแม้แต่จะเป็นลิสต์ด้วยกันเองก็ได้ การเขียนอ้างอิงถึงข้อมูลชนิดลิสต์ทำได้ดังนี้

```
list2 = [1,"two", [3,4]]
```
เป็นการสร้างลิสต์ที่ประกอบด้วยสมาชิก 3 ตัวโดยตัวแรกเป็น ตัวเลข, ตัวที่สองเป็นสายอักขระ, ตัวที่สามเป็นลิสต์ที่มีสมาชิกเป็นตัวเลขสองตัว

การอ้างถึงสมาชิกในลิสต์ทำได้ดังนี้

```
list2[0]
```
จะได้ผลลัพธ์เป็น 1

ด้วยคุณสมบัติและลักษณะข้างต้น จึงสามารถใช้ลิสต์ในการสร้างสแตค (Stack) และคิว (Queue) ได้ ซึ่งจะมีประโยชน์ต่อการสร้างอินเตอร์พรีตเตอร์ภาษา CL (CL-Interpreter) ต่อไป

5. ทัปเปิล (Tuples) ข้อมูลชนิดทัปเปิลมีลักษณะเหมือนข้อมูลชนิดลิสต์ แต่จะต่างกันตรงที่ไม่สามารถแก้ไขข้อมูลของสมาชิกภายในทัปเปิลได้ ทัปเปิลจึงเหมาะแก่การใช้เป็นข้อมูลอ้างอิง การกำหนดทัปเปิลมีวิธีการดังนี้

```
tuple2 = ("one", 2,"three", 4)
```

การจัดการกับทัปเปิลนั้นมีวิธีการเช่นเดียวกับการจัดการกับลิสต์

6. ดิกชันนารี (Dictionary) ข้อมูลชนิดดิกชันนารีจะประกอบไปด้วยคีย์ (Keys) และค่าที่เก็บ (Values) ตัวอย่างของดิกชันนารีเป็นดังนี้

```
dict = {1: "one", 2 : "two", 3: "three"}
```

เราสามารถอ้างอิงข้อมูลในดิกชันนารีทำได้โดย

`dict[key]` จะให้ผลลัพธ์เป็นอ๊อบเจ็คที่ถูกเก็บโดยคีย์นั้น

เช่น `dict[1]` จะให้ผลลัพธ์เป็น "one" เป็นต้น

โดยปกติข้อมูลชนิดลิสต์จะสามารถเรียงลำดับข้อมูลได้ แต่ข้อมูลชนิดดิกชันนารีนั้นจะไม่มีการเรียงลำดับข้อมูล การอ้างอิงทำได้โดยผ่านทางคีย์เท่านั้น

7. ข้อมูลเปล่า (None) ข้อมูลชนิดนี้ใช้ในการระบุคาเริ่มต้นของตัวแปร หรือแสดงว่าไม่มีข้อมูล ซึ่งในการเปรียบเทียบค่านั้น None จะมีค่าเท่ากับ None เท่านั้น

3.4.2 ตัวกระทำ (Operators)

ตัวกระทำต่างๆ ในภาษาไพธอน แบ่งออกเป็น 4 ประเภทคือ ตัวกระทำทางตรรกะ, ตัวกระทำทางการเปรียบเทียบ, ตัวกระทำทางบิตไวส์ และตัวกระทำทางคณิตศาสตร์

1. ตัวกระทำทางตรรกะ (Logical operators) ตัวกระทำทางตรรกะมีทั้งหมด 3 ตัวด้วยกันคือ and, or, not

2. ตัวกระทำทางการเปรียบเทียบ (Comparison operators) ตัวกระทำที่ใช้เปรียบเทียบค่าต่างๆ ได้แก่ <, >, <=, >=, ==, <>, !=, in, not in, is, is not

3. ตัวกระทำทางบิตไวส์ (Bitwise operators) เป็นตัวกระทำที่ใช้คำนวณเลขฐานสอง ได้แก่ << (Shift left), >> (Shift right), &(and), | (or), ^ (xor), ~ (not)

4. ตัวกระทำทางคณิตศาสตร์ (Arithmatic-Style operators) ตัวกระทำทางคณิตศาสตร์นี้สามารถใช้งานไม่เพียงกับตัวเลขเท่านั้น แต่ยังสามารถใช้งานกับข้อมูลชนิดอื่นๆ เช่นสายอักขระได้ ตัวกระทำดังกล่าวได้แก่ ตัวกระทำ +, -, *, /, **, %

5. ลำดับของตัวกระทำ (Precedence) ลำดับของตัวกระทำเป็นดังเช่นโปรแกรมคอมพิวเตอร์ทั่วไปซึ่งมีลำดับดังนี้คือ or, and, not, (<, <=, ==, >=, >, !=, <>, is, in, not, not in), |, ^, &, (<<, >>), (+, -), (*, /, %), **, (unary+, unary-, unary~)

3.4.3 คำสงวน (Reserved words)

คำสงวนในภาษาไพธอนแบ่งเป็น คีย์เวิร์ด(Keywords) และบิวท์อินฟังก์ชั่น(Built-in function)

1. คีย์เวิร์ด (Keywords) ได้แก่

'and', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while'

2. บิวท์อินฟังก์ชั่น (Built-in function) ได้แก่

'import', 'abs', 'apply', 'buffer', 'callable', 'chr', 'cmp', 'coerec', 'compile', 'complex', 'delattr', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float', 'getattr', 'globals', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'intern', 'isins', 'issubclass', 'len', 'list', 'locals', 'long', 'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input', 'reduce', 'reload', 'repr', 'round', 'setattr', 'slide', 'str', 'tuple', 'type', 'vars', 'xrange'

3.4.4 ประโยคในภาษาไพธอน

1. ประโยคกำหนดค่าให้ตัวแปร (Assignment Statement)

การกำหนดค่าให้ตัวแปรใช้เครื่องหมายเท่ากับ(=) ซึ่งการกำหนดค่าให้ตัวแปรนั้นไม่ได้เป็นการนำค่าไปใส่ให้ตัวแปรจริงๆ แต่เป็นการกำหนดให้ตัวแปรอ้างอิงไปยังไพธอนอ๊อบเจ็คที่มีค่านั้นอยู่ โดยไพธอนอ๊อบเจ็คนั้นอาจถูกสร้างขึ้นมาใหม่ตอนที่กำหนดค่า หรือเป็นไพธอนอ๊อบเจ็คที่มีอยู่แล้วจากตัวแปรอื่นก็ได้

การกำหนดค่าให้ตัวแปรในภาษาไพธอน นอกจากจะกำหนดค่าหนึ่งค่าให้กับตัวแปรในรูปแบบ Var = Value แล้ว ยังสามารถกำหนดค่าหนึ่งค่าให้กับตัวแปรหลายๆ ตัวได้ เช่น Var1 = Var2 = Value หรือสามารถกำหนดค่าหลายๆ ค่าให้กับตัวแปรหลายตัวได้ด้วยประโยคเดียว ซึ่งในกรณีหลังนี้ จำนวนค่าที่กำหนดจะต้องเท่ากับจำนวนตัวแปร เช่น (Var1, Var2) = (Value1, Value2)

2. ประโยคควบคุมลำดับการทำงาน (Control Flow Statement)

คำสั่งลูปเป็นคำสั่งที่ใช้ในการควบคุมลำดับการประมวลผลของคำสั่งต่างๆ โดยให้ประมวลผลวนรอบไปเรื่อยๆจนกว่าจะถึงเงื่อนไขที่กำหนด คำสั่งลูปต่างๆได้แก่ IF, WHILE, FOR

- **ประโยค IF** การประมวลผลประโยค if นั้น จะทำการตรวจสอบเงื่อนไขซึ่งกำหนดโดย expression ถ้าเงื่อนไขถูกต้อง จึงทำสเตทเมนต์ที่กำหนด รูปแบบของคำสั่ง IF เป็นดังนี้

```
if <expression>:
      <statement>
elif <expression>:
```

```
        <statement>
else:
        <statement>
```

ตัวอย่างของการใช้คำสั่ง if แสดงได้ดังนี้

```
if  x == 0:
        print 'x equal 0'
elif x >0:
        print 'x is positive number'
else:
        print 'x is negative number'
```

- **ประโยค WHILE** ใช้สำหรับการประมวลผลแบบวนรอบ ซึ่งจะทำงานไปเรื่อยๆ จนกว่าเงื่อนไขหรือ expression จะเป็นเท็จ คำสั่ง WHILE นั้นมีการตรวจสอบเงื่อนไขก่อนการทำคำสั่งในลูปทุกครั้ง รูปแบบคำสั่ง while เป็นดังนี้

```
while <expression> :
        <statement>
```

ตัวอย่างของการใช้คำสั่ง while แสดงได้ดังนี้

```
while 1:
        print 'Infinite loop'
```

- **ประโยค FOR** เป็นประโยคที่ทำการประมวลผลแบบวนรอบจนครบตามจำนวนที่กำหนด คำสั่ง FOR เป็นคำสั่งที่จะวนรอบการทำงานจนครบจำนวนที่กำหนด รูปแบบของคำสั่ง for เป็นดังนี้

```
for <var> in <list>:
        <statement>
```

ตัวอย่างของการใช้คำสั่ง for แสดงได้ดังนี้

```
list1 = [1,'a', [b, c]]
for var1 in list1:
        print var1
```

โดยผลลัพธ์ที่ได้จากการรันคำสั่งเป็นดังนี้

```
1
'a'
[b,c]
```

โดย <RANGE> คือตัวแปรที่เป็นลิสต์ หรือคำสั่งเฉพาะบางคำสั่ง ส่วน <VARIABLE> คือตัวแปรที่เก็บค่าของสมาชิกทุกตัวของ <RANGE> การทำงานของคำสั่ง FOR จะทำงานวนรอบเป็นจำนวนเท่ากับจำนวนสมาชิกของ <RANGE>

3. ประโยคชุดคำสั่ง (Compound Statement)

การกำหนดกลุ่มของซอร์สโค้ดจะใช้การย่อหน้า(Indentation) แทนสัญลักษณ์ ซึ่งต่างจากภาษาอื่นๆ เช่น ภาษา C และ Java ใช้วงเล็บปีกกา และภาษา Pascal ใช้ begin end เป็นตัวกำหนดกลุ่มของซอร์สโค้ด

การที่ไพธอนใช้การย่อหน้าในการกำหนดกลุ่มของซอร์สโค้ดทำให้ช่วยลดความยุ่งยาก และข้อผิดพลาดที่มักเกิดขึ้นจากการลืมพิมพ์สัญลักษณ์ที่ใช้ในการกำหนดกลุ่ม และข้อดีอีกอย่างหนึ่งคือ ทำให้ซอร์สโค้ดอ่านง่ายเนื่องจากทุกครั้งที่ขึ้นกลุ่มใหม่จะต้องทำการย่อหน้าซึ่งทำให้ซอร์สโค้ดมีความเป็นระเบียบ

3.5 สรุป

ภาษาไพธอนเป็นภาษาระดับสูง ที่ถูกประมวลผลแบบอินเตอร์พรีตเตอร์ ภาษาไพธอนถูกออกแบบให้แก้ปัญหาที่พบบ่อยๆ ในการเขียนโปรแกรม ให้สามารถเขียนได้ด้วยโค้ดที่สั้น ทำให้การโปรแกรมง่าย อ่านง่าย และดูแลง่าย จากข้อดีนี้ จึงได้มีการออกแบบภาษา CL ที่สืบทอดคุณลักษณะที่ดีของไพธอนมา โดยรายละเอียดจะได้กล่าวในบทต่อไป

บทที่ 4

ภาษา CL

เนื้อหาในบทนี้จะเป็นการปูพื้นฐานความรู้เกี่ยวกับภาษา CL เพื่อให้ผู้อ่านทำความเข้าใจกับเนื้อหาในบทอื่นๆ ต่อไปได้ สำหรับผู้ที่สนใจเนื้อหาโดยละเอียดของ สามารถศึกษาเพิ่มเติมได้จากวิทยานิพนธ์ของคุณสุพรรณดา โชติพันธ์ [7] เนื้อหาในบทนี้จะขอเริ่มจากการอธิบายหลักการและที่มาของภาษา CL ในหัวข้อที่ 4.1 ตามด้วยความหมายของเวลาที่ใช้ในภาษา CL ในหัวข้อที่ 4.2 ตาม, ด้วยรูปแบบเวลา ในหัวข้อที่ 4.3 จากนั้นจะกล่าวถึงลักษณะการระบุช่วงเวลาที่ใช้ในการเขียนโปรแกรม CL ในหัวข้อ 4.4 ตามมาด้วยรายละเอียดของประโยค (Statement) ในโปรแกรมในหัวข้อที่ 4.5 ตามด้วยแนวคิดเรื่อง Scope ของตัวแปรในหัวข้อที่ 4.6 จากนั้นจะเป็นการนำเนื้อหาในบทนี้มาแสดงในรูปของโปรแกรมตัวอย่างหัวข้อที่ 4.7 และสรุปในหัวข้อที่ 4.8

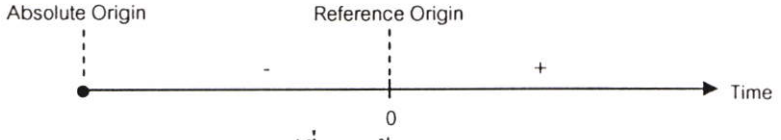
4.1 หลักการและที่มา

ภาษา CL เป็นภาษาคอมพิวเตอร์ที่ใช้สั่งงานคอมพิวเตอร์ตามเวลาและเหตุการณ์ มีจุดเริ่มต้นมาจากการพัฒนาเวปเบราซ์เซอร์ที่สามารถโปรแกรมได้ [1, 2, 3, 4, 5] และได้พัฒนาต่อเนื่องมาจนมาเป็นอินเตอร์พรีตเตอร์ CL [6,7] ในการออกแบบภาษา ผู้ออกแบบภาษาได้เล็งเห็นความสำคัญของภาษาที่มีความเป็นนามธรรมสูง และสามารถโปรแกรมได้ง่าย ดังนั้นผู้ออกแบบจึงเลือกใช้ภาษาไพธอน เป็นต้นแบบในการพัฒนา ทำให้ไวยากรณ์ของภาษา CL ในส่วนที่ไม่เกี่ยวข้องกับเวลามีลักษณะคล้ายภาษาไพธอน

CL Statement จะมีรูปแบบ **"ช่วงเวลา: ประโยค"** ซึ่งหมายถึง ประโยคจะทำงานในช่วงเวลาที่กำหนด ถ้าผู้เขียนโปรแกรมไม่ต้องการระบุช่วงเวลา ก็จะทำให้มีความหมายเช่นในประโยคในภาษาคอมพิวเตอร์ทั่วไป

4.2 ความหมายของเวลา

ผู้ออกแบบภาษา CL ได้ศึกษาธรรมชาติของเวลา และได้เสนอความคิดว่า เวลาเกิดจากการดำเนินไปของกระบวนการ (Process) อันหนึ่งอย่างต่อเนื่อง เมื่อไรที่เกิดมีกระบวนการ ก็จะเกิดเวลาขึ้น แต่ถ้าไม่มี เวลาก็ไม่ปรากฏ เนื่องจากการดำเนินไปของกระบวนการมีความต่อเนื่อง จึงสมมติให้เวลามีความต่อเนื่องด้วย โดยให้มีค่าเพิ่มขึ้นโดยต่อเนื่องอยู่ตลอดไม่ขาดตอน จึงอาจอธิบายได้โดยเส้นลูกศรในระบบ co-ordinate โดยให้เป็นแกนของเวลา ดังรูป ซึ่งถือว่าเป็นแกนที่เพิ่มเติมจากแกน xyz ในระบบเรขาคณิต ซึ่งอาจถือว่าเป็นแกนที่ 4 หรือมิติที่ 4

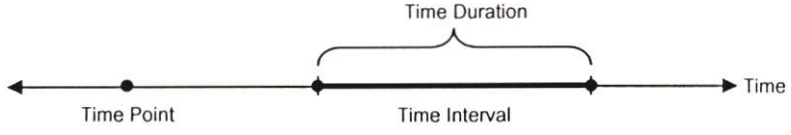


รูปที่ 4.1 เส้นแกนเวลา

ดังนั้นเวลาจะมีค่าเพิ่มขึ้นเรื่อยๆ เมื่อนับจากจุดเริ่มต้นในอุดมคติของการเกิดเวลา(Absolute Origin) แต่ในความเป็นจริงนั้น เราไม่สามารถรู้ถึงจุดกำเนิดของเวลาที่แท้จริงได้ มนุษย์จึงได้กำหนดจุดเริ่มต้นสมมุติของเวลาขึ้นมาเพื่อง่ายต่อการอ้างอิงเวลาเท่านั้น และขอเรียกจุดเริ่มต้นเวลานี้ว่า "จุดเริ่มอ้างอิง"(Reference Origin) ซึ่งในแต่ละอารยธรรมอาจมีจุดเริ่มอ้างอิงที่ต่างกันตามลัทธิความเชื่อทางศาสนา เช่น จุดเริ่มอ้างอิงตามระบบคริสตศักราช และระบบพุทธศักราช เป็นต้น ดังนั้น เวลาที่สนใจเมื่อเทียบกับจุดเริ่มอ้างอิงนี้อาจมีค่าเป็นลบได้ เช่น เมื่อกล่าวถึงเวลาก่อนคริสตศักราช (B.C.) เป็นต้น

จุดต่างๆ บนเส้นแกนเวลานั้น แสดงการอ้างอิงเวลา ณ ขณะใดขณะหนึ่ง จุดนี้เรียกว่า **จุดเวลา *(Time Point)*** โดยจุดเวลาที่อยู่ขวามือของจุดเริ่มอ้างอิง (Reference Origin) มีค่าเป็นบวก และจุดที่อยู่ซ้ายมือของจุดเริ่มอ้างอิง มีค่าเป็นลบ เช่น จุดเวลาที่ 15:00:00 น. วันที่ 10 มิ.ย. พ.ศ. 2546 เป็นจุดเวลาที่เป็นบวกเมื่อเทียบกับจุดเริ่มอ้างอิงตามระบบพุทธศักราช เป็นต้น

ในกรณีที่มีการลากเส้นเชื่อมต่อระหว่าง 2 จุดเวลา จากจุดเวลาที่น้อยกว่าซึ่งเรียกว่าจุดเริ่มต้น ไปยังจุดเวลาที่มีค่ามากกว่าซึ่งเรียกว่าจุดสิ้นสุด จะทำให้เกิด ***ช่วงเวลา(Time Interval)*** ซึ่งประกอบด้วยจุดเวลาหลายๆ จุด ที่เรียงต่อกันอย่างต่อเนื่องไม่ขาดสายจากจุดเริ่มไปจนถึงจุดสิ้นสุด โดย**ความยาวของช่วงเวลา**หรือของเส้นตรงนี้ ถือว่าเป็น ***ระยะเวลา(Time Duration)*** จะสังเกตว่าในกรณีที่จุดเริ่มต้นของช่วงเวลาอยู่ที่ตำแหน่งจุดเริ่มอ้างอิง(Reference Origin) ค่าของจุดสิ้นสุดของช่วงเวลาจะมีค่าเท่ากับระยะเวลาของช่วงเวลานั้น เช่น ณ เวลา 9:00:00 น. วันที่ 17 มิถุนายน พ.ศ. 2546 จะมองได้ว่าเป็น จุดเวลานี้ห่างจากจุดอ้างอิงเป็นระยะเวลา 2546 ปี 7 เดือน 17 วัน และ 9 ชั่วโมง ได้เช่นกัน ในทำนองเดียวกัน จะได้ว่าจุดเริ่มอ้างอิง หมายถึง จุดเวลาที่มีระยะเวลาเป็นศูนย์เมื่อเทียบกับตำแหน่งอ้างอิง หรือตัวมันเอง ดังแสดงในรูป



รูปที่ 4.2 เวลาประเภทต่างๆ บนแกนเวลา

จากแนวคิดเรื่องเวลาที่ได้อธิบายไปแล้ว หัวข้อต่อไปจะกล่าวถึงรูปแบบของเวลาที่ปรากฏในภาษา CL คือ จุดเวลา และช่วงเวลา

4.3 รูปแบบเวลา

- **จุดเวลา** มีลักษณะการระบุเวลาโดยใช้หน่วยเวลา ปี เดือน วัน ชั่วโมง นาที และวินาที การระบุถึงจุดเวลา มีรูปแบบเป็น

HH:MM:SS|dd/mm/yyyy

ตัวอย่างเช่น 7:00:00|12/12/2549 หมายถึง ปีที่ 2549 เดือนที่ 12 วันที่ 12 ชั่วโมงที่ 7 นาทีที่ 0 วินาทีที่ 0 เป็นต้น

- **ช่วงเวลา** ถูกกำหนดโดยจุดเวลาเริ่มต้น p_s และจุดเวลาสิ้นสุด p_e มีรูปแบบเป็น

$[p_s, p_e]$

ตัวอย่างเช่น [7:00:00|12/12/2549, 8:00:00|12/12/2549] หมายถึงช่วงเวลาตั้งแต่ 7 นาฬิกา ของวันที่ 12 เดือนที่ 12 ปีที่ 2550 ถึง 8 นาฬิกา ของวันเดียวกัน

4.4 การระบุช่วงเวลา $[p_s, p_e]$

มีได้ 3 รูปแบบ คือ

1. $[p_s, p_e]$ หมายถึงการระบุช่วงเวลาที่ผู้เขียนโปรแกรมต้องการกำหนดเวลาที่แน่นอนในการเริ่มต้นและสิ้นสุดการทำงานนั้น เช่น ต้องการให้คอมพิวเตอร์เริ่มดาวน์โหลดไฟล์ ณ เวลา 10.00 น. และสิ้นสุดการดาวน์โหลด ณ เวลา 11.00 น. เป็นต้น
2. $[\_,p_e]$ หรือ $[p_s,\_]$ หมายถึงจุดเวลาเริ่มต้นหรือสิ้นสุดไม่มีการระบุค่า จุดเวลาที่ไม่ได้ระบุจะถูกกำหนดโดยระบบเอง
3. [_,_] เป็นการไม่ระบุเวลาทั้งจุดเริ่มต้นและจุดสิ้นสุด แต่ปล่อยให้จุดเวลาทั้งสองถูกกำหนดโดยระบบขณะที่ประมวลผลทำให้ [_,_]:statement มีความหมายเหมือนกับประโยค ในภาษาคอมพิวเตอร์ทั่วไป ที่ไม่มีการระบุเวลาในการทำงาน

4.5 รูปแบบประโยค (Statement) ในโปรแกรม

ภาษา CL มีประโยคหลายแบบ ได้แก่ ประโยคสำหรับกำหนดค่าให้ตัวแปร, ประโยคเรียกใช้ฟังก์ชั่น, ประโยคสำหรับควบคุมลำดับการทำงานในโปรแกรม ประโยคชุดคำสั่ง และประโยคการตอบสนองเหตุการณ์ โดยอธิบายได้ดังนี้

4.5.1 ประโยคสำหรับการกำหนดค่าให้ตัวแปร (Assignment Statement) การกำหนดค่าให้ตัวแปรใช้โอเปอเรเตอร์กำหนดค่า ('=') ด้วยรูปแบบ

<variable-name> = <expression>

4.5.2 ประโยคเรียกใช้ฟังก์ชั่น มีรูปแบบ

<function-name>(<argument>...)

4.5.3 ประโยคการตอบสนองเหตุการณ์ อยู่ในรูปแบบ

เหตุการณ์ → การทำงาน

หรือ Event → Action

โดยที่ "เหตุการณ์" ในที่นี้ คือ Expression หรือ ประโยคคำสั่งที่ให้ค่าออกมาเป็นค่า Boolean "การทำงาน" ในที่นี้ คือ ประโยคคำสั่งใดๆ สำหรับ "เหตุการณ์ → การทำงาน" หมายความว่า ไม่ว่าเมื่อไรก็ตามที่ "เหตุการณ์" (Event) ได้ค่า Boolean เป็น True ออกมา อินเตอร์พรีตเตอร์ภาษา CL จะลงมือกระทำ "การทำงาน" (Action) ทันที

4.5.4 ประโยคสำหรับควบคุมลำดับการทำงานในโปรแกรม (Control Statement) ใน CL มีประโยคสำหรับควบคุมลำดับการทำงานของโปรแกรมทั้งหมด 4 รูปแบบ คือ

- if-statement มีรูปแบบ

  if <expression>: <statementA>

  [else: <statementB>]

- for-statement มีรูปแบบ

  for <control-variable> in <list>: <statement>

- while-statement มีรูปแบบ

  while <expression>:

  <statement>

- repeat-until-statement มีรูปแบบ

  repeat : <statement>

  until <expression>

4.5.5 ประโยคชุดคำสั่ง เกิดจากการนำประโยคแบบต่างๆ มารวมกัน ประโยคชุดคำสั่งแบ่งเป็น 4 แบบ คือ แบบ Sequence, Unordered, Parallel และ Nested

- **Sequential statement** เป็นชุด statement ที่ต้องทำงานเรียง ลำดับทีละอันจากบนลงล่าง โดย statement หลังจะเริ่มทำงานได้ก็ต่อเมื่อ statement ก่อนหน้าทำงานเสร็จสิ้นแล้ว มีรูปแบบ

sequential:

$[s_1,e_1]$:statement$_1$

$[s_2,e_2]$:statement$_2$

.

.

.

ในกรณีที่เรากำหนดจุดเวลาเริ่มต้นหรือสิ้นสุดแบบไม่ระบุให้กับประโยคในชุดคำสั่ง sequential ได้แก่ [_,_]:statement จะทำให้ '_' ที่เป็นจุดเวลาเริ่มต้นหมายถึง "the time **after** the execution of the previous statement" และ '_' ที่เป็นจุดเวลาสิ้นสุดหมายถึง "the time after the execution of this statement is **finished**" ต่อจากนี้สำหรับประโยค sequential บางครั้งเราอาจใช้สัญลักษณ์ 'af' แทน '_' ตัวหน้า และ 'f' แทน '_' ตัวหลัง เพื่อให้สื่อความหมายที่ชัดเจน

นอกจากนี้จุดเวลาเริ่มต้นและหรือจุดเวลาสิ้นสุดของประโยคในชุดคำสั่ง Sequential ยังสามารถกำหนดเป็นค่าตัวแปรได้อีกด้วย โดยค่านี้จะถูก evaluate เมื่อคำสั่งก่อนหน้าการใช้งานตัวแปร ถูกประมวลผลไปแล้ว

- **Unordered statement** เป็นชุด statement ที่ลำดับการทำงานของ statement ย่อย ไม่มีการระบุแน่นอน ส่วนการทำงานของกลุ่ม statement นี้ จะทำได้ทีละหนึ่งเท่านั้น มีรูปแบบ

unordered:

[_,_]:statement$_1$

[_,_]:statement$_2$

.
.
.

- **Parallel statement** เป็นชุด statement ที่แต่ละ statement ย่อย ทำงานไปพร้อมๆ กัน มีรูปแบบ

parallel:

[s_1,e_1]:statement$_1$

[s_2,e_2]:statement$_2$

.
.
.

นอกจากนี้ ชุดคำสั่งทั้ง 3 ยังสามารถใช้ร่วมกันในลักษณะ **Nested** ได้ด้วย

4.6 Scope ของตัวแปรในภาษา CL

ขอบเขตของตัวแปรในภาษา CL มีกำหนดเช่นเดียวกับภาษาคอมพิวเตอร์ทั่วๆไป โดยแบ่งเป็น

1. Global Scope เป็น Scope ระดับบนสุด (Highest level) ของโปรแกรมหรือโปรแกรมหลัก ซึ่งตัวแปรใน Scope นี้จะสามารถอ้างอิงถึงได้ทั้งในโปรแกรมหลัก และโปรแกรมย่อย (ฟังก์ชัน)

2. Local Scope เป็น Scope ที่จำกัดขอบเขตการอ้างถึงตัวแปรอยู่บางส่วนของโปรแกรม ตัวแปรที่เก็บอยู่ใน Local Scope จะถูกอ้างได้จากคำสั่งที่อยู่ใน scope เดียวกันเท่านั้น

4.7 ตัวอย่างโปรแกรมภาษา CL

สมมติว่าเราต้องการสั่งงานให้คอมพิวเตอร์เริ่มเล่นเพลงในช่วงเวลา 20 ถึง 21 น. จากนั้นเปิดโปรแกรม browser แสดงหน้าเว็บข่าวในช่วง 22 ถึง 23 น. และปิดเครื่องตอน 2 น. ตลอดเวลา ตั้งแต่ 22 น. จนถึงวันรุ่งขึ้น ให้รอรับเหตุการณ์รายงานผลการทำประตูของคู่แข่งขันฟุตบอลโลกทางอีเมล์ เมื่อได้รับผลแล้วให้นำผลข้อมูลไปอัพเดทในตารางผลการแข่งขันบนเว็บไซต์ www.mylivefootball.com ดังนั้นโปแกรมภาษา CL อาจเขียนได้ดังนี้

```
[22:0:0|23/6/2005, 1:0:0|24/6/2005]:
  MAIL_AVAILABLE("Live Report") →
    content = readMail()
    update_website(www.mylivefootball.com, content)
[20:0:0|23/6/2005, 21:0:0|23/6/2005]: play_music("song.mp3")
[22:0:0|23/6/2005, 23:0:0|23/6/2005]: browserto(www.news.co.th)
[2:0:0|24/6/2005,_]: shutdown()
```

4.8 สรุป

ภาษา CL เป็นภาษาที่ใช้สั่งงานคอมพิวเตอร์ตามกำหนดเวลาและเหตุการณ์ โดยมีรูปแบบของภาษาคล้ายกับภาษาไพธอน แต่มีการเพิ่มช่วงเวลาเข้าไปแต่ละคำสั่ง รวมทั้งเพิ่มประโยคสำหรับรอรับเหตุการณ์ เพื่ออำนวยความสะดวกในการเขียนโปรแกรมให้มากยิ่งขึ้น

บทที่ 5
โครงสร้างข้อมูลสำหรับเก็บโปรแกรม CL

ในบทนี้เราขอนำเสนอโครงสร้างข้อมูลสำหรับจัดเก็บลำดับการทำงานของโปรแกรม CL โดยจะอธิบายการออกแบบโครงสร้างข้อมูลจากง่ายไปหายาก เราขอเริ่มจากประโยคภาษา CL ที่กำหนดเวลาเป็นค่าคงที่ก่อนในหัวข้อที่ 5.1 จากนั้นเราจะอธิบายถึงโครงสร้างข้อมูลสำหรับเก็บลำดับการทำงานของประโยค CL ดังกล่าว ในหัวข้อที่ 5.2 ตามมาด้วย อัลกอริธึมบนโครงสร้างข้อมูล เช่น การเพิ่มจุดเวลาหรือช่วงเวลา การลบจุดเวลาหรือช่วงเวลา การค้นหาจุดเวลาหรือช่วงเวลา ในหัวข้อที่ 5.3 ตามมาด้วยการวิเคราะห์ประสิทธิภาพของโครงสร้างข้อมูลและอัลกอริธึม จากนั้นเราจะกล่าวถึงการอินเตอร์พรีตประโยค CL แบบต่างๆ เพื่อจัดเก็บในโครงสร้างข้อมูลในหัวข้อที่ 5.5 ต่อมาเราจะเริ่มพิจารณาประโยค CL ที่ยุ่งยากขึ้น นั้นคือ ประโยคที่มีกำหนดเวลาแปรเปลี่ยนได้ ในหัวข้อที่ 5.6 ซึ่งนำมาสู่การปรับปรุงโครงสร้างก่อนหน้านี้ในหัวข้อที่ 5.7 และสรุปในหัวข้อที่ 5.8

5.1 ประโยค CL ที่มีกำหนดเวลาแน่นอน

ประโยค CL ที่มีกำหนดเวลาเริ่มต้นและสิ้นสุดเป็นค่าแน่นอน ถูกใช้สำหรับโปรแกรมการทำงานที่เรามักพบเห็นได้ในชีวิตประจำวัน

ตัวอย่าง

สมมติว่าเราต้องการสั่งงานให้คอมพิวเตอร์เริ่มเล่นเพลงในช่วงเวลา 20 ถึง 21 น. จากนั้นเปิดโปรแกรม browser แสดงหน้าเว็บข่าวในช่วง 22 ถึง 23 น. และปิดเครื่องตอน 2 น. ตลอดเวลา ตั้งแต่ 22 น. จนถึงวันรุ่งขึ้น ให้รอรับเหตุการณ์รายงานผลการทำประตูของคู่แข่งขันฟุตบอลโลกทางอีเมล์ เมื่อได้รับผลแล้วให้นำผลข้อมูลไปอัพเดทในตารางผลการแข่งขันบนเว็บไซต์ www.mylivefootball.com ดังนั้นโปแกรมภาษา CL อาจเขียนได้ดังนี้

```
[22:0:0|23/6/2005, 1:0:0|24/6/2005]:
  MAIL_AVAILABLE("Live Report") →
    content = readMail()
    update_website(www.mylivefootball.com, content)
[20:0:0|23/6/2005, 21:0:0|23/6/2005]: play_music("song.mp3")
[22:0:0|23/6/2005, 23:0:0|23/6/2005]: browserto(www.news.co.th)
[2:0:0|24/6/2005,_]: shutdown()
```

ประโยค CL ที่มีกำหนดเวลาแน่นอน ถือเป็นลักษณะการโปรแกรม CL ที่ง่ายที่สุด ดังนั้นเราจะขอเริ่มออกแบบโครงสร้างข้อมูลจากประโยครูปแบบนี้

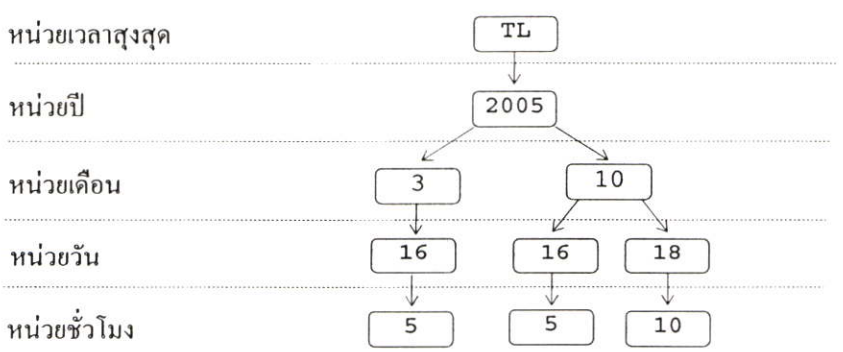
5.2 โครงสร้างข้อมูลสำหรับเก็บประโยค CL ที่มีกำหนดเวลาแน่นอน

จากความจริงที่ว่า จุดเวลาถูกระบุโดยลำดับการนับหน่วยเวลาหลายๆหน่วย โดยนำลำดับหน่วยต่างๆ เหล่านั้นมาเรียงจากเล็กไปหาใหญ่ เช่น จุดเวลา 5:30:10|16/3/2005 หมายถึง วินาทีที่ 10 นาทีที่ 30 ชั่วโมงที่ 5 วันที่ 16 เดือนที่ 3 ปีที่ 2005 จะพบว่าจุดเวลาสามารถกำหนดได้ด้วยหน่วยเวลาที่มีความละเอียดแตกต่างกัน ดังนั้นจุดเวลาที่เก็บในโครงสร้างข้อมูลจะต้องสามารถระบุโดยหน่วยเวลาที่ระดับความละเอียดต่างๆ กันได้หลายๆ หน่วย ในที่นี้เราได้ใช้โครงสร้างข้อมูลแบบ Tree [4] ในการเก็บจุดเวลาต่างๆโดยวิธีดังนี้

- ให้ระดับของโหนดแทนหน่วยเวลา โดยระดับบนสุดเป็นหน่วยสูงสุด ระดับรองลงมาเป็นหน่วยย่อยที่แบ่งย่อยจากหน่วยก่อนหน้ามันลดหลั่นกันไปตามลำดับ
- โหนดแต่ละโหนดแทนลำดับของหน่วยที่จะใช้อ้างในจุดเวลา ยกเว้น root ที่อยู่ที่ระดับหน่วยสูงสุด มีเพียงลำดับเดียว จึงขอแทนด้วย TL เพื่อให้สื่อถึงหน่วยเวลาที่ใหญ่ที่สุด คือ Time Line ซึ่งครอบคลุมเวลาทั้งหมด ทั้งอดีตปัจจุบันและอนาคตกลุ่มโหนดที่ระดับเดียวกันจะต้องเรียงลำดับ โดยโหนดค่าน้อยอยู่ทางซ้าย และโหนดค่ามากอยู่ทางขวา
- การไล่เรียงโหนดจากระดับบนมายังโหนดลูกตามลำดับหน่วยที่ระบุในจุดเวลานั้นๆ เป็นการระบุถึงจุดเวลาที่สมบูรณ์ที่ถูกบันทึกไว้ในโครงสร้างข้อมูล

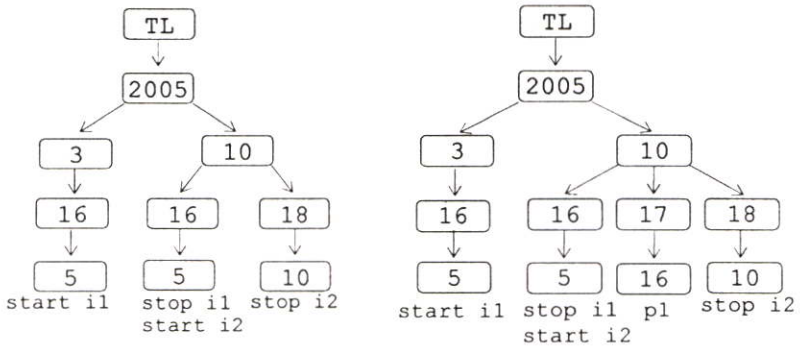
สรุปได้ว่าจุดเวลาแต่ละจุดจะแทนด้วย path จาก root ไปถึง leaf ดังนั้นเพื่อความสะดวกเราจะให้ leaf มีความหมายแทน path ที่มาสิ้นสุดที่ leaf นั้นทำให้ leaf มีความหมายแทนจุดเวลาที่ระบุโดย path ที่มาสิ้นสุดที่ leaf นั้น

สมมติว่าเราจะเก็บจุดเวลา 3 จุด 16/3/2005 เวลา 5 น., 16/10/2005 เวลา 5 น., 18/10/2005 เวลา 10 น. ในโครงสร้างข้อมูล Tree ที่ได้จะเป็นดังรูป



รูปที่ 5.1 Tree ที่เก็บ 3 จุดเวลา

สำหรับการบันทึกช่วงเวลาใน Tree นั้น ทำได้โดยบันทึกทั้งจุดเวลาเริ่มต้นและจุดเวลาสิ้นสุดลงใน Tree พร้อมระบุด้วยว่าจุดใดเป็นจุดเริ่มต้นและจุดใดเป็นจุดสิ้นสุดของช่วงเวลานั้น ดังตัวอย่างการบันทึกช่วงเวลา i1=[16/3/2005 เวลา 5 น., 16/10/2005 เวลา 5 น.] และ i2=[16/10/2005 เวลา 5 น., 18/10/2005 เวลา 10 น.] ใน Tree จะได้ Tree ดังรูป (ก)



ก. เก็บช่วงเวลา ข. เก็บจุดและช่วงเวลา

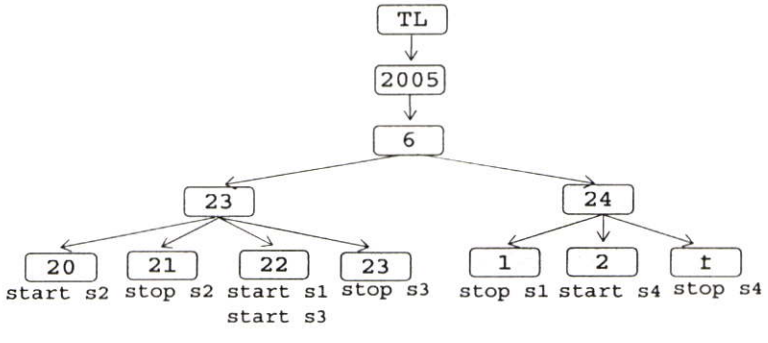
รูปที่ 5.2 Tree เก็บจุดเวลาและช่วงเวลา

รูป (ข) เป็นตัวอย่าง Tree ที่เก็บทั้งจุดและช่วงเวลาไว้ด้วยกัน คือ จุดเวลา p1 ที่ 17/10/2005 เวลา 16 น. และช่วงเวลา i1 = [16/3/2005 เวลา 5 น., 16/10/2005 เวลา 5 น.] และ i2 = [16/10/2005 เวลา 5 น., 18/10/2005 เวลา 10 น.]

ดังนั้นประโยค CL ที่อยู่ในรูปแบบ $[p_s, p_e]$:statement เมื่อถูกเก็บไว้ใน Tree จะมีลักษณะเช่นเดียวกับการเก็บบันทึกช่วงเวลา โดยเราบันทึกข้อมูลว่า start <statement> และ stops <statement> ไว้ที่จุดเวลาเริ่มต้นและสิ้นสุด ตามลำดับ

<u>ตัวอย่าง</u>

โปรแกรม CL ในตัวอย่างก่อนหน้านี้ ถูกเก็บใน Tree ดังรูป



รูปที่ 5.3 การจัดเก็บโปรแกรม CL ใน Tree

สังเกตว่าที่ประโยค s4 ในโปรแกรมช่วงเวลาจะเป็นแบบไม่ระบุเวลาสิ้นสุด จึงหมายถึง 'f' ซึ่งก็จะถูกเก็บใน Tree ดังรูป ในการบันทึก $[p_s,p_e]$:statement ใน Tree นั้น ถ้า p_s และ/หรือ p_e เป็นจุดเวลาแบบไม่ระบุ ("_") เราจะสร้างโหนดแล้วเก็บค่า "af" สำหรับ p_s และ/หรือ สร้างโหนดแล้วเก็บค่า "f" สำหรับ p_e ที่กิ่งขวาสุดของ Tree และบันทึก statement สำหรับช่วงเวลาดังกล่าว

5.3 อัลกอริทึมสำหรับโครงสร้างข้อมูล

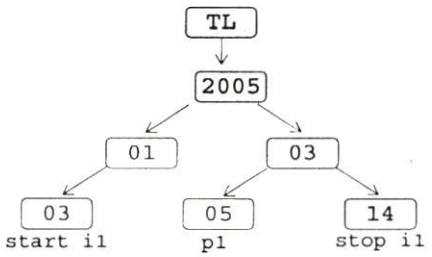
อัลกอริธึมพื้นฐานที่พัฒนาขึ้นมีตั้งแต่ 1) การค้นหาจุดเวลาที่เก็บอยู่ในโครงสร้างข้อมูลโดยใช้ค่าจุดเวลาเป็นคีย์ (random access) 2) การเข้าถึงจุดเวลาตามลำดับ (sequential access) 3) การเพิ่มจุดเวลาและช่วงเวลาลงในโครงสร้างข้อมูล และ 4) การลบจุดเวลาและช่วงเวลาในโครงสร้างข้อมูล

5.3.1 การค้นหาจุดเวลาที่เก็บในโครงสร้างข้อมูลโดยใช้ค่าจุดเวลาเป็นคีย์

การค้นหานี้ อาจเป็นได้ใน 2 ลักษณะคือ 1) การนำเอาค่าจุดเวลาไปค้นหาว่ามีจุดเวลานี้เก็บไว้ในโครงสร้างข้อมูลหรือไม่ 2) การนำเอาค่าจุดเวลาไปค้นหาว่าจุดเวลานี้เป็นจุดเริ่มต้นหรือจุดสิ้นสุดของช่วงเวลาใดที่เก็บในโครงสร้างข้อมูลหรือไม่ ต่อไปเป็นการนำเสนออัลกอริธึมเพื่อการค้นหาดังกล่าว

อัลกอริธึมนี้รับจุดเวลาที่ระบุด้วยหน่วยเวลาหลายหน่วยเรียงลำดับตามหน่วยใหญ่หน่วยเล็ก การค้นหาจุดเวลานี้เริ่มจากนำค่าตัวเลขของหน่วยปี (หน่วยใหญ่สุด) เปรียบเทียบกับโหนด ลูกของ root (TL) ว่าพบค่าหน่วยปีที่โหนดลูกหรือไม่ ถ้าไม่พบก็แสดงว่าไม่มีจุดเวลานั้นเก็บใน Tree และยุติการค้นหา แต่ถ้าพบก็ให้นำค่าตัวเลขหน่วยถัดไปของจุดเวลาไปเปรียบเทียบกับโหนดลูกของโหนดที่ค้นพบ และทำซ้ำเดิมอีกเรื่อยๆ จนกระทั่งหมดหน่วยเวลาที่ใช้ค้นหา แล้วให้ผลลัพธ์เป็นชื่อจุดเวลาที่บันทึกไว้ที่โหนดหรือ leaf สุดท้ายที่ค้นพบค่าหน่วยเวลาสุดท้ายออกมา

ต่อไปเป็นรูปแสดงตัวอย่างการค้นหาจุดเวลาตามอัลกอริธึมนี้เมื่อต้องการค้นหาจุดเวลาด้วยจุดเวลาวันที่ 14 เดือน 3 ปี 2005 เริ่มด้วยการเปรียบเทียบหน่วยปีที่ 2005 กับโหนดลูกของ root ซึ่งพบว่ามีโหนดปีที่ 2005 อยู่จึงนำเอาค่าหน่วยเดือนมาเปรียบเทียบกับโหนดลูกของโหนด 2005 พบว่ามีโหนดเดือนที่ 3 เช่นกัน จึงมีการตรวจสอบต่อไปอีกในหน่วยวัน แล้วสุดท้ายพบว่าวันที่ 14 เป็นค่าหน่วยเวลาสุดท้ายแล้ว จึงให้ผลลัพธ์เป็นชื่อจุดเวลาที่บันทึกไว้ที่โหนดวันที่ 14 ออกไป



รูปที่ 5.4 ขั้นตอนการค้นหาจุดเวลา

รายละเอียดอัลกอริธึมแสดงได้ดังนี้

```
def whatHappenAt(tree, tp):
  root = getroot(tree)
  searchUnit = head(tp)
  if searchUnit is empty:
    return getTPNames(root)
  else:
    children = getAllChildrenOf(root)
    child = find(children, searchUnit)
    if child not found:
      return 'timepoint not found'
    else:
   return whatHappenAt(subtree rooted at child, tail(tp))
```

รูปที่ 5.5 อัลกอริธึมสำหรับการค้นหาจุดเวลา

อัลกอริธึมนี้ จะรับอินพุทเป็น Tree และจุดเวลา tp ที่อยู่ในรูปของลิสต์ของค่าหน่วยเวลา โดยอัลกอริธึมนี้จะอ่านค่าหน่วยเวลาออกมาที่ละหน่วยเวลาด้วยฟังก์ชั่น head ซึ่งค่าที่ได้ถูกนำไปค้นหาโหนดในลิสต์ของโหนดลูกของ root ถ้าไม่พบก็จะแจ้งว่าไม่พบจุดเวลาที่ต้องการค้นหา แต่ถ้าพบ ก็จะเรียกอัลกอริธึมเดิมอีก โดยส่ง subtree ที่มี root เป็นโหนดที่ค้นพบ และลิสต์ของหน่วยเวลาที่เหลือ tail(tp) ไปเป็นพารามิเตอร์ ซึ่งจะกระทำเช่นนี้ไปเรื่อยๆ จนค่าหน่วยเวลาหมดลิสต์ จึงให้ผลลัพท์เป็นชื่อจุดเวลาที่บันทึกอยู่บนโหนดสุดท้ายที่ค้นพบออกมา

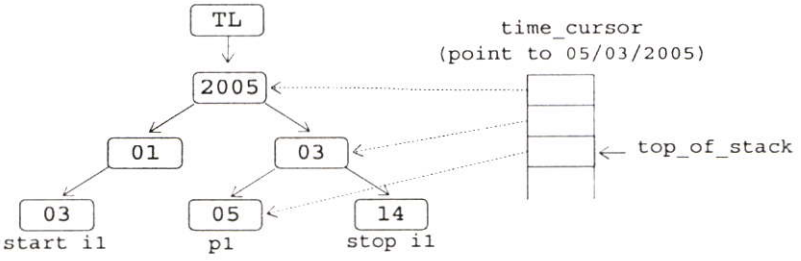
5.3.2 การเข้าถึงจุดเวลาตามลำดับ

เป็นการเข้าถึงจุดเวลาที่อยู่ถัดไปจากจุดเวลาล่าสุดที่เคยอ่านค่าออกมา เป็นการเข้าถึงจุดเวลาตามลำดับของค่าเวลา

เนื่องจากโครงสร้างข้อมูลที่จัดเก็บ เราได้ทำการเรียงลำดับโหนดลูกตามค่าหน่วยเวลาจากน้อยไปหามาก ในตำแหน่งจากซ้ายไปขวา ทำให้โหนด leaf ทั้งหมด มีค่าจุดเวลาเรียงลำดับกัน ทำให้ง่ายต่อการอิมพลีเมนต์อัลกอริธึมนี้

เพื่อให้รู้ตำแหน่งของจุดเวลาล่าสุดที่ถูกอ่านไป เราต้องมีตัวชี้เวลา (time_cursor) ซึ่งค่านี้ต้องถูกเซตก่อนเริ่มใช้งาน และถูกอัพเดทค่าทุกครั้งที่มีการอ่านค่าจุดเวลา ให้ชี้อยู่ที่โหนดจุดเวลาล่าสุดที่อ่านออกไปอยู่เสมอ

เราได้เลือกใช้ stack เพื่ออิมพลีเมนต์ time_cursor โดย stack นี้จะเก็บตัวชี้ ที่ชี้ไปยังโหนดใน Tree ที่แทนจุดเวลา โดยที่ top_of_stack ชี้ไปที่โหนด leaf ดังรูป



รูปที่ 5.6 Tree และ time_cursor

การอัพเดท timer_cursor จะทำโดยการเรียกใช้ฟังก์ชั่น updateCursor ซึ่งมันจะ pop ตัวชี้ โหนดออกมาจาก time_cursor จากนั้น ฟังก์ชั่นนี้จะหาโหนดพี่น้องถัดไปของโหนดที่ตัวชี้อ้างถึง ถ้ามีโหนดถัดไป อัลกอริธึมก็จะ push ตัวชี้ของเวลาถัดไปเข้าไป ใน time_cursor แต่ถ้าไม่มีโหนดพี่น้องถัดไป อัลกอริธึมก็จะ recursive อีกครั้ง เพื่ออัพเดทค่าตัวชี้โหนดที่อยู่ในระดับสูงขึ้นไป หลังจากที่อัพเดทค่าตัวชี้ในระดับสูงเสร็จแล้ว อัลกอริธึมก็จะเลือกเอาเฉพาะโหนดลูกคนแรก push ลงไปใน time_cursor

ในอัลกอริธึมเรากำหนดให้ใช้ตัวแปรแบบ global ชื่อ last_visit ที่เป็นชนิดข้อมูลแบบ time_cursor เพื่อชี้ไปยังกลุ่มโหนดที่แทนจุดเวลาล่าสุดที่เพิ่งอ่านออกไป โดยตัวแปรนี้จะถูก initialize ค่า ด้วย function initialize ให้ตัวชี้ที่อยู่ใน stack ชี้ไปยังโหนดที่เป็น path ไปยังโหนดleaf ซ้ายสุด

ฟังก์ชั่น initialize รับอินพุทเป็นโครงสร้างข้อมูล tree และตัวแปร time_cursor ฟังก์ชั่นนี้จะ traverse ไปบน โหนดลูกที่อยู่ซ้ายมือสุด โหนดที่ visit แล้วจะถูก push ลงเก็บไว้ใน time_cursor ตั้งแต่โหนดในระดับที่รองจาก root จนถึง leaf ตามลำดับ

หลังจากที่ค่าตัวแปร last_visit ถูกกำหนดค่าแล้ว เราสามารถเข้าถึงจุดเวลาได้โดยเรียกฟังก์ชั่น seqAccess ฟังก์ชั่นนี้จะรับอินพุทเป็นจุดเวลาปัจจุบัน ซึ่งฟังก์ชั่นจะหาจุดเวลาถัดจาก last_visit ด้วยการเรียกฟังก์ชั่น updateCursor เก็บไว้ในตัวแปร next_visit จากนั้นฟังก์ชั่นจะนำจุดเวลาปัจจุบันมาเปรียบเทียบกับ next_visit ถ้าจุดเวลาปัจจุบันมีค่าเท่ากับ next_visit ก็จะคืนค่าจุดเวลาถัดไป แต่ถ้าไม่ตรงก็จะแจ้งกลับไปว่าเวลาปัจจุบัน ยังไม่ถึงเวลาถัดไป

เมื่อจุดเวลาปัจจุบันมีค่าเท่ากับ next_visit เราจำเป็นต้องอัพเดทค่า ของ last_visit ให้มีค่าเท่ากับ next_visit เพื่ออัพเดท last_visit ให้ชี้อยู่ที่จุดเวลาที่เพิ่งอ่านออกไปล่าสุดอยู่เสมอ รายละเอียดอัลกอริธึมแสดงได้ดังนี้

```
def initialize(tree, time_cursor):
  root = getroot(tree)
  if root is leaf:
    return
 else:
   children = getAllChildrenOf(root)
   child = head(children)
   push(time_cursor, child)
   return initilize(subtree rooted at child, tail(tp), time_cursor)

def updateCursor(time_cursor):
  if time_cursor is empty stack:
    print 'reach the end of timetable'
  node = pop(time_cursor)
  next_node = next(node)
  if next_node is not exist:
    updateCursor(time_cursor)
    if time_cursor is not empty stack:
      next_node = getFirstChild(getTopOfStack(time_cursor))
      push(time_cursor, next_node)
  else:
    push(time_cursor, next_node)

def seqAccess(current_tp):
  global last_visit
  next_visit = clone(last_visit)
  updateCursor(next_visit)
  if next_visit == current_tp:
    last_visit = next_visit
    node = getTopOfStack(last_visit)
    return getTPNames(node)
  else:
    print 'timepoint not found'

# initialize last_visit when program start up
initialize(tree, last_visit)
```

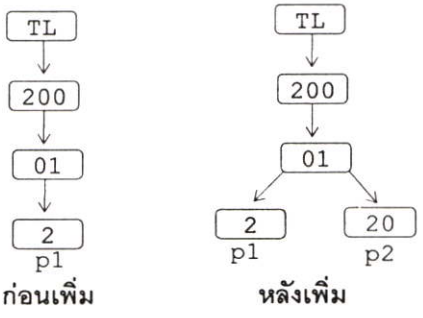
รูปที่ 5.7 อัลกอริธึมสำหรับการเข้าถึงจุดเวลาตามลำดับ

5.3.3 การเพิ่มจุดเวลาและช่วงเวลาลงในโครงสร้างข้อมูล

การเพิ่มจุดเวลาใดๆ กระทำได้โดยการสร้าง path เพื่อแทนจุดเวลาที่ต้องการเพิ่มเข้าไป โดยการเปรียบเทียบหน่วยจุดเวลาใหญ่สุดกับโหนดลูกของ root ลงไปทีละชั้นที่โหนดลูกลำดับต่อมาตามลำดับจนถึงหน่วยสุดท้าย และจะบันทึกชื่อจุดเวลานั้นไว้ที่โหนดหรือ leaf ที่หน่วยย่อยที่สุดไปสิ้นสุด

อัลกอริธึมนี้รับอินพุทเป็น Tree จุดเวลา tp และชื่อจุดเวลา tpname โดยจะอ่านค่าหน่วยเวลาออกมาที่ละหน่วยเวลาด้วยฟังก์ชั่น head ซึ่งค่าที่ได้ถูกนำไปเทียบกับสมาชิกในลิสต์ของโหนดลูกทั้งหมดของ root ถ้าไม่พบก็จะสร้างโหนดนั้นขึ้นมา แล้วเรียกใช้อัลกอริธึมเดิมอีกครั้ง โดยส่ง subtree ที่มี root เป็นโหนดที่ค้นพบ ลิสต์ชุดค่าหน่วยเวลาที่เหลือ tail(tp) และชื่อจุดเวลาที่จะบันทึกลงใน Tree ไปเป็นพารามิเตอร์ ซึ่งจะกระทำเช่นนี้ไปเรื่อยๆ จนกระทั่งหมดค่าหน่วยเวลาของจุดเวลาที่ต้องการบันทึก แล้วจึงบันทึกชื่อจุดเวลานั้นไว้ที่โหนดสุดท้าย (ซึ่งอาจจะเป็น leaf ก็ได้) ที่หน่วยเวลาย่อยสุดท้ายไปสิ้นสุด

ต่อไปเป็นตัวอย่างการเพิ่มจุดเวลา p2 = 20/1/2005 ลงใน Tree ด้านซ้าย ตามอัลกอริธึม ซึ่งจะได้ Tree ด้านขวา ดังรูป



รูปที่ 5.8 วิธีการเพิ่มจุดเวลาลงใน Tree

ส่วนการเพิ่มช่วงเวลา กระทำได้โดยการเพิ่มจุดเริ่มต้น และจุดสิ้นสุดของช่วงเวลาเข้าไปใน Tree โดยข้อมูลที่บันทึกไว้ที่ leaf เป็นชื่อจุดเริ่มต้น และชื่อจุดสิ้นสุดของช่วงเวลาตามลำดับ

รายละเอียดของอัลกอริธึมแสดงได้ดังนี้

```
def insertTPoint(tree, tp, tpname):
  root = getroot(tree)
  searchUnit = head(tp)
  if searchUnit is empty:
    return insertTPName(root, tpname)
 else:
   children = getAllChildrenOf(root)
   child = find(children, searchUnit)
   if child not found:
     child = add child for root and sort
    return insertTPoint(subtree rooted at child, tail(tp), tpname)
```

รูปที่ 5.9 อัลกอริธึมสำหรับการเพิ่มจุดเวลา

5.3.4 การลบจุดเวลาและช่วงเวลาในโครงสร้างข้อมูล

การลบจุดเวลาใดๆ กระทำได้โดยการลบชื่อจุดเวลาออกจากโหนดหรือ leaf ที่บันทึกจุดเวลานั้นอยู่ โดย leaf ใดที่ถูกลบชื่อจุดเวลาที่เก็บอยู่ไปทั้งหมด leaf นั้นก็จะถูกลบออกจาก Tree ด้วย และจะกระทำซ้ำๆ เช่นนี้กับโหนดที่ระดับสูงขึ้นไปตามลำดับ

อัลกอริธึมการลบจุดเวลารับอินพุทเป็น Tree และจุดเวลา tp พร้อมชื่อจุดเวลาที่ต้องการลบ tpname อัลกอริธึมนี้มีลักษณะคล้ายกับการเพิ่มจุดเวลาต่างกันเพียงการลบจุดเวลาเป็นการลบชื่อของจุดเวลาออกจากโหนด (หรือ leaf) และอาจลบโหนดที่ปราศจากชื่อเวลาออกจาก Tree ด้วย รายละเอียดของอัลกอริธึมแสดงได้ดังนี้

```
def removeTPoint(tree, tp, tpname):
  root = getroot(tree)
  searchUnit = head(tp)
  if searchUnit is empty:
    return deleteTPName(root, tpname)
  else:
   children = getAllChildrenOf(root)
   child = find(children, searchUnit)
   if child not found:
     return 'timepoint not found'
   else:
      removeTPoint(subtree rooted at child, tail(tp), tpname)
      if child is a leaf and contain no name:
        remove child from its root
```

รูปที่ 5.10 อัลกอริธึมสำหรับการลบจุดเวลา

ส่วนการลบช่วงเวลาใดๆ กระทำได้โดยลบจุดเวลาเริ่มต้นและจุดเวลาสิ้นสุดออกจากโครงสร้างข้อมูล

5.4 คุณสมบัติของโครงสร้างข้อมูล

โครงสร้างข้อมูลที่ออกแบบขึ้นมามีคุณสมบัติดังนี้

1. ลำดับของการเพิ่มจุดเวลาไม่มีผลต่อโครงสร้างข้อมูล
2. โครงสร้างข้อมูลสามารถเก็บจุดเวลาและช่วงเวลา ที่มีความละเอียดของหน่วยเวลาที่แตกต่างกันได้ และสามารถแทนเวลาในหน่วยความละเอียดที่ไม่จำกัด
3. การเข้าถึงจุดเวลาแบบตามลำดับ ในกรณีที่ดีที่สุดคือเมื่อการอัพเดท next_visit เกิดขึ้นที่ระดับของโหนด leaf เท่านั้น ซึ่งจะใช้จำนวนการเปรียบเทียบเป็นค่าคงที่หนึ่ง ส่วนในกรณีที่

แย่ที่สุดคือเมื่อต้องมีการอัพเดทค่า next_visit ในทุกๆ ระดับของ Tree ซึ่งจะทำให้ใช้จำนวนการเปรียบเทียบตามความสูงของ Tree

4. พื้นที่เก็บข้อมูลสามารถแบ่งได้เป็น 2 ส่วน คือ โหนดที่เป็นโครงสร้างหลัก และชื่อจุดเวลาที่บันทึกไว้ที่โหนดหรือ leaf โดยโครงสร้างหลักใช้พื้นที่ขึ้นอยู่กับลักษณะจุดเวลาที่จัดเก็บ เช่น ถ้าโครงสร้างข้อมูลจัดเก็บจุดเวลาที่มีลำดับที่ของปีและเดือนต่างกันมาก จะทำให้โหนดปีที่และโหนดเดือนที่ ไม่มีการใช้ร่วมกัน ทำให้มีจำนวนโหนดมาก

สำหรับการวิเคราะห์โครงสร้างข้อมูลโดยละเอียดจะขอกล่าวในบทที่ 7

5.5 การบันทึกประโยค CL ในโครงสร้างข้อมูล

ประโยค [p_s, p_e]:statement มีการจัดเก็บในโครงสร้างข้อมูลตามรูปแบบของประโยค ดังนี้

5.5.1 ประโยคสำหรับการกำหนดค่าให้ตัวแปร

ประโยคที่อยู่ในรูปแบบ [p_s, p_e] : <id> = <expression> เราจะจัดเก็บสัญญลักษณ์ start <id> = <expression> ไว้ยังโหนดที่แทนเวลาเริ่มต้น p_s และจัดเก็บสัญญลักษณ์ stop <id> = <expression> ไว้ยังโหนดที่แทนเวลาสิ้นสุด p_e

5.5.2 ประโยคเรียกใช้ฟังก์ชั่น

ประโยคที่อยู่ในรูปแบบ [p_s, p_e] : <function-name>(<argument>...) เราจะจัดเก็บสัญญลักษณ์ start <function-name>(<argument>...) ไว้ยังโหนดที่แทนเวลาเริ่มต้น p_s และจัดเก็บสัญญลักษณ์ stop <function-name>(<argument>...) ไว้ยังโหนดที่แทนเวลาสิ้นสุด p_e

5.5.3 ประโยคตอบสนองเหตุการณ์

ประโยคที่อยู่ในรูปแบบ [p_{s1},p_{e1}] Event $\rightarrow$ [p_{s2},p_{e2}] Action เราจะจัดเก็บสัญญลักษณ์ start Event $\rightarrow$ [p_{s2},p_{e2}] Action ไว้ยังโหนดที่แทนเวลาเริ่มต้น p_{s1} และจัดเก็บสัญญลักษณ์ stop Event $\rightarrow$ [p_{s2},p_{e2}] Action ไว้ยังโหนดที่แทนเวลาสิ้นสุด p_{e1}

5.5.4 ประโยคควบคุมลำดับการทำงาน

- if-statement ที่อยู่ในรูปแบบ

[p_s, p_e]if <expression>: <statementA>[else: <statementB>] เราจะมองว่า statementA และ statementB เป็นเพียงข้อความธรรมดา โดยจัดเก็บสัญญลักษณ์ start if <expression>: <statementA>[else: <statementB>] ไว้ยังโหนดที่แทนเวลาเริ่มต้น p_s และจัดเก็บสัญญลักษณ์ stop if <expression>: <statementA>[else: <statementB>] ไว้ยังโหนดที่แทนเวลาสิ้นสุด p_e

5.5.4 ประโยคชุดคำสั่ง

- Sequential Statement

ประโยคย่อยๆ ภายใน Sequential Statement จะถูกจัดเก็บลงใน Tree ต้นเดียวกันทั้งหมด โดยวิธีการจัดเก็บของประโยคย่อยๆ จะใช้วิธีการที่กล่าวไปแล้ว

- Parallel Statement

ประโยคย่อยๆ ภายใน Sequential Statement จะถูกจัดเก็บลงใน Tree ต้นเดียวกันทั้งหมด เนื่องจากโครงสร้างข้อมูลที่นำเสนอ สามารถเรียงลำดับข้อมูล จากการเพิ่มข้อมูลจุดเวลา ที่ไม่เป็นลำดับได้ ทำให้วิธีการจัดเก็บ Parallel Statement ใช้วิธีเดียวกับ Sequential Statement ได้

- Unordered Statement

เนื่องจากประโยคย่อยๆ ภายใน Unordered Statement ไม่มีการระบุลำดับการประมวลผลที่แน่ชัด ดังนั้นอินเตอร์พรีตเตอร์จะต้องกำหนดลำดับการประมวลผล ก่อนที่จะจัดเก็บ โดยหลังจากที่อินเตอร์พรีตเตอร์กำหนดลำดับให้กับประโยคย่อยแล้ว จุดเวลาที่เป็น '_' จะตีความหมายเช่นเดียวกับ จุดเวลา '_' ใน Sequential Statement และจัดเก็บลงในโครงสร้างข้อมูล

5.6 ประโยค CL ที่มีกำหนดเวลาเปลี่ยนแปลง

จากที่ผ่านมา เรากำหนดว่า $[p_s,p_e]$ ที่ใช้ในภาษา CL เป็นค่าคงที่เท่านั้น มีผลทำให้ง่ายต่อการออกแบบตัวอินเตอร์พรีตเตอร์ แต่กรณีที่จุดเวลาเริ่มต้นและจุดสิ้นสุดใน $[p_s,p_e]$:statement ถูกกำหนดโดยตัวแปรที่มีค่าแปรเปลี่ยนตามคาบเวลา เช่น ทุกๆเดือน ทุกๆ วัน ตามประโยคทำซ้ำ เช่น for-statement รวมทั้งในกรณีที่ค่าของตัวแปร p_s หรือ p_e ถูกกำหนดค่าโดยการทำงานของ statement ก่อนๆ แล้ว อินเตอร์พรีตเตอร์นี้จะไม่สามารถรองรับการประมวลผลได้

ในหัวข้อต่อไป เราจึงขอนำเสนอการปรับปรุงโครงสร้าง Tree ที่ใช้เก็บตารางการทำงานของโปรแกรม CL แบบเดิม เพื่อให้รองรับการจัดเก็บประโยค CL ที่มีกำหนดเวลาเป็นตัวแปร

5.7. โครงสร้างข้อมูลสำหรับเก็บประโยค CL ที่มีกำหนดเวลาเปลี่ยนแปลง

มีการปรับปรุงจาก Tree เดิมใน 3 ลักษณะดังนี้

5.7.1 โครงสร้างข้อมูลสำหรับเก็บประโยค CL ที่มีการทำงานเป็นคาบๆ

ในชีวิตประจำวันจะพบว่ามีการทำงานที่เกิดขึ้นเป็นคาบๆ เป็นประจำ เช่น การจ่ายค่าบิล ณ วันที่ 1 ของทุกๆเดือน ซึ่งการทำงานในลักษณะนี้สามารถเขียนเป็นโปรแกรม CL ได้ดังนี้

ตัวอย่าง

```
[0:0:0|1/01/2006, 0:0:0|2/12/2006]:              (s_n)
  for month in (1,12):
  [0:0:0|1/month/2006,0:0:0|2/month/2006]:
     transfer_money("1000")
```

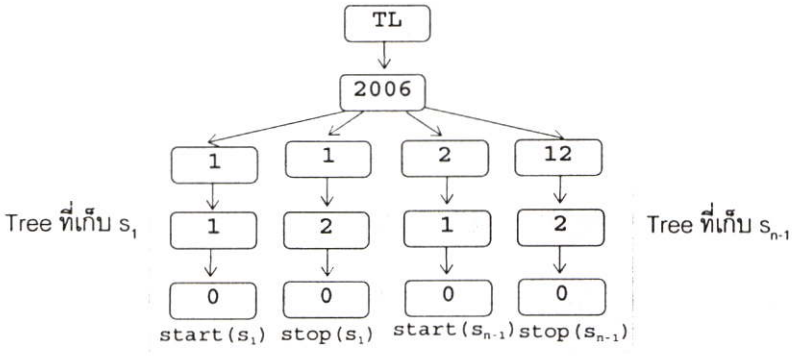
ซึ่งเป็นการสั่งให้คอมพิวเตอร์ โอนเงินเพื่อชำระบิล 1,000 บาท ณ วันที่ 1 และให้เสร็จภายในวันที่ 2 ของทุกๆเดือน ตลอดปี 2006 สังเกตว่าจะมีการนำเอาตัวแปรควบคุม month มาใช้ ใน for-statement เพื่อแปรเปลี่ยนค่าเดือน ในแต่ละคาบการทำงานทีละหนึ่ง ซึ่งแตกต่างจากภาษา CL เดิมที่ไม่สามารถใช้ตัวแปรกับกำหนดเวลาการทำงานได้

เราพบว่าตารางการทำงานซ้ำๆ ที่เป็น n คาบ สามารถแบ่งออกโดยเรียงลำดับเป็นตารางการทำงานคาบที่ 1 ก่อน แล้วตามด้วยตารางการทำงานสำหรับอีก n-1 คาบที่เหลือ โดยวิธีนี้ทำให้สามารถออกแบบโครงสร้าง Tree ที่ใช้จัดเก็บตารางการทำงานซ้ำๆ ที่เป็น n คาบได้ โดยการแบ่งเป็น Tree ที่เก็บตารางการทำงานของคาบที่ 1 แล้วตามด้วย Tree ที่เก็บตารางการทำงานของ n-1 คาบที่เหลือ ดังนั้น Tree ที่เก็บ s_n ดังรูป ประกอบด้วย Tree ที่เก็บ s_1 คือ

```
[0:0:0|1/1/2006,0:0:0|2/1/2006]transfer_money("1000")
```

เชื่อมตามมาด้วย Tree ที่เก็บ s_{n-1} คือ

```
for month in (2,12):
 [0:0:0|01/month/2006, 0:0:0|2/month/2006]
   transfer_money("1000")
```



รูปที่ 5.11 Tree สำหรับเก็บตารางเวลาที่ทำงานเป็นคาบ

5.7.2 โครงสร้างข้อมูลสำหรับเก็บประโยค CL ที่มีกำหนดเวลาเป็นตัวแปร

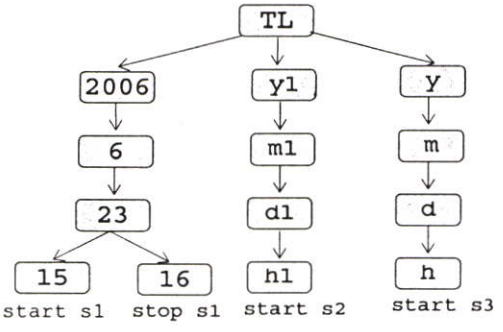
ในบางครั้งตารางเวลาการทำกิจกรรมบางอย่าง อาจมีหมายกำหนดการถูกกำหนดโดยเวลาของผลการทำงานก่อนๆ ดังตัวอย่าง

<u>ตัวอย่าง</u>

เราต้องการสั่งให้คอมพิวเตอร์รอรับข้อความ SMS แจ้งเวลาในการสั่งอาหาร แล้วนำเวลาดังกล่าวมาใช้เป็นตารางเวลาในการสั่งอาหารทางอินเตอร์เน็ต ก่อนหน้าจะสั่งอาหาร 1 ชม.สั่งให้เครื่องล้างจานทำงาน ซึ่งอาจแสดงเป็นโปรแกรม CL ได้ดังนี้

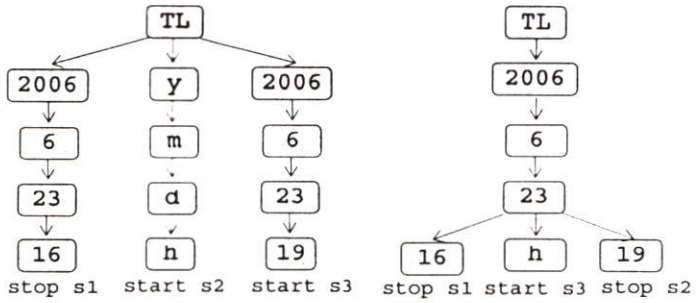
```
[15:00:00|23/6/2005, 18:00:00|23/6/2005]          (s1)
  SMS_AVAILABLE →
    message = getSMS()
    (h,d,m,y) = getOrderTime(message)
    (h1,d1,m1,y1) = (h,d,m,y) - (1 hour)
[h1:0:0|d1/m1/y1,_]start_dish_cleaner()           (s2)
[h:0:0|d/m/y, _] orderFood(pizza)                 (s3)
```

โปรแกรมนี้มีการใช้ตัวแปรใน $[p_s, p_e]$ ของตารางเวลา การจัดเก็บตารางเวลาลงใน Tree ก็เป็นเพียงจัดเก็บตัวแปรลงในโหนด start หรือ stop ของ Tree เพื่อรอการตีค่าดังรูป



รูปที่ 5.12 โครงสร้างตารางเวลาที่เก็บจุดเวลา ที่เป็นตัวแปร

ในบางกรณี Tree ที่สร้างขึ้นอาจสามารถทำให้เล็กลงโดยการลดความซ้ำซ้อนของโหนด ได้แก่ กรณีที่กลุ่มโหนดที่อยู่ในระดับเดียวกัน เมื่อมีโหนดใดอยู่ในระหว่าง 2 โหนดที่มีค่าเหมือนกัน ก็สามารถถูกรวมเป็นโหนดเดียวกันได้ โดยที่ความหมายไม่เปลี่ยน ดังเช่น จาก Tree ข้างล่าง จะเห็นว่ากลุ่มโหนดที่ระดับหน่วยปี เดือน และวัน จะถูกยุบรวมเป็นโหนดเดียวในแต่ละระดับดังรูป



ก. Tree ก่อนการยบโหนด ข. Tree หลังการยุบ

รูปที่ 5.13 การยุบโหนดใน Tree

5.7.3 โครงสร้างข้อมูลสำหรับเก็บประโยค CL ที่มีกำหนดเวลาแบ่งเป็น scope

การที่เรายอมให้ใช้ตัวแปรในการกำหนดค่าของ $[p_s, p_e]$ นั้น อาจเกิดกรณีที่มีการใช้ชื่อตัวแปรชื่อเดียวกันในหลายๆ nested statement ที่อยู่ต่างระดับ (scope) กัน ในกรณีเช่นนี้จะทำให้การตีความหมายตัวแปรผิดพลาดว่าเป็นตัวเดียวกัน ทั้งที่แท้ที่จริงแล้วควรถือว่าตัวแปรพวกนี้เป็นต่างตัวแปรกัน ทำให้มีความจำเป็นที่จะต้องสร้าง Tree ย่อยขึ้นมา สำหรับแต่ละ scope ซึ่งการมี scope นี้จะช่วยแก้ปัญหาในกรณีที่โปรแกรม CL มีชุดประโยคเป็นแบบ parallel ด้วย เช่น

ตัวอย่าง

```
parallel:
 [9|11/12/2006,12|11/12/2006]sequential:          (s1)
  [9|11/12/2006,_]
    h1 =
      getMeetingTime("www.john.com/schedule")     (s1.1)
  [h1|11/12/2006, 12|11/12/2006]
    alert_me("It's meeting time.")                (s1.2)
 [9|11/12/2006,12|11/12/2006]sequential:          (s2)
  [9|11/12/2006,_]
    h2 =
      getLunchTime("www.somchai.com/schedule")    (s2.1)
  [h2|11/12/2006, 12|11/12/2006]
    alert_me("It's lunch time.")                  (s2.2)
```

ซึ่งมีความหมายว่า ในช่วง [9:00:00|11/12/2006, 12:00:00|11/12/2006] ให้คอมพิวเตอร์ไปดึงข้อมูลเวลาการประชุมจากเว็บไซต์ของ John แล้วให้แจ้งเตือนเราที่เวลานั้น ขณะเดียวกันก็โปรแกรมให้ไปดึงข้อมูลกำหนดนัดหมายเวลารับประทานอาหารกลางวัน แล้วเมื่อถึงเวลานั้นก็ให้แจ้งเตือนเราเช่นกัน

จะพบว่าเหล่าตัวแปรใน h1 และใน h2 ที่ปรากฏในประโยค s1 และ s2 เมื่อโปรแกรมทำงาน เราจะไม่สามารถรู้ได้ว่าตัวแปรตัวใดมีค่ามากน้อยกว่ากัน จึงทำให้ s1 และ s2 ไม่สามารถบันทึกใน

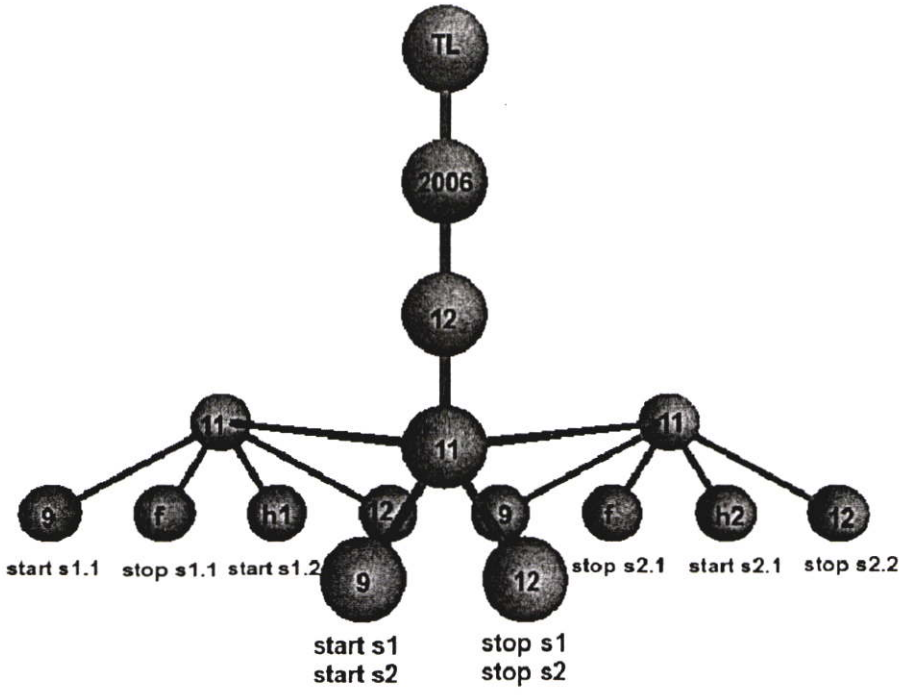
Tree ต้นเดียวกันได้ จึงต้องบันทึกไว้ในต่าง Tree กัน ซึ่งแต่ละ Tree ก็จะถือว่าเป็น scope สำหรับแต่ละกลุ่มตัวแปร

ดังนั้นเมื่อใดที่ Nested statement เกิดขึ้นก็จะมีการสร้าง Tree ย่อยต้นใหม่สำหรับเก็บชุดประโยคใน nested statement นั้น เพื่อให้มี scope เป็นเอกเทศแยกออกมา ในการประมวลผล nested statement ตัว Time Responder จะตรวจทุก statement ใน Tree ไปด้วยกัน แต่จะเลือกอันที่มี start ตรงกับเวลาปัจจุบันขึ้นมา พร้อมด้วยค่า scope ของมัน เพื่อส่งให้อินเตอร์พรีตเตอร์ไปประมวลผล ทำให้ไม่มีการสับสนเรื่อง scope

แต่ละ Scope ถูกกำหนดให้ทำงานได้ในช่วงเวลาหนึ่ง ซึ่งแทนด้วย path ของจุดเวลาเริ่มต้นและสิ้นสุด ซึ่งทั้งสอง path จะมีโหนดร่วมกัน เราจะถือเอาโหนดร่วมที่เป็นหน่วยเวลาเล็กที่สุดเป็นจุดเชื่อม Tree ย่อยๆ ของแต่ละ scope ที่สร้างขึ้น ดังนั้นเพื่อแยก scope แต่ละ statement ออกจากกัน เราจึงแก้ไขโครงสร้างข้อมูลเดิมให้ scope เป็น Tree แยกจากกัน และมีลิงค์เชื่อมต่อระหว่าง parent scope กับ nested scope ผ่านโหนดร่วมดังกล่าว

<u>ตัวอย่าง</u>

จากตัวอย่างที่แล้ว ถ้านำโปรแกรมมาจัดเก็บใน Tree จะเป็นดังนี้



รูปที่ 5.14 Tree ที่มี nested scope

5.8 สรุป

ในบทนี้เราได้นำเสนอโครงสร้างข้อมูลสำหรับจัดเก็บลำดับการทำงานของโปแกรม CL เราได้เริ่มออกแบบโครงสร้างข้อมูลสำหรับประโยค CL ที่มีกำหนดเวลาแน่นอนก่อน โดยออกแบบโครงสร้างข้อมูลจากรูปแบบจุดเวลาที่ใช้ในชีวิตประจำวัน ผลที่ได้คือโครงสร้างข้อมูลแบบ Tree ที่ใช้เก็บจุดเวลาและช่วงเวลา จากการวิเคราะห์เราพบว่าประสิทธิภาพการเพิ่มจุดเวลา การลบจุดเวลา การค้นหาจุดเวลา สามารถทำได้คลาส Logarithmic นอกจากนี้เรายังพิจารณาต่อไปถึงประโยค CL ที่มีกำหนดเวลาเปลี่ยนแปลงได้ อันได้แก่ กำหนดเวลาเปลี่ยนแปลงเป็นคาบเวลา และกำหนดเวลาเปลี่ยนตามผลของการประมวลผลประโยคที่ผ่านมา ซึ่งผลที่ได้ทำให้เราได้โครงสร้างข้อมูลที่เหมาะสำหรับจัดเก็บการจุดเวลาที่เปลี่ยนแปลง

บทที่ 6

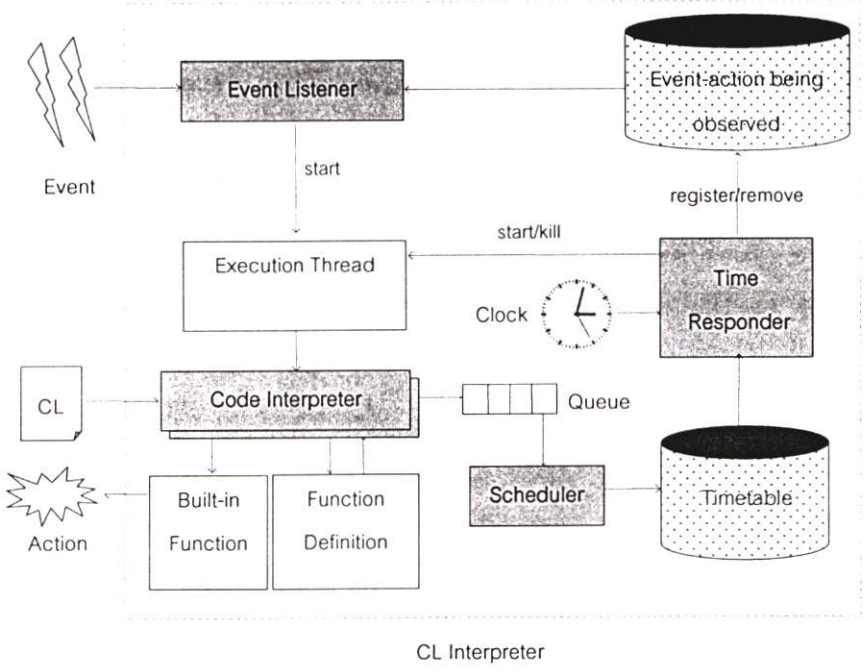
การออกแบบและพัฒนาอินเตอร์พรีตเตอร์ภาษา CL

ในบทนี้ เราจะกล่าวถึงการใช้ประโยชน์โครงสร้างข้อมูลสำหรับเก็บประโยค CL ในอินเตอร์พรีตเตอร์ภาษา CL เนื้อหาของบทนี้เริ่มจากการอธิบายภาพรวมของโครงสร้างอินเตอร์พรีตเตอร์ภาษา CL ในหัวข้อที่ 6.1 จากนั้นจะอธิบายขั้นตอนการทำงานของอินเตอร์พรีตเตอร์ ตั้งแต่จัดเก็บประโยค CL ในโครงสร้างข้อมูลจนกระทั่งประโยคถูกแปลความหมายเป็นการทำงาน ในหัวข้อที่ 6.2 จากนั้นเราจะอธิบายการทำงานของส่วนประกอบที่สำคัญของอินเตอร์พรีตเตอร์ภาษา CL ในหัวข้อที่ 6.3 และสรุป ในหัวข้อที่ 6.4

6.1 โครงสร้างอินเตอร์พรีตเตอร์ CL

Statement ในโปรแกรม CL ที่เก็บใน Tree จะถูกนำมาประมวลผลตามกำหนดเวลาและเงื่อนไขของเหตุการณ์ โดยอินเตอร์พรีตเตอร์ CL ที่มีโครงสร้างหลักประกอบด้วย

1. Clock เป็นนาฬิกาของอินเตอร์พรีตเตอร์
2. Timetable เป็นตารางเวลาของอินเตอร์พรีตเตอร์
3. Event-action being observed เป็นที่เก็บเหตุการณ์ที่อยู่ในความสนใจของอินเตอร์พรีตเตอร์
4. Time Responder เป็นเธรดที่คอยอ่านเวลาปัจจุบันมาจาก Clock เพื่อตรวจสอบกับกำหนดการทำคำสั่งที่เก็บอยู่ในตารางเวลาของอินเตอร์พรีตเตอร์
5. Event Listener เป็นเธรดที่คอยรับเหตุการณ์ภายนอก เพื่อมาตรวจความกับ Event-action being observed
6. Scheduler เป็นเธรดที่คอยจัดเก็บกำหนดเวลาในคิวข้อมูล มาเก็บไว้ในตารางเวลา
7. Code Interpreter เป็นเธรดที่ทำงานตามคำสั่ง CL
8. Execution Thread เป็นไลบรารีที่ใช้จัดการเธรด
9. Built-in Function เป็นไลบรารีที่เก็บฟังก์ชั่นที่รวมอยู่ในอินเตอร์พรีตเตอร์ เป็นฟังชั่นการทำงานพื้นฐานของอินเตอร์พรีตเตอร์ เช่น การพิมพ์ข้อความออกมาทางจอภาพ
10. Function Definition เป็นที่เก็บการนิยามของฟังก์ชั่น ที่กำหนดโดยผู้ใช้



รูปที่ 6.1 สถาปัตยกรรมของอินเตอร์พรีตเตอร์ CL

6.2 ขั้นตอนการประมวลผลประโยคภาษา CL

การทำงานเริ่มต้นจากส่วน Time Responder, Event Listener และ Scheduler จะทำงานเป็น เธรดอิสระ จากนั้น Code Interpreter อ่านประโยคจากโปรแกรม CL เข้าไปตีความหมาย ถ้าเป็นการนิยามฟังก์ชั่น ซึ่งจะยังไม่ทำงาน ให้ส่งไปเก็บไว้ในส่วน Function Definition ก่อน (โดยจะถูกเรียกใช้ในภายหลัง เมื่อมี function call เกิดขึ้น) ส่วนโปรแกรมที่ top-level (ชุด statement จะถูกมองเป็นเพียง statement เดียว) จะถูกจัดเก็บเข้าในคิว เพื่อรอให้ Scheduler บันทึกลงใน Tree

หลังจาก statement/event ถูกบันทึกใน Tree แล้ว จากนั้นจะมีส่วน Time Responder ที่อาศัยตัวแปรชื่อ time_cursor (ซึ่งทำหน้าที่เก็บจุดเวลาสิ้นสุดของ statement/event ล่าสุดที่เพิ่งทำงานแล้วเสร็จหรือจุดเวลาเริ่มต้นของ statement/event ล่าสุดที่ได้ทำงาน) ในการอ้างอิงสำหรับ การอ่านจุดเวลาเริ่มต้นของ statement/event ที่กำลังจะต้องทำงานจาก Tree เพื่อสร้างเธรดที่จะประมวลผลมัน หรือ การอ่านจุดเวลาสิ้นสุดของ statement/event ที่กำลังทำงานอยู่ เพื่อทำลายเธรดที่กำลังประมวลผล statement/event นั้นเมื่อถึงเส้นตายเวลาดังกล่าว เพื่อจะได้ไม่ต้องค้นหาจุดเวลาในอดีตใน Tree ซึ่งไม่มีความจำเป็น ซึ่งจะอ่านได้ค่าจุดเวลาอย่างใดออกมา ขึ้นอยู่กับว่าค่าใดอยู่เป็นลำดับต่อจากค่าจุดเวลาของ time_cursor ซึ่งค่านี้เมื่ออ่านออกมาแล้วจะเก็บไว้ในตัวแปร next_point_to_reach ซึ่งจะเก็บจุดเวลาของสิ่งที่จะเกิดขึ้นถัดไป

ในทุกขณะ Time Responder จะทำการอ่านค่าเวลาปัจจุบันจาก Clock ที่มีความละเอียดสูง มาเทียบกับค่าของ next_point_to_reach ณ ขณะนั้น แล้วพิจารณาว่าถ้าเป็น

กรณีที่ 1 ถ้าค่าตรงกับเวลาปัจจุบัน ให้กำหนด time_cursor เป็นค่า next_point_to_reach แล้วทำการพิจารณาว่าค่าเวลาดังกล่าวเป็น

- จุดเวลา start <statement> ก็จะเรียกให้ส่วน Execution Thread สร้างเธรดเพื่อประมวลผล statement ทันที
- จุดเวลา stop <statement> ก็จะเรียกให้ Execution Thread ทำลายเธรดที่ยังประมวลผล statement นี้ไม่แล้วเสร็จทันที
- จุดเวลา start <event> ก็จะบันทึก event-action statement ไว้ใน Event-action being observed
- จุดเวลา stop <event> ก็จะลบ event-action statement ออกจาก Event-action being observed

กรณีที่ 2 ถ้าค่าเวลาเป็น 'af' แล้ว 'af' นั้นเกี่ยวข้องกับ start <statement> แล้ว ก็จะต้องตรวจสอบว่า statement ก่อนหน้าซึ่งระบุโดยจุดเวลาสิ้นสุดที่เก็บอยู่ที่ time_cursor ได้ทำงานเสร็จสิ้นหรือยัง ถ้าเสร็จแล้วก็ให้กำหนด time_cursor เป็นค่า next_point_to_reach แล้วเรียก Execution Thread ให้สร้างเธรดเพื่อประมวลผล statement ทันที มิฉะนั้นก็จะกลับไปที่การทำงานของ Time Responder อีกครั้ง

กรณีที่ 3 ถ้าค่าเวลาเป็น 'f' แล้ว 'f' นั้นเกี่ยวข้องกับ stop <statement> แล้วก็จะต้องตรวจสอบว่า statement นี้ได้ทำงานเสร็จสิ้นหรือยัง ถ้าเสร็จสิ้นแล้วก็ให้ ให้กำหนด time_cursor เป็นค่า next_point_to_reach มิฉะนั้นก็จะกลับไปที่การทำงานของ Time Responder อีกครั้ง

สำหรับส่วน Execution Thread เป็นส่วนประมวลผล statement ซึ่งจะถูกเรียกให้สร้างเธรดขึ้นมาเพื่อประมวลผลโดย Time Responder หรือ Event Listener แต่จะถูกเรียกให้ทำลาย เธรดโดย Time Responder เท่านั้น

ในส่วน Event Listener นั้น รอรับเหตุการณ์ที่นำมาพิจารณาตลอดเวลา และนำมาเปรียบเทียบกับส่วน event ใน event-action statement ที่เก็บอยู่ใน Event-actions being observed ถ้าพบว่า event นี้เกิดขึ้นก็จะเรียกใช้ Execution Thread เพื่อสร้าง เธรด Code Interpreter ขึ้นมาประมวลผลส่วน action ทันที

6.3 การทำงานของส่วน Code Interpreter

Code Interpreter จะรับอินพุทเป็น scope (เป็น dictionary ที่เก็บค่าตัวแปรสำหรับอ้างใน local scope) และ ประโยค CL การประมวลผลประโยคแยกตามประเภทของประโยค ดังนี้

1. ถ้าเป็น function-call statement แล้ว Code Interpreter ก็จะตรวจสอบดูว่า การเรียกฟังก์ชั่นนั้นเป็นการเรียกฟังก์ชั่นอะไร ถ้าเรียก build-in function แล้ว Code Interpreter ก็จะ

ประมวลผล statement นั้นทันที แต่ถ้าเป็นการเรียก user-defined function แล้ว Code Interpreter ก็จะหานิยามของ ฟังก์ชั่นใน Function Definition แล้วนำมาประมวลผลต่อไป

2. ถ้าเป็นนิยามฟังก์ชั่นก็จะจัดเก็บไว้ในส่วนของ Function Definition
3. ถ้าเป็นประโยคที่มีการทำงานเป็นคาบๆ เช่น for-loop statement, while-loop statement หรือ repeat-until statement แล้ว Code Interpreter ก็จะแตกประโยคออกเป็นการทำงานในรอบแรก และการทำงานในรอบที่เหลือ แล้วจัดเก็บลงในโครงสร้างข้อมูล เพื่อรอเวลาประมวลผลต่อไป
4. ถ้าเป็นประโยคชุดคำสั่งก็จะเอา ประโยคย่อย $[p_s,p_e]$:statement ในชุดคำสั่ง นั้นเก็บลงใน Queue แล้วรอให้ Scheduler จัดเก็บลงใน Tree ในลักษณะของ Customer-Producer เพื่อไม่ให้เสียเวลา โดยส่วนนี้จะเป็นส่วน Producer และ Scheduler เป็น Customer

สำหรับ Pseudo code ของ CL Interpreter แสดงได้ดังข้างล่าง

```
def code_interpreter(scope, statement):
  if statement is function-call function:
    if statement invoke build-in function:
      execute build-in function
    else:
      func = lookup function definition
      code_interpreter(scope, func)
  elif statement is a 'function definition':
    // save  function definition
  elif statement is 'for-loop statement':
    s1 = get the first iteration statement
    sn-1 = get the rest iteration statement
    enqueue(queue, (scope, s1))
    if sn-1 exist: enqueue(queue, (scope, sn-1))
  elif statement is a 'sequential statement' or
       statement is a 'parallel statement':
    create nested_scope
    for stmt in sub statements:
      enqueue(queue, (nested_scope, stmt))
    attachScope(scope, nested_scope)
  elif statement is a 'unordered statement':
    create nested_scope
    random select stmt in sub statements:
      enqueue(queue, (nested_scope, stmt))
    attachScope(scope, nested_scope)
```

รูปที่ 6.2 โค้ดในส่วน Code Interpreter

6.4 การทำงานของส่วน Scheduler

Scheduler จะทำงานเป็นเธรด ที่คอยอ่านค่าจาก queue มาจัดเก็บใน โครงสร้างข้อมูล สิ่งที่อ่านมาได้จาก queue จะเป็น tuple ของ scope และ ประโยค $[p_s, p_e]$: statement ประโยคจะถูกจัดเก็บแยก

ตาม scope เป็น Tree คนละต้น โดยที่ Tree แต่ละต้นจะมีเพียงโหนดเดียวที่เชื่อมต่อถึงกัน ตามที่อธิบายไปในบทที่แล้ว

การจัดเก็บจุดเวลาลงในโครงสร้างข้อมูลแบ่งได้เป็น การจัดเก็บจุดเวลาที่มีค่าแน่นอน กับการจัดเก็บจุดเวลาที่มีค่าเปลี่ยนแปลงได้ ได้แก่ ค่าตัวแปร, ค่า 'af' หรือค่า 'f'

การจัดเก็นจุดเวลาที่มีค่าแน่นอน เราจะแทรกโหนดที่แทนจุดเวลานั้นลงใน Tree ส่วนการเพิ่มจุดเวลาที่มีค่าเปลี่ยนแปลงได้ เราจะสร้างโหนด leaf ที่ต่อจากโหนด leaf ขวาสุดของ Tree

สำหรับ Pseudo code ของ Scheduler แสดงได้ดังข้างล่าง

```
def scheduler():
  whenever queue is not empty:
    (scope, [ps,pe]:statement) = dequeue(queue)
    if ps == '_': ps = 'af'
    if pe == '_': pe = 'f'
    insertTPoint(scope, ps, (start, statement))
    insertTPoint(scope, pe, (stop , statement))

def insertTPoint(scope, timepoint, symbol):
  time_tree = scope.time_tree
  if timepoint is 'ABSOLUTE_TIME':
   /*
   insert path into time_tree to represent timepoint
    append symbol to the created path
  */
  if timepoint is 'af' or 'f' or variable:
   /*
     append to the rightmost path of the time_tree
    append symbol to the created path
  */
```

รูปที่ 6.3 โค้ดในส่วน Scheduler

6.5 การทำงานของส่วน Time Responder

Time Responder ทำงานเป็นเธรด ที่คอยมอนิเตอร์ประโยค CL ที่จัดเก็บในโครงสร้าง เพื่อนำประโยคมาประมวลผลให้ตรงกับเวลาที่ต้องการ

การทำงานของ Time Responder จะประมวลผล nested scope ที่อยู่ที่ลึกที่สุดก่อน จากนั้นก็จะค่อยๆ ประมวลผลประโยคที่อยู่ใน scope ที่สูงขึ้นเรื่อยๆ

เนื่องจาก Time Responder จะอ่านค่าจุดเวลาจาก Tree ตามลำดับเวลาจากน้อยไปหามาก ดังนั้นเราจึงใช้ time_cursor ในการเข้าถึงจุดเวลาตามลำดับ โดยกำหนดให้ Tree ในแต่ละ scope มี time_cursor แยกกัน เมื่อใดก็ตามที่ จุดเวลาถัดจากจุดเวลาที่ time_cursor ชี้อยู่ กับจุดเวลาปัจจุบัน มีค่าเท่ากันก็ให้ นำคำสั่งของจุดเวลานั้นมาประมวลผล และย้าย time_cursor ไปชี้จุดเวลาล่าสุดที่เพิ่งประมวลผลไป

การตีความหมายค่าที่จัดเก็บอยู่ที่โหนด leaf ในรูปแบบ (prefix, statement) มีกรณีต่างๆ ดังนี้

1. (start, event->action) : Time Responder จะจัดเก็บ event->action ไว้ใน Event-action being observed
2. (stop, event->action) : Time Responder จะลบ event->action ออกจาก Event-action being observed
3. (start, statement): Time Responder จะเริ่มสร้างเธรด Code Interpreter เพื่อประมวลผล statement นั้น
4. (stop, statement): Time Responder จะตรวจสอบการทำงานของเธรดที่ประมวลผล statement นั้น ถ้าพบว่าเธรดยังทำงานไม่เสร็จ ก็จะถือว่ามีความผิดของการทำงาน และแจ้งความผิดพลาดให้กับผู้ใช้ในรูปของ exception

สำหรับ Pseudo code ของ Time Responder แสดงได้ดังข้างล่าง

```
def time_responder():
  for every 1 second:
    response_for(rootScope, getCurrentTime())

def response_for(scope, timepoint):
  for nested_scope under scope:
    response_for(nested_scope, timepoint)
  scope.next_point_to_reach = scope.time_cursor_next()
  if scope.next_point_to_reach == timepoint:
    scope.time_cursor = scope.next_point_to_reach
    handle_start_stop(scope)
  elif scope.next_point_to_reach == 'af':
    if the previous statement was complete:
      scope.time_cursor = next_point_to_reach
      handle_start_stop(scope)
  elif scope.next_point_to_reach == 'f':
    if the current statement was complete:
      scope.time_cursor = next_point_to_reach
      handle_start_stop()
  else:
      return 'free'

def handle_start_stop(scope)
  symbol_list = scope.time_cursor.getSymbols()
  for (prefix, stmt) in symbol_list:
    if stmt is 'event-action statement':
      if prefix == 'start':
        observed.insert ((scope, stmt))
      else: observed.remove((scope,stmt))
    else:
      if prefix == 'start':
        startThread(code_interpreter(scope,stmt))
      else:
        stopThread(code_interpreter(scope,stmt))
```

รูปที่ 6.4 โค้ดในส่วน Time Responder

6.6 การทำงานของส่วน Event Listener

Event Listener ทำงานเป็นเธรด คอยฟังเหตุการณ์จากภายนอก เมื่อใดที่พบว่ามีเหตุการณ์เกิดขึ้น ก็จะตรวจสอบกับ Event-action being observed ถ้าพบว่าเหตุการณ์ที่ได้รับมา เป็นเหตุการ์ที่รออยู่ Event Listener ก็จะสร้างเธรด Code Interpreter มาตอบสนองเหตุการณ์

สำหรับ Pseudo code ของ Event Listener แสดงได้ดังข้างล่าง

```
def event_listener():
  wait for event e:
    (scope,event→action) = observed.lookup(e)
    startThread(code_interpreter(scope, action))
```

รูปที่ 6.5 โค้ดในส่วน Event Listener

6.7 สรุป

อินเตอร์พรีตเตอร์ CL ประกอบด้วยเธรดหลักๆ 4 เธรด ได้แก่ Time Responder, Event Listener และ Scheduler โดยเธรดเหล่านี้จะทำงานเป็นอิสระจากกัน และเริ่มทำงานพร้อมกันตั้งแต่ตอนเริ่มระบบอินเตอร์พรีตเตอร์ และทำงานเรื่อยไปจนกระทั่งระบบอินเตอร์พรีตเตอร์ยุติการทำงาน ในระหว่างการประมวลผล อินเตอร์พรีตเตอร์จะมีการสร้างเธรด Code Interpreter เพื่อประมวลผลโปรแกรม จำนวนเธรดนี้จะขึ้นอยู่กับ จำนวนประโยคที่ต้องการให้ทำงานพร้อมๆ กัน

บทที่ 7
การเปรียบเทียบ CL กับ Timer ในลีนุกซ์

Timer เป็นบริการในระบบปฏิบัติการลีนุกซ์ ที่ใช้ตั้งเวลาเพื่อเรียกใช้งานฟังก์ชั่น ซึ่งมีความสามารถคล้ายคลึงกับอินเตอร์พรีตเตอร์ CL เราจึงขอเปรียบเทียบสองระบบนี้ โดยเนื้อหาภายในบทจะเริ่มจากการอธิบายการอ้างอิงเวลาเวลาในลีนุกซ์ ในหัวข้อที่ 7.1 จากนั้นจะกล่าวถึงการทำงานของ Timer ในหัวข้อ 7.2 ตามมาด้วยการเรียกใช้งาน Timer ในหัวข้อ 7.3 จากนั้นจะกล่าวถึงโครงสร้างข้อมูลสำหรับ Timer ในหัวข้อที่ 7.4 ตามมาด้วย อัลกอลิธึมสำหรับโครงสร้างข้อมูล เช่น การเข้าถึงจุดเวลาตามลำดับ การค้นหาจุดเวลา การเพิ่มจุดเวลา และการลบจุดเวลา ในหัวข้อที่ 7.5 ต่อด้วยการวิเคราะห์โครงสร้างข้อมูลในหัวข้อที่ 7.6 จากนั้นจะเป็นผลวิเคราะห์การเปรียบเทียบ Timer และอินเตอร์พรีตเตอร์ CL ในด้านต่างๆ ในหัวข้อที่ 7.7 และสรุปในหัวข้อ 7.8

7.1 การอ้างอิงเวลาในลีนุกซ์

ระบบปฏิบัติการลีนุกซ์[18] ทำงานบนเครื่องคอมพิวเตอร์ส่วนบุคคล ใช้อุปกรณ์หลายอย่างในการนับเวลา ได้แก่ Real-time clock, Time stamp counter (TSC) และ Programmable Interrupt Timer (PIT) ซึ่งแต่ละอุปกรณ์ก็มีการใช้งานแตกต่างกัน โดยในสองอุปกรณ์แรกจะใช้สำหรับเก็บรักษาค่าเวลาปัจจุบัน เพื่อให้ลีนุกซ์อ่านไปใช้ ส่วนอุปกรณ์อันสุดท้ายจะทำงานเป็นตัวสร้างอินเตอร์รัพท์ไปยัง CPU รายละเอียดของแต่ละอุปกรณ์มีดังนี้

- Real-time clock เป็นอุปกรณ์ที่มีอยู่ในคอมพิวเตอร์ทุกเครื่อง มีการทำงานที่เป็นอิสระจาก CPU และชิบอื่นๆ สามารถทำงานได้ตลอดเวลาแม้จะปิดเครื่องไป (ใช้ไฟจากแบตเตอร์รี่) มีการจัดเก็บเวลาอยู่ในรูปแบบ ปี เดือน วัน ชั่วโมง นาที และวินาที ตามลำดับ เวลานี้จะถูกใช้สำหรับเป็นค่าเริ่มต้นให้กับระบบปฏิบัติการ เพื่อให้ระบบปฏิบัติการรู้เวลาที่แท้จริงทุกครั้งที่เริ่มระบบขึ้นมาใหม่
- Time stamp counter (TSC) เป็นรีจิสตี้ขนาด 64 บิทอยู่ภายใน CPU รุ่น Pentium ขึ้นไป ค่า TSC จะถูกนับเพิ่มขึ้น ทุกๆ ครั้งที่ CPU ได้รับสัญญาณ Clock ค่า TSC จะถูกนับเพิ่มไปจนกระทั่ง overflow ก็จะนับวนกลับมาเป็น 0 อีกครั้ง เราสามารถหาคาบเวลา T ของ Clock ได้จากอินเวอร์สของความถี่ CPU (f)

$$T = \frac{1}{f} \qquad [7.1]$$

คาบเวลานี้เมื่อนำไปคูณกับ ค่าใน TSC ก็จะได้เวลาในหน่วยวินาทีออกมา

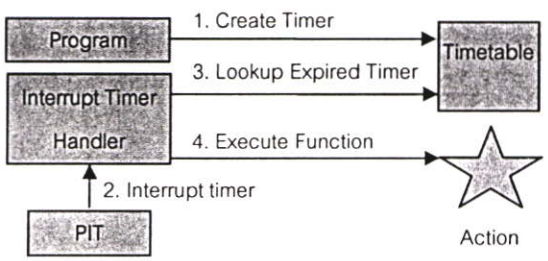
ค่า TSC สามารถแทนเวลาที่มีความระเอียดของเวลาในระดับ nanosecond แต่ระยะเวลาสูงสุดที่ TSC สามารถวัดได้จะไม่มากนัก เพราะค่า TSC จะ overflow ทำให้โปรแกรมเมอร์มักใช้ค่านี้เพื่อวัดเวลาในการประมวลผลคำสั่ง หรืองานที่ต้องการความละเอียดของเวลาสูงเท่านั้น

- Programmable Interrupt Timer (PIT) เป็นชิปที่ติดมากับเครื่องคอมพิวเตอร์ส่วนใหญ่ ที่ผลิตตามโมเดลของ IBM PC เมื่อเริ่มเปิดเครื่อง PIT จะถูก OS โปรแกรมให้ผลิตอินเตอร์รัพท์ โดยในลีนุกซ์ ได้โปรแกรมให้ผลิตอินเตอร์รัพท์ไปขัดจังหวะการทำงาน CPU ทุกๆ 10 ms ซึ่งเราจะเรียกอินเตอร์รัพท์นี้ว่า Interrupt Timer

การเกิด Interrupt Timer แต่ละครั้ง เราจะเรียกว่า 1 tick ซึ่งจำนวน tick จะถูกนับไว้ในตัวแปรขนาด 32 bit ที่ชื่อว่า jiffies เพื่อแทนจุดเวลาปัจจุบันในลีนุกซ์

7.2 การทำงานของ Timer

Timer จะกระทำฟังก์ชั่นงานตามกำหนดเวลา โปรแกรมใดๆ สามารถสร้าง Timer ได้โดยการบันทึกค่าของ Timer ที่ประกอบด้วยจุดเวลาที่หมดอายุและฟังก์ชั่นการทำงานลงในโครงสร้างข้อมูลสำหรับเก็บกำหนดเวลา (Timetable) หลังจากนั้นโครงสร้างข้อมูลดังกล่าวจะถูกโพลลิ่งด้วย Interrupt Timer Handler เพื่อหา Timer ที่หมดอายุ แล้วจึงเรียกใช้งานฟังชั่นใน Timer ขั้นตอนการใช้งาน Timer เป็นดังรูป



รูปที่ 7.1 การทำงานของ Timer

7.3 การเรียกใช้ Timer ในลีนุกซ์

โครงสร้างข้อมูลสำหรับ Timer ในลีนุกซ์มีชื่อว่า struct timer_list มีโครงสร้างดังนี้

```
struct timer_list {
   struct timer_list *prev;
   struct timer_list *next;

   unsigned long expires;
   void (*function)(unsigned long);
   unsigned long arg;
}
```

Timer ในลีนุกซ์จะถูกเก็บเป็นลิงค์ลิสต์สองทาง (Doubly Link List) โดยมี prev ชี้ไปที่ Timer ก่อนหน้าและ next ชี้ไปที่ Timer ตัวถัดไป ส่วนตัวแปร expires, function และ arg หมายถึง จุดเวลาหมดอายุ, ฟังก์ชั่นที่ต้องเรียก และข้อมูลที่เป็นอากิวเมนต์ให้กับฟังก์ชั่น ตามลำดับ

ค่า expires จะอยู่ในรูปของตัวเลข 32 บิท ซึ่งเป็นผลรวมของค่า jiffies กับอายุของ Timer (นับเป็นจำนวน tick) เช่น สมมติว่า ณ เวลาที่จัดเก็บ Timer ค่า jiffies เป็น 50 แล้ว ถ้าเราต้องการให้ Timer มีอายุ 100 tick จะได้ว่า Timer จะมีจุดเวลาหมดอายุที่ tick ที่ 150

การสร้างหรือลบ Timer ในลีนุกซ์ สามารถทำได้ด้วยการเรียกฟังก์ชั่นต่อไปนี้

```
void add_timer(struct timer_list *timer);
int del_timer(struct timer_list * timer);
```

ฟังชั่น add_timer เป็นการเพิ่ม timer เข้าไปเก็บไว้ในตารางเวลา

ฟังก์ชั่น del_timer เป็นการลบ timer ออกจากตารางเวลา

สังเกตว่าการลบ Timer เราจะต้องมีตัวชี้ไปที่ Timer ตัวนั้นก่อน เนื่องจากในลีนุกซ์ ไม่มีฟังก์ชั่นที่รองรับการค้นหา Timer จากจุดเวลาหมดอายุ

7.4 โครงสร้างข้อมูลสำหรับเก็บจุดเวลาในลีนุกซ์

Timer ที่เก็บในโครงสร้างข้อมูลของลีนุกซ์ มิได้เก็บเป็นลิสต์เพียงสายเดียว แต่การทำดรรชนีของ Timer ออกเป็นช่วง ซึ่งแต่ละช่วงสามารถอ้างด้วยค่าจุดเวลา 32 บิท ลีนุกซ์แบ่งตัวเลข 32 บิท ออกเป็น 5 ตัวเลข ตัวเลขแรกได้แก่ 8 บิทซ้ายสุด ตัวเลขตัวถัดมามีขนาด 6 บิทเท่าๆ กัน โดยเราขอตั้งชื่อแต่ละตัวเลขที่ได้เป็น t_5-t_1 ตามลำดับ ดังรูป

t_5 บิทที่ 31-26	t_4 บิทที่ 25-20	t_3 บิทที่ 19-14	t_2 บิทที่ 13-8	t_1 บิทที่ 7-0

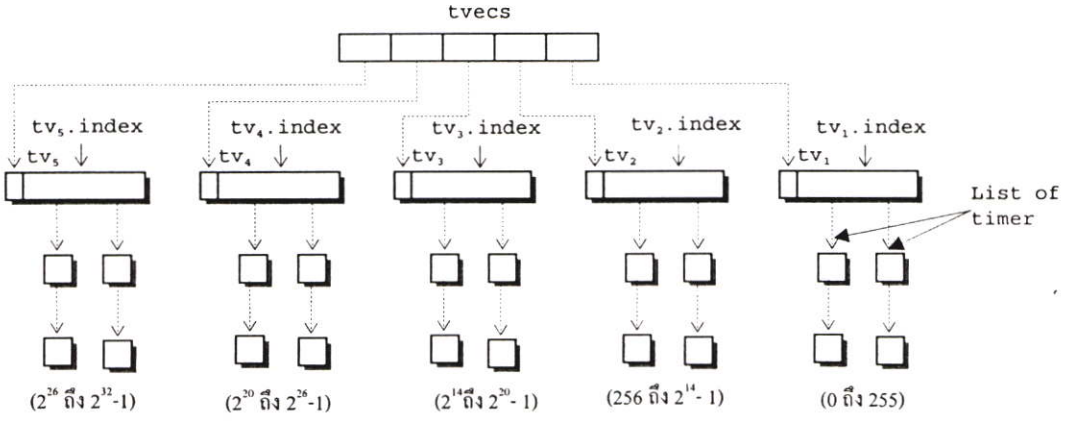
รูปที่ 7.2 การแบ่งกลุ่มของค่าจุดเวลา

ค่าของ t_1 จะมีค่าในช่วง 0-255 และ $t_2 - t_5$ จะมีค่าอยู่ในช่วง 0-63

เราอาจระบุจุดเวลาของลีนุกซ์ในรูปของ tuple (t_5, t_4, t_3, t_2, t_1) โดยโครงสร้างข้อมูลของลีนุกซ์ถูกออกแบบมาเพื่อจัดเก็บจุดเวลาในลักษณะนี้

โครงสร้างข้อมูลของลีนุกซ์มีลักษณะดังรูปด้านล่าง ประกอบด้วยโครงสร้างหลักที่เป็นอาร์เรย์ของตัวชี้ (tvecs) ที่ชี้ไปยังโครงสร้างข้อมูล tv_5-tv_1 ตามลำดับ โดยที่โครงสร้างข้อมูล tv_1 จะ

ประกอบด้วย อินเด็กซ์ (index) และอาร์เรย์ (vec) ขนาด 256 ช่อง ส่วน tv_2-tv_4 จะประกอบด้วย อินเด็กซ์ (index) และ อาร์เรย์ขนาด 64 ช่อง ในแต่ละช่องของ tv_1- tv_5 จะเก็บลิสต์ของ Timer



รูปที่ 7.3 โครงสร้างข้อมูลของลีนุกซ์

เราสามารถจัดเก็บ Timer ที่ expired ตั้งแต่ (0, 0, 0, 0, 0) จนถึง (63, 63, 63, 63, 255) ไว้ใน tv1 ถึง tv5 ตามลำดับ ถ้ากำหนดให้ i คือตัวเลขจำนวนเต็ม และ x คือตัวเลขจำนวนเต็มใดๆ เราสามารถสรุปรูปแบบการจัดเก็บจุดเวลาได้ดังนี้

จุดเวลา (0,0,0,0,i) โดยที่ 0 <= i < 256 จะถูกจัดเก็บในลิสต์ tv_1.vec[i]

จุดเวลา (0,0,0,i,x) โดยที่ 0 < i < 64 จะถูกจัดเก็บในลิสต์ tv_2.vec[i]

จุดเวลา (0,0,i,x,x) โดยที่ 0 < i < 64 จะถูกจัดเก็บในลิสต์ tv_3.vec[i]

จุดเวลา (0,i,x,x,x) โดยที่ 0 < i < 64 จะถูกจัดเก็บในลิสต์ tv_4.vec[i]

จุดเวลา (i,x,x,x,x) โดยที่ 0 < i < 64 จะถูกจัดเก็บในลิสต์ tv_5.vec[i]

การเก็บจุดเวลาด้วยวิธีนี้จะทำให้ได้ว่า Timer ที่หมดอายุ ใน tick ที่ 0 ถึง 255 จะถูกเก็บใน tv_1 และ เก็บ tick ที่ (256 ถึง 2^{14}-1), (2^{14}ถึง 2^{20}- 1), (2^{20} ถึง 2^{26}-1) และ (2^{26} ถึง 2^{32}-1) ไว้ใน tv_2, tv_3, tv_4, tv_5 ตามลำดับ

ในตอนเริ่มต้น index ของ tv_1 – tv_5 จะมีค่าเป็น 0 จากนั้น ทุกๆครั้งที่เกิด Interrupt Timer ค่า tv_1.index จะเพิ่มขึ้น 1 และเราสามารถนำลิสต์ของ Timer tv_1.vec[tv1.index] ไปประมวลผล ค่า tv_1.index นี้จะ เพิ่มขึ้นเรื่อยๆ จนมีค่าเป็น 256 แล้วจึงกลับมาเป็น 0 อีกครั้ง ซึ่งหมายถึงคำสั่งที่จัดเก็บใน tv_1.vec ถูกประมวลผลจนหมดแล้ว ดังนั้น Interrupt Timer Handler จำเป็นต้องเพิ่มค่า tv_2.index ขึ้นหนึ่ง และย้าย Timer จาก tv_2.vec[tv_2.index] มาเก็บใน tv_1.vec ใหม่อีกครั้ง

ค่า tv_2.index จะเพิ่มค่าขึ้นเรื่อยๆ จนมีค่า 64 ก็จะกลับมาเป็น 0 อีกครั้งหนึ่ง ก็จะมีการเพิ่มค่า tv_3.index ขึ้น 1 แล้วย้าย Timer จาก tv_3.vec[tv_3.index] ไปยัง tv_2.vec และในทุกๆ ครั้งที่มีการนับค่า tv4.index, tv5.index จนเต็ม ก็จะมีการย้ายข้อมูลเช่นเดียวกัน

ดังนั้นเราสามารถสรุปการย้าย Timer ให้อยู่ในรูปทั่วไปได้ว่า เมื่อใดก็ตาม ที่ tv_j.vec (j มีค่าอยู่ระหว่าง 1 ถึง 5) ถูกประมวลผลจนหมดแล้ว จะมีการเพิ่มค่า tv_{j+1}.index ขึ้นหนึ่ง และ tv_j จะถูกแทนที่ด้วยค่าใหม่ ที่โหลดมาจาก $tv_{j+1}[tv_{j+1}.index]$

7.5 อัลกอลิธึมบนโครงสร้างข้อมูลของลีนุกซ์

7.5.1 การเข้าถึงจุดเวลาตามลำดับ

การเข้าถึงจุดเวลาตามลำดับ เป็นฟังก์ชั่นที่ถูกเรียกใช้บ่อยๆ จาก Interrupt Timer Handler ฟังก์ชั่นนี้จะอ่านค่า Timer ที่จัดเก็บอยู่ในโครงสร้างข้อมูล โดยเพิ่มค่า tv_1.index ขึ้น 1 จากนั้นจึงอ่านลิสต์ของ Timer จาก tv_1.vec[tv_1.index] ออกไปประมวลผล

7.5.2 การค้นหาจุดเวลา

ฟังก์ชั่นนี้ยังไม่ถูกอิมพลีเมนต์ในลีนุกซ์ เนื่องจากใช้เวลาในการทำงานมาก แต่เราจะอธิบายฟังก์ชั่นนี้ เพื่อเปรียบเทียบกับโครงสร้างข้อมูลของ CL โดยฟังก์ชั่นนี้รับจุดเวลาหมดอายุมาเป็นอาร์กิวส์เมนต์ แล้วนำค่าดังกล่าวมาเปรียบเทียบว่าตกอยู่ในช่วงใด (tv_1-tv_5) จากนั้นจึงคำนวณต่อไปว่าจุดเวลาถูกเก็บที่ลิสต์ใด ใน tv_j จากนั้น เราจึงสามารถสแกนหาจุดเวลาที่ต้องการจากลิสต์ดังกล่าว

การหาค่าลิสต์ที่จัดเก็บจุดเวลาสามารถทำได้โดยการเปรียบเทียบ t_j กับ tv_j.index สำหรับค่า j ย้อนกลับจาก 5 ไปจนถึง 1 ถ้า t_j กับ tv_j.index มีค่าเท่ากัน แสดงว่า จุดเวลาถูกโหลดเข้าไปใน tv_{j-1} แล้ว ดังนั้นจึงให้ดำเนินการต่อในลูปถัดไป มิฉะนั้นแล้ว แสดงว่าจุดเวลาอยู่ในระดับ tv_j เราสามารถนำลิสต์ tv_j.vec[t_j] ไป สแกนหาจุดเวลาที่ต้องการต่อได้

เราสามารถแสดงซูโดโค้ดการค้นหาจุดเวลาในโครงสร้างข้อมูลของลีนุกซ์ได้ดังนี้

```
void find_timer(unsigned long expires):
  for j from 5 to 1:
    tj = get(expire, j)
    if tv[j].index == tj :
      continue
    else:
      return find_in_list(tv[j].vec[tj], expires)
```

รูปที่ 7.4 การค้นหาในโครงสร้างข้อมูลของลีนุกซ์

7.5.3 การเพิ่มจุดเวลา

การเพิ่มจุดเวลา มีการทำงานคล้ายกับการค้นหาจุดเวลา โดยการทำงานในตอนต้นจะเป็นการหาลิสต์ที่เก็บจุดเวลาก่อน จากนั้นจึงนำจุดเวลาที่ต้องการจัดเก็บต่อท้ายลิสต์ดังกล่าว

ซูโดโค้ดของการเพิ่มจุดเวลาในโครงสร้างข้อมูลของลีนุกซ์เป็นดังนี้

```
void add_timer(struct timer_list *timer){
  for j from 5 to 1:
    t_j = get(timer->expire, j)
    if tv[j].index == t_j :
      continue
    else:
      return tv[j].vec[t_j].append(timer)
```

รูปที่ 7.5 การเพิ่มจุดเวลาในโครงสร้างข้อมูลของลีนุกซ์

7.5.4 การลบจุดเวลา

การลบจุดเวลาในลีนุกซ์ รับตัวชี้ timer_list เป็นอาร์กิวเมนต์ นั้นหมายความว่า เราต้องมีตำแหน่งที่จัดเก็บจุดเวลาที่ต้องการลบก่อน การทำงานของฟังก์ชั่นนี้ จะเป็นการลบจุดเวลาออกจากลิงค์ลิสต์สองทาง ซึ่งทำได้ไม่ยากนัก

การลบจุดเวลาโดยที่ไม่รู้ตำแหน่งจัดเก็บจุดเวลาล่วงหน้า ยังไม่มีอิมพลีเมนต์ในลีนุกซ์ แต่เราสามารถใช้วิธีการเดียวกับการค้นหาจุดเวลาได้ แล้วลบจุดเวลานั้นออกจากลิสต์

7.6 การวิเคราะห์ความซับซ้อนของโครงสร้างข้อมูลของลีนุกซ์

กำหนดให้ n คือจำนวนจุดเวลาที่ถูกเก็บในโครงสร้างข้อมูล และจุดเวลาที่จัดเก็บไม่ซ้ำกันเลย

1. การเพิ่มจุดเวลา ประกอบด้วย การค้นหาลิสต์ที่ใช้เก็บจุดเวลานั้น และการนำจุดเวลาไปต่อท้ายลิสต์ ซึ่งทั้งสองส่วนทำงานโดยใช้เวลาคงที่ ไม่เปลี่ยนตามจำนวนจุดเวลาเก็บอยู่ในโครงสร้างข้อมูล ดังนั้นจึงทำให้การเพิ่มจุดเวลาทำได้โดยใช้เวลาคงที่
2. การเข้าถึงจุดเวลาแบบตามลำดับ ในกรณีที่ดีที่สุด จุดเวลาถูกเก็บอยู่ใน tv_1 แล้ว Interrupt Timer Hander สามารถอ่านค่าจุดเวลาปัจจุบันจากตำแหน่ง tv_1.vec[tv_1.index] ได้ทันที แต่ในบางครั้ง ถ้า tv_1.vec ถูกอ่านจนหมดแล้ว แล้วค่าจุดเวลาต้องถูกโหลดมาใหม่จาก tv_2.vec[tv_2.index] ซึ่งต้องคิดเวลาในการย้ายข้อมูลเพิ่มขึ้นมาด้วย
3. การค้นหาจุดเวลานั้น ในกรณีที่จุดเวลาที่ค้นหามีจำนวน tick อยู่ในช่วง 0-255 ซึ่งถูกเก็บอยู่ใน tv_1.vec ดังนั้นเราสามารถอ่านคำสั่งจาก tv_1.vec ได้ในทันที แต่ในกรณีที่จุดเวลามีจำนวน tick มากกว่า 256 เราจำเป็นต้องค้นหาลงไปในลิสต์ด้วย
4. การลบจุดเวลา สามารถวิเคราะห์ความซับซ้อนได้เช่นเดียวกับการค้นหาจุดเวลา

7.7 การเปรียบเทียบการจัดการเวลาใน CL กับการจัดการเวลาของลีนุกซ์

เราสามารถเปรียบเทียบได้ในหลายด้านดังนี้

7.7.1 การเปรียบเทียบความซับซ้อนด้านเวลาของฟังก์ชั่น

เพื่อให้สามารถเปรียบเทียบประสิทธิภาพของโครงสร้างข้อมูลได้ เราขอเปลี่ยนลักษณะการระบุจุดเวลาใน CL ใหม่ให้เหมือนกับลีนุกซ์ โดยเวลาระบุด้วย tuple

$$(t_5, t_4, t_3, t_2, t_1)$$

t_5 - t_2 มีค่าอยู่ในช่วง 0-63 และ t1 มีค่าอยู่ในช่วง 0 - 255 จากการระบุเวลาในรูปแบบนี้ทำให้เราอ้างถึงเวลาได้ไม่เกิน 2^{32} จุดเวลา ซึ่งกระบวนการจัดเก็บจุดเวลาโครงสร้างข้อมูล CL ยังเหมือนเดิมทุกประการ

7.7.1.1 การเข้าถึงจุดเวลาแบบตามลำดับ

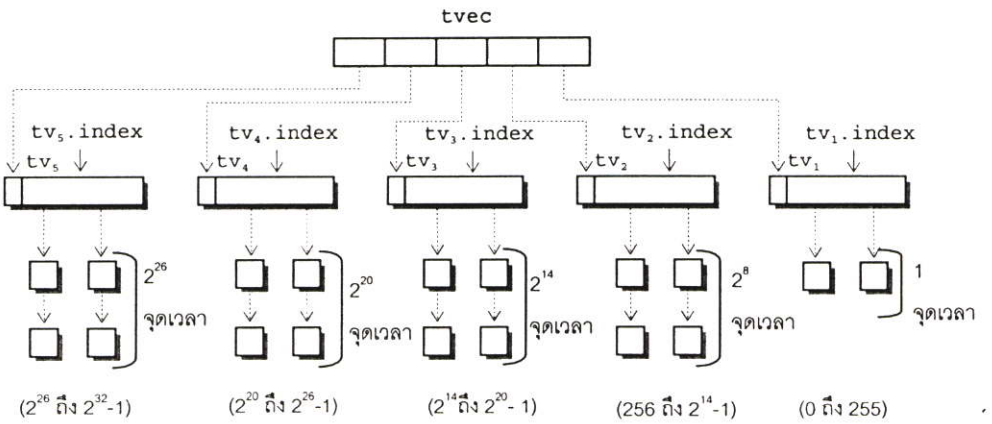
ความซับซ้อนของฟังก์ชั่นนี้ ต้องคิดเวลาทั้งหมด เริ่มนับตั้งแต่ (0, 0, 0, 0, 0) จนกระทั่งจุดเวลา (63, 63, 63, 63, 255) เพื่อให้สะท้อนถึงเวลารวมในการนับแต่ละครั้ง นอกจากนี้การเข้าถึงจุดเวลาในลีนุกซ์ ยังมีคุณลักษณะที่แตกต่างกัน ตามกรณี ทำให้เราพิจารณาเปรียบเทียบโครงสร้างข้อมูลเป็นกรณีต่างๆ ดังนี้

1. กรณีจุดเวลาถูกเก็บเต็มโครงสร้างข้อมูล
2. กรณีจุดเวลาถูกเก็บอยู่ในช่วงขอบเขตบน
3. กรณีจุดเวลาถูกเก็บอยู่ในช่วงขอบเขตล่าง

กรณีที่ 1 : จุดเวลาถูกเก็บเต็มโครงสร้างข้อมูล

ในโครงสร้างข้อมูลของลีนุกซ์ การเข้าถึงจุดข้อมูลตามลำดับ ต้องใช้การย้ายจุดเวลาจากตำแหน่งต่างๆ มายัง tv_1 โดยเราสามารถวิเคราะห์เวลาที่ใช้ได้เป็น เวลาที่ใช้สำหรับการเพิ่มค่า tv.index (c_{inc}) และเวลาที่ใช้ย้ายจุดเวลาจาก tv_j.vec[tv_j.index]ในลิสต์ลงมาใส่ใน tv_{j-1}.vec ถ้ากำหนดให้ c_{move} คือเวลาในการย้ายจุดเวลาแต่ละจุด และ n_{tvj} คือจำนวนจุดเวลาที่จัดเก็บในลิสต์ของ tv_j.vec[tv_j.index] โดยในกรณีที่จุดเวลาถูกเก็บจนเต็ม และไม่มีจุดเวลาใดซ้ำกัน เราจะได้ว่า

- จำนวนจุดเวลาที่จัดเก็บในลิสต์ของ tv_5.vec[tv_5.index] (n_{tv5}) จะเท่ากับ 2^{26} จุดเวลา
- จำนวนจุดเวลาที่จัดเก็บในลิสต์ของ tv_4.vec[tv_4.index] (n_{tv4}) จะเท่ากับ 2^{20} จุดเวลา
- จำนวนจุดเวลาที่จัดเก็บในลิสต์ของ tv_3.vec[tv_3.index] (n_{tv3}) จะเท่ากับ 2^{14} จุดเวลา
- จำนวนจุดเวลาที่จัดเก็บในลิสต์ของ tv_2.vec[tv_2.index] (n_{tv2}) จะเท่ากับ 2^{8} จุดเวลา
- จำนวนจุดเวลาที่จัดเก็บในลิสต์ของ tv_1.vec[tv_1.index] (n_{tv1}) จะเท่ากับ 1 จุดเวลา



รูปที่ 7.6 จำนวนจุดเวลาที่เก็บจัดเก็บได้ในลิสต์

เราสามารถหาเวลารวมในการเข้าถึงจุดเวลา (c_{seq_access})ในการ tick d ครั้งได้ว่า เป็นผลรวมของ

- เวลาที่ใช้ในการเพิ่มค่า tv_1.index และเวลาในการย้ายค่าจาก tv_1.vec[tv_1.index] ออกไปใช้งาน ซึ่งเกิดขึ้น d ครั้ง
- เวลาที่ใช้ในการเพิ่มค่า tv_2.index และเวลาในการย้ายจุดเวลาจากลิสต์ tv_2.vec[tv_2.index] ลงไปเก็บไว้ใน tv_1 ซึ่งจะเกิดขึ้น $\frac{d}{2^8}$ ครั้ง
- เวลาที่ใช้ในการเพิ่มค่า tv_3.index และเวลาในการย้ายจุดเวลาจากลิสต์ tv_3.vec[tv_3.index] ลงไปเก็บไว้ใน tv_2 ซึ่งจะเกิดขึ้นหรือ $\frac{d}{2^{14}}$ ครั้ง
- เวลาที่ใช้ในการเพิ่มค่า tv_4.index และเวลาในการย้ายจุดเวลาจากลิสต์ tv_4.vec[tv_4.index] ลงไปเก็บไว้ใน tv_3 ซึ่งจะเกิดขึ้นหรือ $\frac{d}{2^{20}}$ ครั้ง
- เวลาที่ใช้ในการเพิ่มค่า tv_5.index และเวลาในการย้ายจุดเวลาจากลิสต์ tv_5.vec[tv_5.index] ลงไปเก็บไว้ใน tv_4 ซึ่งจะเกิดขึ้นหรือ $\frac{d}{2^{26}}$ ครั้ง

เราสามารถสรุปเป็นสมการได้ดังนี้

$$c_{seq\_access} = d \times (c_{inc} + n_{tv1} \times c_{move}) + \frac{d}{2^8} \times (c_{inc} + n_{tv2} \times c_{move}) +$$

$$\frac{d}{2^{14}} \times (c_{inc} + n_{tv3} \times c_{move}) + \frac{d}{2^{20}} \times (c_{inc} + n_{tv4} \times c_{move}) +$$

$$\frac{d}{2^{26}} \times (c_{inc} + n_{tv5} \times c_{move}) \qquad [7.2]$$

ถ้าพิจารณาจากสมการดังกล่าวข้างต้น เราจะพบว่าค่า c_{inc} และ c_{move} เป็นค่าคงที่ ส่วน n_{tv1}, n_{tv2}, n_{tv3}, n_{tv4}, n_{tv5} และ d เป็นค่าที่แปรผันได้ ดังนั้นเวลาในการเข้าถึงจุดเวลาตามลำดับในโครงสร้างข้อมูลของลีนุกซ์ จึงอยู่ในระดับ $O(d \times n_{tv1} + \frac{d}{2^8} \times n_{tv2} + \frac{d}{2^{14}} \times n_{tv3} + \frac{d}{2^{20}} \times n_{tv4} + \frac{d}{2^{26}} \times n_{tv5})$

ถ้าเปรียบเทียบโครงสร้างข้อมูลของ CL กับโครงสร้างของลีนุกซ์ เราพบว่า การเข้าถึงจุดเวลาตามลำดับของ CL จะใช้วิธีการอัพเดทค่า cursor ที่เก็บอยู่ในรูปของ Stack ซึ่งเวลาพื้นฐานที่ใช้จะประกอบด้วย เวลาในการ pop ข้อมูลออกมาจาก stack (c_{pop}) และเวลาที่ใช้ในการเป็นตำแหน่งของการชี้ ไปยังโหนดลูกถัดไป (c_{next})

ในกรณีที่จุดเวลาถูกเก็บจนเต็มโครงสร้าง เวลารวมในการเข้าถึงจุดเวลาสำหรับการ tick d ครั้ง จะเป็นผลรวมของ

- เวลาที่ใช้ในการ pop โหนดในระดับหน่วยเวลา t1 ออกมา และเวลาในการจัดเก็บโหนดถัดไป ซึ่งเกิดขึ้น d ครั้ง
- เวลาใช้ในการ pop โหนดในระดับหน่วยเวลา t2 ออกมา และเวลาในการจัดเก็บโหนดถัดไป ซึ่งจะเกิดขึ้น $\frac{d}{2^8}$ ครั้ง
- เวลาที่ใช้ในการ pop โหนดในระดับหน่วยเวลา t3 ออกมา และเวลาในการจัดเก็บโหนดถัดไป ซึ่งจะเกิดขึ้นหรือ $\frac{d}{2^{14}}$ ครั้ง
- เวลาที่ใช้ในการ pop โหนดในระดับหน่วยเวลา t4 ออกมา และเวลาในการจัดเก็บโหนดถัดไป ซึ่งจะเกิดขึ้นหรือ $\frac{d}{2^{20}}$ ครั้ง
- เวลาที่ใช้ในการ pop โหนดในระดับหน่วยเวลา t5 ออกมา และเวลาในการจัดเก็บโหนดถัดไป ซึ่งจะเกิดขึ้นหรือ $\frac{d}{2^{26}}$ ครั้ง
- เวลาที่ใช้สำหรับในการเปรียบเทียบเวลาปัจจุบัน กับจุดเวลาที่อยู่ถัดจาก time_cursor ไป เป็นการตรวจสอบก่อนที่จะย้าย time_cursor ไปชี้ยังจุดเวลาถัดไป ซึ่งค่าที่อยู่ใน stack ขนาด 5 อัน ทำให้ต้องมีการเปรียบเทียบ 5 ครั้ง ซึ่งเวลานี้จะเกิดขึ้น d ครั้ง

เราสามารถสรุปเป็นสมการได้ดังนี้

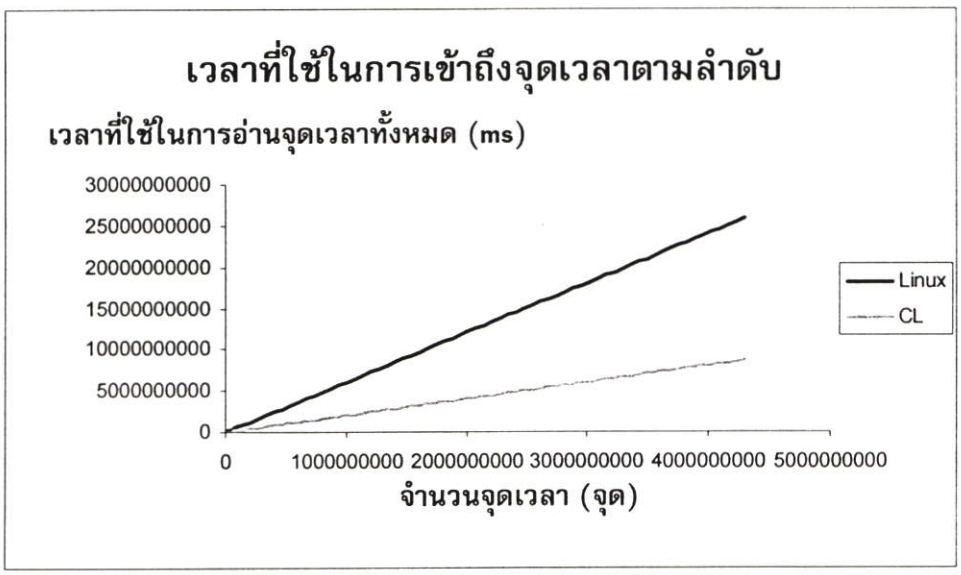
$$c_{seq\_access} = d \times (c_{pop} + c_{next}) + \frac{d}{2^8} \times (c_{pop} + c_{next}) +$$

$$\frac{d}{2^{14}} \times (c_{pop} + c_{next}) + \frac{d}{2^{20}} \times (c_{pop} + c_{next}) +$$

$$\frac{d}{2^{26}} \times (c_{pop} + c_{next}) + 5 \times d \times c_{cmp} \qquad [7.3]$$

เนื่องจากสมการที่ [7.3] ไม่มีส่วนที่เกี่ยวข้องกับจำนวนจุดเวลาที่เก็บในโครงสร้างข้อมูลเลย ดังนั้นเราอาจสรุปได้ว่า ฟังก์ชั่นนี้ใช้เวลาในระดับ *O(d)*

ถ้าสมมติให้ c_{inc}, c_{move}, c_{pop} ,c_{next}และ c_{cmp} มีค่าเท่ากับ 1 มิลลิวินาที แล้วทดลองนำสมการ [7.2] และ [7.3] มาเขียนเป็นกราฟ โดยให้แกน y แทนเวลาสำหรับเข้าถึงจุดเวลา และแกน x แทนจำนวนเวลาที่เก็บอยู่ในโครงสร้างข้อมูล จะได้กราฟดังนี้



รูปที่ 7.7 กราฟเปรียบเทียบเวลาในการเข้าถึงจุดเวลาตามลำดับ

กรณีที่ 2: จุดเวลารวมตัวกันทางด้านขอบเขตบน

สมมติว่ามีจุดเวลาถูกเก็บอยู่ใน tv_5 เท่านั้น ในโครงสร้างข้อมูลของลีนุกซ์จะใช้เวลาคงที่สำหรับ การ tick 2^{26} แรก หลังจากนั้น ก็จะมีการย้ายข้อมูลตามที่ได้กล่าวมาแล้วในกรณีที่ 1 เราสามารถคำนวณหาเวลารวมการเข้าถึงข้อมูลของกรณีนี้ได้โดย นำเวลารวมในกรณีที่ 1 (ใช้การ tick 2^{32} ครั้ง) แล้วลบออกด้วยเวลาที่ต้องใช้ย้ายข้อมูลในการ tick 2^{26} แรก คิดมาเป็นสมการได้เป็น

$$c_{seq\_access} = 2^{32} - 2^{26} \times (c_{inc} + n_{tv1} \times c_{move}) +$$
$$\frac{2^{32} - 2^{26}}{2^{8}} \times (c_{inc} + n_{tv2} \times c_{move}) +$$
$$\frac{2^{32} - 2^{26}}{2^{14}} \times (c_{inc} + n_{tv3} \times c_{move}) +$$
$$\frac{2^{32} - 2^{26}}{2^{20}} \times (c_{inc} + n_{tv4} \times c_{move}) +$$
$$\frac{2^{32} - 2^{26}}{2^{26}} \times (c_{inc} + n_{tv5} \times c_{move})$$

ส่วนในกรณีของโครงสร้างข้อมูล CL จะใช้เวลาตามสมการที่ 7.3

ในกรณีนี้เราสรุปได้ว่า ในกรณีที่ข้อมูลรวมตัวอยู่ที่ด้านขอบเขตบน ถ้ายิ่งรวมตัวหนาแน่นมากเท่าไหร่ ก็จะทำให้เวลารวมในโครงสร้างข้อมูลของลีนุกซ์ลดน้อยลง เพราะในแต่และ tick ที่ไม่มีข้อมูล โครงสร้างข้อมูลของลีนุกซ์ใช้เวลาในการคืนค่าผลลัพธ์ได้รวดเร็วกว่าในโครงสร้างข้อมูลของ CL

กรณีที่ 3: จุดเวลารวมตัวกันทางด้านขอบเขตล่าง

สมมติว่ามีจุดเวลาถูกเก็บอยู่ใน tv1 เท่านั้น เวลารวมในการเข้าถึงจุดเวลาทุกจุดใน tv1 คำนวณได้ด้วยสมการที่ [7.2] โดยกำหนดให้ มีการ tick 255 ครั้ง ซึ่งทำให้ค่า $\frac{d}{2^8}$, $\frac{d}{2^{14}}$, $\frac{d}{2^{20}}$ และ $\frac{d}{2^{26}}$ มีค่าเป็น 0 ทั้งหมด ทำให้เวลารวมสำหรับกรณีนี้คิดเป็น

$$c_{seq\_access} = 255 \times (c_{inc} + n_{tv1} \times c_{move})$$

เนื่องจาก n_{tv1} มีค่าเป็น 1 (จุดเวลาที่จัดเก็บในโครงสร้างข้อมูลไม่ซ้ำกัน) ทำให้เวลารวมสำหรับกรณีนี้ใช้เวลาน้อยมาก

ส่วนโครงสร้างข้อมูลของ CL ใช้เวลารวมตามสมการที่ [7.3] ซึ่งใช้เวลามากกว่า เพราะแต่ละครั้งที่มีอ่านข้อมูลออกไป ต้องมีการเปรียบเทียบค่าเวลาปัจจุบันกับ ค่าจุดเวลาที่ timecursor ชี้อยู่ทุกครั้ง

7.7.1.2 การค้นหาจุดเวลา

การค้นหาจุดเวลาไม่มีอิมพลีเมนต์ในโครงสร้างข้อมูลของลีนุกซ์ แต่เราจะขอเปรียบเทียบโดยอ้างอิงจากวิธีการค้นหาจุดเวลาที่เราได้นำเสนอในตอนต้น โดยเราพิจารณาใน 3 กรณี

1. กรณีจุดเวลาถูกเก็บเต็มโครงสร้างข้อมูล
2. กรณีจุดเวลาถูกเก็บอยู่ในช่วงขอบเขตบน
3. กรณีจุดเวลาถูกเก็บอยู่ในช่วงขอบเขตล่าง

กรณีที่ 1: จุดเวลาถูกเก็บเต็มโครงสร้างข้อมูล

ในโครงสร้างข้อมูลของลีนุกซ์ เราค้นหาจุดเวลาได้โดยคำนวณหาตำแหน่งของลิสต์ที่เก็บจุดเวลา แล้วสแกนหาจุดเวลาในลิสต์อีกทีหนึ่ง ดังนั้นในการค้นหาจุดเวลาจึงเป็นผลรวมของเวลาที่ใช้คำนวณหาตำแหน่งของลิสต์ และเวลาเฉลี่ยที่ใช้ค้นหาจุดเวลาในลิสต์ ขนาดความยาวของลิสต์ใน tv_1-tv_5 มีขนาดลดหลั่นกันไปตามลำดับ ถ้ากำหนดให้ c_{cmp}คือ เวลาในการเปรียบเทียบ และ c_{access_list} คือเวลาที่ใช้ในการอ่านข้อมูลส่วนหัวออกจากลิสต์ เราสามารถหาเวลาที่ใช้ค้นหาจุดเวลา (tp) ในช่วงต่างๆ ได้ตามสมการต่อไปนี้

$$c_{find} = \begin{cases} 5 \times c_{cmp} + \frac{n_{tv1}}{2} \times c_{access\_list} & ,0 < tp < 255 \\ 4 \times c_{cmp} + \frac{n_{tv2}}{2} \times c_{access\_list} & ,256 < tp < 2^{14} - 1 \\ 3 \times c_{cmp} + \frac{n_{tv3}}{2} \times c_{access\_list} & ,2^{14} < tp < 2^{20} - 1 \\ 2 \times c_{cmp} + \frac{n_{tv4}}{2} \times c_{access\_list} & ,2^{20} < tp < 2^{26} - 1 \\ 1 \times c_{cmp} + \frac{n_{tv5}}{2} \times c_{access\_list} & ,2^{26} < tp < 2^{32} - 1 \end{cases} \qquad [7.4]$$

ค่า c_{cmp} และ c_{access_list} เป็นค่าที่คงที่ ส่วน n_{tv1} , n_{tv2} , n_{tv3} , n_{tv4} , n_{tv5} เป็นค่าตัวแปร ดังนั้นเราอาจกล่าวได้ว่า การค้นหาจุดเวลาจะใช้เวลา $O(\frac{n_{tv1}}{2})$, $O(\frac{n_{tv2}}{2}), O(\frac{n_{tv3}}{2})$, $O(\frac{n_{tv4}}{2})$ หรือ $O(\frac{n_{tv5}}{2})$ ขึ้นอยู่กับจุดเวลาที่ค้นหาว่าอยู่ตำแหน่งใด

ในโครงสร้างข้อมูลของ CL ค่าจุดเวลาสามารถนำทางการค้นหาจากโหนด root ไปยังโหนด leaf ได้ โดยมีจำนวนครั้งการ visit เท่ากับความสูงของ Tree (k) การ visit แต่ละครั้งจะใช้การค้นหาแบบ binary search กับโหนดลูกที่เรียงลำดับ ทำให้ใช้การเปรียบเทียบ $\log_2$ (b1), log2 (b2), log2 (b3), log2 (b4), log2 (b5) ครั้ง สำหรับการค้นหาโหนดลูกในระดับที่ 1 (มีจำนวนโหนดลูก b1 โหนด) ระดับที่ 2 (มีจำนวนโหนดลูก b2 โหนด) ระดับที่ 3 (มีจำนวนโหนดลูก b3 โหนด) ระดับที่ 4 (มีจำนวนโหนดลูก b4 โหนด) และระดับที่ 5 (มีจำนวนโหนดลูก b5 โหนด) ตามลำดับ ถ้าให้การเปรียบเทียบแต่ละครั้งใช้เวลา c_{cmp} เราจะได้ว่าเวลาในการค้นหาเป็น

$$c_{find} = (\log_2(b1) + \log_2(b2) + \log_2(b3) + \log_2(b4) + \log_2(b5)) \times c_{cmp} \quad [7.5]$$

ค่า c_{cmp} เป็นค่าคงที่ ทำให้ฟังก์ชั่นนี้ใช้เวลา

$$\begin{aligned} O(c_{find}) &= O((\log_2(b1) + \log_2(b2) + \log_2(b3) + \log_2(b4) + \log_2(b5)) \times c_{cmp}) \\ &= O(\log_2(b1) + \log_2(b2) + \log_2(b3) + \log_2(b4) + \log_2(b5)) \\ &= O(\log_2(b1 \times b2 \times b3 \times b4 \times b5)) \end{aligned}$$

เนื่องจาก b1*b2*b3*b4*b4 เท่ากับจำนวนจุดเวลาเก็บในลิสต์ (n) ดังนั้น

$$O(c_{find}) = O(\log_2(n))$$

เพื่อให้เห็นประสิทธิภาพของการค้นจุดเวลาทั้ง 2 แบบ จะขอพิจารณาเปรียบเทียบในกรณีต่างๆ โดยกำหนดให้ c_{access_list}, c_{cmp} มีค่าเป็น 1 มิลลิวินาที

ตัวอย่างที่ 1

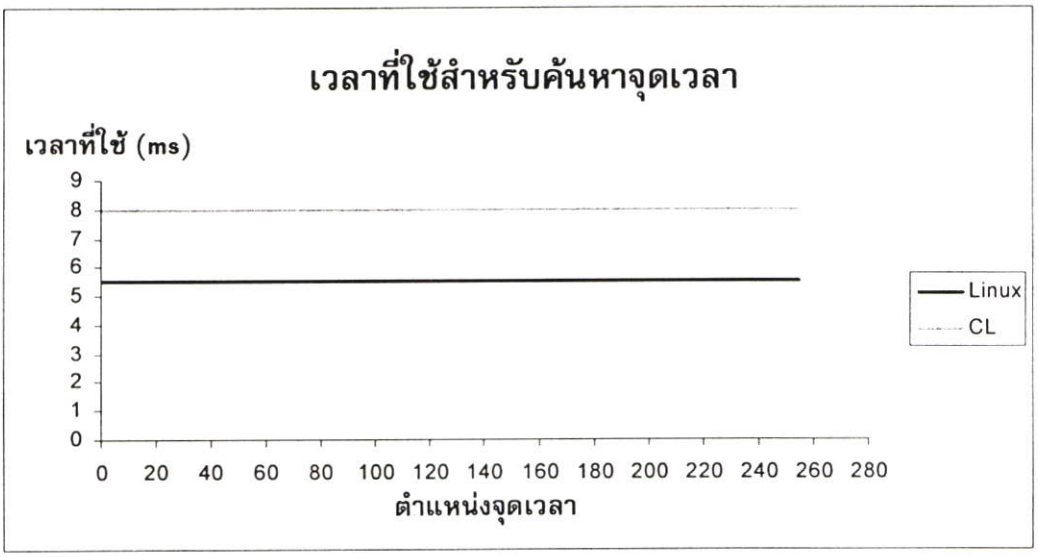
ถ้าเราสมมติว่าโครงสร้างข้อมูลเก็บจุดเวลาทุกจุด ในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,0,0,255) อยู่ก่อนแล้ว และไม่มีจุดเวลาใดเก็บซ้ำกันเลย ค่า c_{cmp} และ c_{access_list} มีค่าเป็น 1 มิลลิวินาที ในกรณีนี้โครงสร้างข้อมูลของลีนุกซ์จะเก็บจุดเวลาใน tv_1.vec ทั้งหมด การค้นหาจุดเวลาใดในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,0,0,255) จะใช้เวลา

$$\begin{aligned} c_{find} &= 5 \times c_{cmp} + \frac{n_{tv}}{2} \times c_{access\_list} \\ &= 5 + \frac{1}{2} \times 1 = 5.5 \text{ มิลลิวินาที} \end{aligned}$$

สำหรับโครงสร้างข้อมูลของ CL ลักษณะของ Tree ในกรณีนี้จะเป็น Tree ที่มีกิ่งเดียว และแตกออกที่ระดับสุดท้ายเท่านั้น ดังนั้นเราจึงคิดเวลาการค้นหาจุดเวลาได้จาก

$$\begin{aligned} c_{find} &= (\log_2(b1) + \log_2(b2) + \log_2(b3) + \log_2(b4) + \log_2(b5)) \times c_{cmp} \\ &= (\log_2(1) + \log_2(1) + \log_2(1) + \log_2(1) + \log_2(256)) \times 1 \\ &= 8 \text{ มิลลิวินาที} \end{aligned}$$

ผลการเปรียบเทียบได้กราฟดังรูปด้านล่าง



รูปที่ 7.8 กราฟเวลาการค้นหาจุดเวลาในช่วงเวลา (0,0,0,0,0) ถึง (0,0,0,0,255)

ตัวอย่างที่ 2

เราเพิ่มจุดเวลาเข้าไปอีก เป็นจุดเวลาในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,0,63,255) และไม่มีจุดเวลาใดเก็บซ้ำกันเลย ในกรณีนี้โครงสร้างข้อมูลของลีนุกซ์จะเก็บจุดเวลา 255 จุดแรกไว้ใน tv_1.vec และจุดเวลาหลังจาก 255 ไว้ใน tv_2.vec การค้นหาจุดเวลา 255 จุดแรกจะใช้เวลาเท่ากับกรณีที่แล้ว แต่เมื่อค้นหาเกินจุดเวลา (0,0,0,0,255) ไปแล้ว การค้นหาจะต้องค้นหาไปบนลิสต์ที่เก็บอยู่ใน tv_2.vec ซึ่งลิสต์ดังกล่าวจะมีขนาด 256 จุดเวลา เราสามารถหาเวลาสำหรับการค้นหาจุดเวลาในลำดับที่เกินจุดเวลา (0,0,0,0,255) ได้จาก

$$c_{find} = 4 \times c_{cmp} + \frac{n_{tv2}}{2} \times c_{access\_list}$$

$$= 4 + \frac{256}{2} \times 1 = 132 \text{ มิลลิวินาที}$$

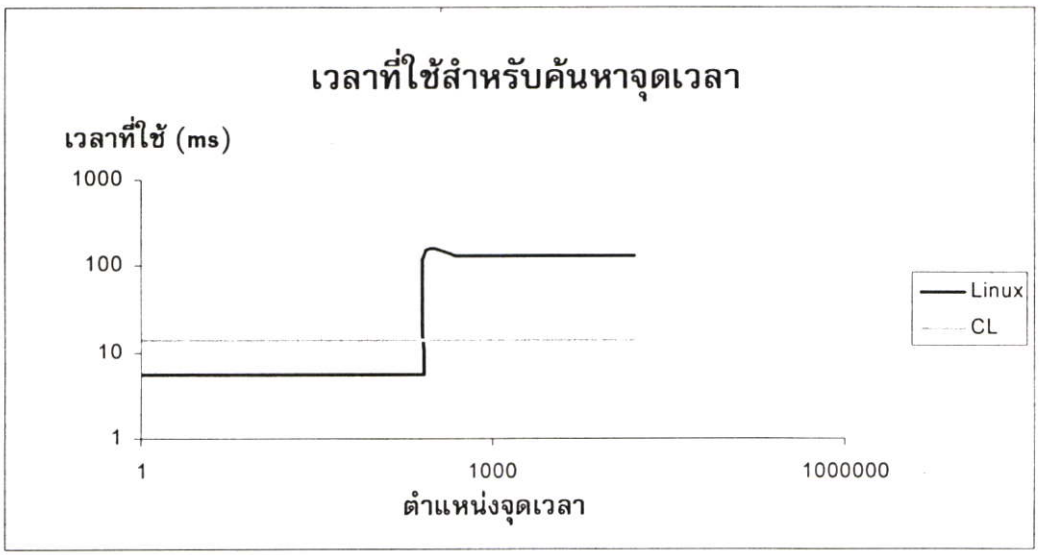
ส่วนโครงสร้างข้อมูลของ CL เมื่อเก็บจุดเวลาในกรณีนี้ จะได้เป็น Tree ที่มี b1, b2, b3 เท่ากับ 1 และมี b4, b5 เท่ากับ 64 และ 256 ตามลำดับ เราสามารถคำนวณหาเวลาค้นหาในทุกจุดเวลาได้ดังนี้

$$c_{find} = (\log_2(b1) + \log_2(b2) + \log_2(b3) + \log_2(b4) + \log_2(b5)) \times c_{cmp}$$

$$= (\log_2(1) + \log_2(1) + \log_2(1) + \log_2(64) + \log_2(256)) \times 1$$

$$= 14 \text{ มิลลิวินาที}$$

ผลการเปรียบเทียบได้กราฟดังรูปด้านล่าง



รูปที่ 7.9 กราฟเวลาการค้นหาจุดเวลา ในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,0,63,255)

ตัวอย่างที่ 3

ถ้าเราสมมติว่าโครงสร้างข้อมูลเก็บจุดเวลาทุกจุด ในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,63,63,255) อยู่ก่อนแล้ว และไม่มีจุดเวลาใดเก็บซ้ำกันเลย

เราสามารถหาเวลาสำหรับการค้นหาจุดเวลาในลำดับที่เกินจุดเวลา (0, 0, 0, 63, 255) ในโครงสร้างข้อมูลของลีนุกซ์ได้จาก

$$c_{find} = 3 \times c_{cmp} + \frac{n_{tv3}}{2} \times c_{access\_list}$$

$$= 3 + \frac{2^{14}}{2} \times 1 = 8195 \text{ มิลลิวินาที}$$

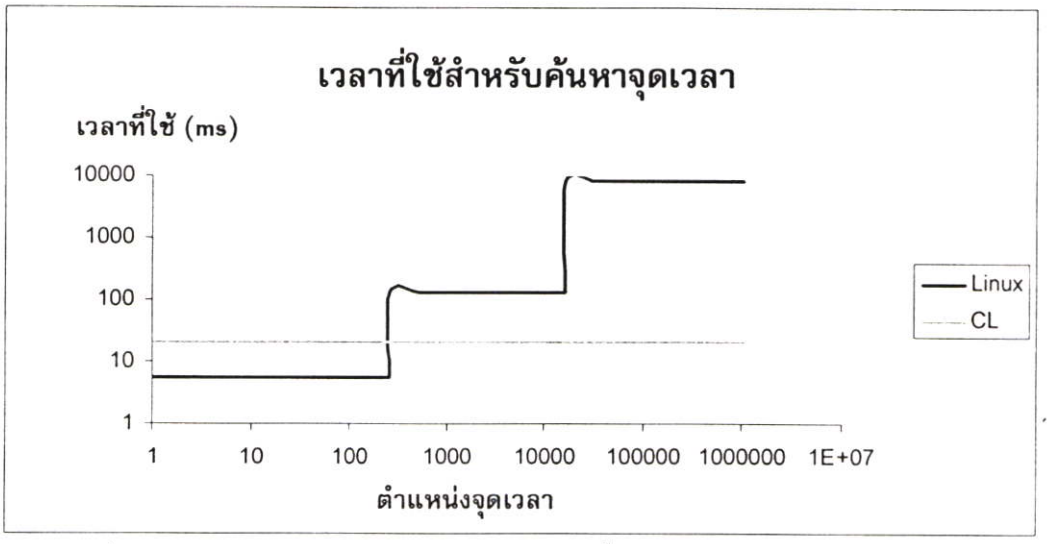
สำหรับโครงสร้างข้อมูลของ CL เราจะใช้เวลาค้นหา

$$c_{find} = (\log_2(b1) + \log_2(b2) + \log_2(b3) + \log_2(b4) + \log_2(b5)) \times c_{cmp}$$

$$= (\log_2(1) + \log_2(1) + \log_2(64) + \log_2(64) + \log_2(256)) \times 1$$

$$= 20 \text{ มิลลิวินาที}$$

ผลการเปรียบเทียบได้กราฟดังรูปด้านล่าง



รูปที่ 7.10 กราฟเวลาการค้นหาจุดเวลาในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,63,63,255)

จากทั้ง 3 ตัวอย่าง แสดงให้เห็นว่า การค้นหาจุดเวลาในโครงสร้างข้อมูลของลีนุกซ์ เมื่อจุดที่ค้นหาอยู่ในตำแหน่งที่ไกลจากปัจจุบัน จะทำให้ใช้เวลาในการค้นหานานขึ้น ในขณะที่การค้นหาจุดเวลาในโครงสร้างของ CL เวลาในการค้นจุดเวลาแต่ละจุดจะเท่าๆ กัน ทุกจุด แต่เวลาจะเพิ่มขึ้นตามจำนวนจุดเวลาที่เก็บในโครงสร้างข้อมูล

กรณีที่ 2: จุดเวลารวมตัวกันทางด้านขอบเขตบน

ในกรณีนี้ การค้นหาจุดเวลาในโครงสร้างข้อมูลของลีนุกซ์จะใช้เวลามาก เพราะว่าจุดเวลาถูกเก็บอยู่ในลักษณะของลิสต์ทางด้านท้ายๆ ของโครงสร้างข้อมูล

ส่วนการค้นหาจุดเวลาที่เก็บอยู่ในโครงสร้างข้อมูลของ CL จะใช้เวลาที่คำนวณตามสมการที่ [7.5] ซึ่งเวลาที่ใช้ขึ้นอยู่กับจำนวนจุดเวลาที่เก็บอยู่ในโครงสร้างข้อมูลเท่านั้น ถ้าเปรียบเทียบด้วยจำนวนจุดเวลาที่เท่ากัน จะพบว่าโครงสร้างข้อมูลของ CL ทำงานได้รวดเร็วกว่าในกรณีนี้

กรณีที่ 3: จุดเวลารวมตัวกันทางด้านขอบเขตล่าง

ในกรณีนี้ โครงสร้างข้อมูลของลีนุกซ์สามารถถูกค้นหาได้รวดเร็ว เพราะสามารถใช้ คีย์จะจุดเวลาเป็นอินเด็กซ์ในค้นหาข้อมูลได้ทันที

ส่วนในโครงสร้างข้อมูลของ CL จะมีการทำงานที่ช้ากว่า เนื่องจากต้องมีการค้นหาโหนดไล่ไปตามแต่ละลำดับชั้นของ Tree

7.7.1.3 การเพิ่มจุดเวลา

การเพิ่มจุดเวลาในโครงสร้างข้อมูลของลีนุกซ์ กระทำโดย คำนวณหาตำแหน่งของลิสต์ที่ควรจะเก็บจุดเวลา จากนั้นจึงนำจุดเวลาไปต่อท้ายลิสต์

เราสามารถวิเคราะห์วิธีการดังกล่าวได้ว่า เวลาที่ใช้ประกอบด้วย เวลาสำหรับคำนวนหาตำแหน่งของลิสต์ และเวลาที่ใช้สำหรับนำจุดเวลาไปต่อท้ายลิสต์ (c_{append_list}) เวลาที่ใช้ในการเพิ่มจุดเวลา (c_{insert}) คำนวณได้จาก

$$c_{insert} = \begin{cases} 5 \times c_{cmp} + c_{append\_list} & ,0 < tp < 255 \\ 4 \times c_{cmp} + c_{append\_list} & ,256 < tp < 2^{14} - 1 \\ 3 \times c_{cmp} + c_{append\_list} & ,2^{14} < tp < 2^{20} - 1 \\ 2 \times c_{cmp} + c_{append\_list} & ,2^{20} < tp < 2^{26} - 1 \\ 1 \times c_{cmp} + c_{append\_list} & ,2^{26} < tp < 2^{32} - 1 \end{cases} \qquad [7.6]$$

ทั้ง c_{cmp}และ c_{append_list} เป็นค่าคงที่ ดังนั้นเราจึงอาจกล่าวได้ว่า การเพิ่มจุดเวลาใช้เวลา *O(1)*

การเพิ่มจุดเวลาในโครงสร้างข้อมูลของ CL กระทำโดยการ visit ไปตามโหนด จากโหนด root ไปยังโหนด leaf เป็นจำนวน k ครั้ง ตามความสูงของ Tree ถ้ายังไม่มีโหนดต้องการ visit ก็สร้างโหนดนั้นขึ้นมา จากกระบวนการอันนี้ ทำให้เราวิเคราะห์ได้ว่าเวลาที่ฟังก์ชั่นนี้ใช้ประกอบด้วยเวลาสำหรับการค้นหาโหนด (ใช้สมการเดียวกับการค้นหาจุดเวลา) และเวลาสำหรับการสร้างโหนดใหม่ ($k \times c_{new\_node}$) โดยเวลาที่ใช้ในการเพิ่มจุดเวลาคำนวณได้จาก

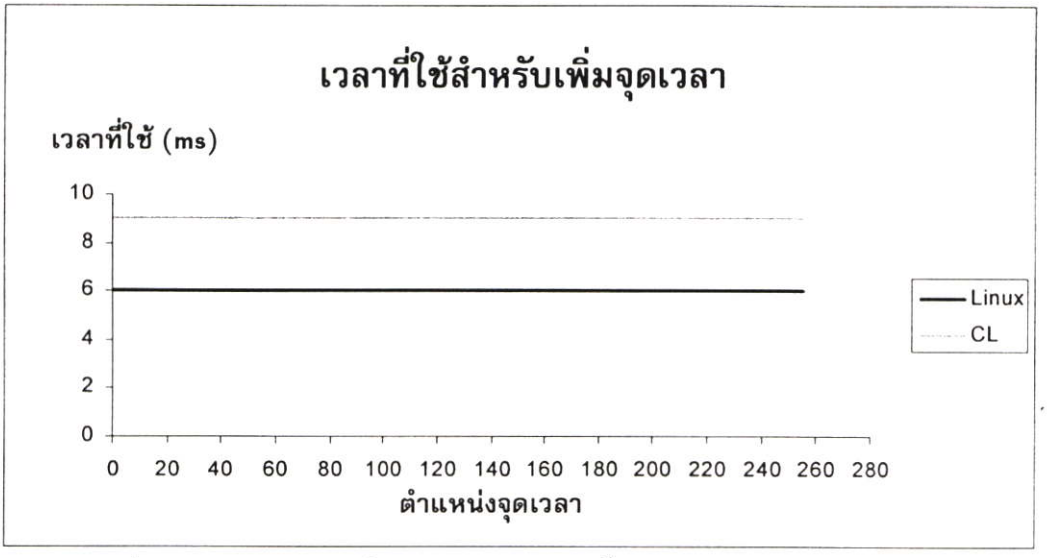
$$c_{insert} = (\log_2 b1 + \log_2 b2 + \log_2 b3 + \log_2 b4 + \log_2 b5) \times c_{cmp} + k \times c_{new\_node} \qquad [7.7]$$

เวลาที่คำนวณได้จากสูตรนี้เป็นเวลาประมาณเท่านั้น เนื่องจาก ค่า c_{new_node} จะบวกเพิ่มเข้าไปในตอนที่โหนดที่ต้องการ visit ยังไม่ถูกสร้างเท่านั้น

เพื่อให้เข้าใจการใช้เวลาในฟังก์ชั่นนี้ จะขอยกตัวอย่างต่างๆ เพื่อทดลองคำนวณค่าเวลาตามสมการ [7.6] และ [7.7] โดยให้ ค่าคงที่ต่างๆ มีค่าเท่ากับ 1 มิลลิวินาที

ตัวอย่างที่ 1

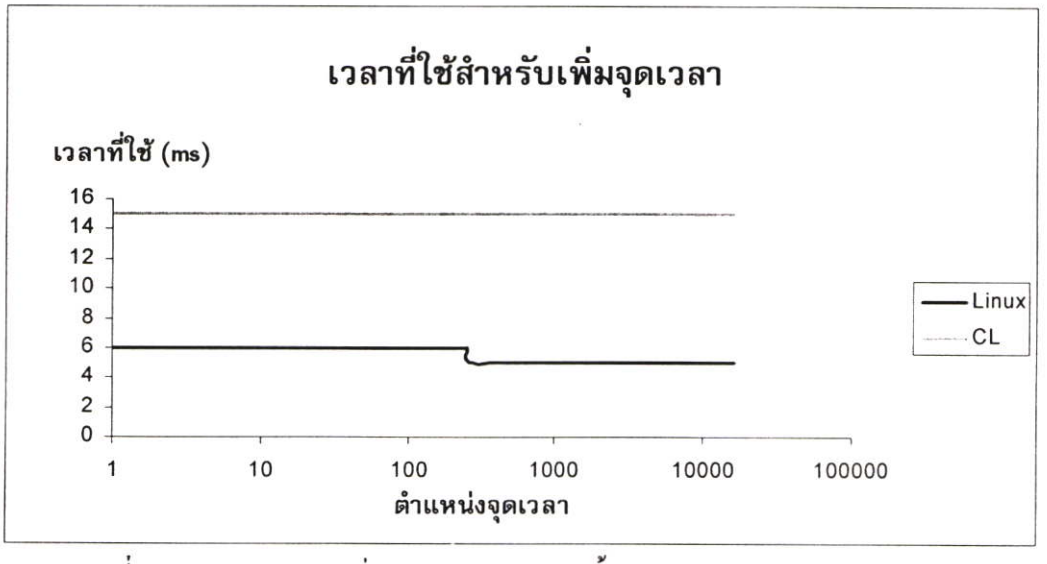
เราสมมติว่าเพิ่มจุดเวลาในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,0,0,255) ในโครงสร้างข้อมูล



รูปที่ 7.11 กราฟเวลาการเพิ่มจุดเวลาในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,0,0,255)

ตัวอย่างที่ 2

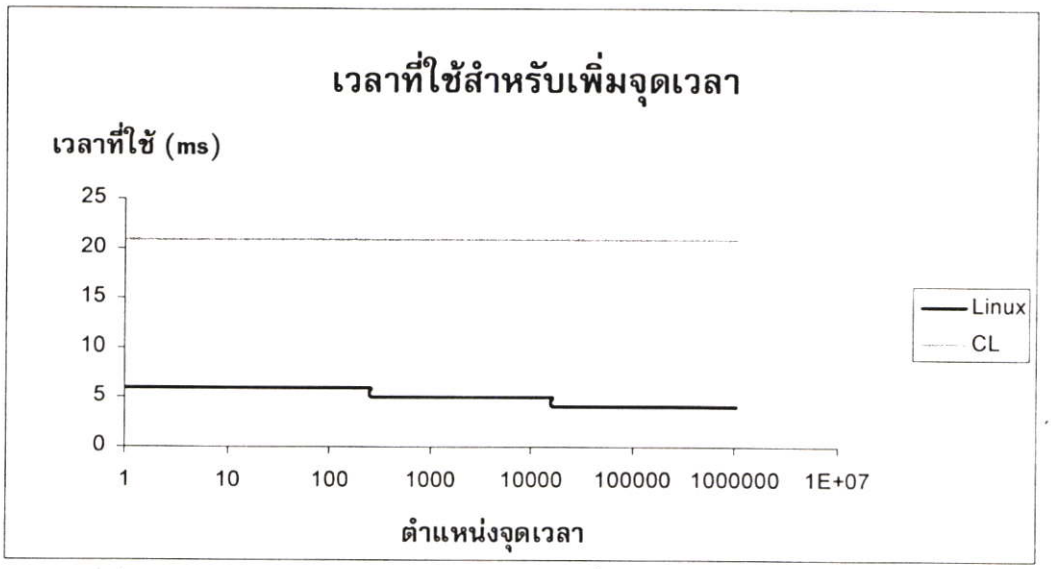
เราสมมติว่าเพิ่มจุดเวลาในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,0,63,255) ในโครงสร้างข้อมูล



รูปที่ 7.12 กราฟเวลาการเพิ่มจุดเวลาในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,0,63,255)

ตัวอย่างที่ 3

เราสมมติว่าเพิ่มจุดเวลาในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,63,63,255) ในโครงสร้างข้อมูล



รูปที่ 7.13 กราฟเวลาการเพิ่มจุดเวลาในช่วงเวลาตั้งแต่ (0,0,0,0,0) ถึง (0,0,63,63,255)

จากตัวอย่างที่ผ่านมา ทำให้สรุปได้ว่าการเพิ่มจุดเวลาในโครงสร้างข้อมูลของลีนุกซ์ใช้เวลาค่อนข้างคงที่ ส่วนโครงสร้างข้อมูลของ CL ใช้เวลาเพิ่มขึ้นในระดับ Logarithm

7.7.1.4 การลบจุดเวลา

ในโครงสร้างข้อมูลของลีนุกซ์ การลบจุดเวลาจะมีวิธีการคล้ายกับการค้นหาจุดเวลา เพียงแต่เพิ่มขั้นตอนการลบจุดเวลาออกจากโครงสร้างข้อมูลหลังจากพบจุดเวลาแล้ว

การวิเคราะห์เวลาที่ใช้ในฟังก์ชั่นนี้ จึงสามารถวิเคราะห์ได้เช่นเดียวกับการค้นหาจุดเวลา โดยเพิ่มเวลาสำหรับการลบจุดเวลาออกจากลิสต์ (c_{delete_list}) เข้าไปในสมการที่ [7.4] ทำให้เราได้ว่า เวลาที่ใช้ลบจุดเวลา (c_{delete}) เป็น

$$c_{delete} = \begin{cases} 5 \times c_{cmp} + \frac{n_{tv1}}{2} \times c_{access\_list} + c_{delete\_list} & ,0 < tp < 255 \\ 4 \times c_{cmp} + \frac{n_{tv2}}{2} \times c_{access\_list} + c_{delete\_list} & ,256 < tp < 2^{14} - 1 \\ 3 \times c_{cmp} + \frac{n_{tv3}}{2} \times c_{access\_list} + c_{delete\_list} & ,2^{14} < tp < 2^{20} - 1 \\ 2 \times c_{cmp} + \frac{n_{tv4}}{2} \times c_{access\_list} + c_{delete\_list} & ,2^{20} < tp < 2^{26} - 1 \\ 1 \times c_{cmp} + \frac{n_{tv5}}{2} \times c_{access\_list} + c_{delete\_list} & ,2^{26} < tp < 2^{32} - 1 \end{cases} \quad [7.8]$$

การลบจุดเวลาที่ค้นหาตำแหน่งพบแล้ว เป็นการลบจุดเวลาออกจาก Doubly Link List ทำให้ค่า c_{delete_list} เป็นค่าคงที่ ดังนั้นเวลาทั้งหมดในการลบจุดเวลาจึงเป็น $O(\frac{n_{lv1}}{2})$, $O(\frac{n_{lv2}}{2})$, $O(\frac{n_{lv3}}{2})$, $O(\frac{n_{lv4}}{2})$ หรือ $O(\frac{n_{lv5}}{2})$ เช่นเดียวกับการค้นหาจุดเวลา

ส่วนในโครงสร้างข้อมูลของ CL การลบจุดเวลาจะใช้วิธีการ visit ไปตามค่าจุดเวลา จากโหนด root จนกระทั่งถึงโหนด leaf ซึ่งมีความสูงของ Tree k node เมื่อถึงโหนด leaf แล้ว ก็ลบจุดเวลาออก จากนั้นระหว่างการ backtrack ก็จะมีการตรวจสอบจากโหนด leaf กลับไปหาโหนด root ถ้าพบว่าโหนด leaf ใดไม่มีจุดเวลาบันทึกอยู่ก็จะลบโหนด leaf นั้นทิ้งเสีย จากขั้นตอนการทำงานนี้ เราสามารถแบ่งประเภทของเวลาที่ใช้ในการลบจุดเวลาได้เป็น เวลาที่ใช้ค้นหาไปยังโหนด leaf (ใช้สูตรเดียวกับการค้นหาจุดเวลา) และ เวลาที่ใช้ตรวจสอบโหนดว่าเป็น leaf โหนด และไม่มีจุดเวลาบันทึกอยู่หรือไม่ (c_{check}) และเวลาที่ใช้ลบโหนด (c_{remove_node}) เวลาที่ใช้ลบจุดเวลาคำนวณได้จาก

$$\begin{aligned} c_{delete} &= c_{find} + k \times (c_{check} + c_{remove\_node}) \\ &= (\log_2 b1 + \log_2 b2 + \log_2 b3 + \log_2 b4 + \log_2 b5) \times c_{cmp} + \\ &\quad k \times (c_{check} + c_{remove\_node}) \end{aligned} \qquad [7.9]$$

จะเห็นว่า ค่า k, c_{check} และ c_{remove_node} เป็นค่าเวลาคงที่ ทำให้ความซับซ้อนของอัลกอลิธึมนี้จึงอยู่ในชั้นของ O(log(n)) การวิเคราะห์ประสิทธิภาพการทำงานของการลบจุดเวลาในกรณีต่างๆ มีลักษณะคล้ายกับการค้นหาจุดเวลา ดังนั้นจึงไม่ขออธิบายซ้ำอีกครั้ง

จากการวิเคราะห์ความซับซ้อนที่ผ่านมาทั้งหมด เราสรุปลงมาไว้ในตารางได้ดังนี้

ตารางที่ 7.1 สรุปการเปรียบเทียบความซับซ้อนของอัลกอลิธึมบนโครงสร้างข้อมูลแต่ละแบบ

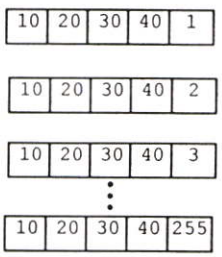
อัลกอลิธึม	โครงสร้างข้อมูลลีนุกซ์	โครงสร้างข้อมูลของ CL	ข้อดี/ข้อเสีย
การเข้าถึงจุดเวลา	$O(d \times n_{tv1} + \frac{d}{2^8} \times n_{tv2} + \frac{d}{2^{14}} \times n_{tv3} + \frac{d}{2^{20}} \times n_{tv4} + \frac{d}{2^{26}} \times n_{tv5})$	$O(d)$	โครงสร้างข้อมูลของ CL เข้าถึงจุดเวลาได้เร็วกว่าในกรณีที่จุดเวลาถูกเก็บอยู่เต็มโครงสร้างข้อมูล แต่จะเข้าถึงข้อมูลช้ากว่าในกรณีที่จุดเวลามีการกระจายตัวอย่างไม่สม่ำเสมอ
การค้นหาจุดเวลา	$O(\frac{n_{tv1}}{2})$, $O(\frac{n_{tv2}}{2}), O(\frac{n_{tv3}}{2})$, $O(\frac{n_{tv4}}{2})$ หรือ $O(\frac{n_{tv5}}{2})$ ขึ้นอยู่กับจุดเวลาที่ต้องการค้นหา	$O(\log_2 n)$	ในกรณีทั่วไปโครงสร้างข้อมูลของ CL ค้นหาจุดเวลาได้รวดเร็วกว่าโครงสร้างข้อมูลของลีนุกซ์ ยกเว้น กรณีเดียวคือ ในกรณีที่จุดเวลาถูกเก็บอยู่ใน tv1 โครงสร้างข้อมูลของลีนุกซ์จะค้นหาข้อมูลได้เร็วกว่า
การเพิ่มจุดเวลา	$O(1)$	$O(\log_2 n)$	โครงสร้างข้อมูลของลีนุกซ์จัดเก็บข้อมูลได้รวดเร็วกว่าโครงสร้างข้อมูลของ CL
การลบจุดเวลา	$O(\frac{n_{tv1}}{2})$, $O(\frac{n_{tv2}}{2}), O(\frac{n_{tv3}}{2})$, $O(\frac{n_{tv4}}{2})$ หรือ $O(\frac{n_{tv5}}{2})$ ขึ้นอยู่กับจุดเวลาที่ต้องการลบ	$O(\log_2 n)$	มีข้อดี/ข้อเสียเช่นเดียวกับการค้นหาจุดเวลา

โครงสร้างข้อมูลของลีนุกซ์ถูกออกแบบมาเพื่อใช้ตั้งเวลาที่เกี่ยวกับ System Programming ซึ่งเป็นการตั้งเวลาสั้นๆ ไม่เกิน 2 ชั่วโมง ทำให้ต้องการความรวดเร็วในการจัดเก็บจุดเวลา ประกอบกับจุดเวลาที่จัดเก็บในโครงสร้างข้อมูลมีความไม่สม่ำเสมอ ดังนั้นการใช้งานโครงสร้างข้อมูลของลีนุกซ์ ในระบบปฏิบัติการลีนุกซ์ จึงมีความเหมาะสมแล้ว

แต่การใช้งานโครงสร้างข้อมูลของ CL ต่างไปจากลีนุกซ์ เนื่องจากจุดเวลาที่จัดเก็บในโครงสร้างข้อมูลเป็นจุดเวลาสำหรับการทำงานในชีวิตประจำวัน เช่น จุดเวลาการเริ่มลงทะเบียนเรียน หรือจุดเวลาในการจ่ายค่าไฟฟ้า ซึ่งมีจุดเวลามีจำนวนมากอย่างไม่จำกัด และกระจายอยู่ทั่วไป ทำให้โครงสร้างข้อมูลของลีนุกซ์ ไม่เหมาะสม ควรใช้โครงสร้างข้อมูลที่ออกแบบใหม่มากกว่า

7.7.2 การเปรียบเทียบหน่วยความจำสำหรับโครงสร้างข้อมูล

ในโครงสร้างข้อมูลของลีนุกซ์ มีการจัดเก็บจุดเวลาที่ซ้ำซ้อนกันเป็นอย่างมาก ตัวอย่างเช่น จุดเวลาในลีนุกซ์ (10,20,30,40,1), (10,20,30,40,2), (10,20,30,40,3) ... (10,20,30,40,255) เมื่อนำมาจัดเก็บเป็นลิสต์ในโครงสร้างข้อมูล จะทำให้ตัวเลข 10, 20, 30, 40 ซ้ำกันหลายครั้ง ดังรูป



รูปที่ 7.14 จำนวนตัวเลขที่เก็บในโครงสร้างข้อมูลของลีนุกซ์

ถ้ามีจุดเวลาทั้งหมด 2^{32} จุด คิดเป็นจำนวนตัวเลขที่ต้องเก็บ

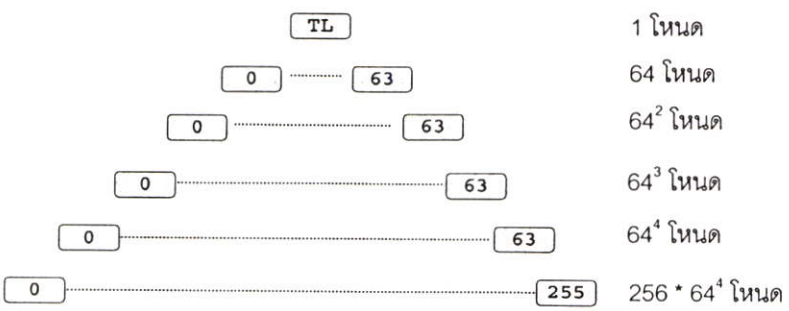
$$\text{จำนวนตัวเลข} = 2^{32} * 5$$

$$= \underline{\mathbf{21,474,836,480}} \text{ ตัว}$$

ในกรณีโครงสร้างข้อมูลของ CL โหนดใน Tree จะลดความซ้ำซ้อนของตัวเลข เช่น ในกรณีมีจุดเวลา 2^{32} เราจะใช้โหนดเพียง (ดูรูปด้านล่างประกอบ)

$$\text{จำนวนโหนด} = 1 + 64 + 64^2 + 64^3 + 64^4 + 256 * 64^4$$

$$= \underline{\mathbf{4,312,010,817}} \text{ โหนด}$$

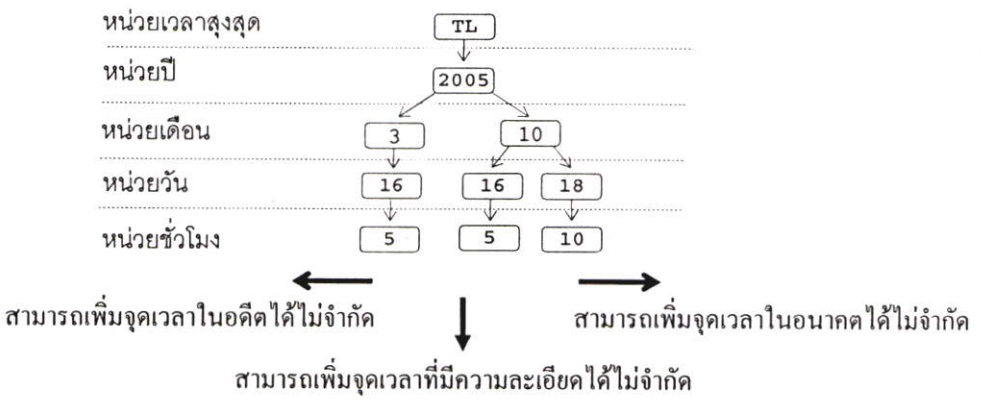


รูปที่ 7.15 จำนวนโหนดในโครงสร้างข้อมูลของ CL

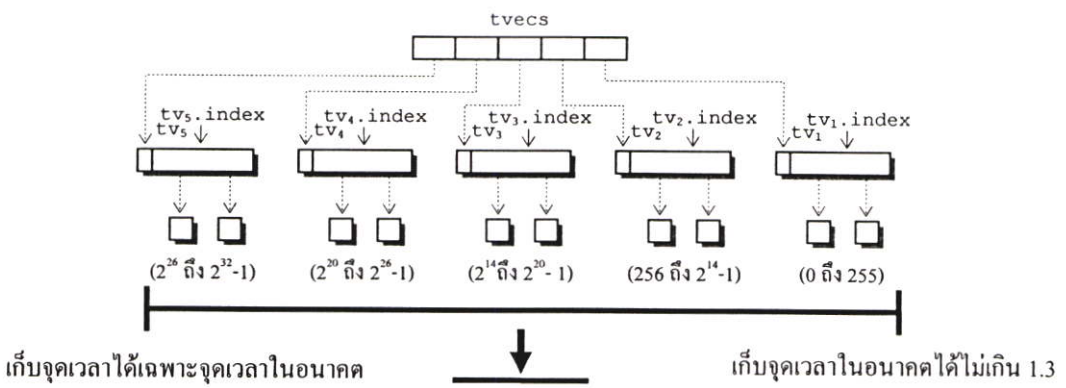
เมื่อนำตัวเลขทั้งสองมาเปรียบเทียบกัน จะพบว่าการจัดเก็บจุดเวลาในโครงสร้างข้อมูลของลีนุกซ์ จะใช้พื้นที่เป็น 4.98 เท่าของพื้นที่จัดเก็บข้อมูลของ CL

7.7.3 การเปรียบเทียบความสามารถในการจัดเก็บเวลา

1. โครงสร้างข้อมูลของ CL สามารถเก็บจุดเวลาที่มีความละเอียดได้ไม่จำกัด ส่วนโครงสร้างข้อมูลของลีนุกซ์ เก็บได้ที่ความละเอียดของเวลาไม่เกิน 10 มิลลิวินาทีเท่านั้น
2. โครงสร้างข้อมูลที่นำเสนอเก็บจุดเวลาได้ทั้งอดีตและอนาคต ที่ไม่ที่สิ้นสุด ส่วนโครงสร้างข้อมูลของลีนุกซ์ ไม่เก็บจุดเวลาในอดีต และเก็บจุดเวลาที่ห่างไกลจากปัจจุบัน ได้ไม่เกิน $(2^{32} - 1) * 10 * 10^{-3}$ วินาที หรือประมาณ 42949672 วินาที หรือ 1.3 ปีเท่านั้น



รูปที่ 7.16 ความสามารถในการจัดเก็บจุดเวลาของโครงสร้างข้อมูล Tree



รูปที่ 7.17 ข้อจำกัดในการจัดเก็บจุดเวลาในโครงสร้างข้อมูลของลีนุกซ์

3. ในโครงสร้างข้อมูลของลีนุกซ์เก็บจุดเวลา แต่ไม่มีการบันทึกความแตกต่างระหว่างจุดเวลาเริ่มต้นและจุดเวลาสิ้นสุด ทำให้ไม่สามารถคงความหมายของช่วงเวลาไว้ได้
4. โครงสร้างข้อมูลที่นำเสนอสามารถถูกโพลลิ่ง ด้วยความถี่ที่ปรับเปลี่ยนได้ โดยที่การทำงานยังคงถูกต้องอยู่ เช่น สมมติว่าจุดเวลาถูกอ้างด้วยหน่วยเวลาความละเอียดระดับชั่วโมง การโพลลิ่งสามารถปรับเปลี่ยนได้ เป็นทุกๆ 30 นาที หรือ ทุกๆ 45 นาที ได้โดยไม่

ทำให้การทำงานของ CL เปลี่ยนไป ต่างจากโครงสร้างข้อมูลของลีนุกซ์ที่ต้องถูกโพลลิ่งทุกๆ 10 มิลลิวินาทีอยู่เสมอ

5. โครงสร้างข้อมูลที่นำเสนอเก็บจุดเวลาแบบสมบูรณ์ (absolute) ทำให้ไม่ว่าข้อมูลนี้จะถูกส่งไปที่ใด คอมพิวเตอร์ดับไป หรือ ระยะเวลาของ interrupt timer เลื่อน เมื่ออินเตอร์พรีตเตอร์กลับมาทำงานอีกครั้งก็ยังคงทำงานได้ถูกต้อง เพราะอินเตอร์พรีตเตอร์ CL อ่านค่าเวลาจริงจากนาฬิกามาเปรียบเทียบทุกครั้ง ต่างจากโครงสร้างข้อมูลของลีนุกซ์ที่อ้างอิงเวลาแบบสัมพัทธ์ (relative) กับเวลาปัจจุบัน ทำให้ในกรณีที่ โครงสร้างข้อมูลนี้ถูกส่งให้ไปทำงานบนเครื่องอื่น หรือระยะเวลาของ interrupt timer เลื่อน จะทำให้จุดเวลาที่เก็บไว้ในโครงสร้างข้อมูลมีความหมายผิดไป
6. โครงสร้างข้อมูลที่นำเสนอสามารถเก็บตัวแปรไว้ใน Tree ได้ รวมทั้งมีการแบ่งแยก scope ซึ่งช่วยให้การตั้งเวลาในรูปแบบที่ซับซ้อนทำได้สะดวกขึ้น

7.7.4 การเปรียบเทียบด้านการจัดการ Process

1. Process ที่ทำงานในลีนุกซ์ถูกสร้างให้ทำงาน โดยปราศจากการตรวจสอบผลการทำงาน ณ เวลาสิ้นสุด และไม่มีการฆ่า Process ที่หมดเวลาประมวลผลแล้ว ทำให้ผู้เขียนโปรแกรมต้องคอยตรวจผลการทำงานของโปรแกรมเอง ซึ่งเกิดความยุ่งยาก ตัวอย่างเช่น การโปรแกรมให้คอมพิวเตอร์ เปิดวาล์วรดน้ำต้นไม้ ตั้งแต่เวลา 18 นาฬิกา จนถึงเวลา 19 นาฬิกา อินเตอร์พรีตเตอร์คอยควบคุมการเริ่มเปิดวาวล์น้ำ และฆ่า Process ของการเปิดวาวล์ ซึ่งช่วยให้ผู้เขียนโปรแกรมให้ความสนใจเฉพาะวิธีการควบคุม IO เท่านั้น ทำให้การเขียนโปรแกรมทำได้ง่ายขึ้น

7.7.5 การเปรียบเทียบด้านรูปแบบภาษา

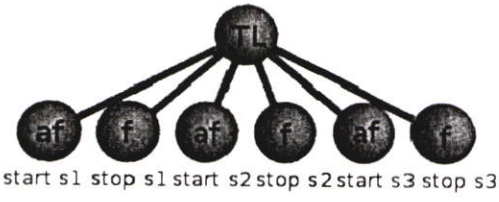
1. การเขียนสั่งงานคอมพิวเตอร์ด้วยภาษา CL ทำได้ง่ายกว่า การเขียนโปรแกรมโดยเรียกผ่านฟังก์ชั่นที่มีในลีนุกซ์โดยตรง เช่น การโปรแกรมให้คอมพิวเตอร์ทำงาน สองอย่างพร้อมกัน ถ้าโปรแกรมในลีนุกซ์ เราจะต้องสร้างเธรด และดูแลจัดการเธรดเอง แต่ถ้าโปรแกรมใน CL เราจะไม่ต้องห่วงการสร้างเธรด หรือทำลายเธรด อินเตอร์พรีตเตอร์จะดูแลเรื่องนี้ให้

7.8 สรุป

จากการเปรียบเทียบด้วยการวิเคราะห์ เราสามารถสรุปได้ว่า โครงสร้างข้อมูลสำหรับเก็บจุดเวลาของลีนุกซ์ เหมาะสำหรับการตั้งเวลาที่มีระยะเวลาไม่นานมาก และมีจุดเวลาที่จัดเก็บไม่มาก ซึ่งไม่สามารถนำมาใช้กับอินเตอร์พรีตเตอร์ CL ได้ เนื่องจาก CL มีการโปรแกรมการตั้งเวลาหลายจุด กระจายอยู่ทั่วไป

ทดลองโหลดโปรแกรมนี้เข้าสู่อินเตอร์พรีตเตอร์ CL อินเตอร์พรีตเตอร์จะวิเคราะห์โค้ดโปรแกรม และสร้าง abstract syntax tree จากนั้นอินเตอร์พรีตเตอร์จะจัดเก็บประโยคจาก abstract syntax tree ลงในโครงสร้างข้อมูล โดยในที่นี้ เราจะใช้สัญลักษณ์ว่า start s1, start s2 และ start s3 เพื่อแทนการสั่งให้เริ่มประมวลผลประโยคที่ 1, 2 และ 3 ตามลำดับ และใช้สัญลักษณ์ว่า stop s1, stop s2 และ stop s3 เพื่อแทนการสั่งให้ยุติการประมวลผลของประโยคที่ 1, 2 และ 3 ตามลำดับ

ความหมายของโปรแกรมนี้ต้องการให้อินเตอร์พรีตเตอร์ประมวลผลประโยคที่ละประโยค จากบนลงล่าง อย่างต่อเนื่องกัน ดังนั้นเราจึงใช้สัญญาลักษณ์ af และ f เพื่อใช้เป็นชื่อโหนดแทนจุดเวลาเริ่มต้น และจุดเวลาสิ้นสุดของแต่ละประโยค โดยกำหนดให้ af มีค่าเท่ากับจุดเวลาสิ้นสุดการประมวลผลของประโยคก่อนหน้า และ f มีค่าเท่ากับจุดเวลาสิ้นสุดของการประมวลผลของประโยคปัจจุบัน โครงสร้างข้อมูลมีลักษณะดังรูป



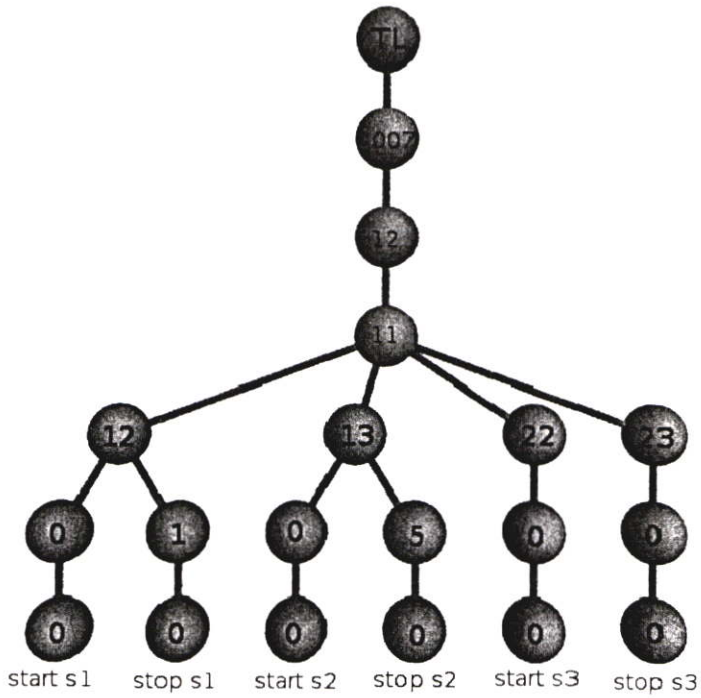
รูปที่ 8.1 โครงสร้างข้อมูลสำหรับโปรแกรม CL ในตัวอย่างที่ 1

ตัวอย่างที่ 2

ตัวอย่างต่อมาเป็นการทดลองอินเตอร์พรีตโปรแกรม CL ที่มีกำหนดเวลาเป็นลำดับ โดยสั่งให้คอมพิวเตอร์ทำงาน คล้ายกับตัวอย่างที่ 1 แต่การทำงานในแต่ละประโยค จะมีกำหนดเวลากำกับอยู่ และกำหนดเวลา มีค่าของประโยคที่ 1 2 และ 3 มีค่าเรียงกันตามลำดับ

```
[12:00:00|11/12/2007, 12:01:00|11/12/2007]:
                    print 'CL Test                              [s1]
[13:00:00|11/12/2007,13:05:00|11/12/2007]:
                    send-email('surachai@localhost')           [s2]
[22:00:00|11/12/2007,23:00:00|11/12/2007]:
                    download("file")                             [s3]
```

โปรแกรมนี้จะถูกจัดเก็บลงในโครงสร้างข้อมูล ตามจุดเวลาเริ่มต้น และจุดเวลาสิ้นสุดของแต่ละประโยค (รูปด้านล่าง) สังเกตว่าจะไม่จุดเวลาใดที่เก็บ เวลาเริ่มต้น หรือเวลาสิ้นสุดไว้ที่โหนด leaf เดียวกัน เนื่องจากจุดเวลาทั้งหมด มีลำดับ และเว้นระยะเวลาระหว่างกัน



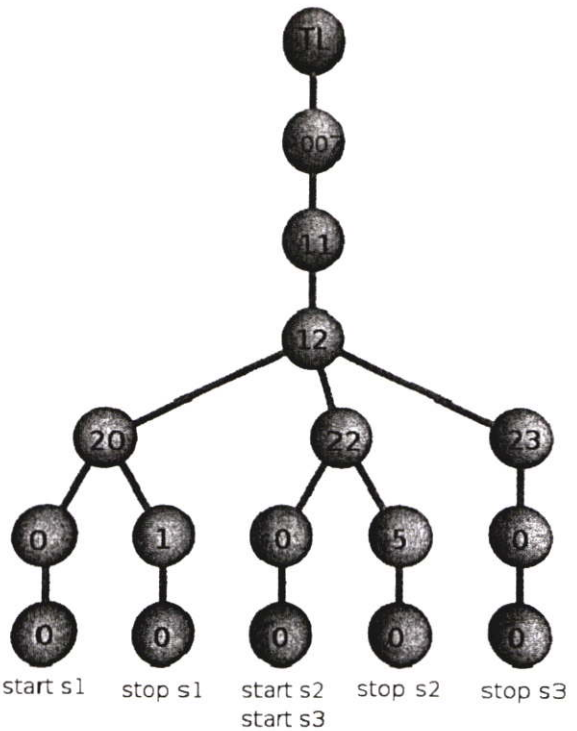
รูปที่ 8.2 โครงสร้างข้อมูลสำหรับโปรแกรม CL ในตัวอย่างที่ 2

ตัวอย่างที่ 3

ในตัวอย่างนี้เป็นการทดลองอินเตอร์พรีตโปรแกรม CL ที่กำหนดเวลามีการคาบเกี่ยวกัน โดยเราสั่งงานคอมพิวเตอร์คล้ายกับตัวอย่างที่ 1 แต่จุดเวลาเริ่มต้นการประมวลของประโยค s2 และ s3 มีค่าเท่ากัน นั้นหมายถึงประโยค s2 และ s3 จะต้องทำงานไปพร้อมๆ กัน

```
parallel:
   [20:00:00|11/12/2007, 20:01:00|11/12/2007]:
                print 'CL Test'                              [s1]
   [22:00:00|11/12/2007, 22:05:00|11/12/2007]:
                send-email('kan2005@gmail.com')              [s2]
   [22:00:00|11/12/2007,23:00:00|11/12/2007]:
                download("file")                             [s3]
```

เมื่อโหลดโปรแกรม CL เข้าไปเก็บไว้ในโครงสร้างข้อมูลแล้ว เราได้โครงสร้างข้อมูลดังรูปด้านล่าง สังเกตว่าที่โหนด leaf ที่แทนเวลา 22:00:00|11/12/2007 จะเก็บคำสั่งการเริ่มประมวลผลของประโยค s2 และ s3 เอาไว้ เพื่อให้ประโยคทั้งสองเริ่มทำงานพร้อมกันในภายหลัง



รูปที่ 8.3 โครงสร้างข้อมูลสำหรับโปรแกรม CL ในตัวอย่างที่ 3

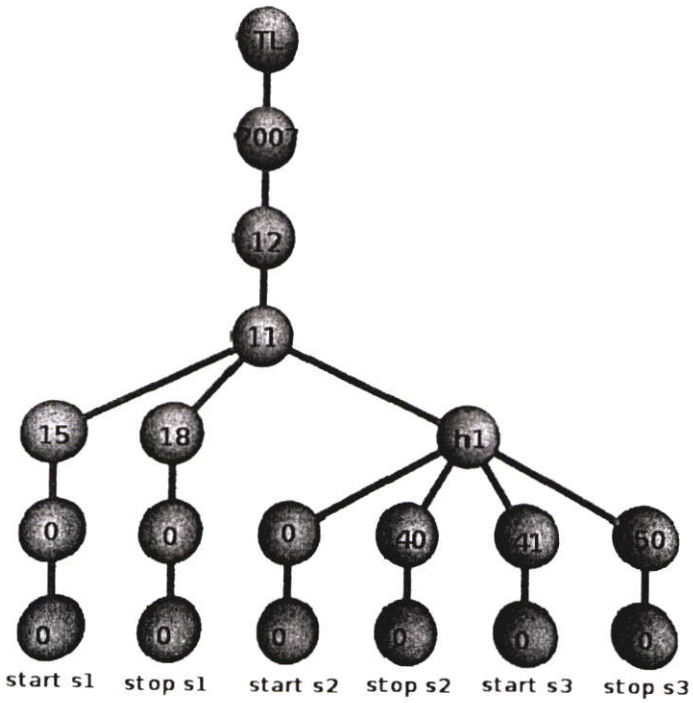
ตัวอย่างที่ 4

ในตัวอย่างนี้เป็นการทดลองการอินเตอร์พรีตโปรแกรม CL ที่ทำงานเป็นแบบ event-action และกำหนดเวลาเป็นตัวแปร โดยสั่งให้คอมพิวเตอร์รอรับเหตุการณ์ การกำหนดเวลาสั่งอาหารผ่าน SMS ในช่วงเวลา 15 นาฬิกา ถึง 18 นาฬิกา ของวันที่ 11 เดือน 12 ปี 2006 เมื่อมี SMS มาในช่วงเวลาดังกล่าว ก็ให้อ่านลำดับชั่วโมงมาจาก SMS แล้วจึงสั่งให้เครื่องล้างจานทำงานในช่วงนาทีที่ 0 ถึงนาทีที่ 40 ในลำดับชั่วโมงนั้น และสั่งอาหารในช่วงเวลาตั้งแต่นาทีที่ 41 ถึงนาทีที่ 50 ในชั่วโมงเดียวกัน เราสามารถโปรแกรมในภาษา CL ได้ดังนี้

```
[15:00:00|11/12/2006, 18:00:00|11/12/2006]:
   SMS_AVAILABLE ->
      h1 = getOrderTime()                                    [s1]
[h1:00:00|11/12/2006, h1:40:00|11/12/2006]:
   start_dish_cleaner()                                      [s2]
[h1:41:00|11/12/2006, h1:50:00|11/12/2006]:
   order_food(pizza)                                         [s3]
```

ประโยคที่ 1 อยู่ในรูปแบบ event->action เราจะคิดเป็นประโยคเดียวและจัดเก็บลงในโครงสร้างข้อมูล ส่วน 2 ประโยคถัดมา กำหนดเวลาเริ่มต้นมีตัวแปรอยู่ในกำหนดเวลาด้วย เราจะ

จัดเก็บประโยคเหล่านี้ไว้หลังเวลา 18:00:00|11/12/2006 โดยอนุญาตให้มีโหนด h1 ได้ โครงสร้างข้อมูล มีลักษณะดังรูป



รูปที่ 8.4 โครงสร้างข้อมูลสำหรับโปรแกรม CL ในตัวอย่างที่ 4

ตัวอย่างที่ 5

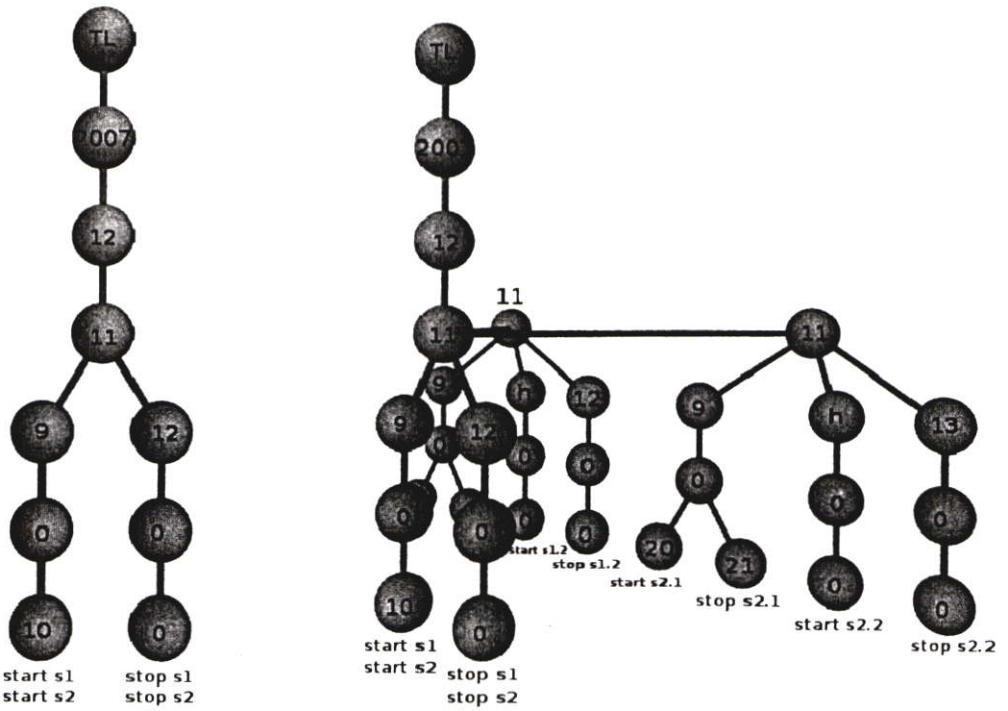
ในตัวอย่างนี้เป็นการทดลองอินเตอร์พรีตโปรแกรม CL ที่มี Nested statement และมีตัวแปรโดยในแต่ละ scope จะมีการใช้ชื่อตัวแปรซ้ำกัน อินเตอร์พรีตเตอร์จะต้องแยก scope การจัดเก็บค่าตัวแปรออกจากกันเพื่อให้การประมวลผลคำสั่งทำงานได้อย่างถูกต้อง

เราสมมติโปรแกรม CL ว่าต้องการให้คอมพิวเตอร์ประมวลผลประโยค แบบ sequential สองคำสั่ง พร้อมๆกัน ในช่วงเวลา 9:00:10 – 12:00:00 ของวันที่ 11/12/2006 โดยในประโยค sequential แรกต้องการให้คอมพิวเตอร์อ่านค่าเวลาหมายกำหนดการณ์การประชุมของ john ออกมา แล้วนำเวลานั้นตั้งเวลาการแจ้งเตือน ส่วนในประโยค sequential ต่อมาก็ให้คอมพิวเตอร์อ่านเวลานัดรับประทานอาหารกลางวันของ surachai ออกมา แล้วแจ้งเตือนผู้ใช้เมื่อถึงเวลาดังกล่าว เราสามารถเขียนความต้องการดังกล่าวในภาษา CL ได้ดังนี้

```
parallel:
  [9:00:10|11/12/2006, 12:00:00|11/12/2006] sequential:      [s1]
    [9:00:20|11/12/2006,  9:00:21|11/12/2006]
      h = getMeetingTime('www.john.com/schedule')            [s1.1]
    [h:00:00|11/12/2006, 12:00:00|11/12/2006]
      alert_me('It is a meeting time.')                      [s1.2]
  [9:00:10|11/12/2006, 12:00:00|11/12/2006]sequential:       [s2]
    [9:00:20|11/12/2006,  9:00:21|11/12/2006]
      h = getLunchTime('www.surachai.com/timetable')         [s2.1]
```

```
[h:00:00|11/12/2006, 13:00:00|11/12/2006]
  alert_me('It is lunch time.')                    [s2.2]
```

เมื่อโปรแกรมนี้ถูกอินเตอร์พรีตในครั้งแรก อินเตอร์พรีตเตอร์จะมองประโยคออกเป็น 2 ประโยค เท่านั้น คือ s1 และ s2 จากนั้นเมื่อถึงเวลา 9:00:10|11/12/2006 อินเตอร์พรีตเตอร์ก็จะอินเตอร์พรีตประโยค s1 และ s2 เกิดเป็น Tree ต้นใหม่ เราสามารถแสดงลักษณะของ Tree ได้ดังรูป ก และ ข ตามลำดับ



(ก) (ข)

รูปที่ 8.5 โครงสร้างข้อมูลสำหรับเก็บโปรแกรมตัวอย่างที่ 5 (ก) Tree ที่เก็บประโยคในครั้งแรก (ข) Tree ภายหลังจากประมวลผลประโยคชุดคำสั่งไปแล้ว

8.3 สรุป

ในบทนี้เราได้นำแนวคิดโครงสร้างข้อมูลมาทดลองใช้งานจริง ซึ่งสามารถได้ผลตามที่ออกแบบภาษาไว้

บทที่ 9
สรุปผลการวิจัย และข้อเสนอแนะ

ในงานวิจัยนี้ได้นำเสนอการออกแบบ และพัฒนาอินเตอร์พรีตเตอร์ภาษา CL โดยเริ่มจากศึกษาลักษณะการทำงานของมนุษย์เรา เราสังเกตว่าคนเรามีแผนกำหนดการ อยู่ก่อนจะทำสิ่งใดๆ ดังนั้นการทำงานของอินเตอร์พรีตเตอร์ภาษา CL จะต้องมีโครงสร้างข้อมูลที่เก็บลำดับการทำงานล่วงหน้าก่อนการทำงานจะเกิดจริง เพื่อออกแบบโครงสร้างข้อมูลดังกล่าว เราได้ศึกษาเวลาที่ใช้ในชีวิตประจำวัน โดยเราพบว่าจุดเริ่มต้นและสิ้นสุดของเวลาหาขอบเขตมิได้ และการอ้างอิงเวลาใช้หน่วยเวลาหลายหน่วย เช่น ปี เดือน วัน ชั่วโมง นาที และวินาที ซึ่งแต่ละหน่วยเวลามีการแบ่งย่อยเป็นหน่วยเวลาเล็กๆ ลงไปไม่มีที่สิ้นสุด จากความจริงดังกล่าว เราจึงได้ออกแบบโครงสร้างข้อมูลแบบ Tree ที่มีสามารถเก็บจุดเวลาในอดีตและอนาคตไม่มีที่สิ้นสุด รวมทั้งเก็บจุดเวลาที่อ้างด้วยหน่วยเวลาซอยย่อยได้อย่างไม่ที่สิ้นสุด

นอกจากนี้เรายังได้พิจารณาลำดับการทำงานในชีวิตประจำวัน ซึ่งเราพบว่าบ้างครั้งมีการกำหนดเวลาการทำงานที่เปลี่ยนแปลงได้อยู่ในแผนของเรา ทั้งในรูปแบบกำหนดเวลาที่เปลี่ยนตามผลของการทำงานที่ผ่านมา หรือ กำหนดเวลาที่เปลี่ยนแปลงเป็นคาบเวลา ตัวอย่างเช่น การนัดหมายส่งบทความที่เราทราบอยู่ว่าจะมีขึ้นในเดือนมกราคม แต่ไม่ทราบแน่ชัดว่าเมื่อไร เมื่อถึงเวลาใกล้เดือนมกราคมเราจึงทราบเวลาที่แน่ชัด หรือ กำหนดการจ่ายค่าน้ำ ที่มาเป็นประจำในวันที่ 25 ของทุกๆ เดือน จากการพิจารณาการเก็บลำดับแผนการทำงานดังกล่าว ทำให้เราสามารถปรับปรุงโครงสร้างข้อมูลให้สามารถเก็บสัญลักษณ์สำหรับค่าเวลาไว้ได้อย่างครอบคลุมหลายๆ กรณี

ในส่วนของการประมวลผลภาษา CL เราได้นำเสนอสถาปัตยกรรมการประมวลผลภาษา CL ใหม่ โดยยังคงความสามารถในการประมวลผลประโยคในรูปแบบเดิมทุกประการ และมีการเพิ่มความสามารถในการประมวลผลประโยคที่มีกำหนดเวลาเปลี่ยนแปลงได้ เช่น กำหนดเวลาเปลี่ยนแปลงตามการทำงานของประโยคก่อนหน้าแล้ว เปลี่ยนแปลงเป็นคาบๆ รวมทั้งจัดการเรื่อง scope ของตัวแปรอีกด้วย

แนวทางที่จะพัฒนาในอนาคต

1. ปรับปรุงโครงสร้างข้อมูลและอัลกอริธึมให้รองรับการค้นหาที่กำหนดจุดเวลา 1 จุด แล้วให้ค่าช่วงเวลาทั้งหมดที่จุดเวลานั้นครอบคลุมเวลานั้นออกมา (stabbing query)
2. พัฒนาอินเตอร์พรีตเตอร์ให้สมบูรณ์เพื่อนำไปใช้ในรูปแบบ Organizer ส่วนบุคคล ที่ผู้ใช้ทั่วไปสามารถโปรแกรมกิจกรรมการทำงานในแต่ละวันได้

3. พัฒนาอินเตอร์พรีตเตอร์ให้ในรูปแบบ Workflow Engine สำหรับองค์กร หรือการจัดการเวลาสำหรับกลุ่มคน
4. ออกแบบภาษา CL ให้สามารถนิยามคลาสได้ และเพิ่มความสามารถอินเตอร์พรีตเตอร์ให้รองรับคลาสได้

บรรณานุกรม

[1] วชิระ ศิริพจนาวรรณ และ สุพัณณดา โชติพันธ์. **"บราวเซอร์ที่โปรแกรมได้"**. ปริญญานิพนธ์ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีเจ้าคุณทหารลาดกระบัง ปีการศึกษา 2543.

[2] ธนวัฒน์ แก้วค้า และ บุญทวี สันติศรีวราภรณ์. **"บราวเซอร์ที่โปรแกรมได้โดยใช้ภาษาไจธอน"**. ปริญญานิพนธ์ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีเจ้าคุณทหารลาดกระบัง ปีการศึกษา 2544.

[3] วิศิษฏ์ หิรัญกิตติ และคณะ. **"บราวเซอร์ที่สามารถโปรแกรมได้"**, การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 6 NCSEC2002. 2545.

[4] ชาคริต เฮงสิริกุล และ เชิดเกียรติ แซ่แต้. **"บราวเซอร์ที่โปรแกรมได้บนเครื่อง Pocket PC"**. ปริญญานิพนธ์ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีเจ้าคุณทหารลาดกระบัง ปีการศึกษา 2546.

[5] วิศิษฏ์ หิรัญกิตติ และคณะ, **"บราวเซอร์ที่สามารถโปรแกรมได้"**, การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 6, 2545

[6] วิศิษฏ์ หิรัญกิตติ และสุพัณณดา โชติพันธ์ **"CL: ภาษาสำหรับการสั่งงานคอมพิวเตอร์ด้วยเวลาและเหตุการณ์"**, การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 7, 2546.

[7] สุพัณณดา โชติพันธ์ **"CL: ภาษาสำหรับการสั่งงานคอมพิวเตอร์ด้วยเวลาและเหตุการณ์"**, วิทยานิพนธ์ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง, 2547. (4.1)

[8] Allen J., **"Maintaining Knowledge about Temporal Intervals."**, Communications of ACM 26(11), pp. 832-843, 1983 (2.1)

[9] Allen J., **"Time and Time Again : The Many Ways to Represent Time"**, International Journal of Intelligent Systems, 6(4), pp. 341-355, July 1991. (2.2)

[10] Allen J., **"Towards a General Theory of Action and Time"**, Artificial Intelligence, 23, pp. 123-154. 1984 (2.3)

[11] A. Galton., **"A Critical Examination of Allen's Theory of Action and Time"**, Artificial - Intelligence, 42, pp. 159-188, 1990. (2.4)

[12] A. Galton, **"Towards a Qualitative Theory of Movement"**, [online] Available: http://citeseer.ist.psu.edu/372.html

[13] Wikipedia, **"Real-Time Computing"** [online]

Available: http://en.wikipedia.org/wiki/Real-time

[14] K. Prouskas and J. Pitt., **"A Real-Time Architecture for Time-Aware Agents"** IEEE transactions on systems, man, and cybernetics , IEEE Explorer, pp. 1553-1568, 2004.

[15] George Varghese and Anthony Lauck, **"Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility"**, IEEE/ACM Transaction on networking, vol. 5, no. 6, pp 824-834, December 1997

[16] Beazley D.M. **Python Essential Reference**. : New Riders. 1999.

[17] Chun W. J. **Core Python Programming**. : Prentice Hall. Upper Saddle River. 2001.

[18] Daniel P. Bovet & Marco Cesati , **Understanding the Linu Kernel:** O'reilly Sebstopol, CA USA January 2001

ภาคผนวก

ผลงานวิจัยที่ได้รับการตีพิมพ์เผยแพร่

[1] วิศิษฏ์ หิรัญกิตติ และสุรชัย ล้อเจริญ **"นามธรรมของตารางลำดับเวลาและการใช้งานสำหรับประมวลผลคำสั่งในภาษา CL"**, การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 9, 2548.

[2] วิศิษฏ์ หิรัญกิตติ และสุรชัย ล้อเจริญ **"การประมวลผลตารางการทำงานของโปรแกรมภาษา CL ที่กำหนดเวลาแปรเปลี่ยนได้"**, การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 10, 2549.

The 9th National Computer Science and Engineering Conference

October 27-28, 2005

University of Thai Chamber of Commerce, Bangkok Thailand

Organized by:

Department of Computer Engineering, School of Engineering,
University of Thai Chamber of Commerce

In Cooperation with:

Electrical Engineering; Electronics, Computer, Telecommunications and Information Technology Association of Thailand (ECTI)

IEEE Communications Society, Thailand Chapter

Sponsored by:

University of Thai Chamber of Commerce

National Electronics and Computer Technology Center (NECTEC)

Sun Microsystems (Thailand)

CS Loxinfo Public Company Limited

Pearson Education Indochina Limited

The OGA Group

นามธรรมของตารางลำดับเวลาและการใช้งานสำหรับประมวลผลคำสั่งในภาษา CL
An Abstraction of Schedules and Its Application to Program Interpretation in CL

วิศิษฏ์ หิรัญกิตติ และ สุรชัย ล้อเจริญ
ห้องวิจัยการสื่อสารและคมนาคมชาญฉลาด ภาควิชาวิศวกรรมคอมพิวเตอร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง กรุงเทพ 10520 ประเทศไทย
E-mail: v_hirankitti@gmail.com, kan2005@gmail.com

บทคัดย่อ

บทความนี้ศึกษาลักษณะนามธรรมของเวลารวมทั้งตารางลำดับเวลาอันประกอบด้วยกลุ่มของจุดเวลาและช่วงเวลาที่เรียงลำดับกัน ซึ่งผลนำไปสู่การประดิษฐ์โครงสร้างข้อมูลของตารางลำดับเวลา ที่มีความสามารถใช้บันทึกจุดเวลาและช่วงเวลา ที่หน่วยความละเอียดของการวัดเวลาที่ระดับหยาบละเอียดได้อย่างไม่จำกัด สามารถบันทึกจำนวนจุดเวลาและช่วงเวลาได้อย่างไม่จำกัด ทั้งเวลาในอดีตและอนาคตอย่างไม่สิ้นสุด รวมทั้งยังได้พัฒนาอัลกอริธึมพื้นฐานที่ใช้ในการจัดการข้อมูลที่บันทึกในตารางเวลา ซึ่งโครงสร้างข้อมูลนี้สามารถนำไปประยุกต์ใช้ในการปรับปรุงอินเตอร์พรีตเตอร์ภาษา CL [1] ให้สามารถประมวลผลคำสั่งและตอบสนองกับเหตุการณ์ตามกำหนดเวลาได้อย่างมีประสิทธิภาพมากขึ้น

Abstract

In this paper we studied an abstraction of time and a time schedule. The latter is defined to be an ordered list of time points and/or time intervals. The result of the study led us to invent a tree-like abstract data structure of a time schedule which can store time points and intervals, with different details of time measurements, in any number, and in any range of time scale ranging from the endless past to the endless future. In addition, we also developed some basic algorithms for processing the data stored in this data structure. Later this data structure has been applied to improve the CL interpreter so that it can interpret CL statements according to a time schedule more efficiently.

Keyword: Time Representation, Scheduling, Real-time Programming, Software Agent

1. บทนำ

เวลาเป็นสิ่งที่มนุษย์กำหนดขึ้นเพื่อแสดงถึงการดำเนินไปของธรรมชาติ ในบทความวิจัยชิ้นนี้ เราได้ทำการศึกษาธรรมชาติที่เป็นนามธรรมของเวลา โดยจำแนกเวลาออกเป็น 3 ประเภท คือ ช่วงเวลา จุดเวลา และระยะเวลา โดยศึกษาคุณสมบัติทางคณิตศาสตร์ของเวลาทั้ง 3 ประเภท หลังจากนั้นเราศึกษาลำดับของจุดเวลาและช่วงเวลาที่อยู่ในรูปของตารางลำดับเวลา

บทความในส่วนที่เหลือจะกล่าวถึงงานวิจัยที่เกี่ยวข้องในหัวข้อที่ 2 ความหมายของเวลาและประเภทของเวลา รวมทั้งตารางลำดับเวลาในหัวข้อที่ 3 จากนั้นเรานำเสนอโครงสร้างข้อมูลสำหรับตารางลำดับเวลาในหัวข้อที่ 4 และอัลกอริทึมเพื่อการประมวลผลข้อมูลในตารางลำดับเวลาในหัวข้อที่ 5 ส่วนหัวข้อที่ 6 จะเป็นเรื่องการประยุกต์ใช้โครงสร้างข้อมูลของตารางเวลาในภาษา CL หัวข้อที่ 7 แสดงผลการทดลองและขอสรุปงานวิจัยในหัวข้อที่ 8

2. งานวิจัยที่เกี่ยวข้อง

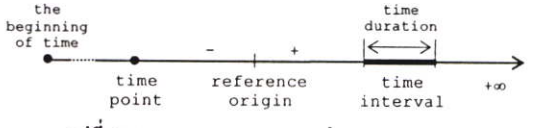
การศึกษาเรื่องเวลามีมานานแล้ว ดังเช่น Allen [3, 4] ได้แบ่งประเภทเวลาออกเป็น จุดเวลา และ ช่วงเวลา โดยเขาได้ใช้ช่วงเวลาเป็นส่วนประกอบพื้นฐานในการนิยามความสัมพันธ์ของเวลาและตรรกศาสตร์เวลา ซึ่งความสัมพันธ์ของเวลาที่ได้ถูกนำมาใช้สร้างเงื่อนไขลำดับเวลาของเหตุการณ์และการทำงานที่สัมพันธ์กับเวลา ส่วน Kawalski และ Sergot [5] ได้

ศึกษารูปแบบของเวลาในทางตรรกะและนำเสนอวิธีการเพิ่มและแก้ไขความรู้ในฐานความรู้ โดยอาศัยลำดับการเกิดขึ้นของเหตุการณ์ที่ระบุโดยจุดเวลา จะพบว่างานวิจัยทั้งสองมุ่งเน้นการใช้ประโยชน์จากเวลาประเภทจุดเวลาและช่วงเวลา แต่ในชีวิตประจำวันยังมีเวลาอีกหนึ่งประเภทคือระยะเวลา นอกจากนี้ตารางลำดับเวลาก็มีความสำคัญในชีวิตประจำวัน จึงได้มีการศึกษาในบทความนี้

3. เวลา

เวลาเกิดจากความสามารถในการแบ่งแยกถึงการเกิดก่อนหลัง เรามักแทนเวลาด้วยการเปลี่ยนแปลงที่ต่อเนื่องไม่มีที่สิ้นสุด เวลาสามารถแทนด้วยเส้นจำนวนจริงที่ค่าเพิ่มขึ้นเรื่อยๆ ไม่มีที่สิ้นสุดเรียกว่า "เส้นแกนเวลา (Time Line)" จุดใดๆบนเส้นแกนเวลานี้ขอเรียกว่า "จุดเวลา (time point)"และช่วงที่เป็นเส้นเชื่อมระหว่างจุดเวลาสองจุดที่ไม่ใช่จุดเดียวกัน ขอเรียกว่า "ช่วงเวลา (time interval)" ส่วนความยาวของเส้นดังกล่าวขอเรียกว่า "ระยะเวลา (time duration)" ด้วยเหตุนี้เราจึงแบ่งเวลาเป็น 3 ประเภทคือ จุดเวลา ช่วงเวลา และระยะเวลา

เพื่อความสะดวกในการกำหนดค่าให้จุดเวลา จึงมีการกำหนดจุดอ้างอิง (reference origin) คล้ายจุดเริ่มต้นสมมติไว้บนแกนเวลา โดยให้จุดเวลาที่อยู่ขวามือของจุดอ้างอิงนี้มีค่าเป็นบวก ส่วนจุดเวลาที่อยู่ทางซ้ายมือให้มีค่าเป็นลบ เช่น การอ้างอิงจุดเวลาด้วยระบบคริสตศักราช จุดเวลาที่อยู่หลังและก่อนจุดเวลาวันที่ 1 ม.ค. ค.ศ 1 จะถือว่ามีค่าเป็นบวกและลบตามลำดับ ดังแสดงได้ตามรูป



รูปที่ 1 แสดง แกนเวลา จุดเวลา ช่วงเวลา และระยะเวลา

เพื่อให้การศึกษาเรื่องเวลามีความชัดเจน ต่อไปจึงขออธิบายเวลาทั้ง 3 ประเภท ในรูปแบบคณิตศาสตร์

3.1 จุดเวลา ช่วงเวลา และระยะเวลา

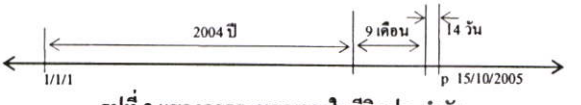
เนื่องจากเส้นแกนเวลาประกอบขึ้นจากจุดเวลาทั้งหมด จึงสามารถแทนด้วยเซตของจุดเวลาทั้งหมด สมมติให้ T แทนแกนเวลา ดังนั้น จุดเวลาใดๆ คือ $p_i \in T$

ส่วนช่วงเวลา คือ ช่วงที่เป็นเส้นเชื่อมระหว่างจุดเวลาสองจุด p_s และ p_e โดยที่ถ้า $p_s < p_e$ ซึ่งหมายถึง p_s และ p_e คือจุดเริ่มต้นและจุดสิ้นสุดของช่วงเวลาตามลำดับ เราสามารถแทนช่วงเวลาด้วยเซตต่อเนื่องของจุดเวลาตั้งแต่ p_s ถึง p_e คือ $\{ p \mid p_s \leq p < p_e \}$ หรือเขียนอย่างสั้นๆว่า $[p_s, p_e)$ จะพบว่าเส้นแกนเวลาก็คือช่วงเวลา [the beginning of time, $+\infty$) (ซึ่งอาจหมายถึงช่วงตั้งแต่จุดกำเนิดของเวลาไปไม่มีที่สิ้นสุด)

ดังนั้นระยะเวลาสำหรับช่วงเวลา $[p_s, p_e)$ จึงเท่ากับ $p_e - p_s$ ซึ่งพบว่าอยู่ในรูปของค่าปริมาณเหมือนกับค่าปริมาณของสสาร เช่น มวล เป็นต้น ซึ่งระยะเวลาที่ปริมาณต่างๆ จะมีการกำหนดหน่วยให้ เช่น เป็นหน่วยวินาที วัน ปี เป็นต้น

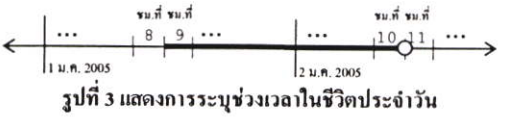
3.2 การระบุเวลาในชีวิตประจำวัน

ในชีวิตประจำวันเพื่อให้ง่ายต่อการระบุจุดเวลา จึงกำหนดให้จุดเวลาถูกอ้างด้วยลำดับการนับหน่วยเวลาต่างๆจากจุดอ้างอิงถึงจุดเวลานั้น เช่น จุดเวลา p ในรูปข้างล่าง ถูกกำหนดโดยลำดับของหน่วยเวลาจากจุดอ้างอิง (1/1/1) ไปยังจุดเวลา p จากรูปตำแหน่งจุดเวลา p ถือว่าได้ผ่านจุดอ้างอิงไปเป็นระยะเวลา 2004 ปี แล้วย่างเข้าสู่ปีที่ 2005 และได้ผ่านจุดที่ครบ 2004 ปีไป 9 เดือน แล้วย่างเข้าสู่เดือนที่ 10 และได้ผ่านจุดที่ครบ 9 เดือนไป 14 วัน แล้วย่างเข้าวันที่ 15 ดังนั้นจึงเรียกจุดเวลา p เป็น วันที่ 15 เดือนที่ 10 ปีที่ 2005 ดังรูป สรุปได้ว่าจุดเวลาที่ผ่านจุดอ้างอิงไปเป็นระยะเวลา x หน่วยแล้ว จะถูกเรียกว่าเป็น "จุดเวลาอยู่ที่ลำดับที่ $x + 1$ ของหน่วยนั้น"



รูปที่ 2 แสดงการระบุจุดเวลาในชีวิตประจำวัน

สำหรับรูปแบบช่วงเวลาที่ใช้ในชีวิตประจำวันเช่น ช่วงเวลาระหว่าง 9 นาฬิกา 1 ม.ค. 2005 ถึง 11 นาฬิกา 2 ม.ค. 2005 หมายถึง ช่วงเวลาตั้งแต่ จุดเวลาชั่วโมงที่ 9 วันที่ 1 เดือนที่ 1 ปีที่ 2005 ไปจนถึง ครบจุดเวลาชั่วโมงที่ 10 วันที่ 2 เดือนที่ 1 ปีที่ 2005 แต่ไม่รวมจุดเวลาชั่วโมงที่ 11 ดังรูป



รูปที่ 3 แสดงการระบุช่วงเวลาในชีวิตประจำวัน

หลังจากที่ได้นิยามความหมายของเวลาทั้ง 3 ประเภทแล้ว ต่อไปเราขอกล่าวถึงความสัมพันธ์ทางคณิตศาสตร์ที่เป็นไปได้ระหว่างจุดเวลาและช่วงเวลา

3.3 ความสัมพันธ์ทางคณิตศาสตร์ระหว่างจุดเวลาและช่วงเวลา

ความสัมพันธ์ระหว่าง 2 จุดเวลา p_1 และ p_2 มีหลายลักษณะ เช่น จุดเวลาสองจุดเป็นจุดเดียวกัน ต่างกัน เกิดก่อนหรือเกิดหลัง ซึ่งสรุปได้ดังตาราง

ตารางที่ 1 ความสัมพันธ์ระหว่างจุดเวลากับจุดเวลา

ความสัมพันธ์	ความสัมพันธ์
1. $p_1=p_2$ (p_1 equal p_2)	4. $p_1 \neq p_2$ (p_1 not equal p_2)
2. $p_1<p_2$ (p_1 before p_2)	5. $p_1 \leq p_2$ (p_1 before or equal p_2)
3. $p_1>p_2$ (p_1 after p_2)	6. $p_1 \geq p_2$ (p_1 after or equal p_2)

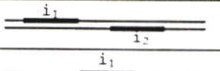
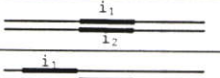
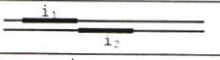
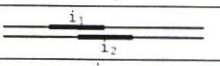
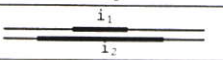
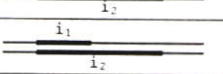
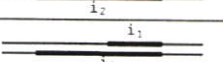
ความสัมพันธ์ระหว่างจุดเวลา p และช่วงเวลา i สามารถพิจารณาได้เป็น p เกิดก่อนหรือเกิดหลัง i หรือ p อยู่ภายใน i ที่จุดเริ่มต้น ระหว่าง หรือ จุดปลาย ซึ่งความสัมพันธ์นี้ได้เสนอโดยคุณสุพัณณคา โชติพันธ์ [2] สรุปได้ดังนี้

ตารางที่ 2 ความสัมพันธ์ระหว่างจุดเวลากับช่วงเวลา

ความสัมพันธ์	ความสัมพันธ์
1. $p < i$ (p before i)	4. $p\ s\ i$ (p starts i)
2. $p > i$ (p after i)	5. $p\ f\ i$ (p finishes i)
3. $p\ d\ i$ (p during i)	

สำหรับความสัมพันธ์ระหว่างช่วงเวลา Allen [3] ได้เสนอความสัมพันธ์ระหว่างช่วงเวลาและช่วงเวลา ได้แก่ ก่อน เท่ากัน ต่อเนื่องกัน ซ้อนกัน อยู่ระหว่าง เป็นช่วงเริ่มต้น และเป็นช่วงสิ้นสุด ซึ่งสรุปได้ดังนี้

ตารางที่ 3 ความสัมพันธ์ระหว่างช่วงเวลากับช่วงเวลา

ความสัมพันธ์	ลักษณะความสัมพันธ์
1. $i_1 < i_2$ (i_1 before i_2)	
2. $i_1 = i_2$ (i_1 equal i_2)	
3. $i_1\ m\ i_2$ (i_1 meets i_2)	
4. $i_1\ o\ i_2$ (i_1 overlaps i_2)	
5. $i_1\ d\ i_2$ (i_1 during i_2)	
6. $i_1\ s\ i_2$ (i_1 starts i_2)	
7. $i_1\ f\ i_2$ (i_1 finishes i_2)	
ความสัมพันธ์ที่เหลือจะเป็นความสัมพันธ์แบบตรงกันข้ามกับความสัมพันธ์ในหมายเลข 2, 3, 4, 5, 6,7	

3.4 ตารางลำดับเวลา

ในชีวิตประจำวัน เราพบว่ามักมีการใช้กลุ่มของจุดเวลาและช่วงเวลา ในลักษณะที่เรียงกันตามลำดับ เช่น ตารางเวลาการเดินรถ ปฏิทิน เป็นต้น เพื่อความกระชับในการเรียกตารางลำดับเวลา ต่อจากนี้เราขอเรียกเป็นสั้นๆว่า "ตารางเวลา (Schedule)" ในทางคอมพิวเตอร์เราสามารถแทนตารางเวลาด้วยลิสต์ที่เรียงลำดับสมาชิกที่เป็นจุดเวลาและ/หรือช่วงเวลาจากน้อยไปหามาก (ถ้าสมาชิกคู่ใดเกิด ณ จุดเวลาเดียวกัน เราสามารถเรียงลำดับคู่นี้สลับกันได้) เช่น [[1/1/2005, 20/1/2005), [25/1/2005, 30/1/2005], 25/1/2005, 31/1/2005] เพื่อความชัดเจนขอให้คำจำกัดความของคำว่า "มาก่อน" ดังนี้ :

จุดเวลา p_1 มาก่อน p_2 ก็ต่อเมื่อ $p_1 < p_2$

จุดเวลา p มาก่อนช่วงเวลา i ก็ต่อเมื่อ $p < i$

ช่วงเวลา i มาก่อนจุดเวลา p ก็ต่อเมื่อ $p > i$, $p\ d\ i$, หรือ $p\ f\ i$

ช่วงเวลา i_1 มาก่อน i_2 ก็ต่อเมื่อ $i_1<i_2$, $i_1\ o\ i_2$, $i_1\ m\ i_2$, หรือ $i_2\ f\ i_1$

ส่วนคำว่า "เกิด ณ จุดเวลาเดียวกันหรือเกิดพร้อมกัน" ขอให้คำจำกัดความดังนี้

จุดเวลา p_1 เกิดพร้อมกันกับจุดเวลา p_2 ก็ต่อเมื่อ $p_1 = p_2$

จุดเวลา p เกิดพร้อมกันกับช่วงเวลา i ก็ต่อเมื่อ $p\ s\ i$

ช่วงเวลา i_1 เกิดพร้อมกันกับ i_2 ก็ต่อเมื่อ $i_1=i_2$ หรือ $i_1\ s\ i_2$

4. โครงสร้างข้อมูลสำหรับตารางเวลา

เพื่อนำตารางเวลามาใช้ประโยชน์ในทางคอมพิวเตอร์ เราจึงได้คิดค้นโครงสร้างข้อมูลของตารางเวลาขึ้น สำหรับใช้บันทึกและประมวลผลข้อมูลจุดและช่วงเวลาที่เรียงลำดับกัน โดยที่มีระดับความละเอียดของหน่วยเวลาที่แตกต่างกันได้อย่างไม่จำกัด และยังสามารถบันทึกข้อมูลเวลาได้อย่างไม่จำกัดจำนวนและครอบคลุมช่วงเวลาทั้งอดีตและอนาคตได้อย่างไม่มีที่สิ้นสุด

4.1 การออกแบบโครงสร้างข้อมูล

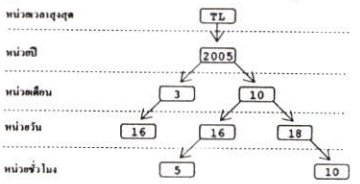
จุดประสงค์เราคือต้องการออกแบบโครงสร้างข้อมูลสำหรับตารางเวลาซึ่งใช้เก็บจุดเวลาและช่วงเวลาที่เรียงลำดับกัน การออกแบบขอเริ่มพิจารณาจากความจริงที่ว่าช่วงเวลาประกอบจากจุดเวลา 2 จุด ดังนั้นโครงสร้างข้อมูลที่ออกแบบจะต้องสามารถใช้เก็บจุดเวลาได้เป็นอันดับแรก

เนื่องจากจุดเวลาในชีวิตประจำวันถูกระบุโดยลำดับการนับหน่วยเวลาหลายๆหน่วยจากจุดอ้างอิงถึงจุดเวลานั้น โดยนำลำดับหน่วยต่างๆมาเรียงจากหน่วยเล็กไปหาใหญ่ เช่น จุดเวลา 16 มี.ค. 2005 (วันที่ 16 เดือนที่ 3 ปีที่ 2005) หรือจุดเวลา 16 มี.ค. 2005 5 นาฬิกา 30 นาที 10 วินาที (วินาทีที่ 10 นาทีที่ 30 ชั่วโมงที่ 5 วันที่ 16 เดือนที่ 3 ปีที่ 2005) จะพบว่าจุดเวลาสามารถกำหนดได้ด้วยหน่วยเวลาที่มีความละเอียดแตกต่างกัน ดังนั้นจุดเวลาที่เก็บในโครงสร้างข้อมูลจะต้องสามารถระบุโดยหน่วยเวลาที่ระดับความละเอียดต่างๆ กันได้หลายๆหน่วย

สมมติว่าเราจะเก็บจุดเวลา 3 จุด 16/3/2005, 16/10/2005 เวลา 5 นาฬิกา, 18/10/2005 เวลา 10 นาฬิกา ดังนั้นโครงสร้างข้อมูลจะต้องสามารถอ้างอิงถึงหน่วย ปี เดือน วัน ชั่วโมง นาที และวินาที ที่ใช้อ้างอิงโดยจุดเวลาได้ ซึ่งเราได้ออกแบบโดยใช้ Tree สำหรับการเก็บจุดเวลาต่างๆ ดังนี้

- ให้ระดับของโหนดแทนหน่วยเวลา โดยระดับบนสุดเป็นหน่วยสูงสุด (เท่ากับหน่วยขนาดแกนเวลา) ระดับรองลงมาเป็นหน่วยย่อยที่แบ่งย่อยจากหน่วยก่อนหน้าลดหลั่นตามลำดับ
- โหนดแต่ละโหนดแทนลำดับของหน่วยที่จะใช้อ้างโดยจุดเวลา ยกเว้น root ที่อยู่ที่ระดับหน่วยสูงสุด มีเพียงลำดับเดียว จึงขอแทนด้วย TL เพื่อให้สื่อถึงหน่วยเวลาของ Time Line กลุ่มโหนดที่ระดับเดียวกันจะต้องเรียงลำดับของโหนดค่าน้อยอยู่ทางซ้าย โหนดค่ามากอยู่ทางขวา
- การเชื่อมโหนดจากระดับบนมายังโหนดลูกตามลำดับหน่วยที่ระบุในจุดเวลานั้นๆ เป็นการระบุถึงจุดเวลาที่สมบูรณ์ที่ถูกบันทึกไว้ในโครงสร้าง

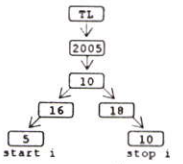
จากตัวอย่างจุดเวลาทั้ง 3 จุดข้างต้น เมื่อถูกบันทึกในโครงสร้างข้อมูลแล้ว จะได้ Tree ลักษณะดังรูป



รูปที่ 4 แสดง Tree ที่เก็บจุดเวลา

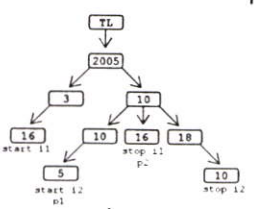
สรุปได้ว่าจุดเวลาแต่ละจุดจะแทนด้วย path จาก root ไปถึง leaf เพื่อความสะดวกดังนั้นเราจะให้ leaf มีความหมายแทน path ที่มาสิ้นสุดที่ leaf นั้นทำให้ leaf มีความหมายแทนจุดเวลาที่ระบุโดย path ที่มาสิ้นสุดที่ leaf นั้น

สำหรับการบันทึกช่วงเวลาใน Tree นั้นเนื่องจากช่วงเวลาระบุโดยจุดเวลาเริ่มต้นและจุดเวลาสิ้นสุด ดังนั้นการบันทึกช่วงเวลาทำได้โดยบันทึกจุดเวลาทั้ง 2 ลงในโครงสร้าง พร้อมระบุว่าจุดใดเป็นจุดเริ่มต้นและจุดใดเป็นจุดสิ้นสุดของช่วงเวลาใด ดังตัวอย่าง การบันทึกช่วงเวลา i = [16/10/2005 เวลา 5 นาฬิกา, 18/10/2005 เวลา 10 นาฬิกา) ใน Tree จะให้ผลดังนี้



รูปที่ 5 แสดง Tree ที่เก็บช่วงเวลา

รูปด้านล่างเป็นตัวอย่าง Tree ที่เก็บจุดเวลา p1 ที่ 10/10/2005 เวลา 5 นาฬิกา และ p2 ที่ 16/10/2005 และช่วงเวลา i1 = [16/3/2005, 16/10/2005) และ i2 = [10/10/2005 เวลา 5 นาฬิกา, 18/10/2005 เวลา 10 นาฬิกา) เพื่อให้สามารถระบุถึงตำแหน่งของจุดและช่วงเวลาที่เก็บใน Tree ต่อจากนี้ไปเราจะบันทึกชื่อของจุดและช่วงเวลาที่ตำแหน่งนั้นๆ ของมันใน Tree



รูปที่ 6 แสดง Tree ที่เก็บจุดเวลาและช่วงเวลา

จากลักษณะของโครงสร้าง Tree ที่ได้ออกแบบมานี้ พบว่าโครงสร้างนี้สามารถเก็บรักษาคุณสมบัติทางคณิตศาสตร์ของจุดเวลาและช่วงวลา ตลอดจนความความสัมพันธ์ของจุดเวลาและช่วงเวลาไว้ได้ เช่น จากรูป

- path ของ p1 อยู่ทางซ้ายของ path p2 จะหมายถึง p1 < p2
- p2 อยู่ระหว่าง path จุดเริ่มต้นของ i2 และจุดสิ้นสุดของ i2 จะหมายถึง p2 d i2
- จุด p1 อยู่ตำแหน่งเดียวกับจุดเริ่ม i2 จะหมายถึง p1 s i2

- path จุดเริ่มต้นของ i2 อยู่ระหว่าง path จุดเริ่มต้นของ i1 และจุดสิ้นสุดของ i1 จะหมายถึง i1 o i2

ต่อจากนี้ เราจะพัฒนาอัลกอลิธึมพื้นฐานเพื่อใช้ประมวลผลจุดเวลาและช่วงเวลาที่เก็บในโครงสร้างข้อมูลตารางเวลา

5. อัลกอริทึมสำหรับโครงสร้างข้อมูลตารางเวลา

อัลกอลิธึมพื้นฐานบางส่วนที่พัฒนาขึ้นมีตั้งแต่ 1) การค้นหาจุดเวลาที่เก็บอยู่ในตารางเวลา 2) การเพิ่มจุดเวลาและช่วงเวลาลงในตารางเวลา และ 3) การลบจุดเวลาและช่วงเวลาในตารางเวลา

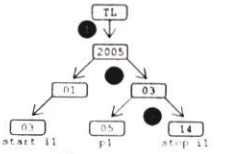
นับจากนี้ เราจะใช้จุดเวลาในรูปแบบชีวิตประจำวัน ซึ่งอ้างจุดเวลาโดยการนับหน่วยเวลา ปี เดือน วัน ฯลฯ ตามลำดับ

5.1 การค้นหาจุดเวลาที่เก็บในตารางเวลา

การค้นหาจุดเวลาที่เก็บอยู่ในตารางเวลา อาจเป็นได้ใน 2 ลักษณะคือ 1) การนำเอาค่าจุดเวลาไปค้นหาว่ามีจุดเวลานี้เก็บไว้ในตารางเวลาหรือไม่ 2) การนำเอาค่าจุดเวลาไปค้นหาว่าจุดเวลานี้เป็นจุดเริ่มต้นหรือจุดสิ้นสุดของช่วงเวลาใดที่เก็บในตารางเวลาหรือไม่ ต่อไปเป็นการนำเสนออัลกอลิธึมเพื่อการค้นหาดังกล่าว

อัลกอลิธึมนี้รับจุดเวลาที่ระบุด้วยหน่วยเวลาหลายหน่วยเรียงลำดับตามหน่วยใหญ่หน่วยเล็ก การค้นหาจุดเวลานี้เริ่มจากนำค่าตัวเลขของหน่วยปี (หน่วยใหญ่สุด) เปรียบเทียบกับโหนดลูกของ root ว่าพบค่าหน่วยปีที่โหนดลูกหรือไม่ ถ้าไม่พบก็แสดงว่าไม่มีจุดเวลานั้นเก็บใน Tree และยุติการค้นหา แต่ถ้าพบก็ให้นำค่าตัวเลขหน่วยถัดไปของจุดเวลาไปเปรียบเทียบกับโหนดลูกของโหนดที่ค้นพบ และทำซ้ำเดิมอีกเรื่อยๆ จนกระทั่งหมดหน่วยเวลาที่ใช้ค้นหา แล้วให้ผลลัพธ์เป็นชื่อจุดเวลาที่บันทึกไว้ที่โหนดหรือ leaf สุดท้ายที่ค้นพบค่าหน่วยเวลาสุดท้ายออกมา

ต่อไปเป็นรูปแสดงตัวอย่างการค้นหาจุดเวลาตามอัลกอลิธึมนี้เมื่อต้องการค้นหาจุดเวลาด้วยจุดเวลาวันที่ 14 เดือน 3 ปี 2005 เริ่มด้วยการเปรียบเทียบหน่วยปีที่ 2005 กับโหนดลูกของ root ซึ่งพบว่ามีโหนดปีที่ 2005 อยู่จึงนำเอาค่าหน่วยเดือนมาเปรียบเทียบกับโหนดลูกของโหนด 2005 พบว่ามีโหนดเดือนที่ 3 เช่นกัน จึงมีการตรวจสอบต่อไปอีกในหน่วยวัน แล้วสุดท้ายพบว่าวันที่ 14 เป็นค่าหน่วยเวลาสุดท้ายแล้ว จึงให้ผลลัพธ์เป็นชื่อจุดเวลาที่บันทึกไว้ที่โหนดวันที่ 14 ออกไป



รูปที่ 7 แสดงขั้นตอนการค้นหาจุดเวลา

รายละเอียดอัลกอลิธึมแสดงได้ดังนี้

```
procedure whatHappenAt(tree, tp):
  root = getroot(tree)
  searchUnit = head(tp)
  if searchUnit is empty:
    return getTPNames(root)
  else:
    children = getAllChildrenOf(root)
    child = find(children, searchUnit)
    if child not found:
      return 'timepoint not found'
    else:
      return whatHappenAt(subtree rooted at child,
                                          tail(tp))
```

รูปที่ 8 แสดงอัลกอลิธึมสำหรับการค้นหาจุดเวลา

อัลกอลิธึมนี้ จะรับอินพุทเป็น Tree ตารางเวลา และจุดเวลา tp ที่อยู่ในรูปของลิสต์ของค่าหน่วยเวลา โดยอัลกอลิธึมนี้จะอ่านค่าหน่วยเวลาออกมาที่ละหน่วยเวลาด้วยฟังก์ชั่น head ซึ่งค่าที่ได้ถูกนำไปค้นหาโหนดในลิสต์ของโหนดลูกของ root ถ้าไม่พบก็จะแจ้งว่าไม่พบจุดเวลาที่ต้องการค้นหา แต่ถ้าพบ ก็จะเรียกอัลกอลิธึมเดิมอีก โดยส่ง subtree ที่มี root เป็นโหนดที่ค้นพบ และลิสต์ของหน่วยเวลาที่เหลือ tail(tp) ไปเป็นพารามิเตอร์ ซึ่งจะกระทำเช่นนี้ไปเรื่อยๆ จนค่าหน่วยเวลาหมดลิสต์ จึงให้ผลลัพท์เป็นชื่อจุดเวลาที่บันทึกอยู่บนโหนดสุดท้ายที่ค้นพบออกมา

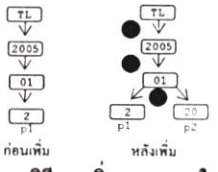
5.2 การเพิ่มจุดเวลาและช่วงเวลาลงในตารางเวลา

การเพิ่มจุดเวลาใดๆ กระทำได้โดยการสร้าง path เพื่อแทนจุดเวลาที่ต้องการเพิ่มเข้าไป โดยการเปรียบเทียบหน่วยจุดเวลาใหญ่สุดกับโหนดลูกของ root ลงไปทีละชั้นที่โหนดลูกลำดับต่อมาตามลำดับจนถึงหน่วยสุดท้าย และจะบันทึกชื่อจุดเวลานั้นไว้ที่โหนดหรือ leaf ที่หน่วยย่อยที่สุดไปสิ้นสุด

อัลกอลิธึมนี้รับอินพุทเป็น Tree ตารางเวลา จุดเวลา tp และชื่อจุดเวลา tpname โดยจะอ่านค่าหน่วยเวลาออกมาที่ละหน่วยเวลาด้วยฟังก์ชั่น head ซึ่งค่าที่ได้ถูกนำไปเทียบกับสมาชิกในลิสต์ของโหนดลูกทั้งหมดของ root ถ้าไม่พบก็จะสร้างโหนด

นั้นขึ้นมา แล้วเรียกใช้อัลกอลิธึมเดิมอีกครั้ง โดยส่ง subtree ที่มี root เป็นโหนดที่ค้นพบ ลิสต์ชุดค่าหน่วยเวลาที่เหลือ tail(tp) และชื่อจุดเวลาที่จะบันทึกลงใน Tree ไปเป็นพารามิเตอร์ ซึ่งจะกระทำเช่นนี้ไปเรื่อยๆ จนกระทั่งหมดค่าหน่วยเวลาของจุดเวลาที่ต้องการบันทึก แล้วจึงบันทึกชื่อจุดเวลานั้นไว้ที่โหนดสุดท้าย (ซึ่งอาจจะเป็น leaf ก็ได้) ที่หน่วยเวลาย่อยสุดท้ายไปสิ้นสุด

ต่อไปเป็นตัวอย่างการเพิ่มจุดเวลา p2 = 20/1/2005 ลงใน Tree ด้านซ้าย ตามอัลกอลิธึม ซึ่งจะได้ Tree ด้านขวา ดังรูป



รูปที่ 9 แสดงวิธีการเพิ่มจุดเวลาลงใน Tree

ส่วนการเพิ่มช่วงเวลา กระทำได้โดยการเพิ่มจุดเริ่มต้น และจุดสิ้นสุดของช่วงเวลาเข้าไปใน Tree โดยข้อมูลที่บันทึกไว้ที่ leaf เป็นชื่อจุดเริ่มต้นและชื่อจุดสิ้นสุดของช่วงเวลาตามลำดับ

รายละเอียดของอัลกอลิธึมแสดงได้ดังนี้

```
procedure insertTPoint(tree, tp, tpname):
  root = getroot(tree)
  searchUnit = head(tp)
  if searchUnit is empty:
    return insertTPName(root, tpname)
  else:
    children = getAllChildrenOf(root)
    child = find(children, searchUnit)
    if child not found:
      child = add child for root and sort
    return insertTPoint(subtree rooted at child,
                        tail(tp), tpname)
```

รูปที่ 10 แสดงอัลกอลิธึมสำหรับการเพิ่มจุดเวลา

5.3 การลบจุดเวลาและช่วงเวลาในตารางเวลา

การลบจุดเวลาใดๆ กระทำได้โดยการลบชื่อจุดเวลาออกจากโหนดหรือ leaf ที่บันทึกจุดเวลานั้นอยู่ โดย leaf ใดที่ถูกลบชื่อจุดเวลาที่เก็บอยู่ไปทั้งหมด leaf นั้นก็จะถูกลบออกจาก Tree ด้วย และจะกระทำซ้ำๆ เช่นนี้กับโหนดที่ระดับสูงขึ้นไปตามลำดับ

อัลกอลิธึมการลบจุดเวลารับอินพุทเป็น Tree ตารางเวลาและจุดเวลา tp พร้อมชื่อจุดเวลาที่ต้องการลบ tpname อัลกอลิธึมนี้มีลักษณะคล้ายกับการเพิ่มจุดเวลาต่างกันเพียงการลบจุดเวลาเป็นการลบชื่อของจุดเวลาออกจากโหนด(หรือ leaf) และอาจลบโหนดที่ปราศจากชื่อเวลาออกจาก Tree ด้วย รายละเอียดของอัลกอลิธึมแสดงได้ดังนี้

```
procedure removeTPoint(tree, tp, tpname):
  root = getroot(tree)
  searchUnit = head(tp)
  if searchUnit is empty:
    return deleteTPName(root, tpname)
  else:
    children = getAllChildrenOf(root)
    child = find(children, searchUnit)
    if child not found:
      return 'timepoint not found'
    else:
      removeTPoint(subtree rooted at child,
                                tail(tp), tpname)
      if child is a leaf and contain no name:
        remove child from its root
```

รูปที่ 11 แสดงอัลกอลิธึมสำหรับการลบจุดเวลา

ส่วนการลบช่วงเวลาใดๆ กระทำได้โดยลบจุดเวลาเริ่มต้นและจุดเวลาสิ้นสุดออกจากตารางเวลา

5.4 การวิเคราะห์คุณสมบัติของโครงสร้างข้อมูลตารางเวลา

กำหนดให้ k คือจำนวนค่าหน่วยเวลาที่ใช้ระบุจุดเวลา (หรือกล่าวอีกอย่างหนึ่งว่าเป็น จำนวนโหนดโดยเฉลี่ยในแต่ละ path ที่แทนจุดเวลา) และ n คือจำนวนจุดเวลาที่ถูกเก็บ โครงสร้างข้อมูลตารางเวลาที่ออกแบบขึ้นมามีคุณสมบัติดังนี้

1. ลำดับของการเพิ่มจุดเวลาไม่มีผลต่อโครงสร้างข้อมูล

2. โครงสร้างข้อมูลสามารถเก็บจุดเวลาและช่วงเวลา ที่มีความละเอียดของหน่วยเวลาที่แตกต่างกันได้ และสามารถแทนเวลาในหน่วยความละเอียดที่ไม่จำกัด

3. การเพิ่มและลบข้อมูลเข้าไปในโครงสร้างข้อมูลใช้การเปรียบเทียบอยู่ในชั้นของ $O(k)$

4. การค้นหาช่วงเวลาโดยใช้จุดเวลานั้น ในกรณีที่แย่ที่สุดใช้การเปรียบเทียบไม่เกินความสูงของ Tree หรือ $O(\log kn)$

5. พื้นที่เก็บข้อมูลสามารถแบ่งได้เป็น 2 ส่วน คือ โหนดที่เป็นโครงสร้างหลัก และชื่อจุดเวลาที่บันทึกไว้ที่โหนดหรือ leaf โดยโครงสร้างหลักใช้พื้นที่ขึ้นอยู่กับลักษณะจุดเวลาที่จัดเก็บ เช่น ถ้าตารางเวลาประกอบด้วยจุดเวลาที่มีลำดับที่ของปีและเดือนต่างกันมาก จะทำให้โหนดปีที่และโหนดเดือนที่ ไม่มีการใช้ร่วมกัน ทำให้มีจำนวนโหนดมาก ดังนั้นในกรณีที่แย่ที่สุด (ไม่มีการใช้โหนดร่วมกันเลย) จะต้องใช้พื้นที่เพื่อเก็บโครงสร้างหลักประมาณ kn ส่วนชื่อจุดเวลาที่บันทึกไว้ที่โหนดและ leaf จะใช้พื้นที่ตามจำนวนจุดเวลาที่เก็บ หรือ $O(n)$

6. การประยุกต์ใช้งานโครงสร้างข้อมูลตารางเวลาใน CL

โครงสร้างข้อมูลตารางเวลาที่ได้นำเสนอไปสามารถนำไปประยุกต์ใช้กับโปรแกรมที่เกี่ยวข้องกับการจัดการเวลา เช่น ออแกไนเซอร์ส่วนบุคคล ระบบทำงานอัตโนมัติตามตารางเวลา ในบทความนี้ได้มีการนำโครงสร้างข้อมูลนี้ไปใช้กับกับอินเตอร์พรีตเตอร์ภาษา CL [1]

6.1 ภาษา CL (Communication Language)

ภาษา CL ออกแบบมาเพื่อใช้งานเป็นภาษาคอมพิวเตอร์ทั่วไป แต่มีความพิเศษที่ผู้เขียนโปรแกรมสามารถใช้ CL สั่งงานคอมพิวเตอร์ให้ทำงานตามกำหนดเวลาและการเกิดขึ้นของเหตุการณ์ได้

ในภาษา CL ผู้เขียนโปรแกรมสามารถระบุช่วงเวลาให้กับคำสั่งในรูปแบบของ $[p_s,p_e]$:statement ซึ่งหมายถึงต้องการให้อินเตอร์พรีตเตอร์ภาษา CL ประมวลผลคำสั่งในช่วงเวลา $[p_s,p_e)$ (เพื่อให้ง่ายต่อการเขียนโปรแกรมใน CL เราจะเขียนโค้ดช่วงเวลาว่า $[p_s,p_e]$ เพื่อให้หมายถึงช่วงเวลา $[p_s,p_e)$)

ส่วนการโปรแกรมให้คอมพิวเตอร์กระทำคำสั่ง 'action' เพื่อตอบสนองต่อเหตุการณ์ 'event' จะอยู่ในรูปแบบของ $[p_s,p_e]$:event–>action ซึ่งหมายถึงว่า เมื่อมี event เกิดขึ้นในช่วงเวลา $[p_s,p_e)$ แล้วให้คอมพิวเตอร์ตอบสนองด้วยการประมวลผลคำสั่งที่ระบุในส่วนของ action ซึ่งเป็น CL statement ใดๆ

การระบุช่วงเวลาให้กับแต่ละคำสั่งและแต่ละเหตุการณ์นั้นมีได้ทั้งหมด 4 รูปแบบคือ (1) แบบระบุเวลา $[p_s,p_e]$:statement ระบุจุดเวลาเริ่มต้นและสิ้นสุดที่แน่นอนให้ statement ทำงาน (2) แบบกึ่งระบุเวลา $[\_,p_e]$:statement หรือ $[p_s,\_]$:statement ระบุจุดเริ่มต้นหรือจุดสิ้นสุดให้ statement ทำงาน แต่ไม่ระบุจุดเวลาที่เหลือ (3) แบบไม่ระบุเวลา [_,_]:statement ไม่มีการระบุเวลาทั้งจุดเริ่มต้นและจุดสิ้นสุดของการทำงาน แต่ปล่อยให้จุดเวลาทั้งสองถูกกำหนดโดยระบบขณะทำคำสั่ง ทำให้คำสั่งแบบนี้เหมือนกับคำสั่งในภาษาคอมพิวเตอร์ทั่วไป ผู้เขียนโปรแกรมสามารถเขียนโค้ดเพียง statement เพื่อให้หมายถึง [_,_]:statement ได้

นอกจากนี้ในภาษา CL ยังสามารถจัดกลุ่มคำสั่งรวมกันเป็นชุดคำสั่ง (Compound statement) ได้ โดยได้ออกแบบประเภทของชุดคำสั่งเพื่อเลียนแบบการทำงานของมนุษย์ ประกอบด้วย

- ชุดคำสั่งแบบเรียงลำดับ อินเตอร์พรีตเตอร์ CL จะทำงานตามคำสั่งย่อยที่ละคำสั่งอย่างเป็นลำดับ โดยคำสั่งถัดไปจะทำงานได้ เมื่อคำสั่งก่อนหน้าได้ทำงานเสร็จสิ้นแล้วเท่านั้น มีรูปแบบเป็น

  $[s_1,e_1]$:statement$_1$
  $[s_2,e_2]$:statement$_2$...

- ชุดคำสั่งแบบไม่มีลำดับ เป็นชุดคำสั่งที่ไม่ได้ให้ความสำคัญต่อลำดับการทำงานของแต่ละคำสั่งย่อย โดยคำสั่งย่อยใดจะทำงานก่อนหรือหลังก็ได้ แต่ให้ทำได้ครั้งละหนึ่งคำสั่งเท่านั้น ในแต่ละคำสั่งย่อยจะนำหน้าด้วย # และไม่ระบุเวลาเริ่มต้นและสิ้นสุด มีรูปแบบเป็น

  #[_,_]:statement$_1$
  #[_,_]:statement$_2$...

- ชุดคำสั่งแบบขนาน เป็นชุดคำสั่งที่แต่ละคำสั่งจะทำงานไปพร้อมๆกันอย่างอิสระมีรูปแบบเป็น

  \\[s_1,e_1]:statement$_1$
  \\[s_2,e_2]:statement$_2$...

ทั้ง 3 แบบชุดคำสั่งยังสามารถใช้ร่วมกันในแบบ nested ได้ด้วย

6.2 วิธีประมวลผลคำสั่งแบบเดิมของอินเตอร์พรีตเตอร์ CL

วิธีประมวลผลคำสั่งแบบเดิมของอินเตอร์พรีตเตอร์ CL [1,2] จะประมวลผลคำสั่งทีละคำสั่งจากบนลงล่าง (ดูโค้ดโปรแกรมข้างล่างประกอบ ซึ่งเขียนด้วยภาษา Python) ในการประมวลผลคำสั่ง $[p_s,p_e]$:statement หนึ่งๆ อินเตอร์พรีตเตอร์จะหยุดรอจนถึงจุดเวลา p_s แล้วจึงประมวลผล statement หลังจากนั้นจะรอจนกระทั่งถึงจุดเวลา p_e จึงประมวลผลคำสั่งต่อไป ซึ่งวิธีนี้ใช้ได้ดีกับชุดคำสั่งแบบเรียงลำดับ และชุดคำสั่งแบบไม่มีลำดับ แต่จะมีปัญหาเกิดขึ้นในกรณีการประมวลผลชุดคำสั่งแบบขนานซึ่งอินเตอร์พรีตเตอร์ CL จะต้องสร้างเธรดขึ้นมาประมวลผลพร้อมๆ กันตามจำนวนคำสั่งย่อยที่อยู่ในชุดคำสั่งแบบขนาน ซึ่งเธรดที่สร้างขึ้นมานั้น จะไม่ทำงานทันที แต่จะหยุดรอตามเวลา p_s ที่อยู่ในแต่ละคำสั่งย่อยๆ นั้น ยิ่งมีจำนวนคำสั่งย่อยมาก ก็จะทำให้มีการสร้างเธรดขึ้นมารอเป็นจำนวนมากด้วย

ดังนั้นเพื่อแก้ปัญหานี้จึงได้นำโครงสร้างข้อมูลตารางเวลาที่นำเสนอข้างต้นมาใช้เป็นตารางเวลาของชุดคำสั่งสำหรับการประมวลผลของอินเตอร์พรีตเตอร์ ซึ่งวิธีการเดิมไม่ได้ใช้ตารางเวลาสำหรับการนี้

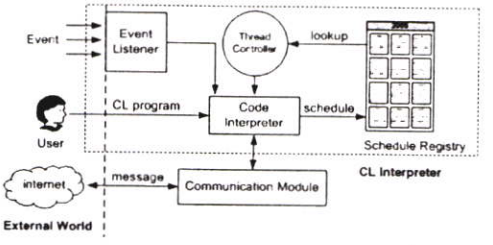
```
def main():
  program = read(ProgramFile)
  error = checkSyntax&Parser(program)
  if error != 0 : print(error)
  else : cl_interpreter(program)
def cl_interpreter([element1, …, elementn]):
  for elementi in [element1, …, elementn] :
  if (elementType(elementi) == "function_def"):
    define_function(elementi)
  elif (elementType(elementi) == "event_listen"):
    createThread(eventListener(elementi))
  else: execute_interval_statement(elementi)
def execute_interval_statement([Ps,Pe]:statement) :
  if (Pe != "_"):
    Pe = expr_eval(Pe)
    if (Ps != "_"):
      Ps' = expr_eval(Ps)
      wait(Ps' - getCurrentTime())
      createThread(execute_statement(statement))
      wait(Pe' - getCurrentTime())
      if (process_stack != []):
        clear(process_stack)
        exit()
  else:
    if (Ps != "_"):
      Ps' = expr_eval(Ps)
      wait(Ps' - getCurrentTime())
      execute_statement(statement)
def execute_statement(statement):
  statement.status = RUNNING
  try:
    if statementType(statement) == "funct-call" :
      exec(statement)
    if statementType(statement) == "sequence" :
      sequence execute subStatement in statement:
        execute_interval_statement(subStatement)
    if statementType(statement) == "unordered" :
      random execute subStatement in statement:
        execute_interval_statement(subStatement)
    if statementType(statement) == "parallel" :
      for subStatement in statement:
        createThread(
          execute_interval_statement(
            subStatement)))
    if statementType(statement) == "if-stmt":
      "IF" (cond): then-stmt = statement
      if expr_eval(cond):
        execute_interval_statement(then-stmt)
    if statementType(statement) == "while-stmt" :
      "WHILE" (cond) : while-stmt = statement
      while expr_eval(cond):
        execute_interval_statement(while-stmt)
    statement.status = SUCCESS
  exeception:
    statement.status = FAIL
```

รูปที่ 12 แสดงโค้ดเดิมของอินเตอร์พรีตเตอร์ CL

6.3 การปรับปรุงวิธีประมวลผลคำสั่งของอินเตอร์พรีตเตอร์ CL

เราได้ปรับปรุงวิธีการประมวลผลชุดคำสั่งแบบขนานของอินเตอร์พรีตเตอร์ CL จากเดิมที่มีการสร้างเธรดขึ้นมารอไว้ตามจำนวนคำสั่งย่อย โดยเปลี่ยนมาเป็นการนำคำสั่งย่อยเหล่านั้นไปบันทึกไว้ในตารางเวลา (Schedule Registry) แล้วคำสั่งย่อยๆ เหล่านั้นจะถูกประมวลผลในภายหลังโดยส่วนควบคุมการสร้างเธรด (Thread Controller) ของอินเตอร์พรีตเตอร์ ซึ่งจะช่วยลดจำนวนเธรดที่สร้างขึ้นมารอได้ ส่วนการประมวลผลชุดคำสั่งประเภทอื่นรวมทั้งคำสั่งที่ทำงานตามเหตุการณ์ภายนอกวิธีการเดิมนั้นยังใช้งานได้ดี

คำสั่งแต่ละคำสั่งที่ถูกนำมาเก็บไว้ในตารางเวลาจะถูกส่วนควบคุมการสร้างเธรดตรวจสอบทุกๆ 1 วินาที เมื่อใดก็ตามที่พบว่ามีคำสั่งรอการประมวลผลอยู่ มันก็จะสร้างเธรดสำหรับประมวลผลคำสั่งนั้นทันที ดังนั้นสถาปัตยากรรมของอินเตอร์พรีตเตอร์ CL ที่ปรับปรุงสามารถแสดงได้ดังนี้



รูปที่ 13 แสดงสถาปัตยากรรมของอินเตอร์พรีตเตอร์ CL ที่ปรับปรุงใหม่

อินเตอร์พรีตเตอร์ที่ปรับปรุงมีความแตกต่างกับอินเตอร์พรีตเตอร์เดิม (ดูโค้ดในรูปที่ 12) เพียงนิยามของฟังก์ชั่น execute_statement เท่านั้น ซึ่งนิยามที่ปรับปรุงใหม่ของฟังก์ชั่น execute _statement จะแสดงในรูปที่ 14

การทำงานของอินเตอร์พรีตเตอร์ที่ปรับปรุงใหม่ (ดูโค้ดทั้งในรูปที่ 12 และ 14 ประกอบ) เริ่มจากได้รับชุดคำสั่ง CL มาจากผู้ใช้ อินเตอร์พรีตเตอร์ที่ปรับปรุงก็จะประมวลผลไปที่ละคำสั่ง ถ้าเป็นนิยามฟังก์ชั่นมันจะเก็บไว้ในหน่วยความจำ แต่ถ้าเป็นการสั่งงานด้วยเหตุการณ์ มันจะสร้างตัวจัดการเหตุการณ์ (Event Listener) ขึ้นมารอติดตามฟังเหตุการณ์ที่ระบุไว้ ส่วนคำสั่งประเภท statement มันจะเรียกใช้ฟังก์ชั่น execute_interval_statement มาประมวลผลคำสั่งนั้น ฟังก์ชัน execute_interval_statement จะประมวลผลคำสั่งในรูป $[p_s,p_e]$:statement โดยอ่านค่าเวลาปัจจุบันขึ้นมาเปรียบเทียบกับจุดเวลา p_s จากนั้นจะคำนวณหาระยะเวลาที่จะต้องหยุดรอ แล้วรอจนครบกำหนดเวลา จากนั้นฟังก์ชั่นนี้ก็จะสร้างเธรดของฟังก์ชั่น execute_statement เพื่อเริ่มประมวลผลคำสั่ง แล้วหยุดรออีกครั้งจนกระทั่งถึงจุดเวลา p_e จากนั้นจึงมีการ

ตรวจสอบว่าเธรดที่สร้างขึ้นมาสามารถประมวลผลได้สำเร็จหรือไม่ ถ้าไม่ก็จะหยุดการประมวลผลคำสั่งและแจ้งข้อผิดพลาดออกมา

การประมวลผล statement ใดๆ จะใช้ฟังก์ชั่น execute_statement ซึ่งจะเปลี่ยนสถานะของ statement ที่เริ่มทำงานเป็น RUNNING และถ้าการประมวลผลสำเร็จมันก็จะเปลี่ยนสถานะ statement นั้นเป็น SUCCESS แต่ถ้าระหว่างทำคำสั่งเกิด exception ขึ้นก็จะเปลี่ยนสถานะ statement นั้นเป็น FAIL

```
def execute_statement(statement):
  statement.status = RUNNING
  try:
    if statementType(statement) == "funct-call" :
      exec(statement)
    if statementType(statement) == "sequence" :
      sequence execute subStatement in statement:
        execute_interval_statement(subStatement)
    if statementType(statement) == "unordered" :
      random execute subStatement in statement:
        execute_interval_statement(subStatement)
    if statementType(statement) == "parallel" :
      for subStatement in statement:
        [Ps,Pe]:stmt = subStatement
        stmt.status = WAITING
                    # เราให้ root ใช้แทน sub tree ที่มี root เป็นรูทโหนด
        insertTPoint (root,Ps,start(stmt))
        insertTPoint (root,Pe,stop(stmt))
    if statementType(statement) == "if-stmt":
      "IF" (cond): then-stmt = statement
      if expr_eval(cond):
        execute_interval_statement(then-stmt)
    if statementType(statement) == "while-stmt" :
      "WHILE" (cond) : while-stmt = statement
      while expr_eval(cond):
        execute_interval_statement(while-stmt)
    statement.status = SUCCESS
  exeception:
    statement.status = FAIL
```

รูปที่ 14 แสดงการประมวลผลคำสั่ง

ฟังก์ชั่น execute_statement ประมวลผล statement แยกตามประเภท ถ้าเป็นประเภทที่เรียกใช้ฟังก์ชั่น (function call) จะถูกประมวลผล statement นั้นทันที แต่ถ้าเป็นชุดคำสั่งแบบเรียงลำดับ จะประมวลผลคำสั่งย่อยทีละคำสั่งจนหมด ส่วนชุดคำสั่งแบบไม่มีลำดับ จะเลือกคำสั่งย่อยมาประมวลผลทีละคำสั่งจนหมด **และในกรณีที่เป็นชุดคำสั่งแบบขนาน จะนำคำสั่งนั้นไปเก็บไว้ในตารางเวลาก่อน** พร้อมกับกำหนดค่าสถานะของคำสั่งนั้นเป็น WAITING เพื่อรอให้ส่วนควบคุมการสร้างเธรดมาเรียกไปประมวลผลต่อไป **การประมวลผลชุดคำสั่งแบบขนานนี้เป็นส่วนที่ได้ทำการปรับปรุงใหม่ จึงได้แสดงเป็นตัวอักษรทึบในโค้ดโปรแกรมรูปที่ 14**

นอกจากจะได้ปรับปรุงโค้ดฟังก์ชั่น execute_statement แล้ว เรายังได้เพิ่มเติมส่วนควบคุมการสร้างเธรด (Thread Controller) เข้ามาในอินเตอร์พรีตเตอร์ปรับปรุงใหม่ด้วย ซึ่งนิยามเป็นคลาสดังแสดงในรูปที่ 15

การทำงานของส่วนควบคุมการสร้างเธรดนี้จะทำงานเป็นเธรดอิสระจากอินเตอร์พรีตเตอร์ โดยคอยตรวจสอบกำหนดการทำคำสั่งที่ถูกบันทึกไว้ในตารางเวลาก่อนหน้านี้ เมื่อพบว่ามีคำสั่งใดถึงกำหนดทำงาน มันก็จะสร้างเธรดขึ้นมาประมวลผลคำสั่งนั้น หรือถ้ามีคำสั่งใดหมดเวลาการทำงานตามที่ถูกระบุในโค้ด แต่ยังกำลังทำงานอยู่ มันก็จะแจ้งข้อผิดพลาดออกมาในรูปของ execpetion ให้กับผู้ใช้

```
class ThreadController(Thread):
  def run ():
    runningList = empty list
    while (true):
      currentTime = getCurrentTime()
      occurs = whatHappenAt(root, currenTime)
      for occur in occurs:
        if occur is 'start(stmt)':
          runningList.append(stmt)
          createThread(execute_statement(stmt))
        if occur is 'stop(stmt)':
          runningList.remove(stmt)
          if stmt.status == RUNNING:
            raise Expired statement is running
```

รูปที่ 15 แสดงส่วนควบคุมการสร้างเธรด

7. ผลการทดลอง

เพื่อวัดประสิทธิภาพการประมวลผลคำสั่งโดยอินเตอร์พรีตเตอร์ CL ที่ปรับปรุง เราได้ใส่อัลกอลิธึมการประมวลผลคำสั่งแบบใหม่ลงในอินเตอร์พรีตเตอร์ CL เดิมแล้วทำการเปรียบเทียบการทำงานกับอินเตอร์พรีตเตอร์ตัวเดิม โดยได้ทดลองรันอินเตอร์พรีตเตอร์ทั้งสองกับตัวอย่างชุดคำสั่ง CL ซึ่งประกอบด้วยชุดคำสั่งแบบเรียงลำดับ ขนาน และไม่มีลำดับ ดังนี้

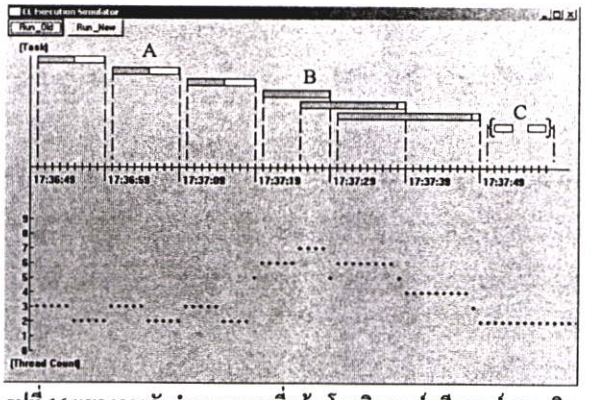
```
currentTime=getCurrentTime()
[currentTime,currentTime+10]: doAction(5)
[currentTime+11,currentTime+20]: doAction(5)
[currentTime+21,currentTime+30]: doAction(5)
\\[currentTime+31,currentTime+40]: doAction(9)
\\[currentTime+36,currentTime+50]: doAction(13)
\\[currentTime+41,currentTime+60]: doAction(18)
[currentTime+61,currentTime+70]:
  #[_,_]: doAction(5)
  #[_,_]: doAction(5)
```

ชุดคำสั่ง CL นี้ทำงานโดยเรียกใช้ฟังก์ชั่น doAction() ซึ่งเป็นการหน่วงเวลาตามหน่วยวินาที

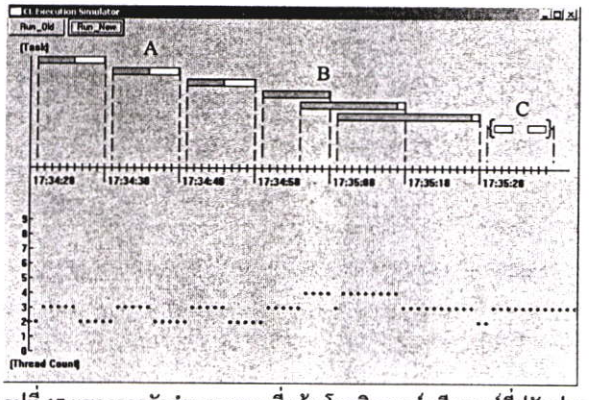
การประมวลผลของอินเตอร์พรีตเตอร์ทั้งสองกับตัวอย่างชุดคำสั่งนี้จะถูกวิเคราะห์และแสดงผลออกมาในรูปของบาร์

ชาร์ท (Barchart) ด้านล่าง โดยให้แต่ละบาร์แทนช่วงเวลา $[p_s,p_e]$ ที่ระบุไว้ที่แต่ละคำสั่ง ส่วนช่วงเวลาที่อินเตอร์พรีตเตอร์ใช้ในการประมวลผลคำสั่งนั้นจริง จะแสดงด้วยแถบสีเทาที่อยู่ช่วงต้นของแต่ละบาร์ ส่วนบาร์ที่ตำแหน่ง C ในรูปจะแทนช่วงเวลาของการประมวลผลชุดคำสั่งที่ไม่มีลำดับ ซึ่งสามารถระบุได้เฉพาะจุดเริ่มต้นและสิ้นสุดของทั้งชุดคำสั่งเท่านั้น

เพื่อตรวจสอบจำนวนของเธรดที่สร้างขึ้นสำหรับการประมวลผลคำสั่งของอินเตอร์พรีตเตอร์ทั้งสอง เราสามารถแสดงให้เห็นได้จากกราฟใต้บาร์ชาร์ท กราฟที่ว่านี้ใช้แสดงจำนวนเธรดที่เกิดขึ้นที่สัมพันธ์กับลำดับการประมวลผลชุดคำสั่งโดยอินเตอร์พรีตเตอร์ โดยแกนตั้งแสดงจำนวนเธรดที่เกิดขึ้นในระหว่างการประมวลผลคำสั่งของบาร์ด้านบน และในแกนนอนจะแทนเวลาที่ดำเนินไป



รูปที่ 16 แสดงการวัดจำนวนเธรดที่สร้างโดยอินเตอร์พรีตเตอร์ CL เดิม



รูปที่ 17 แสดงการวัดจำนวนเธรดที่สร้างโดยอินเตอร์พรีตเตอร์ที่ปรับปรุง

จากการทดลองเปรียบเทียบระหว่างการทำงานของอินเตอร์พรีตเตอร์เดิมและที่ปรับปรุงใหม่ โดยได้ทดลองรันกับตัวอย่างชุดคำสั่ง CL ข้างต้น ให้ผลแสดงได้ดัง รูปที่ 16 และ 17

ผลการทดลองนี้แสดงให้เห็นว่าอินเตอร์พรีตเตอร์เดิมและที่ปรับปรุง ไม่มีความแตกต่างของจำนวนเธรดที่สร้างขึ้นสำหรับการประมวลผลชุดคำสั่งแบบเรียงลำดับและแบบไม่มีลำดับ ตัวอย่างเช่น ที่ช่วงเวลาระบุโดย A ซึ่งเป็นการประมวลผลชุดคำสั่งแบบเรียงลำดับ อินเตอร์พรีตเตอร์ทั้ง 2 จะสร้างเธรดจำนวน 3 เท่ากัน ส่วนกรณีการประมวลผลชุดคำสั่งแบบขนาน อินเตอร์พรีตเตอร์เดิมมีการสร้างจำนวนเธรดมากกว่าของอินเตอร์พรีตเตอร์ที่ปรับปรุง เช่น ที่ช่วงเวลาระบุโดย B ซึ่งเป็นการประมวลผลชุดคำสั่งแบบขนาน อินเตอร์พรีตเตอร์เดิมจะสร้างเธรดจำนวน 7 ขณะที่อินเตอร์พรีตเตอร์ที่ปรับปรุงสร้างเธรดเพียง 4 เธรด

ดังนั้นสรุปได้ว่า การใช้ตารางเวลาสำหรับการควบคุมการประมวลผลคำสั่งตามเวลาในอินเตอร์พรีตเตอร์ CL จะช่วยลดจำนวนเธรดในการประมวลผลคำสั่งแบบขนาน เป็นการเพิ่มประสิทธิภาพการประมวลผลคำสั่งให้กับอินเตอร์พรีตเตอร์ CL อย่างมาก

8. บทสรุป

งานวิจัยนี้ได้ศึกษาลักษณะนามธรรมของเวลาและตารางเวลา แล้วนำเสนอโครงสร้างข้อมูลตารางเวลาสำหรับใช้ประมวลผลจุดเวลาและช่วงเวลาที่เรียงลำดับกัน โครงสร้างข้อมูลดังกล่าวได้นำไปใช้ปรับปรุงประสิทธิภาพการประมวลผลคำสั่งของอินเตอร์พรีตเตอร์ภาษา CL ให้ดียิ่งขึ้น

9. เอกสารอ้างอิง

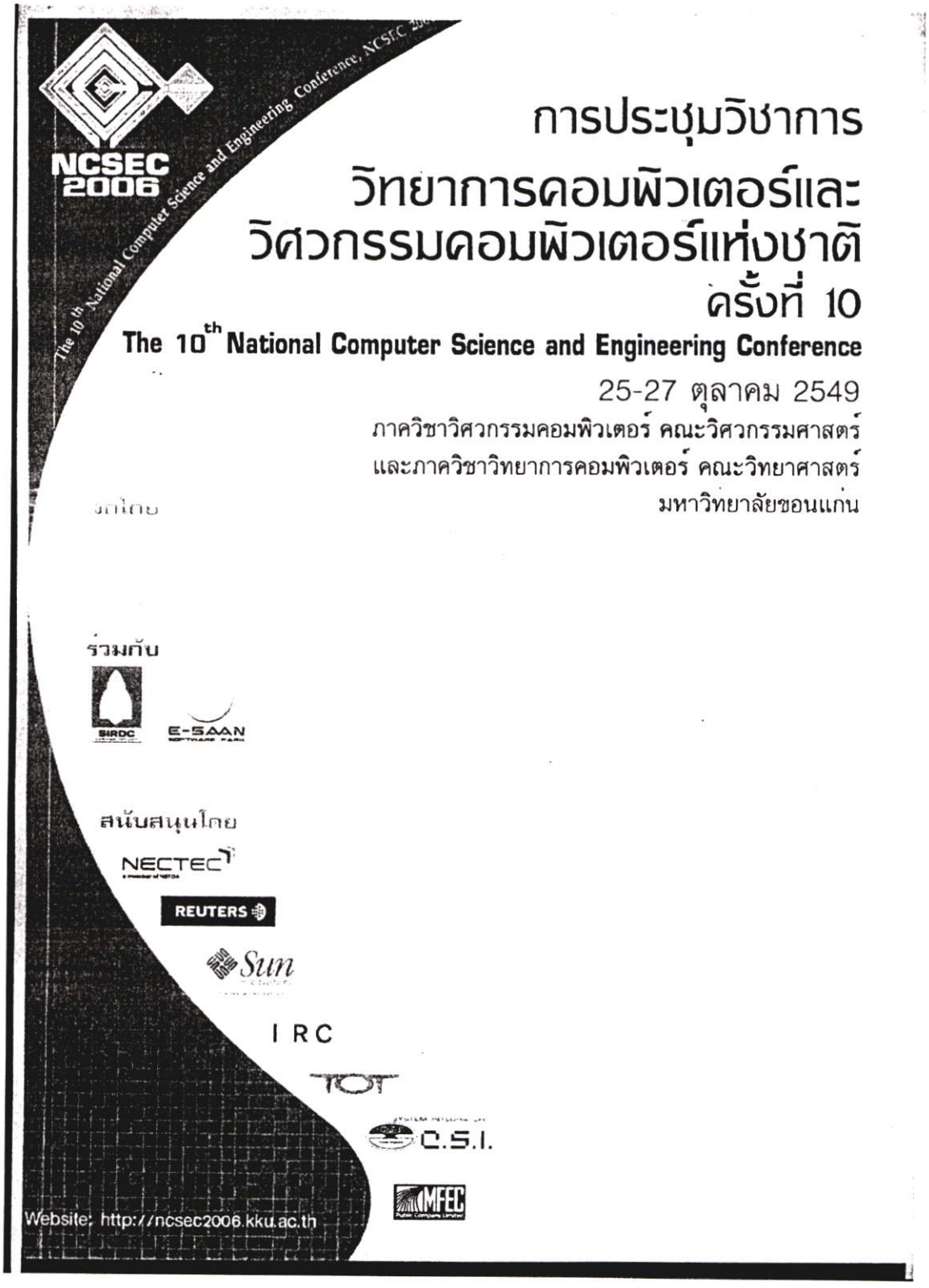
[1] วิศิษฏ์ หิรัญกิตติ และสุพัฒณคา โชติพันธ์ "CL: ภาษาสำหรับการสั่งงานคอมพิวเตอร์ด้วยเวลาและเหตุการณ์", การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 7 NCSEC2003, 2546.

[2] สุพัฒณคา โชติพันธ์ "CL: ภาษาสำหรับการสั่งงานคอมพิวเตอร์ด้วยเวลาและเหตุการณ์", วิทยานิพนธ์ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีเจ้าคุณทหารลาดกระบัง, 2547.

[3] Allen J., "Maintaining knowledge about temporal intervals", Communications of ACM 26(11), 1983, pp.832-843.

[4] Allen J., "Action and Event in Interval Temporal Logic", Journal of Logic and Computation, Special Issue on Actions and Processes, 1994

[5] Kawalski R. and Sergot M., "A Logic-based Calculus of Events", New Generation Computing,4,1986,pp. 67-95


NCSEC 2006
The 10th National Computer Science and Engineering Conference, NCSEC 2006
การประชุมวิชาการ
วิทยาการคอมพิวเตอร์และ
วิศวกรรมคอมพิวเตอร์แห่งชาติ
ครั้งที่ 10
The 10th National Computer Science and Engineering Conference
25-27 ตุลาคม 2549
ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์
และภาควิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์
มหาวิทยาลัยขอนแก่น
ร่วมกับ
SIRDC
E-SAAN
สนับสนุนโดย
NECTEC
REUTERS
Sun
IRC
TOT
C.S.I.
MFEC
Website: http://ncsec2006.kku.ac.th

การประมวลผลตารางการทำงานของโปรแกรมภาษา CL ที่กำหนดเวลาแปรเปลี่ยนได้

Interpretation of CL Program Schedules Governed by Varying Time

วิศิษฏ์ หิรัญกิตติ และสุรชัย ล้อเจริญ
ห้องวิจัยการสื่อสารและคมนาคมชาญฉลาด ภาควิชาวิศวกรรมคอมพิวเตอร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง กรุงเทพ 10520 ประเทศไทย
E-mail: v.hirankitti@gmail.com, kan2005@gmail.com

บทคัดย่อ

CL เป็นภาษาคอมพิวเตอร์แนวใหม่ที่ใช้สั่งงานคอมพิวเตอร์ให้ทำงานตามกำหนดเวลาและตามการเกิดของเหตุการณ์หนึ่งๆ CL ที่ได้ออกแบบไว้ตาม [1,2,3,4] ยังมีข้อจำกัดที่ไม่สามารถรองรับประโยคคำสั่งตามกำหนดเวลาที่แปรผันไปตามค่าตัวแปรได้ ซึ่งการแก้ไขข้อจำกัดนี้จะทำให้การทำงานตามคำสั่ง CL มีความยืดหยุ่นมากขึ้น คือ สามารถทำงานเป็นคาบๆ ซ้ำๆ และทำงานตามกำหนดเวลาที่ถูกกำหนดได้จากผลการทำคำสั่งที่มีมาก่อนหน้าได้ ดังนั้นในบทความนี้จึงขอเสนอวิธีการปรับปรุงภาษา CL โดยใช้วิธีการใหม่ในการแทนโครงสร้างข้อมูลภายในที่ใช้เก็บลำดับการทำงานของโปรแกรม CL เพื่อให้รองรับกำหนดเวลาของคำสั่งที่ระบุโดยตัวแปร รวมทั้งได้ปรับปรุงอินเตอร์พรีตเตอร์เพื่อการประมวลผลคำสั่งดังกล่าว

Abstract

CL is a novel computer language which is used to instruct a computer to interpret commands according to certain schedules and events. The CL, as that designed and proposed in [1,2,3,4], still has one limitation that it cannot support statements governed by time which varies by values of variables. Having lifted this limitation, the improved CL can now handle more flexible statements which are programmed to run according to a periodic schedule and/or to run according to the schedule dynamically determined by some outcome of the interpretation of some previous statements. Therefore, in this paper we shall propose a way to overcome such limitation of the current CL by proposing an extension of its internal data structure for storing the program schedules which contain statements governed by time variables. Additionally, we also improve the CL interpreter to cope with the interpretation of this kind of statements.

Keyword: Agent Programming Language, Real-time Programming

1. บทนำ

การโปรแกรมคอมพิวเตอร์ ให้สามารถทำกิจกรรมต่างๆ ตามตารางเวลา และเหตุการณ์หนึ่งๆ ได้นั้น จะเป็นประโยชน์อย่างมาก เพราะเป็นการลดภาระในการทำงานซ้ำซาก เพื่อที่เราจะได้มีเวลามากขึ้นในการทำงานที่อาศัยความคิดสร้างสรรค์ ด้วยความต้องการดังกล่าวได้นำไปสู่การพัฒนาเว็บบราวเซอร์ที่สามารถโปรแกรมกิจกรรมการสื่อสารได้ [1] และเอเจนต์สื่อสารที่ชาญฉลาด [3]

อาจเรียกได้ว่า CL เป็นภาษาเดียวที่ได้พัฒนาขึ้นมาสำหรับการสั่งงานคอมพิวเตอร์ตามกำหนดเวลาและเหตุการณ์ [2,5] อย่างไรก็ตามพบว่ามีงานวิจัยที่เกี่ยวข้อง ได้แก่ [6, 7] พยายามออกแบบสถาปัตยกรรมเอเจนต์ที่สามารถสื่อสารและทำงานร่วมกับมนุษย์ โดยเอเจนต์สามารถทำงานตามตารางเวลาที่ตนสร้างขึ้นมาเอง เพื่อให้สามารถทำงานได้บรรลุเป้าหมายตามที่มนุษย์สั่งการ ซึ่งพบว่าการทำงานของเอเจนต์ในลักษณะนี้แตกต่างจากการทำงานของเอเจนต์ภาษา CL ซึ่งจะถูกโปรแกรม

ด้วยตารางการทำงานโดยมนุษย์ เพื่อให้ทำคำสั่งโดยไม่จำเป็นต้องสร้างตารางเวลาขึ้นเอง เนื่องจากเป็นปัญหาที่ซับซ้อนเกินไป ดังนั้นวิธีนี้จะใช้ได้กับเฉพาะลักษณะงานบางอย่างเท่านั้น ไม่สามารถใช้กับงานทั่วไปดังเช่นภาษา CL ได้

ในส่วนของภาษา CL ที่ได้ออกแบบไว้แล้วตาม [1, 2, 3, 4, 5] นั้นยังมีข้อจำกัดที่ไม่สามารถรองรับประโยคคำสั่งตามกำหนดเวลาที่แปรเปลี่ยนไปตามค่าตัวแปรได้ ซึ่งการแก้ไขข้อจำกัดนี้จะทำให้การทำงานตามคำสั่ง CL มีความยืดหยุ่นมากขึ้น คือ สามารถทำงานเป็นคาบๆ ซ้ำๆ และทำงานตามกำหนดเวลาที่ถูกกำหนดจากผลการทำคำสั่งที่มีมาก่อนได้ ดังนั้นในบทความนี้จึงเสนอวิธีการปรับปรุง CL โดยใช้วิธีการใหม่ในการแทนโครงสร้างข้อมูลภายในที่ใช้เก็บลำดับการทำงานของโปรแกรม CL ให้รองรับกำหนดเวลาของคำสั่งที่ระบุด้วยตัวแปรได้ รวมทั้งได้พัฒนาอินเตอร์พรีตเตอร์เพื่อประมวลผลคำสั่งดังกล่าว

สำหรับเนื้อหาในบทความเราจะกล่าวย้อนถึงรูปแบบภาษา CL ที่ได้เสนอไว้ใน [1, 2, 3, 5] ในหัวข้อที่ 2 จากนั้นเราจะกล่าวถึงโครงสร้างข้อมูลที่ใช้เก็บตารางเวลาของภาษา CL ในงานวิจัยเดิมในหัวข้อที่ 3 แล้วเป็นขั้นตอนการทำงานของอินเตอร์พรีตเตอร์เดิมในหัวข้อที่ 4 และข้อจำกัดของภาษา CL ปัจจุบันในหัวข้อ 5 แล้วจะเสนอวิธีการแก้ไขข้อจำกัดโดยการพิจารณาถึงปัญหาที่เกิดจากการกำหนดจุดเวลาเริ่มต้นและสิ้นสุดที่เปลี่ยนค่าได้ของประโยคคำสั่งทั้งในแบบที่เปลี่ยนตามรอบเวลาและแบบที่เปลี่ยนค่าตามผลของคำสั่งก่อนหน้าในหัวข้อ 6 ส่วนอินเตอร์พรีตเตอร์ที่ปรับปรุงใหม่จะนำเสนอในหัวข้อ 7 แล้วเป็นผลการทดลองในส่วนอินเตอร์พรีตเตอร์ในหัวข้อที่ 8 และวิเคราะห์ผลการวิจัยในหัวข้อที่ 9 ตามด้วยบทสรุป

2. รูปแบบภาษา CL

ใน CL ผู้เขียนโปรแกรมสามารถระบุช่วงเวลาให้กับคำสั่งในรูปแบบ $[p_s,p_e]$:statement ซึ่งหมายถึง ต้องการให้ statement ทำงานตั้งแต่จุดเวลา p_s แล้วเสร็จไม่เกิน p_e โดยที่รูปแบบของจุดเวลาคือ HH:MM:SS|dd/mm/yyyy เช่น 7:00:00|12/12/2549 หมายถึง ปีที่ 2549 เดือนที่ 12 วันที่ 12 และชั่วโมงที่ 7 เป็นต้น

ส่วนการโปรแกรมให้คอมพิวเตอร์ทำคำสั่งตอบสนองต่อเหตุการณ์จะอยู่ในรูป $[p_{s1},p_{e1}]$:event → $[p_{s2},p_{e2}]$:statement ซึ่งหมายถึง เมื่อมี event เกิดขึ้นในช่วงเวลาตั้งแต่ p_{s1} ถึง p_{e2} แล้วให้ตอบสนองด้วยการทำ statement ตั้งแต่ p_{s2} แล้วเสร็จไม่เกิน p_{e2}

- **การระบุช่วงเวลา $[p_s,p_e]$ ใน $[p_s,p_e]$:statement ใน CL** มีได้ 3 รูปแบบ

1. $[p_s,p_e]$ หมายถึงจุดเวลาเริ่มต้นและสิ้นสุดมีการระบุค่า
2. $[\_,p_e]$ หรือ $[p_s,\_]$ หมายถึงจุดเวลาเริ่มต้นหรือสิ้นสุดไม่มีการระบุค่า
3. $[\_,\_]$ เป็นการไม่ระบุเวลาทั้งจุดเริ่มต้นและสิ้นสุด แต่ปล่อยให้จุดเวลาทั้งสองกำหนดโดยระบบขณะประมวลผล

- **Statement แบบต่างๆในภาษา CL** มี 5 ประเภท ดังนี้

1. ประเภทใช้กำหนดค่าให้กับตัวแปรมีรูปแบบ

```
<variable-name> = <expression>
```

2. ประเภทใช้เรียกฟังก์ชั่น มีรูปแบบ

```
<function-name>(<argument>…)
```

3. ประเภท Event-action เป็น statement ที่รอการเกิดขึ้นของเหตุการณ์ แล้วลงมือทำคำสั่ง มีรูปแบบ

```
event → statement
```

4. ประเภทควบคุมทิศทางโฟล์วการทำงานของโปรแกรม ได้แก่

- if-statement มีรูปแบบ

```
if <expression>: <statementA>
else: <statementB>
```

- while-statement มีรูปแบบ

```
while <expression>: statement
```

- for-statement มีรูปแบบ

```
for <control-variable> in <list> : <statement>
```

- repeat-until-statement มีรูปแบบ

```
repeat : <statement>
until <expression>
```

5. ประเภทชุด statement ซึ่งเกิดจากการรวมเอาหลายๆ statement ไว้ด้วยกัน แบ่งเป็น

- Sequential statement เป็นชุด statement ที่ต้องทำงานเรียงลำดับทีละอันจากบนลงล่าง โดย statement หลังจะเริ่มทำงานได้ก็ต่อเมื่อ statement ก่อนหน้าทำงานเสร็จสิ้นแล้ว มีรูปแบบ

```
sequential:
        [s1,e1] : statement1
        [s2,e2] : statement2...
```

ในกรณีที่เรากำหนดจุดเวลาเริ่มต้นหรือสิ้นสุดแบบไม่ระบุให้กับประโยคในชุดคำสั่ง sequential ได้แก่ `[_,_]:statement` จะทำให้ '_' ที่เป็นจุดเวลาเริ่มต้นหมายถึง "the time after the execution of the previous statement" และ '_' ที่เป็นจุดเวลาสิ้นสุดหมายถึง "the time after the execution of this statement is finished" ทำให้ [_,_]:statement มีความหมายเหมือนกับคำสั่ง statement ในภาษาคอมพิวเตอร์ทั่วไป ที่ไม่มีการระบุเวลาในการทำคำสั่ง ต่อจากนี้สำหรับประโยค sequential บางครั้งเราอาจใช้สัญลักษณ์ 'af' แทน '_' ตัวหน้า และ 'f' แทน '_' ตัวหลัง เพื่อให้สื่อความหมายที่ชัดเจน

- Unordered statement เป็นชุด statement ที่ลำดับการทำงานของ statement ย่อย ไม่มีการระบุแน่นอน ส่วนการทำงานของกลุ่ม statement นี้ จะทำได้ทีละหนึ่งเท่านั้น มีรูปแบบ

```
unordered:
        [_,_]:statement1
        [_,_]:statement2...
```

- Parallel statement เป็นชุด statement ที่แต่ละ statement ย่อยทำงานไปพร้อมๆ กันได้ มีรูปแบบ

```
parallel:
        [s1,e1]:statement1
        [s2,e2]:statement2...
```

นอกจากนี้ ชุดคำสั่งทั้ง 3 ยังสามารถใช้ผสมกันในลักษณะ nested ได้ ต่อไปนี้เป็นตัวอย่างโปรแกรมภาษา CL อย่างง่าย

ตัวอย่างที่ 1 สมมติว่าเราต้องการสั่งให้คอมพิวเตอร์เล่นเพลงในช่วงเวลา 20 ถึง 21 น. จากนั้นเปิด browser แสดงหน้าเว็บข่าวในช่วง 22 ถึง 23 น. และปิดเครื่องตอน 2 น. ตลอดเวลาตั้งแต่ 22 น. จนถึงวันรุ่งขึ้นให้รอรับรายงานผลการทำประตูของคู่แข่งขันฟุตบอลโลกทางอีเมล์ เมื่อได้รับแล้วให้นำผลไปอัพเดทในตารางผลการแข่งขันบนเว็บไซต์ www.mylivefootball.com ดังนั้นโปแกรม CL อาจเขียนได้ดังนี้

```
[22:0:0|23/6/2005, 1:0:0|24/6/2005]:                    (s1)
  MAIL_AVAILABLE("Live Report") →
    content = readMail()
    update_website(www.mylivefootball.com, content)
[20:0:0|23/6/2005, 21:0:0|23/6/2005]:                   (s2)
  play_music("song.mp3")
[22:0:0|23/6/2005, 23:0:0|23/6/2005]:                   (s3)
  browserto(www.news.co.th)
[2:0:0|24/6/2005,_]: shutdown()                         (s4)
```

ในหัวข้อต่อไปเราจะขออธิบายโครงสร้างข้อมูลที่ใช้เก็บโปรแกรม CL เพื่อการประมวลผลของ CL อินเตอร์พรีตเตอร์

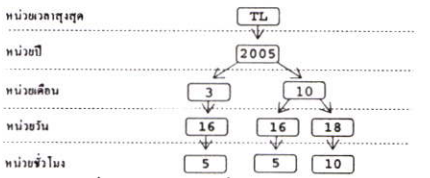
3. โครงสร้างข้อมูลสำหรับเก็บตารางเวลาโปรแกรม CL

จากความจริงที่ว่า จุดเวลาถูกระบุโดยลำดับการนับหน่วยเวลาหลายๆ หน่วย โดยนำลำดับหน่วยต่างๆ มาเรียงจากเล็กไปหาใหญ่ เช่น จุดเวลา 5:30:10|16/3/2005 หมายถึง วินาทีที่ 10 นาทีที่ 30 ชั่วโมงที่ 5 วันที่ 16 เดือนที่ 3 ปีที่ 2005 จะพบว่าจุดเวลาสามารถกำหนดได้ด้วยหน่วยเวลาที่มีความละเอียดแตกต่างกัน ดังนั้นจุดเวลาที่เก็บในโครงสร้างข้อมูลจะต้องสามารถระบุโดยหน่วยเวลาที่ระดับความละเอียดต่างๆ กันได้หลายๆหน่วย ในที่นี้เราได้ใช้โครงสร้าง Tree [4] ในการเก็บจุดเวลาโดยวิธีดังนี้

- ให้ระดับของโหนดแทนหน่วยเวลา โดยระดับบนสุดเป็นหน่วยสูงสุด ระดับรองลงมาเป็นหน่วยย่อยที่แบ่งย่อยจากหน่วยก่อนหน้ามันลดหลั่นกันไปตามลำดับ
- โหนดแต่ละโหนดแทนลำดับของหน่วยที่จะใช้อ้างในจุดเวลา ยกเว้น root ที่อยู่ที่ระดับหน่วยสูงสุด มีเพียงลำดับเดียว จึงขอแทนด้วย TL เพื่อให้สื่อถึงหน่วยเวลาที่ใหญ่ที่สุด คือ Time Line ซึ่งครอบคลุมเวลาทั้งหมด ทั้งอดีตปัจจุบันและอนาคต กลุ่มโหนดที่ระดับเดียวกันจะต้องเรียงลำดับ โดยโหนดค่าน้อยอยู่ทางซ้าย และโหนดค่ามากอยู่ทางขวา
- การไล่เรียงโหนดจากระดับบนมายังโหนดลูกตามลำดับหน่วยที่ระบุในจุดเวลานั้นๆ เป็นการระบุถึงจุดเวลาที่สมบูรณ์ที่ถูกบันทึกไว้ในโครงสร้างข้อมูล

สรุปได้ว่าจุดเวลาแต่ละจุดจะแทนด้วย path จาก root ไปถึง leaf ดังนั้นเพื่อความสะดวกเราจะให้ leaf มีความหมายแทน path ที่มาสิ้นสุดที่ leaf นั้นทำให้ leaf มีความหมายแทนจุดเวลาที่ระบุโดย path ที่มาสิ้นสุดที่ leaf นั้น

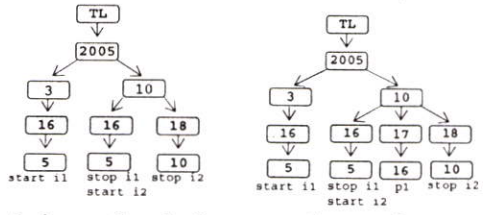
สมมติว่าเราจะเก็บจุดเวลา 3 จุด 16/3/2005 เวลา 5 น., 16/10/2005 เวลา 5 น., 18/10/2005 เวลา 10 น. ในโครงสร้างข้อมูล Tree ที่ได้จะเป็นดังรูป



รูปที่ 1 แสดง Tree ที่เก็บ 3 จุดเวลา

สำหรับการบันทึกช่วงเวลาใน Tree นั้น ทำได้โดยบันทึกทั้งจุดเวลาเริ่มต้นและจุดเวลาสิ้นสุดลงใน Tree พร้อมระบุด้วยว่า

จุดใดเป็นจุดเริ่มต้นและจุดใดเป็นจุดสิ้นสุดของช่วงเวลานั้น ดังตัวอย่าง การบันทึกช่วงเวลา i1=[16/3/2005 เวลา 5 น., 16/10/2005 เวลา 5 น.] และ i2=[16/10/2005 เวลา 5 น., 18/10/2005 เวลา 10 น.] ใน Tree จะได้ Tree ดังรูป (ก)



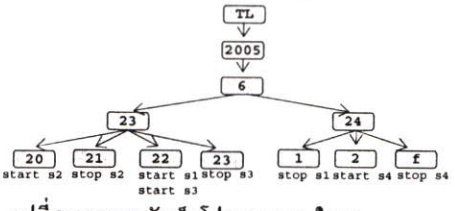
ก . เก็บช่วงเวลาเพียงอย่างเดียว ข. เก็บจุดและช่วงเวลา

รูปที่ 2 แสดง Tree

รูป (ข) เป็นตัวอย่าง Tree ที่เก็บทั้งจุดและช่วงเวลาไว้ด้วยกัน คือ จุดเวลา p1 ที่ 17/10/2005 เวลา 16 น. และช่วงเวลา i1 = [16/3/2005 เวลา 5 น., 16/10/2005 เวลา 5 น.] และ i2 = [16/10/2005 เวลา 5 น., 18/10/2005 เวลา 10 น.]

ดังนั้นประโยค [p_s, p_e]:statement เมื่อเก็บไว้ใน Tree จะเหมือนกับการเก็บช่วงเวลา โดยเราบันทึก start<statement> และ stop <statement> ไว้ที่จุดเวลาเริ่มต้นและสิ้นสุด ตามลำดับ

ตัวอย่างที่ 2 โปรแกรม CL ในตัวอย่างที่ 1 ถูกเก็บใน Tree ดังรูป



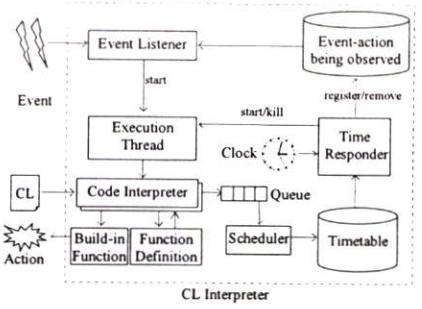
รูปที่ 3 แสดงการจัดเก็บโปรแกรม CL ใน Tree

สังเกตว่าที่ประโยค s4 ในโปรแกรมช่วงเวลาจะเป็นแบบไม่ระบุเวลาสิ้นสุด จึงหมายถึง 'f' ซึ่งก็จะถูกเก็บใน Tree ดังรูปที่ 3 ในการบันทึก [p_s,p_e]:statement ใน Tree นั้น ถ้า p_s และ/หรือ p_e เป็นจุดเวลาแบบไม่ระบุ ("_") เราจะสร้างโหนดแล้วเก็บค่า "af" สำหรับ p_s และ/หรือ สร้างโหนดแล้วเก็บค่า "f" สำหรับ p_e ซึ่งในกรณีนี้เราเก็บค่า "f" ไว้ที่กิ่งขวาสุดของ Tree และบันทึก statement สำหรับช่วงเวลาดังกล่าว

4. โครงสร้างอินเตอร์พรีตเตอร์ CL

Statement ในโปรแกรม CL ที่เก็บใน Tree จะถูกนำมาประมวลผลตามกำหนดเวลาและเงื่อนไขของเหตุการณ์โดย CL อินเตอร์พรีตเตอร์ ซึ่งมีส่วนประกอบต่างๆ แสดงได้ดังรูป

การทำงานเริ่มต้นจากส่วน Time Responder, Event Listener และ Scheduler จะทำงานเป็นเธรดอิสระ จากนั้น Code Interpreter อ่านประโยคจากโปรแกรม CL เข้าไปตีความหมาย ถ้าเป็นการนิยามฟังก์ชั่น ซึ่งจะยังไม่ทำงาน ให้ส่งไปเก็บไว้ในส่วน Function Definition ก่อน (โดยจะถูกเรียกใช้ในภายหลังเมื่อมี function call เกิดขึ้น) ส่วนโปรแกรมที่ top-level (ชุด statement จะถูกมองเป็นเพียง statement เดียว) จะถูกจัดเก็บเข้าในคิว เพื่อรอให้ Scheduler บันทึกลงใน Tree



รูปที่ 4 แสดงสถาปัตยกรรมของอินเตอร์พรีตเตอร์ CL

หลังจาก statement/event ถูกบันทึกใน Tree แล้ว จากนั้นจะมีส่วน Time Responder ที่อาศัยตัวแปรชื่อ `time_cursor` (ซึ่งทำหน้าที่เก็บจุดเวลาสิ้นสุดของ statement/event ล่าสุดที่เพิ่งทำงานแล้วเสร็จหรือจุดเวลาเริ่มต้นของ statement/event ล่าสุดที่ได้ทำงานแล้วแต่ยังไม่เสร็จ) ในการอ้างอิงสำหรับ การอ่านจุดเวลาเริ่มต้นของ statement/event ที่กำลังจะต้องทำงานจาก Tree เพื่อสร้างเธรดที่จะประมวลผลมัน หรือ การอ่านจุดเวลาสิ้นสุดของ statement/event ที่กำลังทำงานอยู่ เพื่อใช้เป็นจุดเวลาสำหรับการทำลายเธรดที่กำลังประมวลผล statement/event นั้นเมื่อถึงเส้นตายเวลาดังกล่าว

ประโยชน์ของการใช้ `time_cursor` เพื่อทำให้ Time Responder ไม่ต้องพิจารณาจุดเริ่มต้น/สิ้นสุดของแต่ละ statement ใน Tree ที่ได้ผ่านพ้นเวลาของ `time_cursor` ไปแล้ว ซึ่ง Time Responder จะอ่านได้ค่าจุดเวลาอย่างใดออกมาจาก Tree ขึ้นอยู่กับว่าค่าใดอยู่เป็นลำดับต่อจากค่าจุดเวลาของ

time_cursor ซึ่งค่านี้เมื่ออ่านออกมาแล้วจะเก็บไว้ในตัวแปร next_point_to_reach ที่ใช้เก็บจุดเวลาของสิ่งที่จะเกิดขึ้นต่อไป

ในทุกขณะ Time Responder จะทำการอ่านค่าเวลาปัจจุบันจาก Clock ที่มีความละเอียดสูง มาเทียบกับค่าของ next_point_ to_reach ณ ขณะนั้น แล้วพิจารณาว่า

กรณีที่ 1 ถ้าค่าตรงกับเวลาปัจจุบัน ให้กำหนด time_cursor เป็นค่า next_point_to_reach แล้วทำการพิจารณาว่าค่าเวลาดังกล่าวว่าเป็น

- จุดเวลา start <statement> ก็จะเรียกให้ส่วน Execution Thread สร้างเธรดเพื่อประมวลผล statement ได้ทันที
- จุดเวลา stop <statement> ก็จะเรียกให้ Execution Thread ทำลายเธรดที่ยังประมวลผล statement นี้ไม่แล้วเสร็จทันที
- จุดเวลา start <event> ก็จะบันทึก event-action statement ไว้ใน Event-action being observed
- จุดเวลา stop <event> ก็จะลบ event-action statement ออกจาก Event-action being observed

กรณีที่ 2 ถ้าค่าเวลาเป็น 'af' สำหรับ start <statement> แล้ว ก็จะต้องตรวจสอบว่า statement ก่อนหน้าซึ่งระบุโดยจุดเวลาสิ้นสุดที่เก็บอยู่ที่ time_cursor ได้ทำงานเสร็จสิ้นแล้วหรือยัง ถ้าเสร็จแล้วก็ให้กำหนด time_cursor เป็นค่า next_point_to_reach แล้วเรียกให้ Execution Thread สร้างเธรดเพื่อประมวลผล <statement> ทันที มิฉะนั้น Time Responder ก็จะกลับไปพิจารณาค่า next_point_to_reach เทียบกับค่าเวลาปัจจุบันอีกครั้ง

กรณีที่ 3 ถ้าค่าเวลาเป็น 'f' สำหรับ stop <statement> แล้วก็จะต้องตรวจสอบว่า statement นี้ได้ทำงานเสร็จสิ้นหรือยัง ถ้าเสร็จแล้วก็ให้กำหนด time_cursor เป็นค่า next_point_to_reach มิฉะนั้น Time Responder ก็จะกลับไปพิจารณาค่า next_point_to_reach เทียบกับค่าเวลาปัจจุบันอีกครั้ง

สำหรับส่วน Execution Thread ซึ่งเป็นส่วนประมวลผล statement จะถูกเรียกให้สร้างเธรดขึ้นมาเพื่อประมวลผลหรือให้ทำลาย เธรดโดย Time Responder (นอกจากนี้ส่วน Execution Thread ยังถูกเรียกใช้โดย Event Listener ด้วย)

ในส่วน Event Listener นั้นจะรอรับเหตุการณ์ต่างๆที่เกิดขึ้นมาพิจารณาตลอดเวลา แล้วนำมาเปรียบเทียบกับส่วน event ใน event-action statement ที่เก็บอยู่ใน Event-actions being observed ถ้าพบว่า event นี้เกิดขึ้นก็จะเรียกใช้ Execution Thread เพื่อสร้าง เธรด Code Interpreter ขึ้นมาประมวลผลส่วน action ทันที

Code Interpreter จะประมวลผล statement ใน 3 รูปแบบ คือ (1) ถ้าเป็นการเรียกใช้ function ที่เป็นแบบ build-in ก็จะสั่งให้ทำงานได้เลย (2) ถ้าเป็นนิยามฟังก์ชั่นก็จะจัดเก็บไว้ในส่วนของ Function Definition หรือ (3) ถ้าเป็นชุด statement ก็จะเอาประโยค $[p_s,p_e]$:statement ในชุด statement นั้นเก็บลงใน Queue แล้วรอให้ Scheduler จัดเก็บลงใน Tree ในลักษณะของ Customer-Producer เพื่อไม่ให้เสียเวลาในการประมวลผล โดยส่วนนี้จะเป็นส่วน Producer และ Scheduler จะเป็น Customer สำหรับ Pseudo code ของ CL Interpreter แสดงได้ดังข้างล่าง

Code Interpreter

```
def code_interpreter(statement):
  if statement is build-in function: execute statement
  elif statement is a 'function definition':
    // save function declaration in function definition
  elif statement is a 'sequential statement' or
       statement is a 'parallel statement':
    for stmt in sub statements: enqueue(queue, stmt)
  elif statement is a 'unordered statement':
    random stmt in sub statements: enqueue(queue, stmt)
```

Scheduler

```
def scheduler():
  whenever queue is not empty:
    [p_s,p_e]:statement = dequeue(queue)
    if p_s == '_': p_s = 'af'
    if p_e == '_': p_e = 'f'
    insertTPoint(time_tree, p_s, (start, statement))
    insertTPoint(time_tree, p_e, (stop, statement))

def insertTPoint(timepoint, symbol):
  global time_tree
  if timepoint is 'ABSOLUTE_TIME':
    /*insert path into time_tree to represent timepoint
    append symbol to created path */
  if timepoint is 'af' or 'f':
    /* append to the rightmost path of the time_tree
     append symbol to created path  */
```

Time Responder

```
def time_responder():
  for every 1 second:
    response_for(getCurrentTime())

def response_for(timepoint):
  global time_cursor, next_point_to_reach
  if next_point_to_reach == timepoint:
    time_cursor = next_point_to_reach
    next_point_to_reach = time_cursor.next()
    handle_start_stop()
  elif next_point_to_reach == 'af':
```

```
    if the previous statement was complete:
      time_cursor = next_point_to_reach
      next_point_to_reach = time_cursor.next()
      handle_start_stop()
  elif next_point_to_reach == 'f':
    if the current statement was complete:
      time_cursor = next_point_to_reach
      next_point_to_reach = time_cursor.next()
      handle_start_stop()
  else: return 'Not_Found'

def handle_start_stop()
  symbol_list = time_cursor.getSymbols()
  for (prefix, stmt) in symbol_list:
    if stmt is 'event-action statement':
      if prefix == 'start': observed.insert(stmt)
      else: observed.remove(stmt)
    else:
      if prefix == 'start':
        startThread(code_interpreter(stmt))
      else: stopThread(code_interpreter(stmt))
```

```
Event Listener
def event_listener():
  wait for event e:
    (event → action) = observed.lookup(e)
    startThread(code_interpret(action))
```

5. ข้อจำกัดของภาษา CL ในปัจจุบัน

ที่ผ่านมาเราได้กำหนดให้ $[p_s,p_e]$ ในภาษา CL เป็นค่าคงที่เท่านั้น ซึ่งมีผลทำให้ง่ายต่อการออกแบบอินเตอร์พรีตเตอร์ แต่กรณีที่จุดเวลาเริ่มต้นและสิ้นสุดใน $[p_s,p_e]$:statement ถูกกำหนดโดยตัวแปรที่มีค่าเปลี่ยนตามคาบเวลา เช่น ทุกๆ เดือน ทุกๆ วัน ตามประโยคทำซ้ำ เช่น for-statement รวมทั้งในกรณีที่ค่าตัวแปร p_s หรือ p_e ถูกกำหนดค่าโดยการทำงานของ statement ก่อนๆ แล้ว อินเตอร์พรีตเตอร์นี้จะไม่สามารถรองรับการประมวลผลได้

สำหรับวิธีการแก้ไขข้อจำกัดนี้ เราจะขอนำเสนอโดยเริ่มจากการเสนอโครงสร้าง Tree แบบใหม่ที่ใช้เก็บตารางการทำงานของโปรแกรม CL โดยดัดแปลงจากโครงสร้าง Tree แบบเดิม รวมทั้งเสนออินเตอร์พรีตเตอร์ตัวใหม่สำหรับการประมวลผลคำสั่งในตารางเวลาแบบที่ใช้ตัวแปรในการระบุช่วงเวลา

6. โครงสร้าง Tree แบบใหม่

มีการปรับปรุงโครงสร้าง Tree เดิมใน 3 ลักษณะดังนี้

6.1 Tree สำหรับเก็บตารางเวลาที่มีการทำงานเป็นคาบๆ

ในชีวิตประจำวันจะพบว่ามีการทำงานที่เกิดขึ้นประจำเป็นคาบๆ เช่น การจ่ายค่าบิล ณ วันที่ 1 ของทุกเดือน ซึ่งการทำงานในลักษณะนี้สามารถเขียนเป็นโปรแกรม CL ได้ดังนี้

ตัวอย่างที่ 3

```
[0:0:0|1/1/2006, 0:0:0|2/12/2006]:                (Sn)
  for month in (1,12):
  [0:0:0|1/month/2006,0:0:0|2/month/2006]:
     transfer_money("1000")
```

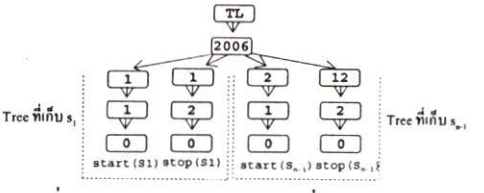
ซึ่งเป็นการสั่งให้คอมพิวเตอร์โอนเงินชำระบิล 1,000 บาท ณ วันที่ 1 และให้เสร็จภายในวันที่ 2 ของทุกเดือนตลอดปี 2006 พบว่ามีการนำเอาตัวแปรควบคุม month มาใช้ใน for-statement เพื่อเปลี่ยนค่าเดือนในแต่ละคาบการทำงาน ซึ่งแตกต่างจากภาษา CL เดิมที่ไม่สามารถใช้ตัวแปรกำหนดเวลาการทำงานได้

เราพบว่าตารางการทำงานซ้ำๆ ที่เป็น n คาบ สามารถแบ่งแยกออกโดยเรียงลำดับเป็นตารางการทำงานคาบที่ 1 ก่อน แล้วตามด้วยตารางการทำงานสำหรับอีก n-1 คาบที่เหลือ โดยวิธีนี้ทำให้สามารถออกแบบโครงสร้าง Tree ที่ใช้จัดเก็บตารางการทำงานซ้ำๆ ที่เป็น n คาบได้ โดยการแบ่งเป็น Tree ที่เก็บตารางการทำงานของคาบที่ 1 แล้วตามด้วย Tree ที่เก็บตารางการทำงานของ n-1 คาบที่เหลือ จากตัวอย่างที่ 3 ดังนั้น Tree ที่เก็บ S_n ดังรูปที่ 5 ประกอบด้วย Tree ที่เก็บ S_1 คือ

```
[0:0:0|1/1/2006,0:0:0|2/1/2006] transfer_money("1000")
```

แล้วเชื่อมต่อด้วย Tree ที่เก็บ S_{n-1} คือ

```
for month in (2,12):
 [0:0:0|01/month/2006, 0:0:0|2/month/2006]
   transfer_money("1000")
```



รูปที่ 5 Tree สำหรับเก็บตารางเวลาที่ทำงานเป็น n คาบ

6.2 Tree สำหรับเก็บตารางเวลาที่มีกำหนดเวลาเป็นตัวแปร

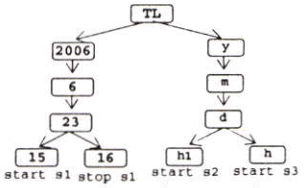
บางครั้งตารางเวลากิจกรรมบางอย่างอาจมีหมายกำหนดการที่ถูกกำหนดโดยค่าเวลาของผลการทำงานก่อนๆ ดังตัวอย่าง

ตัวอย่างที่ 4 เราต้องการสั่งให้คอมพิวเตอร์รอรับข้อความสั้นแจ้งเวลาในการสั่งอาหาร แล้วนำเวลาดังกล่าวมาเป็นเวลาในการสั่งอาหารทางอินเตอร์เน็ต ก่อนหน้าจะสั่งอาหาร 1 ชม.สั่งให้เครื่องล้างจานทำงาน ซึ่งอาจแสดงเป็นโปรแกรม CL ได้ดังนี้

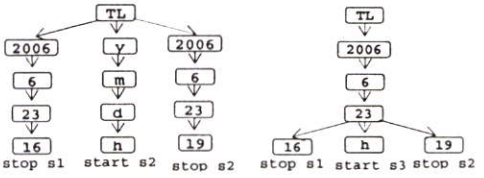
```
[15:00:00|23/6/2005, 18:00:00|23/6/2005]          (s1)
  SMS_AVAILABLE →
    message = getSMS()
    (h,d,m,y) = getOrderTime(message)
    h1 = h - 1
[h1:0:0|d/m/y,_] start_dish_cleaner()             (s2)
```

```
[h:0:0|d/m/y, _] orderFood(pizza)                    (s3)
```

โปรแกรมนี้มีการใช้ตัวแปรใน [p_s, p_e] ของตารางเวลา การจัดเก็บตารางเวลาลงใน Tree ก็เป็นเพียงการจัดเก็บตัวแปรลงใน โหนด start หรือ stop ของ Tree เพื่อรอการตีค่าดังรูป



รูปที่ 6 โครงสร้างตารางเวลาที่เก็บจุดเวลา ที่เป็นตัวแปร



(ก) Tree ก่อนการยุบโหนด (ข) Tree หลังการยุบโหนด

รูปที่ 7 การยุบโหนดใน Tree

บางครั้ง Tree ที่สร้างขึ้นอาจทำให้เล็กลงโดยลดความซ้ำซ้อนของโหนด เช่น กรณีที่กลุ่มโหนดในระดับเดียวกัน เมื่อมีโหนดใดอยู่ในระหว่าง 2 โหนดที่มีค่าเหมือนกัน จะสามารถถูกรวมเป็นโหนดเดียวได้โดยที่ความหมายไม่เปลี่ยน เช่น จาก Tree ในรูปที่ 7 จะเห็นว่ากลุ่มโหนดที่อยู่ระดับเดียวกันจะถูกยุบรวม

6.3 Tree ที่รองรับ scope

การที่เรายอมให้ใช้ตัวแปรใน [p_s, p_e] อาจเกิดกรณีที่มีการใช้ชื่อตัวแปรชื่อเดียวกันในหลายๆ nested statement ที่อยู่ต่างระดับ (scope) กัน ในกรณีเช่นนี้จะทำให้การตีความหมายตัวแปรผิดพลาดว่าเป็นตัวเดียวกัน ทั้งที่จริงแล้วควรถือว่าตัวแปรเหล่านี้เป็นต่างตัวแปรกัน ทำให้เราจำเป็นต้องสร้าง Tree ย่อยๆ ขึ้นมาสำหรับแต่ละ scope ซึ่งการมี scope นี้จะช่วยแก้ปัญหาในกรณีที่โปรแกรม CL มีชุดคำสั่งเป็นแบบ parallel ด้วย เช่น

ตัวอย่างที่ 5

```
parallel:

 [9:0:0|11/12/2006,12:0:0|11/12/2006]sequential:(s1)
  [9:0:0|11/12/2006,_]                              (s1.1)
   (h1:m1:s1|d1/M1/Y1) =
     getMeetingTime("www.john.com/schedule")
  [h1:m1:s1|d1/M1/Y1, 12:0:0|11/12/2006]            (s1.2)
    alert_me("It's meeting time.")

 [9:0:0|11/12/2006,12:0:0|11/12/2006]sequential (s2)
  [9:0:0|11/12/2006,_]                              (s2.1)
   (h2:m2:s2|d2/M2/Y2) =
```

```
     getLunchTime("www.somchai.com/timetable")
  [h2:m2:s2|d2/M2/Y2, 12:0:0|11/12/2006]            (s2.2)
    alert_me("It's lunch time.")
```

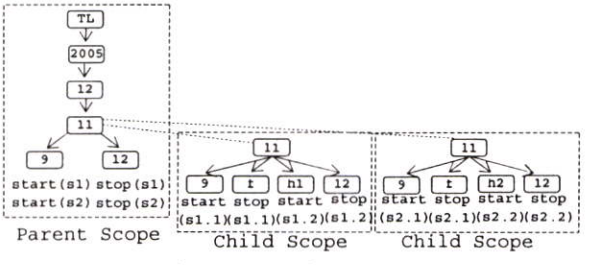
ซึ่งหมายถึง ในช่วง [9:00:00|11/12/2006, 12:00:00|11/12/2006] ให้คอมพิวเตอร์ไปดึงข้อมูลเวลาการประชุมจากเว็บไซต์ของ John แล้วให้แจ้งเตือนเรา ณ. เวลานั้น ขณะเดียวกันก็ให้ไปดึงข้อมูลเวลานัดหมายรับประทานอาหารเที่ยงแล้วให้แจ้งเตือนเราเมื่อถึงเวลานั้น จะพบว่าเหล่าตัวแปรใน h1:m1:s1|d1/M1/Y1 และใน h2:m2:s2|d2/M2/Y2 ที่ปรากฏในประโยค s1 และ s2 เมื่อโปรแกรมทำงาน เราจะไม่สามารถรู้ได้ว่าตัวแปรตัวใดมีค่ามากน้อยกว่ากัน จึงทำให้ s1 และ s2 ไม่สามารถบันทึกใน Tree ต้นเดียวกันได้ จึงต้องบันทึกไว้ในต่าง Tree กัน ซึ่งแต่ละ Tree ก็จะถือว่าเป็น scope สำหรับแต่ละกลุ่มตัวแปร

ดังนั้นเมื่อใดที่มี nested statement เกิดขึ้นก็จะมีการสร้าง Tree ย่อยต้นใหม่สำหรับเก็บชุดคำสั่งใน nested statement นั้น เพื่อให้มี scope เป็นเอกเทศแยกออกมา ในการประมวลผล nested statement ตัว Time Responder จะตรวจทุก statement ใน Tree ไปด้วยกัน แต่จะเลือกอันที่มี start ตรงกับเวลาปัจจุบันขึ้นมา พร้อมด้วยค่า scope ของมัน เพื่อส่งให้อินเตอร์พรีตเตอร์ไปประมวลผล ทำให้ไม่มีการสับสนเรื่อง scope

แต่ละ scope เป็นขอบเขตการใช้งานและการอ้างอิงตัวแปรทั้งหมดในชุด statement ระดับเดียวกัน ซึ่งก็คือ Tree ย่อย ที่เกิดจากการสร้างตารางเวลาการทำงานของชุด statement นั้น

เพื่อเป็นการแยก scope ย่อยออกจาก scope หลัก เราจะทำการสร้างลิงค์เชื่อม Tree ตารางเวลาของ Scope หลักไปยัง Tree ตารางเวลาของ Scope ย่อยๆ ตัวอย่างดังรูปที่ 9

ตัวอย่างที่ 6 จากตัวอย่างที่ 5 เมื่อเก็บใน Tree แยกเป็น scope หลัก และ scope ย่อย จะเป็นดังนี้



รูปที่ 9 แสดง Tree ที่มี nested scope

7. อินเตอร์พรีตเตอร์ CL ที่ปรับปรุงใหม่

มีการปรับปรุงส่วนต่างๆ ของอินเตอร์พรีตเตอร์เดิม โดยที่ส่วนที่เพิ่มเติมนั้นสังเกตได้จากในนิยามฟังก์ชั่นเราจะเขียนเป็นตัวหนา ดังต่อไปนี้

- ส่วน Code Interpreter

  - เพิ่มการประมวลผล statement ที่ทำซ้ำๆ ตามคาบเวลา

  - มีการกำหนด scope ให้กับการประมวลผลชุด statement ทำให้ scope ถูกสร้างใหม่ทุกครั้งที่มีการประมวลผลชุดคำสั่ง

ซึ่งผลการปรับปรุงได้เป็นดังนี้

Code Interpreter

```
def code_interpreter(scope, statement):
  if statement is build-in function:
    execute statement
  elif statement is a 'function definition':
    // save function declaration in function definition
  elif statement is 'for-loop statement':
    s1 = get the first iteration statement
    sn-1 = get the rest iteration statement
    enqueue(queue, (scope, s1))
    if sn-1 exist: enqueue(queue, (scope, sn-1))
  elif statement is a 'sequential statement' or
       statement is a 'parallel statement':
    create nested_scope
    for stmt in sub statements:
      enqueue(queue, (nested_scope, stmt))
    attachScope(scope, nested_scope)
  elif statement is a 'unordered statement':
    create nested_scope
    random select stmt in sub statements:
      enqueue(queue, (nested_scope, stmt))
    attachScope(scope, nested_scope)
```

- ส่วน Scheduler มีการปรับปรุงให้สามารถจัดเก็บ statement ไว้ใน time_tree ที่แยกตาม scope และสามารถเก็บตัวแปรใน Tree ที่เก็บตารางเวลาได้ ผลการปรับปรุงเป็นดังนี้

Scheduler

```
def scheduler():
  whenever queue is not empty:
    (scope, [p*,p*]:statement) = dequeue(queue)
    if p* == '_': p* = 'af'
    if p* == '_': p* = 'f'
    insertTPoint(scope, p*, (start, statement))
    insertTPoint(scope, p*, (stop , statement))

def insertTPoint(scope, timepoint, symbol):
  time_tree = scope.time_tree
  if timepoint is 'ABSOLUTE_TIME':
   /*insert path into time_tree to represent timepoint
   merge variable node if neighbor node is equal
   append symbol to the created path*/
  if timepoint is 'af' or 'f' or variable:
   /*append to the rightmost path of the time_tree
    append symbol to the created path*/
```

- ส่วน Time Responder มีการปรับปรุงให้ทำการเฝ้าสังเกต scope ทั้งหมด โดย statement ที่พบจะถูกประมวลผลโดยอ้างอิงกับ scope ของมัน ซึ่งได้ผลเป็นดังนี้

Time Responder

```
def time_responder():
  for every 1 second:
    response_for(rootScope, getCurrentTime())

def response_for(scope, timepoint):
  for nested_scope under scope:
    response_for(nested_scope, timepoint)
  if scope.next_point == timepoint:
    scope.time_cursor = scope.next_point
    scope.next_point = scope.time_cursor_next()
    handle_start_stop(scope)
  elif scope.next_point == 'af':
    if the previous statement was complete:
      scope.time_cursor = next_point
      scope.next_point = scope.time_cursor_next()
      handle_start_stop(scope)
  elif scope.next_point == 'f':
    if the current statement was complete:
      scope.time_cursor = next_point
      scope.next_point = scope.time_cursor_next()
      handle_start_stop()
  else: return 'Not_Found'

def handle_start_stop(scope)
  symbol_list = scope.time_cursor.getSymbols()
  for (prefix, stmt) in symbol_list:
    if stmt is 'event-action statement':
      if prefix == 'start':
        observed.insert ((scope, stmt))
      else: observed.remove((scope,stmt))
    else:
      if prefix == 'start':
        startThread(code_interpreter(scope,stmt))
      else:
        stopThread(code_interpreter(scope,stmt))
```

- ส่วน Event Listener มีการแก้ไขให้ประมวลผล action โดยอ้างอิงกับ scope ดังนี้

Event Listener

```
def event_listener():
  wait for event e:
    (scope,event→action) = observed.lookup(e)
    startThread(code_interpreter(scope, action))
```

8. ผลการทดลอง

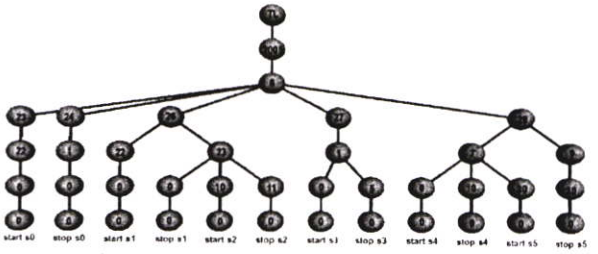
เราได้พัฒนาอินเตอร์พรีตเตอร์ที่ปรับปรุงใหม่ให้มีการแสดง Tree ที่เก็บตารางเวลาออกมาเป็นรูปภาพ 3 มิติในขณะที่รันโปรแกรม เพื่อให้เห็น Tree ของ scope ย่อยที่ถูกย่อยออกมาจาก Tree ที่เป็น scope หลัก

จากนั้นเมื่อนำอินเตอร์พรีตเตอร์นี้ไปรันกับตัวอย่างโปรแกรม CL ต่อไปนี้ จะแสดง Tree ออกมาดังรูปที่ 10 และ 11

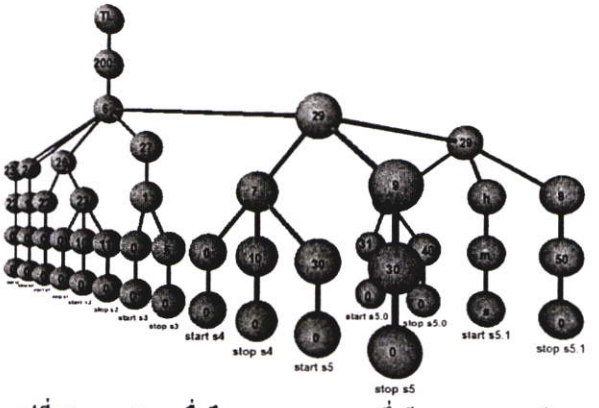
```
[22:0:0|23/6/2005, 1:0:0|24/6/2005] MAIL_AVAILABLE ->
  update_website('www.football.com')                  (s0)
[22:0:0|26/6/2005, 23:0:0|26/6/2005]
  browserto('www.news.co.th')                         (s1)
[23:10:0|26/6/2005, 23:11:0|26/6/2005]
  prompt('What time do you want to start working
          tomorrow?',h, m, s)                         (s2)
[1:0:0|27/6/2005, 1:5:0|27/6/2005] standby()          (s3)
[7:0:0|29/6/2005, 7:10:0|29/6/2005] wakeup()          (s4)
[7:30:0|29|6/2005, 9:30:0|29/6/2005] sequential:      (s5)
```

```
[7:31:0|29/6/2005, 7:40:0|29/6/2005]checkmail()(s5.1)
[h:m:s|29/6/2005, 8:50:0|29/6/2005]
  report_working_status()                    (s5.2)
```

เราพบว่าอินเตอร์พรีตเตอร์ที่ปรับปรุงใหม่ สามารถแก้ไขข้อจำกัดของอินเตอร์แบบเดิมได้ และช่วยให้การ debug โปรแกรมทำได้สะดวกขึ้น



รูปที่ 10 แสดง Tree เมื่อประมวลผลโปรแกรมในครั้งแรก



รูปที่ 11 แสดง Tree เมื่อถึง 7:30:0|29/6/2005 ซึ่งมีการแตก scope ย่อย

9. การวิเคราะห์ Tree และอินเตอร์พรีตเตอร์แบบใหม่

การปรับปรุงรูปแบบของ Tree เพื่อการจัดเก็บตัวแปรเวลาที่เป็นกำหนดการของ statement รวมทั้งการจัดการ scope ทำให้โครงสร้างอินเตอร์พรีเตอร์มีการเปลี่ยนแปลงไปไม่มากไปจากเดิม และข้อดีของโครงสร้าง Tree ที่ออกแบบไว้คือ สามารถบันทึกตารางการทำงานของโปรแกรมที่ดำเนินการไปแล้วในอดีตรวมกับตารางเวลาในปัจจุบันและอนาคตไว้ด้วยกัน ซึ่งง่ายต่อการตรวจสอบความถูกต้องโดยใช้ debugger

นอกจากนี้ เมื่อพิจารณาเปรียบเทียบอินเตอร์พรีตเตอร์แบบใหม่กับแบบเดิม จะพบว่าอินเตอร์พรีตเตอร์แบบใหม่มีข้อดีคือ

1. สามารถประมวลผล statement ที่มีกำหนดเวลาเป็นตัวแปรได้
2. สามารถประมวลผล statement ที่เกิดซ้ำๆ เป็นคาบๆ ได้
3. สามารถรองรับ scope ของ nested statement ได้

แต่อย่างไรก็ตามโดยลักษณะเฉพาะของการประมวลผลโปรแกรม CL จะต้องเป็นไปตามเงื่อนไขที่ว่าตารางเวลาการทำงานของโปรแกรมจะต้องถูกจัดเก็บใน Tree ให้เสร็จสิ้นก่อนกำหนดการทำคำสั่งแรกของตารางเวลานั้นจะเริ่มต้นขึ้น

10. บทสรุป

ในบทความนี้เราได้ปรับปรุงภาษา CL ให้มีความสามารถมากขึ้น โดยได้ปรับปรุงให้รองรับตารางการทำงานที่ทำซ้ำๆเป็นคาบๆได้ และตารางการทำงานที่ยืดหยุ่น ในลักษณะที่หมายกำหนดการการทำงานถูกกำหนดค่าได้ด้วยตัวแปร

11. เอกสารอ้างอิง

[1] วิศิษฏ์ หิรัญกิตติ และคณะ, "บราวเซอร์ที่สามารถโปรแกรมได้", การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 6, 2545

[2] วิศิษฏ์ หิรัญกิตติ และสุพัฒณคา โชติพันธ์ "CL: ภาษาสำหรับการสั่งงานคอมพิวเตอร์ด้วยเวลาและเหตุการณ์", การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 7, 2546.

[3] วิศิษฏ์ หิรัญกิตติ และสุพัฒณคา โชติพันธ์ "ภาษาสำหรับพัฒนาเอเจนต์เพื่อการสื่อสาร", การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 8, 2547.

[4] วิศิษฏ์ หิรัญกิตติ และสุรชัย ล้อเจริญ "นามธรรมของตารางลำดับเวลาและการใช้งานสำหรับประมวลผลคำสั่งในภาษา CL", การประชุมวิทยาการและวิศวกรรมคอมพิวเตอร์แห่งชาติ ครั้งที่ 9, 2548.

[5] สุพัฒณคา โชติพันธ์ "CL: ภาษาสำหรับการสั่งงานคอมพิวเตอร์ด้วยเวลาและเหตุการณ์", วิทยานิพนธ์ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง, 2547.

[6] K. Prouskas and J. Pitt., "A Real-Time Architecture for Time-Aware Agents" IEEE transactions on systems, man, and cybernetics , IEEE Explorer, pp. 1553-1568, 2004.

[7] K. Prouskas and J. Pitt., "Towards a Real-Time Architecture for Time-Aware Agents." In Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002), volume 1, pp. 92-93, 2002.

ประวัติผู้เขียน

ชื่อ-นามสกุล	นายสุรชัย ล้อเจริญ
วัน เดือน ปีเกิด	11 ธันวาคม 2523
ที่อยู่	13/1 หมู่ 2 ต.อุโลกสี่หมื่น อ.ท่ามะกา จ.กาญจนบุรี
ประวัติการศึกษา	2545 วิศวกรรมศาสตร์บัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง