บทที่ 6

การทดลองแก้ปัญหาการขนส่ง

ในบทนี้จะกล่าวถึงการทดลองนำภาษาที่ออกแบบไปแก้ปัญหาการขนส่ง พร้อมทั้งเปรียบเทียบกับวิธีการเขียนโปรแกรมในส่วนของการซิงโครไนซ์ กับภาษาธรรมดา และวัดประสิทธิภาพของอินเตอร์พรีทเตอร์ได้แก่การวัดความเร็วในการประมวลผล และวัดเวลาที่ใช้ในการจัดลำดับงาน

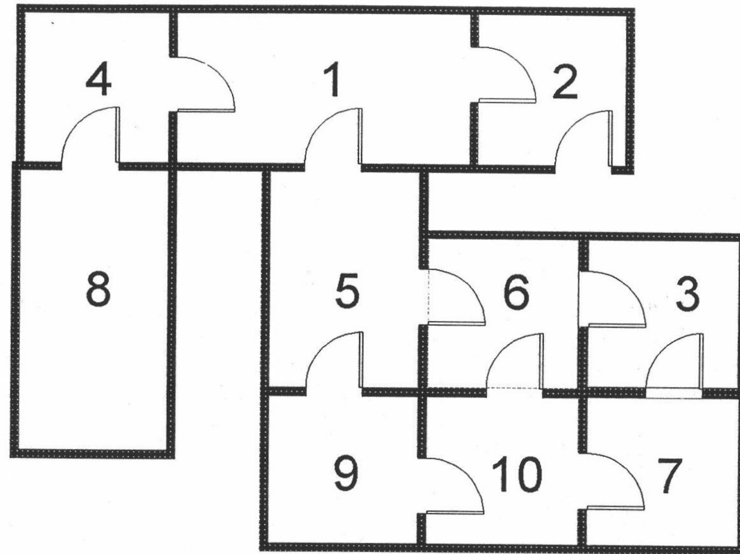
6.1 ปัญหา

กำหนดให้หุ่นยนต์มีเป้าหมายในการเดินทางจากเพื่อไปรับสัมภาระที่ตำแหน่งต้นทางเพื่อขนส่งไปที่ตำแหน่งปลายทาง หุ่นแต่ละตัวจะมีหน้าที่ในการเลือกเส้นทางในการเดินทาง และเดินทางไปตามเส้นทางที่กำหนด แต่ในระบบจะมีหุ่นยนต์อยู่หลายตัวที่จะใช้เส้นทางร่วมกัน ดังนั้น หากมีหุ่นมากกว่าหนึ่งตัวต้องการใช้เส้นทางเดียวกัน จะมีหุ่นเพียงตัวเดียวที่เคลื่อนที่ผ่านเส้นทางได้ หุ่นตัวที่เหลือจะต้องรอจนกว่าเส้นทางจะว่าง

หุ่นยนต์แต่ละตัวจะรับคำสั่งจาก job generator ซึ่งจะให้หุ่นยนต์เคลื่อนที่จากตำแหน่งปัจจุบันไปสู่ตำแหน่งปลายทาง เมื่อถึงปลายทางแล้วจะหยุดอยู่ที่โหนด นั้นเพื่อรอรับคำสั่งจาก job generator ต่อไป

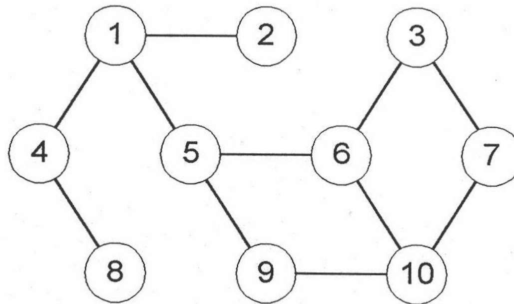
ในการเดินทางจากตำแหน่งหนึ่งไปสู่ตำแหน่งหนึ่ง หุ่นยนต์จะต้องคำนวณหาเส้นทางในการเดินทาง โดยหุ่นยนต์ทุกตัวจะใช้โปรแกรมในการหาเส้นทางโปรแกรมเดียวกัน

สมมุติให้แบบแปลนของโรงงานเป็นห้องต่างๆ 10 ห้องที่มีช่องต่อถึงกันตามรูปที่ 6.1



รูปที่ 6.1 แบบแปลนโรงงาน

แผนที่ในการเดินทางประกอบด้วยโหนด (ซึ่งแทนด้วยวงกลม) ซึ่งโหนด แต่ละโหนดจะถูกเชื่อมต่อกันด้วย ลิงค์ (แทนด้วยเส้นตรง) จะแทนด้วยเส้นทางตามรูปที่ 6.2



รูปที่ 6.2 เส้นทางการเดินทาง

หุ่นยนต์แต่ละตัวจะได้รับมอบหมายงานให้ไปที่โหนดหนึ่งโดยสุ่มจาก job generator โดยจะเลือกหุ่นยนต์ตัวที่ว่าง job generator จะส่งคำสั่งไปให้หุ่นตัวที่ต้องการ เมื่อหุ่นยนต์รับคำสั่งแล้วจะเคลื่อนที่โดยมีข้อจำกัดว่าหุ่นยนต์จะต้องอยู่บนโหนด หรือบนลิงค์ ซึ่งโหนดหนึ่งๆอาจมีหุ่นยนต์อยู่มากกว่าหนึ่งตัว แต่จะมีหุ่นยนต์อยู่บนลิงค์ มากกว่าหนึ่งตัวไม่ได้ ในการเดินทางผ่านลิงค์ จะต้องใช้ระยะเวลาในการเดินทางช่วงหนึ่ง หากมีหุ่นยนต์มากกว่า 1 ตัวต้องการเดินทางผ่านเส้นทางเดียวกัน จะมีหุ่นยนต์เพียงตัวเดียวที่ได้เดินก่อนส่วนตัวที่เหลือจะต้องรอที่โหนดจนกว่าหุ่นตัวแรกผ่านลิงค์ไปถึงโหนดก่อน ซึ่งและการทำงานโดยรหัสเทียมดังนี้

```
Job generator :  random robot number
                 if robot is not busy then
                         random job and send to robot

Robot :          waiting for job from  job generator
                 searching route
                 move to target position
```

6.2 โครงสร้างข้อมูล และการแก้ปัญหา

6.2.1 เส้นทาง

จากรูปแผนที่การเดินทางข้างต้น เมื่อจำแนกตามโหนด และลิงค์ที่เชื่อมต่อกับโหนดนั้นๆ จะพบว่าโหนดทั้งหมดมี 10 โหนด และแต่ละโหนดมีลิงค์ที่เชื่อมต่อไม่เกินสามลิงค์ได้ เราจะให้ความสัมพันธ์ระหว่างโหนดโดยกำหนดว่าโหนดนั้น เชื่อมกับโหนดใด

Node	connect link
1	2 4 5
2	1
3	6 7
4	1 8
5	1 6 9
6	3 5 10
7	3 10
8	4
9	5 10
10	6 7 9

ในระบบมีทั้งหมด 10 โหนด และลิงค์ ที่เชื่อมต่อโหนดหนึ่ง มีไม่เกิน 3 ลิงค์ ดังนั้นเราจะแทน เส้นทางด้วยตัวแปร array สองมิติ คือ

Route = array [1..Maximum node , 1..Maximum link per node]

แต่เนื่องจากภาษาที่ออกแบบไว้จะใช้ตัวแปรแบบอะเรย์ได้เพียงมิติเดียว ดังนั้นจะแทนเส้นทางด้วย

*Route = array [1 to Max. node * Max. link per node]*

ในกรณีที่โหนดใดมีเส้นทางเชื่อมต่อน้อยกว่าสามลิงค์ จะให้ค่าลิงค์ที่ว่างอยู่มีค่าเท่ากับศูนย์

6.2.2 การค้นหาเส้นทางการเดินทาง

ในการค้นหาเส้นทางจากจุดเริ่มต้นไปสู่เป้าหมาย จะใช้การค้นหาแบบ recursive โดยเริ่มค้นหาจากตำแหน่งเป้าหมายว่ามีโหนด ใดเชื่อมต่อกับเป้าหมายบ้าง เมื่อได้โหนด ที่เชื่อมต่อกับตำแหน่งเป้าหมาย ก็จะ recursive เพื่อค้นหาตำแหน่งที่เชื่อมต่อกับโหนดนั้นๆ จนกว่าจะพบ

จุดเริ่มต้น เช่น หากต้องการ ค้นหาเส้นทางจากโหนด 2 ไปโหนด 9 จะเริ่มจากค้นว่ามีโหนดใดเชื่อมกับโหนด 9 จากตารางเราพบว่าโหนด 9 เชื่อมกับโหนด 5 และจะค้นหาต่อไปว่าโหนดใดเชื่อมกับโหนด 5 ทำเช่นนี้เป็นจนกว่าจะพบโหนด 2

```
Search(2,9)     --->    Search(5,9)
                        Search(1,5)
                        Search(2,1)
                        Search(2,2)
```

เส้นทางที่ได้คือค่าของเป้าหมายในการ recursive แต่ละครั้ง

เส้นทางที่ได้คือ 2 , 1 , 5 , 9

เขียนเป็น pseudo ดังนี้

```
pseudo code : SearchAndmove(A, B)
        IF (A =B) THEN   RETURN
        SearchAndMove( A , all node which connect to B);
        IF found solution THEN move from current position to B
```

6.2.3 สถานะของลิงค์

เนื่องจากในลิงค์ ใดๆ จะมีหุ่นยนต์ได้ไม่เกินหนึ่งตัวดังนั้นเราต้องสร้างตารางในการเก็บค่าสถานะของลิงค์ ดังรูป

node	1	2	3	4	5	6	7	8	9	10
10										
9										
8										
7										
6										
5										
4			2							
3										
2										
1										

รูปที่ 6.3 link status

เราจะแทนสถานะของลิงค์ ด้วย array สองมิติ

Link status = array [1..max node , 1..max node]

โดยเมื่อลิงค์ว่างจะแทนค่า 0 และเมื่อหุ่นยนต์ตัวใดจะจองถนนจะเซ็ตค่า link status ด้วยค่า robot no เช่นหากหุ่นยนต์ตัวที่สอง กำลังเดินทางอยู่ระหว่างโหนดที่ 3 และโหนดที่ 4 จะแทนค่าใน link[3,4] ด้วยค่า 2 เพื่อป้องกันหุ่นตัวอื่นเข้ามาในลิงค์นี้

6.2.4 การเดินจากโหนดหนึ่งไปสู่โหนดหนึ่ง

คำสั่ง moveเป็นคำสั่งที่ใช้ในการเคลื่อนที่จากโหนดหนึ่งไปสู่โหนดหนึ่ง โดยก่อนการเดินทางผ่านลิงค์ใดๆ จะต้องตรวจสอบก่อนว่าลิงค์นั้นว่างหรือไม่ หากลิงค์ว่างจะเดินทางได้ หากลิงค์ไม่ว่างจะต้องรอจนกว่าจะว่าง ซึ่งเขียนเป็นรหัสเทียม ดังนี้

```
pseudo code: Move from A to B
        while (current position <> B)
            enter critical region
                IF GetLinkStatus from A to B = ready
                current position = B
            exit critical region
```

6.2.5 Job generator

จะสร้างคำสั่งให้หุ่นยนต์แต่ละตัวเคลื่อนที่ โดย job generator จะเลือกหุ่นยนต์ ขึ้นมาหนึ่งตัวและจะตรวจสอบว่าหุ่นยนต์ตัวนี้ว่างหรือไม่ หากว่าก็จะสุ่มตำแหน่งและส่งข้อมูลให้ทำงานตามคำสั่งต่อไป

```
pseudo code : Job generator
        loop forever
            Robot no = random robot number
            if robot is ready
                TARGET = random position
                send TARGET to robot
```

6.2.6 ชุดคำสั่งของหุ่นยนต์

หุ่นยนต์ตัวที่ว่างจะรอรับคำสั่งจาก job generator เมื่อหุ่นยนต์ได้รับคำสั่ง จะเปลี่ยนสถานะจากว่างเป็นกำลังทำงาน และจะเริ่มเดินทางจากตำแหน่งปัจจุบันไปสู่จุดหมาย เมื่อถึงจุดหมายแล้วจะเปลี่ยนสถานะจากกำลังทำงานเป็นสถานะว่าง เพื่อรอคำสั่งต่อไป

```
pseudo code : Robot_n
    loop forever
        wait for job from job generator
        set status to busy
        search and move
        set status to ready
```

6.3 เปรียบเทียบกับภาษาธรรมดา

เมื่อเปรียบเทียบการ implement ปัญหานี้ เมื่อโดยใช้ภาษาที่มีความสามารถประมวลผลพร้อมกันกับภาษาที่ประมวลผลแบบปกติในภาษาธรรมดา (ในตัวอย่างนี้จะแสดงโดยใช้ภาษา pascal) เราจำเป็นต้องเพิ่มส่วนของโปรแกรมเพื่อให้โปรแกรมย่อยของหุ่นยนต์ทำงานพร้อมๆกัน ในการจัดการให้โปรแกรมหุ่นยนต์ทั้งสามทำงานพร้อมกันจำเป็นต้องกำหนดให้มีคำสั่งเพื่อใช้ในการสลับการทำงานของโปรแกรมย่อยต่างๆ แทรกอยู่ในโปรแกรมย่อยของหุ่นยนต์ ดังนี้

```
type    process = (R1,R2,R3);
var     LastExecution = process
Procedure ContextSwitch;
begin
    case LastExecution of
        R1: LastExecution := R2;
        R2: LastExecution := R3;
        R3: LastExecution := R1;
    end; {case}
    RobotNo := integer(R1);
    case status [ RobotNo] of
        ready   : GenJob(RobotNo);
        busy    : SearchAndMove(RobotNo,A,B);
        move    : Move(RobotNo,A,B);
        wait    : DoNop
    end; {case}
end;
```

โดย procedure Context Switch จะทำหน้าที่ในการตรวจสอบว่าการประมวลผลครั้งสุดท้ายประมวลผลโปรแกรมใด แล้วจะสั่งให้โปรแกรมย่อยถัดมาได้ประมวลผลตามลำดับ เช่นหากเดิมหุ่นยนต์หมายเลขหนึ่งประมวลผล จะสลับมาประมวลผลหุ่นยนต์หมายเลขสอง จากนั้นจะพิจารณาว่าขณะนี้หุ่นหมายเลขสองกำลังอยู่ในสถานะใดแล้ว จึงจะประมวลผลตามสถานะนั้นๆ

ในขณะที่ประมวลผลฟังก์ชันใดๆ หากต้องการให้หุ่นยนต์ตัวอื่นประมวลผลด้วยเราต้องใส่คำสั่ง ContextSwitch เพื่อเข้าไปในทุกๆฟังก์ชัน เช่นในฟังก์ชัน move เป็นโปรแกรมย่อยที่จะตรวจสอบว่าเส้นทางที่ต้องการเคลื่อนที่ว่างหรือไม่ หากว่างจะสั่งให้หุ่นยนต์เคลื่อนที่ หากในขณะนั้นต้องการสลับการทำงานไปที่หุ่นยนต์ตัวอื่นเราจะต้องแทรกคำสั่ง context switch เข้าไป

```
procedure Move;
begin
    While CurrentPosition[RobotNo] <> To[RobotNo]  do
        if GetLinkStatus(From[RobotNo],To[RobotNo])=ready then
        begin
            SetLinkStatus(A,B,Busy);
            CurrentPosition[RobotNo] := To[RobotNo];
            SetLinkStatus(A,B, Ready)
        end;
    ContextSwitch;
end;
```

เมื่อเขียนด้วย concurrent language เราไม่จำเป็นต้องควบคุมการทำงานพร้อมกัน เพราะว่า อินเตอร์พรีเตอร์จะสลับกระบวนการให้เองโดยไม่ต้องเขียนโปรแกรมย่อย context switch ในส่วนของโปรแกรมย่อยใดๆที่ต้องการให้ทำงานแบบพร้อมกันจะใช้คำสั่ง process เช่นหากต้องการให้กระบวนการของหุ่นยนต์เคลื่อนที่พร้อมๆกัน

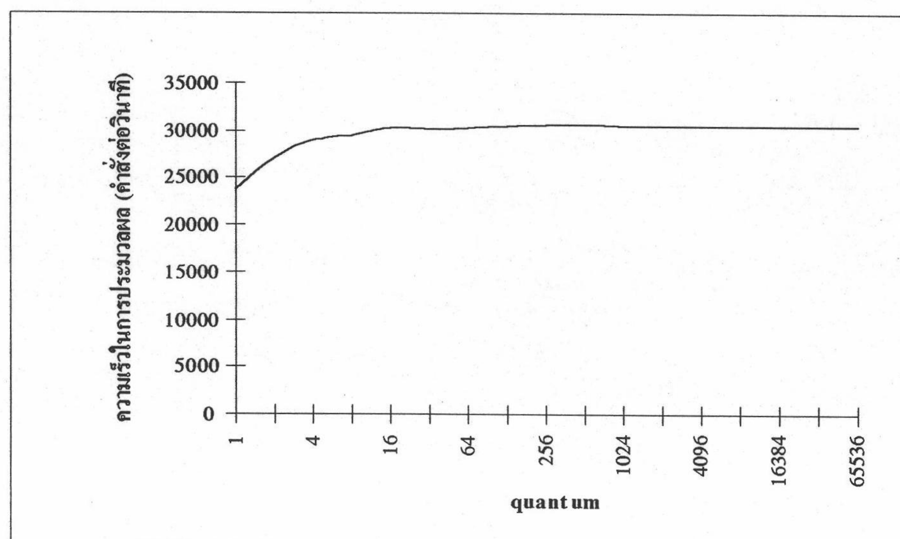
```
function move( RobotNo,A,B ) {
        while (CurrentPos[RobotNo] <> B)
                if (GetLinkStatusFrom( A,B )==Ready) {
                        SetLinkStatus(A,B,Busy);
                        CurrentPos[RobotNo] = B;
                        SetLinkStatus(A,B,ready);
                }
}

Process Robot1()        {
        move(RobotNo,A,B);
}

Process Robot2()        {
        move(RobotNo,A,B);
}
```

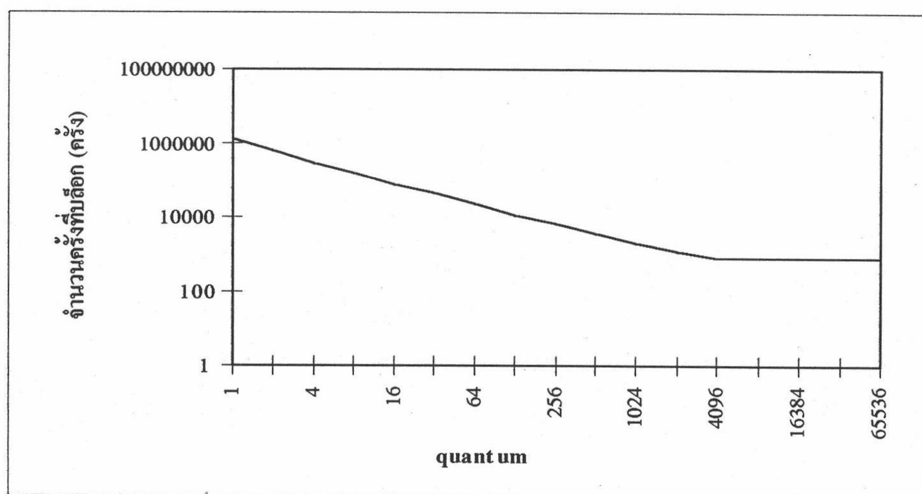
6.4 ผลกระทบจากการเปลี่ยนเวลาควอนตัม

จากการทดลองวัดเวลาและจำนวนคำสั่งที่ถูกประมวลผล โดยทำการทดลองบนเครื่องIBM-PC compatible ซึ่งมีไมโครโปรเซสเซอร์ 80486 ความเร็ว 33 Mhz และ RAM 8 Mbyte โดยเปลี่ยนแปลงจำนวนควอนตัมที่แตกต่างกัน จะผลการทดลอง



รูปที่ 6.4 กราฟแสดงความสัมพันธ์ระหว่างความเร็วกับควอนตัม

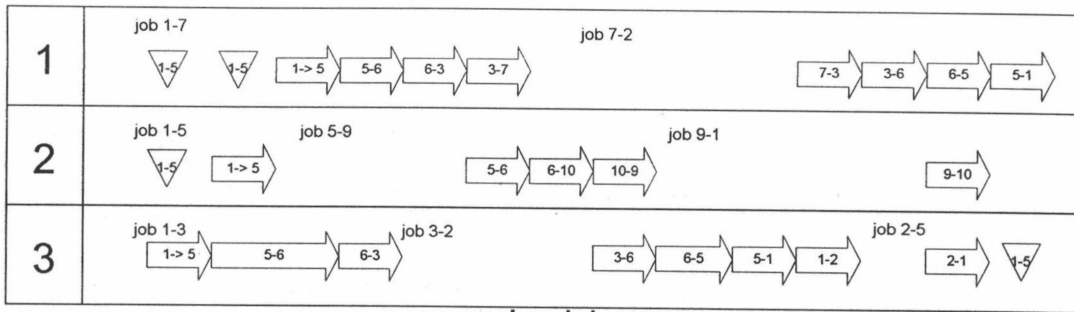
จากผลการทดลอง พบว่าจำนวนควอนตัมที่เปลี่ยนแปลงไป มีผลทำให้ประสิทธิภาพของระบบเปลี่ยนแปลง โดยเมื่อจำนวนควอนตัมมีค่าน้อย อินเตอร์พรีทเตอร์จะใช้เวลาในการจัดลำดับกระบวนการมาก เช่นค่าควอนตัมมีค่าเป็นหนึ่ง ในทุกๆคำสั่งที่ประมวลแล้วจะถูกบล็อกทันที ทำให้เสียเวลาในการเปลี่ยนให้กระบวนการอื่นเข้ามาประมวลผลแทน ดังนั้นการเพิ่มควอนตัมทำให้ประสิทธิภาพในด้านความเร็วของอินเตอร์พรีทเตอร์มีค่าสูงขึ้น และจำนวนครั้งที่ถูกบล็อกน้อยลง



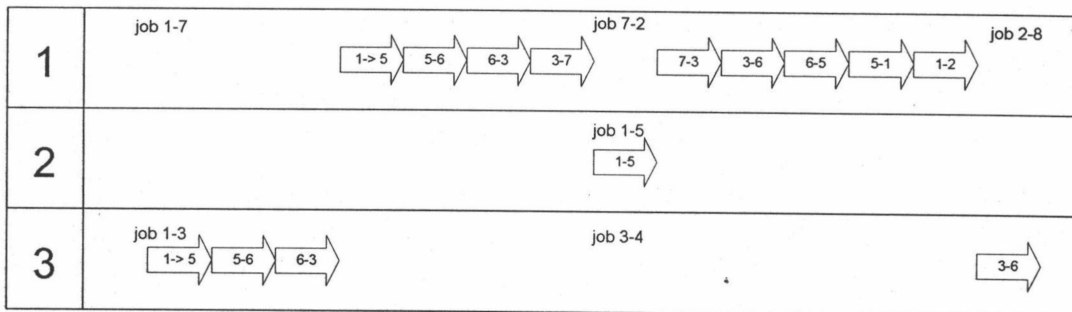
รูปที่ 6.5 กราฟแสดงความสัมพันธ์ระหว่างจำนวนครั้งที่บล็อกกับควอนตัม

เมื่อเพิ่มควอนตัมมากขึ้น (ระหว่าง 1 ถึง 5000 ควอนตัม) จำนวนครั้งที่ถูกบล็อกจะลงลด แต่เมื่อเพิ่มจำนวนควอนตัมขึ้นไปอีก (มากกว่า 5000 คำสั่ง) จำนวนครั้งที่กระบวนการที่ถูกบล็อกจะมีค่าใกล้เคียงกัน (ประมาณ 794 ครั้ง) ทั้งนี้เนื่องจากเมื่อควอนตัมมีค่ามากเกินไป กระบวนการจะไม่ถูกบล็อกเพราะ execution จนหมดควอนตัม แต่จะถูกบล็อกเพราะขั้นตอนในการป้องกันการใช้ทรัพยากรร่วม โดยโปรแกรมเอง

เมื่อนำผลการทดลองรันโปรแกรมการขนส่งโดยกำหนดให้ค่าควอนตัมมีค่าแตกต่างกันสองค่าคือควอนตัม 1 และ 1,000 จะได้ผลการทดลองดังนี้



แผนภาพการเคลื่อนที่เมื่อควอนตัม = 1



แผนภาพการเคลื่อนที่เมื่อควอนตัม = 1,000

▽ wait

⇨ move

รูปที่ 6.6 แผนภาพแสดงการเคลื่อนที่ของหุ่นยนต์

จากผลการรันโปรแกรมเมื่อกำหนดให้ควอนตัมมีค่าเท่ากับหนึ่ง หุ่นยนต์หมายเลขหนึ่งถูกมอบหมายให้เดินทางจากโหนด 1 ไปสู่โหนด 7(job 1-7 ในรูป), หุ่นยนต์หมายเลขสองถูกมอบหมายให้เดินทางจากโหนด 1 ไป โหนด 5 และ หุ่นยนต์หมายเลขสามถูกมอบหมายให้เดินทางจากโหนด 1 ไป โหนด 3 แต่เนื่องจากกระบวนการของหุ่นยนต์หมายเลขสามได้ประมวลผลก่อน หุ่นยนต์หมายเลขสามจึงเดินทางในเส้นทาง 1-5 (ซึ่งแสดงด้วยรูปลูกศรแทนการเคลื่อนที่) ในขณะที่กำลังเคลื่อนที่ เกิดการสลับลำดับงานไปที่หุ่นยนต์หมายเลขหนึ่งที่ต้องการเคลื่อนที่จากโหนด 1-5 เช่นกัน หุ่นยนต์หมายเลขสองจะต้องรอให้เส้นทาง 1-5 ว่างเสียก่อน (ซึ่งแสดงด้วยรูปสามเหลี่ยมแทนการรอคอย) เมื่อหุ่นยนต์หมายเลขสามเคลื่อนที่ถึงโหนด 5 เกิดการสลับกระบวนการพอดีทำให้หุ่นยนต์หมายเลขสองเคลื่อนที่จาก 1-5 ในขณะที่หุ่นยนต์หมายเลขสองกำลังเคลื่อนที่อยู่นั้น หุ่นหมายเลขหนึ่งก็เคลื่อนที่จาก 5-6 ดังรูป

เส้นทางการเดินทางของหุ่นยนต์หมายเลขหนึ่งคือ

1 - 5 - 6 - 3 - 7 - 3 - 6 - 5 - 1

เส้นทางการเดินทางของหุ่นยนต์หมายเลขสองคือ

1 - 5 - 6 - 10 - 9 - 10

เส้นทางการเดินทางของหุ่นยนต์หมายเลขสามคือ

1 - 5 - 6 - 3 - 5 - 1 - 2 - 5

เราจะพบว่ากระบวนการทุกกระบวนการสามารถประมวลผลขนานกันได้ โดยพิจารณาจากรูปในการเคลื่อนที่ในขณะเวลาหนึ่งๆจะพบว่าหุ่นแต่ละตัวสามารถเคลื่อนที่พร้อมๆกันได้

เมื่อเพิ่มค่าควอนตัมเท่ากับ 1,000 ควอนตัม จากรูปเราจะพบว่าหุ่นยนต์หมายเลขสามถูกมอบหมายให้เดินทางจากโหนด 1 ไป โหนด 3 และเคลื่อนที่ผ่านเส้นทาง 1-5 , 5-6 , 6-3 จนถึงเป้าหมายที่กำหนดแล้วเกิดการสลับกระบวนการไปที่หุ่นยนต์หมายเลขหนึ่ง หุ่นยนต์หมายเลขหนึ่งถูกมอบหมายให้เดินทางจากโหนด 1 ไป โหนด 7 จะเคลื่อนที่จาก 1-5 , 5-6 , 6-3 , 3-7 ตามลำดับจนถึงเป้าหมาย จะพบว่าหุ่นยนต์แต่ละตัวจะรับคำสั่งและเคลื่อนที่ไปสู่เป้าหมายโดยหุ่นตัวอื่นไม่ได้ประมวลผลเลย ดังนั้นการเพิ่มจำนวนควอนตัมแม้จะทำให้ความเร็วในการประมวลผลเร็วขึ้น แต่อินเตอร์พรีทเตอร์จะสูญเสียคุณสมบัติของโปรแกรมแบบขนานไป เพราะมีเพียงกระบวนการเดียวที่ได้ครอบครองทรัพยากรนานมาก จนทำให้กระบวนการอื่นเสมือนกับหยุดนิ่ง

6.5 การวัดเวลาที่ใช้ในการจัดลำดับงาน

ในการวัดเวลาที่ใช้ในการจัดลำดับกระบวนการ เราไม่สามารถวัดจากเวลาย่อยในการประมวลผลหนึ่งคำสั่ง ทั้งนี้เป็นเพราะฟังก์ชันที่ใช้ในการจับเวลาของโปรแกรมเป็นฟังก์ชันที่เรียกใช้ฟังก์ชันของดอส (dos function call) ซึ่งมีความละเอียดต่ำ ดังนั้นการวัดอาจทำได้โดยอ้อมคือ วัดเวลาในการประมวลผลของโปรแกรมหนึ่งซึ่งมีกระบวนการเดียว โดยจะทำการทดลองสองครั้งคือในครั้งแรกจะกำหนดให้อินเตอร์พรีเตอร์ประมวลผลโดยให้มีเวลาควอนตัมมีค่ามาก เพื่อไม่ให้เกิดการสลับกระบวนการเลย และทำการทดลองซ้ำอีกครั้งเพื่อโดยกำหนดให้ควอนตัมมีค่าเท่ากับหนึ่ง เพื่อให้มีจำนวนครั้งที่สลับกระบวนการจำนวนมาก

การทดลองนี้ในโปรแกรมเพื่อคำนวนหาค่าอนุกรม fibonaci ซึ่งได้แสดงโปรแกรมในภาคผนวก ได้ผลการทดลองดังนี้

จำนวนครั้งที่สลับกระบวนการ	เวลาที่ใช้ในการประมวลผล (วินาที)	เวลาที่ใช้ในการสลับกระบวนการ (วินาที)	เวลารวม (วินาที)
0	96.50	5.61	102.11
1,651,481	96.50	0	96.50

โดยประมวลผลทั้งสิ้น 1,651,481 คำสั่ง เราจะพบว่าในกรณีที่ช้าที่สุดคือมีการสลับกระบวนการทุกครั้งพบว่า

เวลาที่ใช้ในการสลับกระบวนการ = 5.61 (วินาที) / 1,651,481 (ครั้ง)

= 3.4 ไมโครวินาทีต่อครั้ง

เมื่อเทียบกับเวลาในการประมวลผลโดยรวม

= 5.61 (วินาที) / 102.11 (วินาที)

= 5.49 % ของเวลาประมวลผลรวม