บทที่ 5

อินเตอร์พรีเตอร์ (INTERPRETER)

จากขั้นตอนการคอมไพล์ ภาษาต้นแบบจะถูกเปลี่ยนให้อยู่ในรูปของรหัสกลาง ที่สามารถจะนำมาใช้ดำเนินการ (execution) ได้ อินเตอร์พรีเตอร์จะเป็นเครื่องมือในการที่จะนำเอารหัสกลางที่ได้มาประมวลผล ซึ่งในบทนี้จะอธิบายถึงสภาพของหน่วยความจำในขณะกระทำการอันประกอบด้วย เซกเมนต์รหัส (code segment), เซกเมนต์ข้อมูล (data segment) โดยกล่าวถึงการจัดพื้นที่รหัสกลางในเซกเมนต์รหัส การจัดสรรหน่วยความจำให้กับตัวแปรต่างๆในเซกเมนต์ข้อมูลและการทำงานของสแต็ก (stack) ใน เซกเมนต์สแต็ก (stack segment) การตรวจสอบสแต็กล้น(stack over flow), สแต็กขาด (stack underflow) นอกไปจากนี้จะกล่าวถึงการจัดสรรพื้นที่ให้กับกระบวนการและการจัดโครงสร้างข้อมูลให้กระบวนการ

5.1.รันไทม์เอนวิรอนเมนต์ (Runtime environment)

ในขณะที่อินเตอร์พรีเตอร์ประมวลผล อินเตอร์พรีเตอร์จะสร้างรันไทม์เอนวิรอนเมนต์ซึ่งเป็นโครงสร้างข้อมูลที่ในเก็บและประมวลผลข้อมูลในขณะที่ดำเนินการ ซึ่งประกอบด้วยเซกเมนต์รหัส เซกเมนต์ข้อมูล และ เซกเมนต์สแต็ก

5.1.1.เซกเมนต์รหัส (code segment)

เนื่องจากอินเตอร์พรีเตอร์ใช้รหัสกลางซึ่งเป็นรหัสไบต์ ดังนั้นเพื่อสะดวกในการอ่านคำสั่ง (instruction) เราจะออกแบบให้เซกเมนต์รหัสเป็นอะเรย์ชนิด 8 บิต ดังนี้

CS : array [1 to max. size of CS] of byte

รหัสกลางจะมีความยาว 1 ไบต์ รหัสกลางแต่ละคำสั่งจะมีจำนวนตัวถูกกระทำ (operand)ที่แตกต่างกันระหว่างศูนย์ถึงสามตัว เนื่องจากโครงสร้างของข้อมูลที่ได้ออกแบบไว้เป็นข้อมูลชนิด untype ความยาว 16 บิต ดังนั้นตัวถูกกระทำ (operand) หนึ่งตัวจะมีความยาวเท่ากับสองไบต์ รหัสกลางที่ได้จากขั้นตอนคอมไพเลอร์จะประกอบด้วยรหัสคำสั่ง (instruction code) จำนวน 1 ไบต์ และตามด้วยตัวถูกกระทำของรหัสคำสั่งนั้นๆ ซึ่งมีความยาวสองเท่าของ

จำนวนตัวถูกกระทำ ซึ่งในส่วนนี้ถือเป็นหนึ่งชุดคำสั่ง เมื่อจบหนึ่งชุดคำสั่งแล้วจะเป็นรหัสคำสั่งและตัวถูกกระทำสลับกันจนจบโปรแกรม

รหัสคำสั่ง	ตัวถูกกระทำ	ความยาว
Literal	integer	2
LvalueGV,RvalueGV,LValueLV,RvalueLV	address of variable	2
Plus,Minus,Multiple,Div		0
Jump,Jump if zero(JZ),Call	code segment address	2
Set,SetGV		0
Equal,And,Or,Not		0
FetchGV,Index		0
Return0,return1		0
Send,Receive		0
Function	#Parameter, #Local var	4
Process	PID, #Parameter, #Local var	6
LT,LE,GE,GT,NE		0
Wait, Signal	address of global variable	2
Print,PrintCh		0

ตารางแสดงรหัสกลางและจำนวนตัวถูกกระทำ

รหัสกลางสามารถจำแนกออกเป็นกลุ่มต่างๆ ได้ดังนี้

1.รหัสกลางที่เป็นคำสั่งกระทำการ (operator) ได้แก่คำสั่ง Plus, Minus, Multiple, Div, Equal, And, Or, Not, LT, LE, GE, GT, และ NE โดยมีตัวถูกกระทำเป็นข้อมูลที่อยู่ในสแต็กเซกเมนต์ เมื่ออินเตอร์พรีเตอร์พบคำสั่งเหล่านี้ อ่านตัวถูกกระทำจากเซกเมนต์สแต็ก และดำเนินการตามคำสั่ง และใส่ผลลัพท์ที่ได้กลับเข้าไปในเซกเมนต์สแต็ก

2.คำสั่งกระโดดไปที่ตำแหน่งอื่น เป็นคำสั่งที่ใช้เปลี่ยนตำแหน่งของตัวชี้คำสั่งเพื่อให้อินเตอร์พรีเตอร์ไปทำงานที่ตำแหน่งใหม่ เช่น

Jump	คำสั่งกระโดดโดยไม่มีเงื่อนไข
Jump if zero(JZ)	คำสั่งกระโดด ถ้าค่าบนสุดของสแต็กเป็นศูนย์
Call	สร้างเฟรมใหม่ และกระโดด

3.คำสั่งที่ใช้ในการอ่านค่าตัวแปร จากตำแหน่งที่กำหนดเพื่อมาใส่ไว้ที่บนสุดของเซกเมนต์สแต็ก ได้แก่คำสั่ง

LvalGV	อ่านค่าตำแหน่งที่กำหนดของตัวแปรส่วนกลาง
RvalGV	อ่านค่าที่ตำแหน่งที่กำหนดของตัวแปรส่วนกลาง
LvalLV	อ่านค่าตำแหน่งที่กำหนดของตัวแปรเฉพาะที่
RvalLV	อ่านค่าที่ตำแหน่งที่กำหนดของตัวแปรเฉพาะที่
Fetch	อ่านค่าโดยใช้ตำแหน่งของ top of stack

4.คำสั่งในการกำหนดค่า ได้แก่

Set	กำหนดค่าให้กับตัวแปรเฉพาะ
SetGV	กำหนดค่าให้กับตัวแปรส่วนกลาง

5.คำสั่งที่ใช้ในการประสานจังหวะของกระบวนการ ได้แก่ send,receive,wait,signal

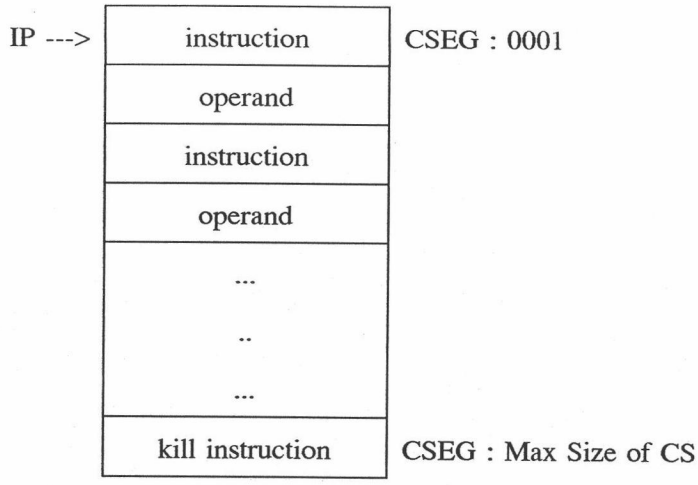
6.คำสั่งในการประกาศโปรแกรมย่อย ได้แก่

Function	เพื่อใช้ประกาศฟังก์ชัน
Process	เพื่อใช้ประกาศกระบวนการ

7.คำสั่งในการแสดงผลได้แก่ คำสั่ง Print และ PrintCh

เซกเมนต์รหัสประกอบด้วยสแต็กที่ใช้เก็บรหัสไบต์ และตัวถูกกระทำ ของรหัสไบต์นั้นๆ เรียงกันตามลำดับ จนจบชุดคำสั่ง อินเตอร์พรีเตอร์จะมีตัวชี้คำสั่ง (instruction pointer, IP) เป็นตัวบอกว่าปัจจุบันกำลังกระทำการอยู่ในตำแหน่งใด ขั้นตอนในการอ่านรหัสไบต์แต่ละ คำสั่งจะอ่านรหัสไบต์มา 1 คำสั่งจากนั้น จะนำคำสั่งไปเปิดตารางเพื่อค้นหาว่าคำสั่งนั้นมีตัวถูกกระทำกี่ตัว และจะอ่านรหัสไบต์ต่อมาเป็นถูกกระทำของคำสั่งข้างต้น และจะเลื่อน IP ไปชี้อยู่ที่คำสั่งถัดไป

เมื่อเปรียบเทียบกับ ภาษาจาวา(JAVA) จะพบว่าจาวามีความสลับซับซ้อนมากกว่าทั้งนี้เนื่องจากจาวาออกแบบใช้งานบนอินเตอร์เนตที่มีความซับซ้อนมากกว่าในงานควบคุมขนาดเล็ก โดยจาวาจะมีลักษณะคล้ายภาษา C++ ซึ่งสามารถเขียนโปรแกรมแบบ object oriented programming แต่ในงานควบคุมขนาดเล็กมักจะไม่นิยมใช้ภาษาที่มีความสลับซับซ้อนมากเกินไป เพราะว่าทำให้การ implement runtime environment บนโปรเซสเซอร์ที่แตกต่างกันทำได้ยาก นอกจากนี้ในขณะที่เริ่มทำงานวิจัยชิ้นนี้ (พ.ศ.2537) ยังไม่มีมาตรฐานรหัสกลางที่สามารถประมวลผลพร้อมกันที่มีความเหมาะสม

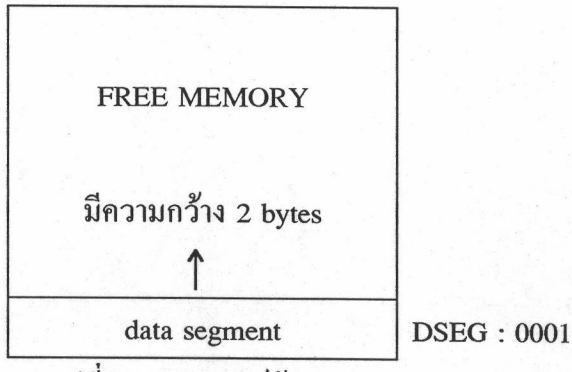


รูปที่ 5.1 เซกเมนต์รหัส

5.1.2.เซกเมนต์ข้อมูล (Data segment)

สำหรับชนิดข้อมูลในภาษาต้นแบบออกแบบให้เป็นภาษาที่มีข้อมูลที่มีขนาดคงที่ คือเป็นข้อมูลที่มีความยาว 16 บิต ดังนั้น เซกเมนต์ข้อมูลซึ่งใช้ในการเก็บค่าตัวแปรที่เป็นตัวแปรส่วนกลางออกแบบให้เป็นอะเรย์ชนิด 16 บิต

DS : array [1..size of DS] of integer



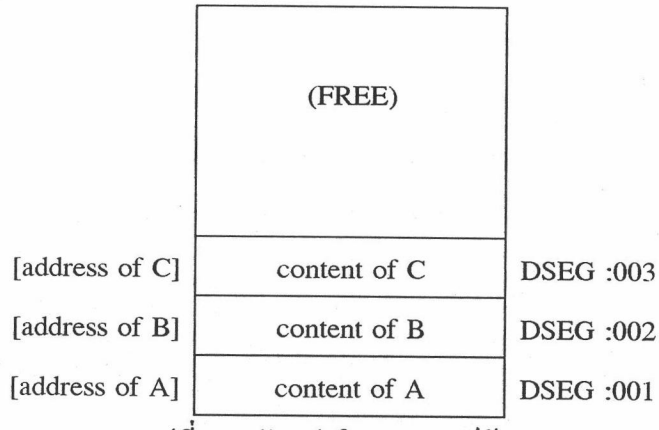
รูปที่ 5.2 เซกเมนต์ข้อมูล

เซกเมนต์ข้อมูลใช้ในการเก็บค่าตัวแปรส่วนกลาง โดยแบ่งตัวแปรออกเป็น 2 ชนิด คือ ตัวแปรที่มีขนาดคงที่ (fix size) คือ จำนวนเต็มชนิด 16 bit และตัวแปรแบบอะเรย์ (array) ที่สามารถกำหนดขนาดได้ ตำแน่งต่างๆของตัวแปรส่วนกลางจะถูกกำหนดในขั้นตอนการคอมไพล์โปรแกรม

ในการประกาศตัวแปรส่วนกลางชนิดจำนวนเต็ม คอมไพเลอร์จะจัดสรรพื้นที่ในเซกเมนต์ข้อมูล โดยเริ่มจาก SSEG : 001 เป็นต้นไป เช่น ในการประกาศตัวแปร A,B,C

GLOBAL A,B,C

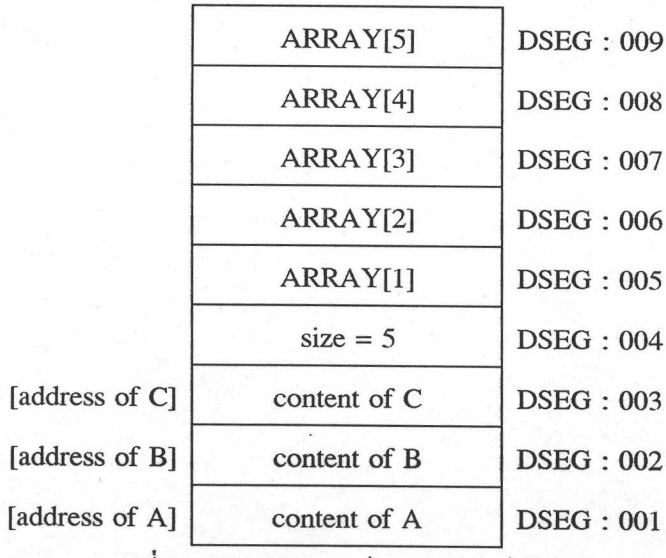
จะได้



รูปที่ 5.3 ตัวแปรในเซกเมนต์ข้อมูล

ตัวแปรอะเรย์ ประกอบด้วย array descriptor และข้อมูล โดย descriptor จะใช้ในการเก็บจำนวนข้อมูล เช่น หากประกาศตัวแปร ARRAY จำนวน 5 ตัว ดังนี้

GLOBAL ARRAY[5]



รูปที่ 5.4 ตัวแปรอะเรย์ในเซกเมนต์ข้อมูล

การอ้างอิงถึงตัวแปร ARRAY จะใช้แอดเดรส DSEG:004 แต่อินเตอร์พรีเตอร์จะต้องคำนวณ index ของ element ใดๆ เช่น การอ้างอิง ตัวแปร

ARRAY[I] จะหมายถึง DSEG : (ARRAY + I + 1)

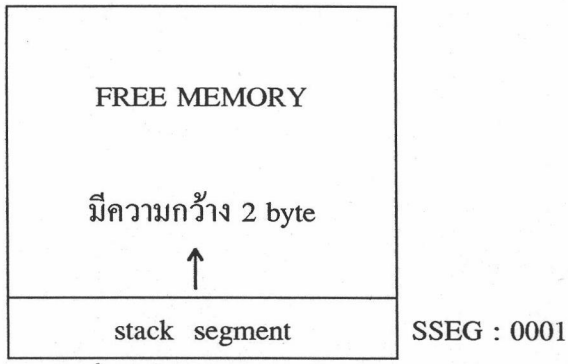
การตรวจสอบช่วงของตัวแปรอะเรย์ (range checking)

ในการตรวจสอบช่วงของการอ้างอิงตัวแปรอะเรย์ จะทำในขณะที่ดำเนินการ (runtime checking) โดยจะตรวจสอบในขณะที่ประมวลผลคำสั่ง index โดยจะเปรียบเทียบ index ว่ามีค่าระหว่าง 1 จนถึงขนาดของอะเรย์หรือไม่ หากไม่อยู่ในช่สงที่กำหนดแสดงว่ามีการอ้างถึงตัวแปรอะเรย์นอกช่วงที่ได้ประกาศไว้ก่อนหน้านี้

5.1.3.เซกเมนต์สแต็ก (Stack segment)

สำหรับชนิดข้อมูล ในภาษาต้นแบบออกแบบให้เป็นภาษาที่มีข้อมูลที่มีขนาดคงที่ คือ เป็นข้อมูลที่มีความยาว 16 บิต ดังนั้น เซกเมนต์ข้อมูลซึ่งใช้ในการเก็บค่าตัวแปรที่เป็นตัวแปรส่วนกลางออกแบบให้เป็นอะเรย์ชนิด 16 บิต

SS : array [1..size of SS] of integer



รูปที่ 5.5 เซกเมนต์สแต็ก

เซกเมนต์สแต็กจะใช้ในการคำนวณค่าต่างๆ และใช้ในการเก็บค่าตัวแปรเฉพาะที่ ในการอินเตอร์พรีท์เราจะใช้เซกเมนต์สแต็กเก็บค่าในการคำนวณเช่น

นิพจน์

*A + B * C*

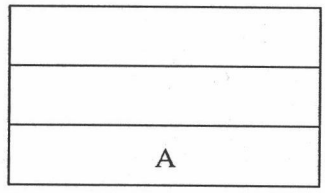
จะถูกแปลเป็นรหัสกลาง

1 : R-valueLV [addr A]
2 : R-valueLV [addr B]
3 : R-valueLV [addr C]
4 : MULTIPLY
5 : PLUS

อินเตอร์พรีเตอร์จะโหลดค่า A ใส่ในเซกเมนต์สแต็ก และโหลดค่า B ,C ตามลำดับ เมื่ออินเตอร์พรีเตอร์พบคำสั่งคูณจะ pop ค่าจากเซกเมนต์สแต็กมาสองค่าและนำมาคูณกัน ผลที่ได้จะ

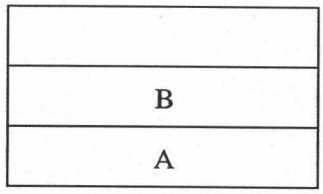
push ลงสแต็กอีกครั้ง และเมื่อพบคำสั่งบวกก็จะทำการ pop ค่าจากสแต็กมาสองค่า (คือค่า B*C และค่า A ตามลำดับ) เมื่อนำมาบวกแล้วค่าที่ได้จะเก็บไว้ในสแต็กเพื่อนำไปใช้ต่อไป

เมื่อ ดำเนินการคำสั่ง R-valueLV [addr A]



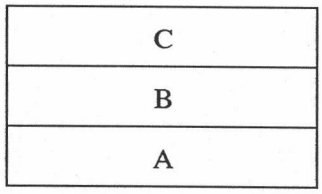
1: R-valueLV [Address A] <-- Stack pointer

เมื่อ ดำเนินการคำสั่ง R-valueLV [addr B]



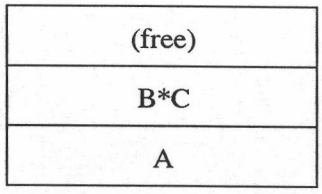
2: R-valueLV [Address B] <-- Stack pointer

เมื่อ ดำเนินการคำสั่ง R-valueLV [addr C]



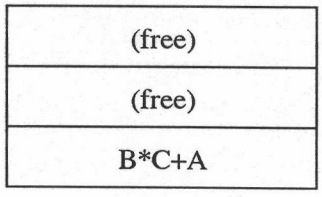
3: R-valueLV [Address C] <-- Stack pointer

เมื่อ ดำเนินการคำสั่ง multiply



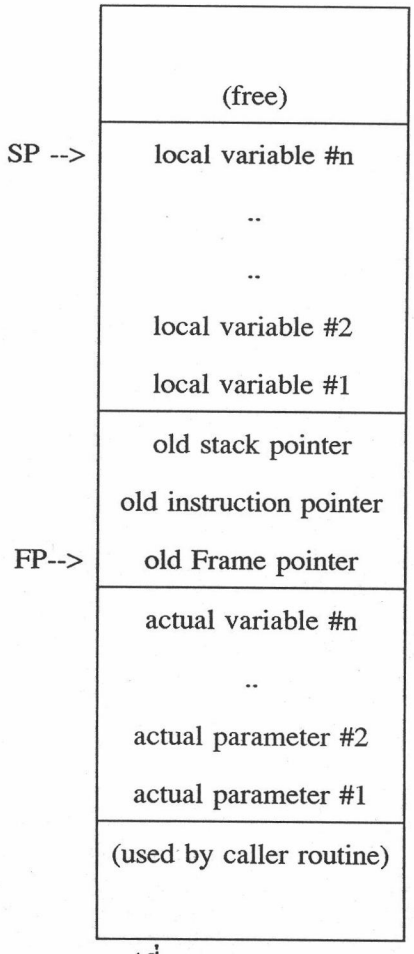
4: Multiply <-- Stack pointer

เมื่อ ดำเนินการคำสั่ง plus



5: Plus <-- Stack pointer

ในการเรียกโปรแกรมย่อย อินเตอร์พรีเตอร์จะต้องเก็บค่า machine status ซึ่งได้แก่ค่า frame pointer (FP), stack pointer (SP), instruction pointer (IP) ดังรูป



รูปที่ 5.6 machine status

ก่อนการเรียกโปรแกรมย่อยจะมีการโหลดค่าพารามิเตอร์ส่งผ่าน (passing parameter) มาให้ไว้ในเซกเมนต์สแต็ก เมื่ออินเตอร์พรีเตอร์พบคำสั่ง call จะทำการ push ค่า FP,IP,SP ของ frame เก่าไว้ ก่อนที่จะไปดำเนินการ ณ.ตำแหน่งของโปรแกรมย่อย เมื่อดำเนินการ จนพบคำสั่ง return จะทำการลบ frame เก่าโดยการเลื่อน FP มาที่ FP ของ frame เก่า และหากเป็นการ return แบบที่มีการส่งค่ากลับ ก็จะลอกค่าบนสุดของสแต็ก (top of stack) ของ frame ปัจจุบัน (ค่าที่ SP ชี้อยู่) ไปสู่ frame เดิมก่อนการเรียก เพื่อส่งค่าของ ฟังก์ชันไปสู่โปรแกรมที่เรียกมา พร้อมทั้งเปลี่ยนค่า SP และ IP เป็นค่าเก่าก่อนการเรียกโปรแกรมย่อย

ยกตัวอย่างเช่นในภาษาต้นแบบกำหนดฟังก์ชันที่มีพารามิเตอร์สามตัว และตัวแปรเฉพาะที่หนึ่งตัวดังนี้

```
function F1(a,b,c)
{
        local_var = 0;                  // define local variable
        // other statement
}
```

ถ้าจะได้รหัสกลาง คือ

function 3 1
(other intermediate code)

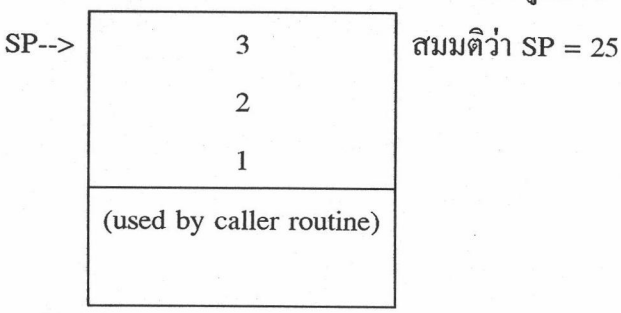
เมื่อมีการเรียกฟังก์ชัน F1 โดยมีพารามิเตอร์คือ 1, 2 และ 3 ตามลำดับ

F1 (1, 2, 3);

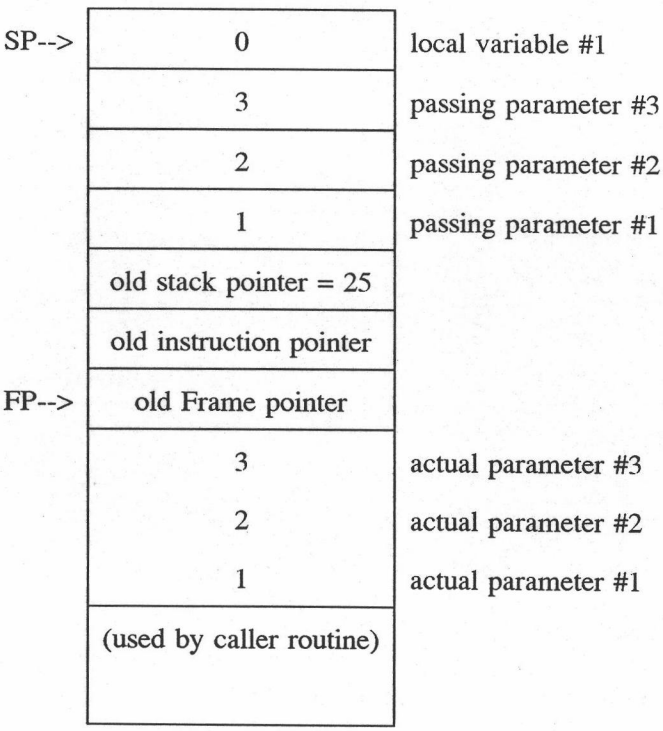
จะได้รหัสกลาง คือ

literal 1
literal 2
literal 3
call (address of F1)

เมื่ออินเตอร์พรีเตอร์ประมวลผลก่อนถึงคำสั่ง call เซกเมนต์สแต็กจะมีข้อมูลดังนี้



เมื่อดำเนินการคำสั่ง Call และ function 3 0



การตรวจสอบเซกเมนต์สแต็ก (stack checking)

1.การตรวจสอบสแต็กขาด (stack underflow)

อินเตอร์พรีเตอร์จะไม่ทำการตรวจสอบสแต็กขาด ทั้งนี้เป็นเพราะในขั้นตอนการคอมไพล์โปรแกรมต้นแบบจะทำการสร้าง parse tree ในการตรวจสอบวากยสัมพันธ์ของภาษา ในขั้นตอนนี้จะทำให้เกิดคำสั่ง push stack และ pop stack เราจะพบว่าการ pop stack จะเกิดขึ้นในกรณีต่อไปนี้คือ

1.ใน assignment statement

2.เมื่อพบ operator

3.เมื่อมีการ reference index ของตัวแปร array

หากพิจารณาจาก parse tree ของกรณีดังกล่าวข้างต้น เราจะพบว่าคำสั่งที่จะทำให้เกิดการ pop stack จะไม่อยู่ที่ใบของต้นไม้เลย ส่วนคำสั่งที่อยู่ที่ใบจะเป็นคำสั่งที่ทำให้เกิดการ push stack เมื่อเปลี่ยนต้นไม้ให้อยู่ในรูปของ reverse polish notation จะทำให้คำสั่งเหล่านี้ จะต้องมีการ push ค่าลงใน stack ก่อนการ pop stack เสมอ ดังนั้นในการ pop stack ทุกครั้งจึงไม่มีโอกาสในการ under flow

2.การตรวจสอบสแต็กล้น (stack overflow)

หากทำการตรวจสอบสแต็กล้นทุกครั้ง ก่อนที่จะ push ค่าลงบนสแต็ก วิธีนี้จะประกันได้ว่าสแต็กจะไม่ล้น แต่เทคนิคนี้จะทำให้ประสิทธิภาพด้านความเร็วของอินเตอร์พรีเตอร์ลดลงอย่างมาก เนื่องจากต้องสูญเสียเวลาส่วนใหญ่ในการทำการตรวจสอบ overflow หากพิจารณาการเกิดการ over flow จะเกิดขึ้นในกรณีต่อไปนี้

1.เมื่อพบรหัสไบต์ที่เป็นตัวดำเนินการ (operator) โดย unary operator จะ push stack หนึ่งชั้น หากเป็น binary operator หรือ logical operator จะ push stack สองชั้น ดังนั้นสแต็กจะไม่ overflow เมื่อมีพื้นที่ในสแต็กเหลือมากกว่าสองชั้น

2.ในการ call ไปยังโปรแกรมย่อย จะมีการสร้าง stack frame ซึ่งขนาดของ stack frame จะมีส่วนคงที่คือค่า machine status ได้แก่ instruction pointer(IP), stack pointer(SP) ,frame pointer(FP) ซึ่งมีขนาด 6 ไบต์ และ ส่วนที่แปรผันได้แก่จำนวน passing parameter และตัวแปร local ในโปรแกรมย่อยนั้นๆ ดังนั้นการเกิด overflow จะเกิดเมื่อมีพื้นที่เหลือในสแต็กก่อนการเรียกโปรแกรมย่อยน้อยกว่าจำนวน ตัวแปรที่ใช้ x 2 + 6 ไบต์

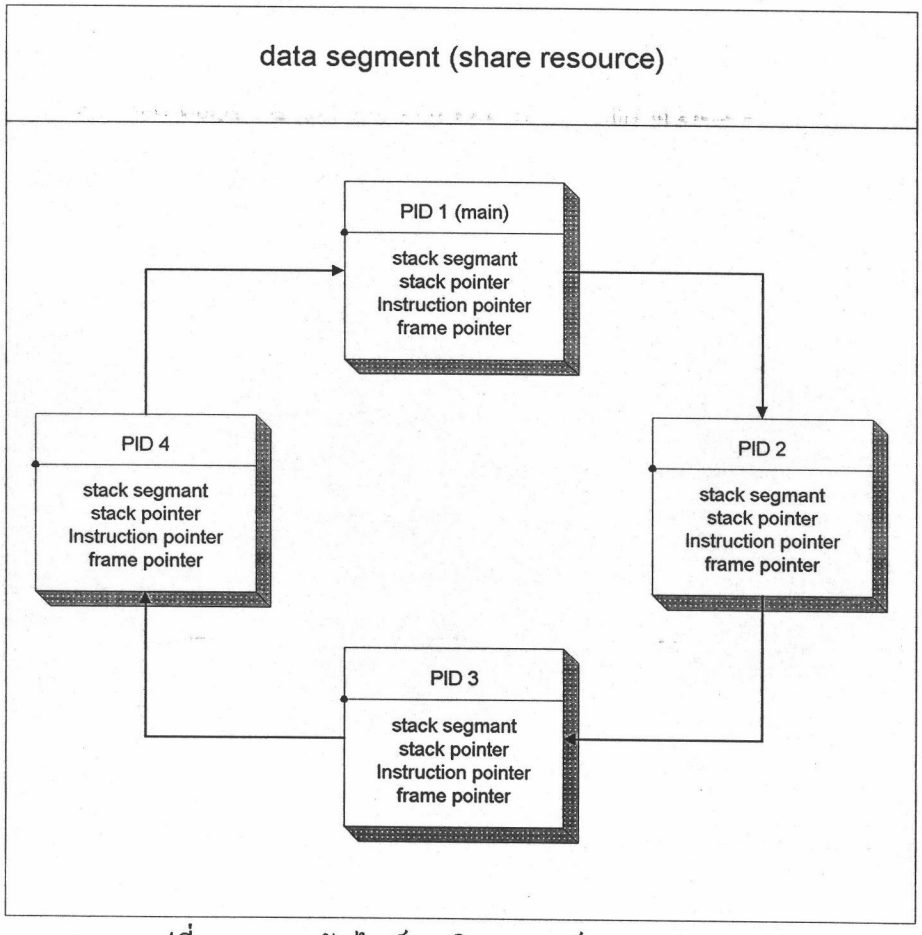
3.ในสเตตเมนต์ทำซ้ำ (repetition statement) จะไม่มีการสร้างสแต็กเฟรมขึ้นใหม่ เพราะเนื่องจากขั้นตอนการทำแปลสเตตเมนต์ทำซ้ำ คำสั่งเหล่านี้จะถูกเปลี่ยนให้อยู่ในรูปของคำสั่ง jump เมื่ออินเตอร์พรีเตอร์พบคำสั่ง jump จะทำการ pop stack ก่อนที่จะมีการเลื่อน instruction

pointer หากมีคำสั่งสเตตเมนต์ทำซ้ำ ที่มีจำนวนลูป (loop) จำนวนมาก ก็จะไม่ทำให้เกิดการ overflow

ดังนั้นเพื่อประสิทธิภาพด้านความเร็วของอินเตอร์พรีเตอร์ เราจะทำการเช็คเมื่อมีการ call subroutine เท่านั้น เพราะมีการใช้สแต็ก จำนวนมาก แต่จะไม่ตรวจสอบเมื่อพบคำสั่งที่ เป็นคำสั่ง push stack ทุกครั้งเพราะมีโอกาสในการ overflow น้อยมาก

5.2 กระบวนการ

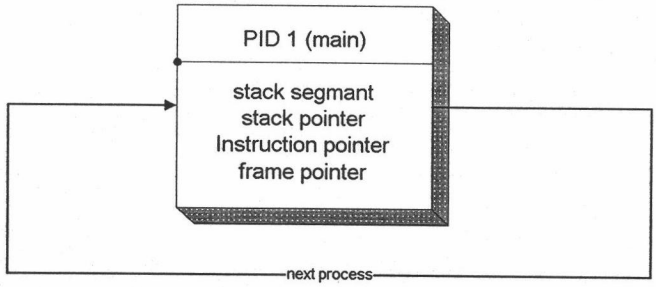
กระบวนการหนึ่งจะประกอบด้วย รันไทม์เอนวิรอนเมนต์ของตนเองที่ทำงานเป็นอิสระจากกระบวนการอื่นๆ ดังนั้นกระบวนการหนึ่งๆจึงจะต้องมีคุณสมบัติ (property) อันได้แก่ หมายเลขประจำกระบวนการ, เซกเมนต์สแต็ก (stack segment), ตัวชี้คำสั่ง(instruction pointer), ตัวชี้สแต็ก (stack pointer), และตัวชี้เฟรม (frame pointer) และมีส่วนที่กระบวนการต่างๆ สามารถใช้งานร่วมกัน ได้แก่เซกเมนต์ข้อมูลซึ่งจัดเป็นทรัพยากรร่วม (share resource) กระบวนการต่างๆเหล่านี้จะเชื่อมต่อกันเป็นลิสต์แบบวงกลม (circular link list) ดังรูป



รูปที่ 5.7 แสดงรันไทม์เอนวิรอนเมนต์ของกระบวนการ

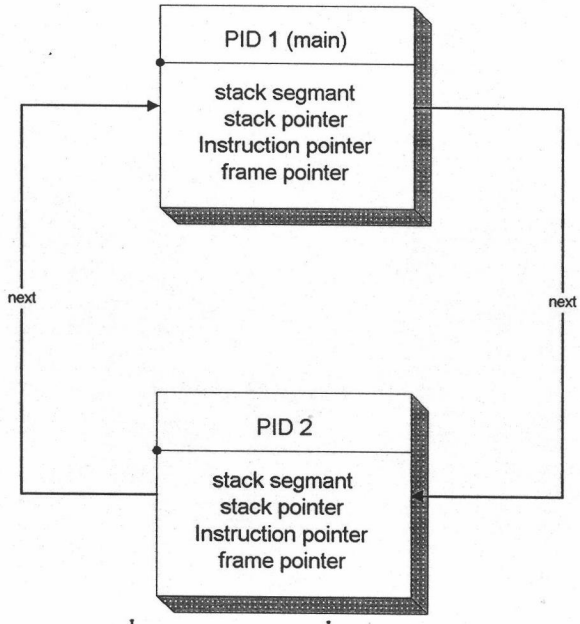
5.2.1 การเกิดของกระบวนการ

เมื่อเริ่มต้นการทำงาน อินเตอร์พรีเตอร์จะสร้างกระบวนการขึ้นเพื่อเป็นกระบวนการหลักในการทำงานสำหรับฟังก์ชันรัน (run function) ซึ่งกำหนดให้เป็นกระบวนการที่จะถูกดำเนินการก่อน โดยกระบวนการนี้จะมีเลขประจำตัว (process ID) เท่ากับหนึ่ง และมีตัวชี้กระบวนการถัดไป (next pointer) ชี้ไปที่ตนเอง ดังรูป



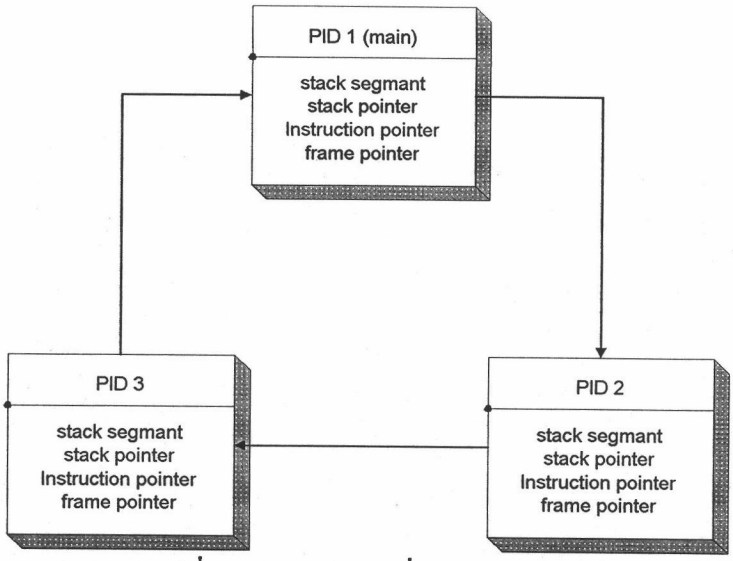
รูปที่ 5.8 แสดงรันไทม์เอนวิรอนเมนต์ของกระบวนการใหม่

เมื่อมีคำสั่งในการเรียกกระบวนการ (process call) อินเตอร์พรีเตอร์ จะสร้างกระบวนการขึ้นมาใหม่เพื่อรองรับคำสั่ง กระบวนการที่เกิดขึ้นจะมีเลขประจำตัวเพิ่มขึ้นจากของเดิม รันไทม์เอนวิรอนเมนต์ของกระบวนการสุดท้ายจะชี้ไปที่กระบวนการใหม่ที่ถูกสร้างขึ้น กระบวนการใหม่จะชี้ไปที่กระบวนการแรกสุดเป็นลิงค์ลิสต์แบบวงกลม (circular link list)



รูปที่ 5.9 แสดงการเพิ่มกระบวนการ

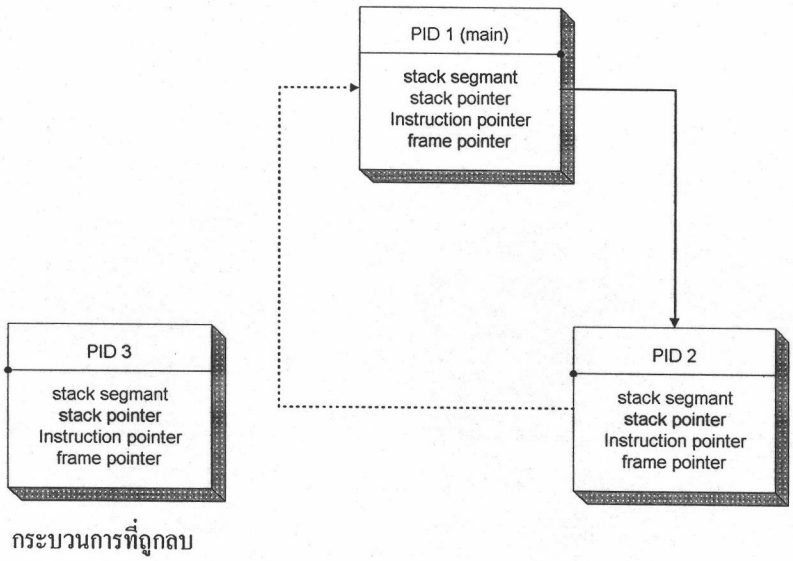
ถ้ามีกระบวนการที่สามเกิดขึ้นมาอีก



รูปที่ 5.10 แสดงการเพิ่มกระบวนการ

5.2.2 การจบของกระบวนการ

เมื่อมีกระบวนการที่ทำงานที่มอบหมายให้สิ้นสุดลงด้วยคำสั่ง kill process การลบรันไทม์เอนวิรอนเมนต์ของกระบวนการทำได้โดยการเปลี่ยนตัวชี้ของกระบวนการก่อนหน้าที่ชี้มาที่กระบวนการที่ต้องการลบ ไปที่กระบวนการถัดไปดังรูป



รูปที่ 5.11 แสดงการเพิ่มกระบวนการ

5.2.3 การตรวจสอบการสิ้นสุดของโปรแกรม

การตรวจสอบการสิ้นสุดของโปรแกรมทั้งหมด ไม่จำเป็นต้องตรวจสอบจากตัวชี้คำสั่ง (instruction pointer) ก็ได้ เมื่อกระบวนการที่กำลังประมวลผลพบคำสั่ง kill process กระบวนการเหล่านั้นจะถูกลบ จนกระทั่งกระบวนการทุกๆกระบวนการถูกลบจนหมด เมื่อรันไทม์เอนวิรอนเมนต์ลิสต์เป็นลิสต์ว่าง

5.2.4 ตัวจัดกำหนดการ (Scheduler)

รันไทม์เอนวิรอนเมนต์ออกแบบเป็นลิสต์แบบวงกลม กระบวนการต่างๆถูกกำหนดให้ประมวลผลด้วยจำนวนรหัสกลาง ซึ่งเรียกว่า ควอนตัม (quantum) เมื่อกระบวนการที่เข้ามาครอบครองซีพียูเข้ามาประมวลผล อินเตอร์พรีเตอร์ จะนับจำนวนรหัสกลางที่ถูกประมวลผล กระบวนการจะถูกประมวลผลจนกระทั่งจำนวนรหัสกลางที่ประมวลผลแล้วมีค่ามากกว่าควอนตัม กระบวนการนี้จะถูกบล็อก และอินเตอร์พรีเตอร์จะเลื่อนรันไทม์เอนวิรอนเมนต์ลิสต์ไปที่กระบวนการถัดไป ซึ่งจะทำให้กระบวนการนั้นเข้าครอบครองซีพียูเพื่อประมวลผลต่อไป

ตัวจัดกำหนดการจะจัดคิวแบบวนรอบ (round robin,RR) โดยตัวจัดกำหนดการจะนำคำสั่งจากเซกเมนต์รหัสจากรันไทม์เอนวิรอนเมนต์ของกระบวนการแรกมาประมวลผลครั้งละหนึ่งคำสั่ง และจะเพิ่มค่าของตัวนับควอนตัม (quantum count) จนกระทั่งประมวลผลคำสั่งของกระบวนการนี้จนครบควอนตัมที่กำหนดไว้ ตัวจัดกำหนดการจะเลื่อนรันไทม์เอนวิรอนต์เมนต์ลิสต์ไปที่กระบวนการถัดมา และจะเริ่มต้นนับจำนวนควอนตัมของกระบวนการปัจจุบันหมุนเวียนไปเรื่อยๆจนครบทุกกระบวนการ

กระบวนการที่ได้ประมวลผลจะถูกบล็อกเมื่อหมดเวลาควอนตัมของกระบวนการนั้น นอกจากนั้นหากกระบวนการยังอาจถูกบล็อกได้หากใช้ทรัพยากรร่วม ในขณะที่มีกระบวนการอื่นใช้อยู่ ซึ่งจะได้กว่าถึงในหัวข้อถัดไป

5.2.5 เซมาฟอร์ (semaphore)

ในการควบคุมการใช้ทรัพยากรร่วม เซมาฟอร์เป็นตัวแปรร่วมที่ใช้ในการประสานจังหวะ โดยผ่านคำสั่งพื้นฐานสองคำสั่งคือ wait และ signal เมื่อมีกระบวนการที่ต้องการใช้ทรัพยากรร่วม และจะป้องกันไม่ให้กระบวนการอื่นเข้ามาใช้ หากมีกระบวนการอื่นต้องการใช้ตัวแปรร่วม จะต้องรอให้กระบวนการแรกประมวลผลในบริเวณวิกฤตจนเสร็จเสียก่อน

ตัวแปรเซมาฟอร์จะกำหนดให้เป็นตัวแปรส่วนกลาง (global variable) และต้องกำหนดให้มีค่าเริ่มต้นเท่ากับหนึ่ง เมื่อกระบวนการที่ต้องการใช้ทรัพยากรร่วม จะใช้คำสั่ง wait เพื่อกำหนดจุดเริ่มต้นของการใช้ทรัพยากรร่วม อินเตอร์พรีเตอร์จะลดค่าของ semaphore ลงหนึ่งและจะประมวลผลสเตตเมนต์ต่อไป หากในช่วงนี้เกิดหมดเวลาควอนตัม แล้วมีกระบวนการอื่นต้องการใช้ทรัพยากรร่วมโดยการประมวลผลคำสั่ง wait (semaphore) กระบวนการนี้จะถูกบล็อกเพราะมีกระบวนการอื่นใช้อยู่ จนกระทั้งกระบวนการแรกประมวลผลเสร็จแล้วจะให้สัญญาณ signal(semaphore) อินเตอร์พรีเตอร์จะเพิ่มค่าตัวแปรเซมาฟอร์ขึ้นอีกหนึ่ง เพื่อให้กระบวนการอื่นสามารถประมวลผลต่อไปได้ ตัวอย่างเช่น

```
Process P1( )
{
        wait (semaphore);
        access share resource
        signal (semaphore);
}
Process P2( )
{
        wait (semaphore);
        access share resource
        signal (semaphore);
}
```

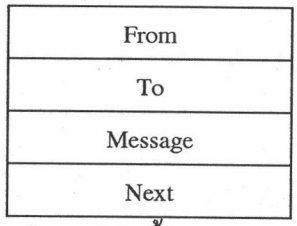
หากกระบวนการ P1 ก่อนและสามารถเข้าครอบครองทรัพยากรร่วมได้ อินเตอร์พรีเตอร์จะลดค่า semaphore ลงเหลือศูนย์

```
P1:   wait (semaphore);                          // decrease semaphore to 0
      ...
      access share resource                      // time slice limit
      ...
                          P2: wait (semaphore);  // process block
P1:   continue execution
      signal (semaphore);                        // increase semaphore to 1
      ...
      ...
      blocked
                          P2: wait (semaphore);  // success
                              ...
                              signal (semaphore);
```

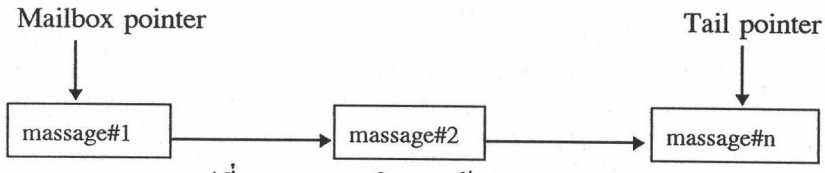
ในขณะที่ P1 กำลังอยู่ในบริเวณวิกฤต และ context switch เนื่องจากหมดเวลาควอนตัม ทำให้ P2 ประมวลผลคำสั่ง wait (semaphore) แต่เนื่องจากค่า semaphore มีค่าเท่ากับศูนย์ กระบวนการ P2 จะถูกบล็อก P1จะประมวลผลต่อจนเสร็จและจะประมวลผลคำสั่ง signal (semaphore) ทำให้ค่า semaphore มีค่าเป็นหนึ่ง เมื่อกระบวนการ P2 ได้ประมวลผลอีกครั้งถึงจะสามารถเข้าใช้ทรัพยากรร่วมได้

5.5.6 ระบบการส่งข่าวสาร (message passing)

ในระบบนี้ออกแบบให้ระบบการส่งข่าวสารเป็นแบบ asynchronous message passing โดยมีโครงสร้างข้อมูลแบบมีช่องทางเดียวในการส่งข้อมูล โดยในข่าวสารแต่ละฉบับจะประกอบด้วย กระบวนการผู้รับ กระบวนการผู้ส่ง และข้อความที่ส่งซึ่งมีขนาด 16 บิต



ข่าวสารต่างๆจะอยู่ในรูปของลิงค์ลิสต์ และมีตัวชี้ตำแหน่งอยู่สองตัวคือ Mailbox pointer จะชี้ไปที่ส่วนหัวของลิสต์ และ tail pointer จะชี้ไปที่ท้ายลิสต์



รูปที่ 5.12 แสดงโครงสร้างของ mailbox

ในการเพิ่มข้อมูลจะเพิ่มข้อมูลต่อท้ายจากข่าวสารฉบับสุดท้ายโดยต่อท้ายจากจุดที่ tail pointer ชี้อยู่ แต่เมื่อมีกระบวนการมารับข่าวสารจะเริ่มหาข้อความจากส่วนหัวของลิสต์ก่อน ดังนั้นการให้บริการนี้จะเป็นแบบมาก่อนออกก่อน (first in first out)

การออกแบบเป็นตู้ไปรษณีย์แบบลิงค์ลิสต์นี้จะทำให้กระบวนการผู้ส่งสามารถส่งข้อมูลล่วงหน้าก่อนที่ผู้รับจะมารับได้ ซึ่งหากไม่กำหนดขนาดของลิงค์ลิสต์จะทำให้กระบวนการผู้ส่งไม่ถูกบล็อกเลยเพราะสามารถส่งข้อมูลเก็บไว้ในบัฟเฟอร์ได้