

บทที่ 5 การประสานงานระหว่างโปรเซส (1)

วัตถุประสงค์ของเนื้อหา

- ศึกษาถึงกรณีประเด็นปัญหาส่วนวิกฤติ และเหตุผลของความจำเป็นในการใช้งานส่วนวิกฤติ
- ศึกษาขั้นตอนวิธีต่างๆ ที่ถูกนำมาใช้สร้างส่วนวิกฤติ
- ศึกษา system call ที่จำเป็นของระบบปฏิบัติการที่นำมาใช้สร้างส่วนวิกฤติ

สิ่งที่คาดหวังจากการเรียนในบทนี้

- นักศึกษาเข้าใจถึงเหตุผลของการใช้งานส่วนวิกฤติ และเข้าใจถึงกลไกการทำงานของส่วนวิกฤติ
- นักศึกษาเข้าใจถึงหลักการพื้นฐานของการทำงานของฮาร์ดแวร์คอมพิวเตอร์ และระบบปฏิบัติการ ในส่วนที่เกี่ยวข้องกับส่วนวิกฤติ
- นักศึกษาเข้าใจถึงการประยุกต์กลไกต่างๆ ของฮาร์ดแวร์ และระบบปฏิบัติการ เพื่อสนับสนุนทฤษฎีของส่วนวิกฤติ

วัตถุประสงค์ของปฏิบัติการท้ายบท

- นักศึกษานำเอาขั้นตอนวิธีต่างๆ ที่ได้เรียนรู้ ไปใช้สร้างส่วนวิกฤติ
- นักศึกษาประยุกต์กลไกของระบบปฏิบัติการในการจัดการส่วนวิกฤติ

สิ่งที่คาดหวังจากปฏิบัติการท้ายบท

- นักศึกษาเข้าใจและเห็นภาพถึงสถานะแข่งขัน และเข้าใจถึงกลไกการทำงานของส่วนวิกฤติ

เวลาที่ใช้ในการเรียนการสอน

- ทฤษฎี 2 ชั่วโมง
 - สภาวะแข่งขันและทฤษฎีพื้นฐานของส่วนวิกฤติ 0.5 ชั่วโมง
 - การสร้างส่วนวิกฤติด้วยกลไกทางซอฟต์แวร์ และชุดคำสั่งที่สนับสนุนโดยทางฮาร์ดแวร์ 1.5 ชั่วโมง
- ปฏิบัติ 2 ชั่วโมง
 - ศึกษาถึงกรณีประเด็นปัญหาส่วนวิกฤติ 0.5 ชั่วโมง
 - ศึกษาถึงการสร้างส่วนวิกฤติด้วยกลไกทางซอฟต์แวร์และฮาร์ดแวร์ 1.5 ชั่วโมง

ครั้งที่ 5 การประสานงานระหว่างโพรเซส

5.1 ปัญหาเกี่ยวกับการประสานงานระหว่างโพรเซส

ในการทำงานของคอมพิวเตอร์ที่มีหลายโพรเซสทำงานไปพร้อมๆ กัน ปัญหาที่เรามักจะพบก็คือเรื่องการทำอย่างไรให้โพรเซสที่ใช้ข้อมูลร่วมกันนั้น จะสามารถแลกเปลี่ยนข้อมูลซึ่งกันและกันโดยข้อมูลเหล่านั้นต้องแน่ใจว่าคือข้อมูลชุดเดียวกัน ซึ่งการกระทำดังกล่าวต้องอาศัยกลไกการให้โพรเซสทั้งสองที่ติดต่อกัน สอดประสานงานกันในการให้และรับข้อมูลอย่างถูกต้อง

ตัวอย่างเช่นในกรณีปัญหาผู้ผลิตและผู้บริโภค (consumer-producer problem) ที่ได้กล่าวมาในบทก่อนหน้า หากเราจัดการแบบ bounded buffer หรือมีขนาดพื้นที่เก็บข้อมูลที่จำกัด โดยใช้ circular queue ที่สามารถเก็บข้อมูลได้เพียง $n-1$ จากจำนวนหน่วยข้อมูลที่มีอยู่ทั้งหมด (เหตุผลดังกล่าว นักศึกษาได้เรียนมาแล้วในวิชาโครงสร้างข้อมูล ซึ่งถ้าเราจัดเก็บตามจำนวนที่มี จะพบปัญหาว่าเราจะไม่สามารถแยกสภาวะข้อมูลเต็มพื้นที่กับข้อมูลว่างออกจากกันได้) เพื่อความสะดวกรวดเร็วในการตรวจสอบว่าคิวเต็มหรือว่างอยู่หรือไม่ แทนที่เราจะต้องนำค่าตำแหน่งหัวคิว บวกหนึ่งแล้วหารหาเศษจากจำนวนข้อมูลที่มี แล้วไปเทียบกับตำแหน่งหาง เราอาจจะกำหนดตัวแปรนับเพิ่มอีกหนึ่งตัว เพื่อบอกจำนวนข้อมูลที่มีอยู่ ณ ปัจจุบันในคิว โดยตัวแปรนี้จะถูกเพิ่มค่าขึ้นหนึ่ง เมื่อชุดคำสั่งผู้ผลิต เพิ่มข้อมูลหนึ่งหน่วยลงในคิว และตัวแปรดังกล่าวนี้จะถูกลดค่าลงหนึ่ง เมื่อชุดคำสั่งผู้บริโภค อ่านข้อมูลออกมาจากคิว ดังเช่น

ผู้ผลิต (Producer)

```
while (true){
    //Block if buffer is full
    while(counter==BUF_SIZE);
    buffer[head] = newData;
    head = (head+1)%BUF_SIZE;
    counter++;
}
```

ผู้บริโภค (Consumer)

```
while (true){
    //Block if buffer is empty
    while(counter==0);
    data = buffer[tail];
    tail = (tail+1)%BUF_SIZE;
    counter--;
}
```

จากขั้นตอนวิธีข้างต้น ข้อความสั่งที่ปรากฏนั้น เมื่อแปลออกมาเป็นภาษาเครื่องแล้ว จะใช้คำสั่งหลายคำสั่ง ส่งผลให้มีโอกาสที่ในขณะโพรเซสทั้งสองกำลังประมวลอยู่นั้น อาจเกิดการสลับการทำงานจากโพรเซสหนึ่งไปยังอีกโพรเซสหนึ่ง (ในกรณีที่มีซีพียูคอร์เดียว) หรืออาจกำลังทำงานไปพร้อมกันทั้งสองโพรเซส (ในกรณีที่มีซีพียูหลายคอร์) และจะมีโอกาสสูงที่ค่าในตัวแปรนับ (counter) อาจจะไม่มีการเปลี่ยนค่าจากโพรเซสหนึ่ง แต่อีกโพรเซสคาดหวังว่าค่าดังกล่าวจะถูกเปลี่ยนไปแล้ว หากสมมติว่าโพรเซสทั้งสองมีชุดคำสั่งในระดับภาษาเครื่องในการเพิ่มหรือลดค่าในตัวนับดังนี้

ผู้ผลิต (Producer) counter++

```
register1 = counter
register1 = register1 + 1
counter = register1
```

ผู้บริโภค (Consumer) counter--

```
register2 = counter
register2 = register2 - 1
counter = register2
```

สมมติว่าในขณะกำลังประมวลนั้น มีการสลับแบ่งแต่ละโพรเซสเข้ามาทำงานดังเช่น

```
S0: producer execute register1 = counter {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = counter {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute counter = register1 {count = 6}
S5: consumer execute counter = register2 {count = 4}
```

จะเห็นว่า ในขณะที่ชุดคำสั่งของผู้ผลิตอ่านค่าจากตัวแปรนับเพื่อนำมาเพิ่มค่า ชุดคำสั่งของผู้บริโภคก็อ่านตัวแปรนับดังกล่าวแทบจะพร้อมกัน แล้วนำไปประมวลผลต่อ ส่งผลให้การเปลี่ยนแปลงค่าของโปรเซสที่เปลี่ยนค่าในตัวแปรนับก่อนจะถูกทับโดยค่าที่เปลี่ยนโดยอีกโปรเซสหนึ่งซึ่งกระทำไปคนละทาง ลักษณะเช่นนี้เราเรียกว่าเป็น **สภาวะแข่งขัน (race condition)** ของสองโปรเซสที่พยายามจะเปลี่ยนแปลงค่าที่ใช้ร่วมกันไปคนละทาง การแก้ไขสภาวะเช่นนี้กระทำโดยในขณะที่โปรเซสหนึ่งกำลังจัดการกับตัวแปรที่ใช้ร่วมกันระหว่างโปรเซส จะต้องหยุดมิให้โปรเซสอื่นๆ เข้าถึงตัวแปรดังกล่าว จนกว่าโปรเซสนั้นๆ จัดการกับตัวแปรร่วมเสร็จสิ้น แล้วจึงปล่อยให้โปรเซสอื่นเข้าใช้ตัวแปรร่วมนั้นต่อไปในภายหลัง

อีกตัวอย่างหนึ่งที่เรามาจะเคยเห็นกันในโปรแกรมขนาดใหญ่ อย่างเช่นในเกมคอมพิวเตอร์ ที่เราพบบั๊กตรงจุดที่เราสั่งให้ companion ของเราหยิบของ ในขณะที่เราหยิบของชิ้นนั้นพร้อมกันในเวลาเดียวกัน เราอาจจะพบว่าทั้งเราและ companion อาจจะได้ของชิ้นนั้นมาทั้งสองฝ่าย ให้นักศึกษาลองพิจารณาตัวอย่างโค้ดสั้นๆ ต่อไปนี้

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

int stop;
int ball;

void *player(void *selector);

int main(void){
    pthread_t tid[2];           // Thread ID
    pthread_attr_t attr[2];     // Thread attributes
    int section[2]={0,1};
    stop = 0;
    ball = 0;
    char text[8];
    int round = 1;

    srand(time(NULL));

    pthread_attr_init(&attr[0]); // Get default attributes
    pthread_attr_init(&attr[1]); // Get default attributes

    // Create 2 threads
    pthread_create(&tid[0],&attr[0],player,(void *)&section[0]);
    pthread_create(&tid[1],&attr[1],player,(void *)&section[1]);

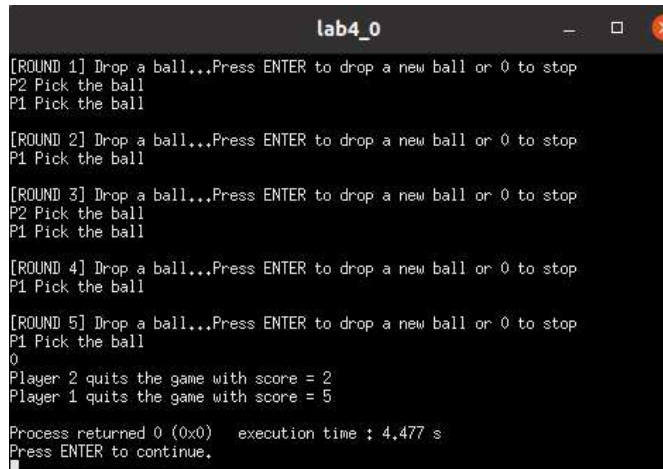
    while(1){
        printf("[ROUND %d] Drop a ball...",round);
        ball = 1;
        printf("Press ENTER to drop a new ball or 0 to stop\n");
        fgets(text,8,stdin);
        if(text[0]=='0')break;
        round++;
    }
    stop = 1;
    // Wait until all threads finish
    pthread_join(tid[0],NULL);
    pthread_join(tid[1],NULL);

    return 0;
}

void *player(void *selector){
    int no=(int)*((int *)selector);
    int score=0;

    while(1){
        if(ball>0){
            printf("P%d Pick the ball\n",no+1);
            usleep(rand()%10000);
            ball = 0;
            score++;
        }
        usleep(rand()%200000);
        if(stop) break;
    }
    printf("Player %d quits the game with score = %d\n",no+1,score);
    pthread_exit(0);
}
```

ในโค้ดตัวอย่าง เเรดหลักทำหน้าที่กำหนดค่า ball ให้เป็น 1 ในขณะที่เเรดลูกสองเเรดที่แตกขึ้นมาใหม่ ทำหน้าที่เป็นผู้เล่นสองคน ที่จะเฝ้าดูว่ามีบอลอยู่หรือไม่ หากมีจะหยิบบอลขึ้นมา ซึ่งในสภาพการใช้งานจริง อย่างเช่นในเกมคอมพิวเตอร์ ก็จะต้องแสดงอนิเมชันของการก้มไปหยิบบอล และอาจจะมึงานอื่นๆ ที่ต้องทำในช่วงระหว่างนั้น ในที่นี้จึงเลียนแบบโดยการใช้การเรียก usleep() เพื่อหน่วงเวลาเพียงเล็กน้อยระหว่างที่เห็นบอล และการหยิบบอล เมื่อเราลองรันโปรแกรมตัวอย่าง จะพบว่า มีบางครั้งที่มีผู้เล่นทั้งสองคนต่างได้บอลขึ้นมาทั้งคู่ ทั้งๆ ที่มีบอลเพียงลูกเดียวเท่านั้น



```

lab4_0
[ROUND 1] Drop a ball...Press ENTER to drop a new ball or 0 to stop
P2 Pick the ball
P1 Pick the ball

[ROUND 2] Drop a ball...Press ENTER to drop a new ball or 0 to stop
P1 Pick the ball

[ROUND 3] Drop a ball...Press ENTER to drop a new ball or 0 to stop
P2 Pick the ball
P1 Pick the ball

[ROUND 4] Drop a ball...Press ENTER to drop a new ball or 0 to stop
P1 Pick the ball

[ROUND 5] Drop a ball...Press ENTER to drop a new ball or 0 to stop
P1 Pick the ball
0
Player 2 quits the game with score = 2
Player 1 quits the game with score = 5

Process returned 0 (0x0)   execution time : 4.477 s
Press ENTER to continue.

```

ตัวอย่างข้างต้นเป็นอีกตัวอย่างของสภาวะแข่งขัน ที่แสดงให้เห็นถึงการที่มีเเรดหลายตัว พยายามแย่งชิงทรัพยากรซึ่งกันและกัน แต่ด้วยกลไกการทำงานของโปรแกรมดังที่ได้อธิบายข้างต้น จึงเกิดปัญหามีความผิดพลาดของข้อมูลเกิดขึ้น

5.2 ปัญหาส่วนวิกฤติ (Critical-Section Problem)

- สมมติว่ามีระบบคอมพิวเตอร์หนึ่งซึ่งมีโปรเซสทั้งหมด n โปรเซสคือ $\{p_0, p_1, \dots, p_{n-1}\}$
- แต่ละโปรเซสมีส่วนของชุดคำสั่งซึ่งถือว่าเป็นส่วนวิกฤติ ซึ่งมีลักษณะคือ
 - อาจใช้ทรัพยากรบางอย่างร่วมกัน เช่น การใช้ตัวแปรร่วมกัน ตารางข้อมูลร่วมกัน เขียนอ่านไฟล์เดียวกัน หรืออื่นใด
 - เมื่อโปรเซสหนึ่งทำงานไปถึงส่วนวิกฤติ โปรเซสอื่นจะไม่สามารถเข้าไปทำงานในส่วนวิกฤติดังกล่าวได้
- เกิดปัญหาที่เกี่ยวข้องกับส่วนวิกฤติ ซึ่งจะต้องมีการออกแบบขั้นตอนวิธีหรือกลไกในการแก้ไขปัญหาดังกล่าว
- ข้อกำหนดในขั้นต้น แต่ละโปรเซสเมื่อทำงานถึงจุดที่จะเป็นส่วนเริ่มต้นส่วนวิกฤติ (entry section) จะต้องร้องขออนุญาตในการที่จะเข้าส่วนวิกฤติ และเมื่อโปรเซสดังกล่าวทำงานภายในส่วนวิกฤติเสร็จแล้วก็จะแจ้งออกจากส่วนวิกฤติ (exit section) เพื่อไปทำงานในส่วนที่เหลือของโปรเซส (remainder section)
- การแก้ไขปัญหาส่วนวิกฤตินี้ จะยุ่งยากมากในกรณีของระบบปฏิบัติการที่รองรับ preemptive เนื่องจากโปรเซสหนึ่งๆ อาจจะสามารถถูกขัดจังหวะ หรือถูกปลดออกจากสภาวะ running ได้ และระบบคอมพิวเตอร์ที่มีซีพียูหลายหน่วย เพราะโปรเซสแต่ละตัวที่มีส่วนวิกฤติร่วมกัน อาจกำลังรันอยู่บนซีพียูคนละหน่วย การตรวจสอบและป้องกันไม่ให้แต่ละโปรเซสเข้าส่วนวิกฤติพร้อมกันจึงยากขึ้น

```

do {
    entry section
    critical section
    exit section
    remainder section
} while (TRUE);

```

การที่จะแก้ไขปัญหเกี่ยวกับส่วนวิกฤตินี้ จำเป็นต้องคำนึงถึงปัจจัยที่จำเป็นดังต่อไปนี้

1. จะมีเพียงโปรเซสเดียวเท่านั้นที่จะสามารถเข้าสู่ส่วนวิกฤติได้ (mutual exclusion)
2. จะต้องรับประกันความคืบหน้าในการทำงานของทุกๆ โปรเซสที่มีส่วนวิกฤติร่วมกัน (progress) หมายความว่า โปรเซสที่กำลังทำงานอยู่ในส่วน remainder section จะไม่สามารถไปแย่งชิงการเข้าสู่ส่วนวิกฤติพร้อมกับโปรเซสอื่นๆ ที่ยังไม่ได้เข้าไปทำงานในส่วนวิกฤติ (ยกเว้นแต่ว่าทุกโปรเซสได้พ้นจากส่วนวิกฤติไปหมดแล้ว) ทั้งนี้เพื่อป้องกันมิให้โปรเซสที่ผ่านส่วนวิกฤติไปแล้ว สามารถไปแย่งเข้าสู่ส่วนวิกฤติในการทำงานรอบถัดไป จนกระทั่งมีบางโปรเซสที่ยังมิได้เข้าสู่ส่วนวิกฤติ ไม่มีโอกาสได้เข้าไปทำงานในส่วนวิกฤติเลย (ทำให้โปรเซสนั้นไม่คืบหน้าในการประมวลผลนั่นเอง)
3. แต่ละโปรเซสจะไม่รอคอยการเข้าสู่ส่วนวิกฤติแบบไม่มีกำหนด (bounded waiting) ทั้งนี้เนื่องจากมีหลายโปรเซสที่กำลังรอเข้าสู่ส่วนวิกฤติ และมีเพียงโปรเซสเดียวในแต่ละช่วงเวลาเท่านั้นที่จะเข้าสู่ส่วนวิกฤติได้ โปรเซสหนึ่งจะต้องไม่รอเข้าสู่ส่วนวิกฤตินานเกินไปกว่าโปรเซสอื่นในจำนวนที่กำหนดค่าหนึ่ง (bound) ได้เข้าไปในส่วนวิกฤติก่อนที่โปรเซสนี้จะมีสิทธิเข้าไปบ้าง

ทั้งนี้เรานับว่า แต่ละโปรเซสจะใช้เวลาประมวลผลในระยะเวลาหนึ่ง (nonzero speed-ไม่ใช่ว่าทำงานเสร็จในทันที) และแต่ละโปรเซสที่รอนั้นน่าจะทำงานช้าหรือเร็วแตกต่างกันไปได้

ในทางปฏิบัติ โปรเซสของระบบปฏิบัติการนั้นมีโอกาสสูงที่จะเกิดสภาวะแข่งขัน เนื่องจากทรัพยากรต่างๆ ที่มีจำกัดหรือโครงสร้างการจัดการทรัพยากรที่ซ้ำซ้อนกัน โปรเซสที่ร้องขอบริการจากระบบปฏิบัติการ (system call) จึงมีโอกาสสูงที่จะเข้าสู่สภาวะแย่งชิงนี้ได้ อาทิเช่น โปรเซสแรกร้องขอเปิดไฟล์ในหน่วยความจำสำรอง ในขณะที่โปรเซสที่สองก็ร้องขอเปิดไฟล์ตัวเดียวกันตามปกติแล้ว การจัดการสภาวะแข่งขันแบบนี้เป็นหน้าที่ของระบบปฏิบัติการที่จะต้องถูกออกแบบให้แก้ไขสถานการณ์ดังกล่าว

ในส่วนของโปรเซสของเคอร์เนลที่ทำหน้าที่ให้บริการ (system call) จากโปรเซสของผู้ใช้นั้น ระบบปฏิบัติการอาจออกแบบโปรเซสที่กำลังทำงานอยู่ในโหมดของเคอร์เนล ในรูปแบบของ preemptive kernels ซึ่งโปรเซสดังกล่าวนี้อาจจะถูกถอด (preempted) จากซีพียู เพื่อให้โปรเซสอื่นที่ร้องขอบริการของเคอร์เนลตัวได้มีโอกาสได้ทำบ้าง หรืออาจจะเป็นแบบ nonpreemptive ซึ่งจะมีโปรเซสที่กำลังทำงานอยู่ในโหมดของเคอร์เนลเพียงตัวเดียวกำลังทำงานในระบบ โดยชุดคำสั่งที่รันในระดับเคอร์เนลจะต้องทำงานให้เสร็จสิ้นจนกลับมาสู่โหมดผู้ใช้ โปรเซสอื่นที่ร้องขอบริการจากระบบจึงจะสามารถเข้าสู่โหมดเคอร์เนลได้ ระบบปฏิบัติการบางตัวจะเป็นแบบ nonpreemptive kernel เช่น ลินุกซ์เวอร์ชัน 2.4 หรือก่อนหน้า แต่ระบบปฏิบัติการปัจจุบันจะเป็นแบบ preemptive kernel แล้วทั้งนี้เพื่อแก้ปัญหาโปรเซสที่ทำงานแล้วเกิดติดตายในเคอร์เนลโหมดไม่ส่งผลทำให้ระบบทั้งตัวหยุดการทำงาน และยังทำให้การตอบสนองของทุกๆ โปรเซสดีขึ้น (ไม่ต้องหยุดนานหากมีบางโปรเซสทำงานในโหมดเคอร์เนลนาน)

ปัญหาเช่นนี้ได้เกิดอยู่แค่ในระบบปฏิบัติการหลายภารกิจเท่านั้น แต่ในการเขียนซอฟต์แวร์แบบเดิมๆ ที่ฮาร์ดแวร์มีกลไกการอินเทอร์รัปต์ ก็มีโอกาที่จะเกิดปัญหาที่คล้ายคลึงกันนี้ได้กับฟังก์ชันจัดการอินเทอร์รัปต์ต่างๆ ที่อาจจะทำงานทับซ้อนกันขึ้นในระบบ ปัญหาอาจจะเกิดได้ระหว่างโปรแกรมที่กำลังรันอยู่ ณ ขณะนั้น กับฟังก์ชันจัดการอินเทอร์รัปต์ หรือระหว่างฟังก์ชันจัดการอินเทอร์รัปต์ด้วยกัน (ปัญหานี้เห็นได้ชัดยิ่งขึ้นในกรณีที่กำหนดให้คอมไพเลอร์ยูนิกซ์ -optimize ซึ่งคอมไพเลอร์จะยูนิกซ์โดยจัดเก็บค่าในตัวแปรที่เปลี่ยนแปลงบ่อย ณ ขณะนั้นไว้ในเรจิสเตอร์ และไม่ได้อัปเดตค่าในตัวแปรตลอดเวลา ทำให้เวลาเกิดอินเทอร์รัปต์ ฟังก์ชันจัดการ

อินเทอร์รัปต์เมื่ออ่านค่าในตัวแปรดังกล่าวนั้น จะได้ค่าที่ไม่อัปเดตล่าสุด -ตรงนี้ภาษาซีจึงมีค่าสวอน volatile ไว้ให้ใช้กับการนิยามตัวแปรที่มีความเสี่ยงต่อปัญหานี้ เพื่อแจ้งคอมไพเลอร์ให้แปลโค้ดในลักษณะที่ต้องอัปเดตค่าในตัวแปรทันทีเมื่อค่าถูกเปลี่ยน)

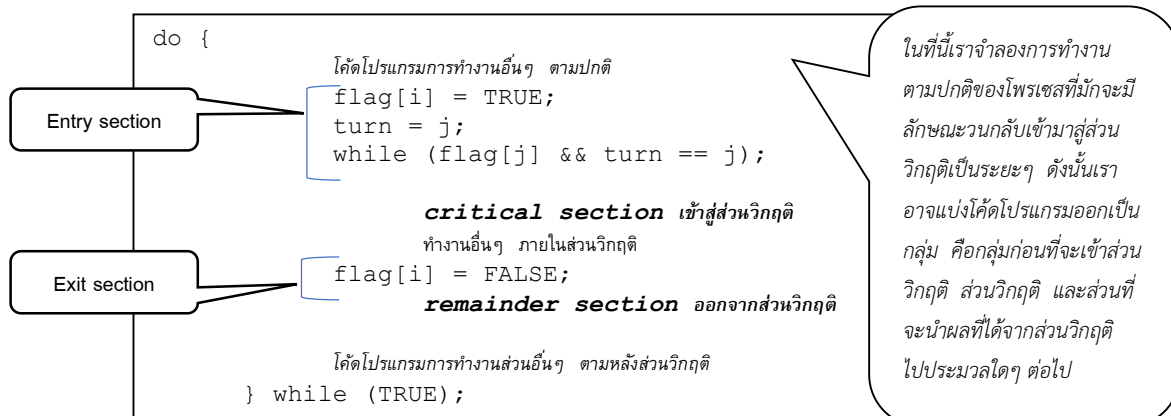
การแก้ไขปัญหาข้างต้นนี้ จึงต้องกลับมาจัดการที่กลไกการอินเทอร์รัปต์ ซึ่งอาจจะกระทำในลักษณะดังเช่น

- การหยุดการอินเทอร์รัปต์ชั่วคราว (disable interrupt) เมื่อจะจัดการทรัพยากรที่มีโอกาสแข่งขัน และอนุญาตให้เกิดอินเทอร์รัปต์ต่อได้ (enable interrupt) เมื่อพ้นช่วงดังกล่าวแล้ว
- แต่การแก้ไขปัญหาด้วยการยกเลิกอินเทอร์รัปต์เป็นเรื่องเสี่ยงพอสมควร โดยเฉพาะกับระบบปฏิบัติการปัจจุบันที่ขับเคลื่อนด้วยอินเทอร์รัปต์ ปัญหาที่เกิดขึ้นมาใหม่มีดังนี้
 - ระบบปฏิบัติการจะเข้ามาแทรกกลไกการทำงานได้อย่างไร หากโพรเซสไม่ยอมจบ CPU burst cycle ด้วยตนเอง (ตามกลไก non-preemptive ตามปกติ) (เพราะกลไกการทำ preemption ของระบบปฏิบัติการก็อาศัยการอินเทอร์รัปต์ด้วยเช่นกัน)
 - หากโพรเซสที่ยกเลิกอินเทอร์รัปต์ ใช้เวลาจัดการทรัพยากรนานมาก จนกระทั่งงานอื่นในระบบเสียหาย ไม่อัปเดตในช่วงเวลาที่เหมาะสม (เกิดความไม่ราบรื่นในการทำงานโดยรวม ไปจนถึงความเสียหายไม่ทันเวลาในระบบปฏิบัติการทันเวลา)
 - ในกรณีที่มีซีพียูมากกว่าหนึ่งตัว การยกเลิกอินเทอร์รัปต์จะต้องกระทำทุกตัวหรือไม่ แล้วจะสั่งการอย่างไร จะยกเลิกเพียงบางตัวได้หรือไม่ แล้วตัวใดจะมีผลกระทบบ้าง

5.3 วิธีการแก้ไขปัญหาที่นำเสนอโดยปีเตอร์สัน (Peterson's Solution)

แนวทางการแก้ไขปัญหของปีเตอร์สัน (Gary L. Peterson) ออกแบบมาสำหรับปัญหาการแย่งชิงทรัพยากรที่มีกลไกง่ายๆ ไม่ซับซ้อน แม้ว่าในปัจจุบันวิธีแนวนี้ไม่ใช่วิธีที่นำไปใช้งานแพร่หลายแล้ว แต่ก็นับเป็นวิธีที่น่าสนใจศึกษาวิธีหนึ่ง

- ใช้กับการสภาวะแข่งขัน (race condition) ระหว่างสองโพรเซส
- อนุমানว่าคำสั่งที่ใช้ในการนำค่าจากเรจิสเตอร์ในซีพียูไปเก็บในตัวแปร (store) และคำสั่งที่ใช้อ่านค่าจากตัวแปรมาใส่ในซีพียูเพื่อประมวลใดๆ ต่อไป (load) นั้นเป็น atomic หมายความว่า ซีพียูจะไม่สามารถหยุดชุดคำสั่งที่กำลังทำนี้ครึ่งทางเพื่อไปทำงานอื่นใดก่อนที่จะเสร็จสิ้นได้
- โพรเซสทั้งสองจะใช้ตัวแปรเหล่านี้ร่วมกัน
 - `int turn;`
 - `bool flag[2];`
- ตัวแปร turn จะเป็นตัวบ่งบอกว่า ณ ปัจจุบัน โพรเซสใดจะได้รับอนุญาตให้เข้าส่วนวิกฤติได้
- อะเรย์ flag ใช้ในการกำหนดว่าโพรเซสใดพร้อมที่จะเข้าส่วนวิกฤติ



อย่าลืมว่า... เวลาเราจะพิจารณาว่ากลไกการจัดการปัญหาการแย่งชิงนี้สามารถทำงานได้โดยสมบูรณ์ เราจะต้องมั่นใจว่า

- มีโพรเซสเดียวที่จะหลุดเข้าไปในส่วนวิกฤติได้ในเวลาใดเวลาหนึ่ง (mutual exclusion)
- แต่ละโพรเซสต้องสามารถทำงานต่อไปได้ (progress)
- หากหยุดรอเพื่อเข้าสู่ส่วนวิกฤติ จะต้องไม่ใช้การหยุดรอโดยไม่มีกำหนด (bounded waiting)

จากขั้นตอนวิธีข้างบน สมมติว่ามีสองโพรเซสคือ P_0 และ P_1 โดย $flag[0]$ ใช้โดย P_0 และ $flag[1]$ ใช้โดย P_1

ในจังหวะเริ่มต้น สมมติว่า $turn$ เป็น 0 หมายถึง P_0 ได้รับอนุญาตให้เข้าสู่ส่วนวิกฤติ เมื่อ P_0 ทำงานมาถึงโค้ดด้านบน $flag[0]$ จะถูกเซตเป็น true เป็นการบอกว่า P_0 พร้อมจะเข้าสู่ส่วนวิกฤติแล้ว จากนั้น P_0 เซตค่า $turn$ เป็น 1 เพื่อบอกว่า P_1 ได้รับอนุญาตให้เข้าสู่ส่วนวิกฤติ **ตามหลัง P_0** จากนั้น P_0 ตรวจสอบค่าใน $flag[1]$ ว่าได้แจ้งเข้าสู่ส่วนวิกฤติไปแล้วหรือไม่ (ซึ่งเป็นการบ่งบอกว่า ณ ขณะนี้ P_1 อาจกำลังอยู่ในส่วนวิกฤติ) และ $turn$ ว่าถูกเซตเป็นของอีกโพรเซสแล้วหรือไม่ ในจุดนี้เป็นการทดสอบเพื่อความมั่นใจว่า P_1 ต้องไม่อยู่ในส่วนวิกฤติ (หากสถานะทั้งสองไม่เป็นไปตามที่เข้าใจ คือ P_1 อาจจะเข้าไปอยู่ในส่วนวิกฤติแล้วเพราะ $flag[1]$ เป็นจริง และ ณ ขณะนี้เป็นคราวของ P_1 จะเข้าสู่ส่วนวิกฤติ P_0 จะหยุดรอ ณ จุดนี้) จากนั้นจึงเข้าสู่ส่วนวิกฤติ และที่ท้ายส่วนวิกฤติ P_0 จะเซตค่า $flag[0]$ เป็นเท็จ เพื่อบ่งบอกว่า ณ ขณะนี้ P_0 เสร็จสิ้นการทำงานในส่วนวิกฤติแล้ว

ในกรณีที่ทั้ง P_0 และ P_1 กำลังเข้าสู่ส่วนวิกฤติพร้อมกัน เราจะเห็นว่า $flag[0]$ และ $flag[1]$ ถูกเซตเป็นจริงทั้งคู่ แต่เมื่อโพรเซสทั้งสองทำงานไปถึงบรรทัด $turn=j$ ซึ่งใน P_0 เปลี่ยนค่าใน $turn$ เป็น 1 ในขณะที่ P_1 เปลี่ยนค่าใน $turn$ เป็น 0 (นี่คือกลไกการ store ของซีพียูนั่นเอง) เนื่องจากขั้นตอนการเปลี่ยนค่าในตัวแปร $turn$ จะกระทำแบบ atomic (คือซีพียูจะไม่สามารถขัดจังหวะไปทำงานอื่นในระหว่างการเปลี่ยนค่านี้ได้) หมายความว่า ผลของการเซตค่านี้จะไม่เกิดพร้อมกันจริงๆ แต่จะต้องมีโพรเซสใดโพรเซสหนึ่งเซตค่าในตัวแปรได้ก่อน ทำให้ตัวแปรที่มาทีหลังเซตค่าในตัวแปร $turn$ ไปให้กับโพรเซสก่อนหน้า ดังนั้นในจังหวะต่อไปที่เป็นการวนรอบรอบว่าสามารถเข้าสู่ส่วนวิกฤติได้หรือไม่นั้น จะมีเพียงโพรเซสเดียวเท่านั้นที่มีสิทธิเดินหน้าต่อเข้าไปทำงานได้นั่นเอง

ประเด็นปัญหาเกี่ยวกับสถาปัตยกรรมซีพียูสมัยใหม่

การสร้างส่วนวิกฤติโดยอาศัยวิธีของปีเตอร์สันนั้น อาจประสบปัญหาได้ในสถาปัตยกรรมคอมพิวเตอร์ในปัจจุบัน ด้วยเหตุผลต่างๆ อาทิเช่น

- คอมไพเลอร์มีกลไกการยุบคำสั่ง (optimization) ที่มีความซับซ้อนมากขึ้น โดยอาจเลือกสลับชุดคำสั่งที่คอมไพเลอร์คิดเองว่าจะไม่มีปัญหาในการคำนวณ ณ จุดดังกล่าว แต่ไม่ได้คำนึงถึงกรณีที่มีหลายโพรเซส/เธรด ที่อาจจะจัดการกับตัวแปรตัวเดียวกัน และการสลับชุดคำสั่งอาจส่งผลเสียหายได้
- ซีพียูในปัจจุบัน นำเอาคำสั่ง (instruction) ไปแปลงเป็นคำสั่งย่อย (micro-ops) และวงจรในซีพียูอาจจะสลับคำสั่งที่คิดว่าไม่มีความสัมพันธ์กันได้ (out-of-order execution) ซึ่งการกระทำดังกล่าวจะไม่ได้คำนึงถึงกรณีที่มีหลายโพรเซส/เธรด และอาจส่งผลต่อการจัดการส่วนวิกฤติได้

ลองพิจารณากรณีตัวอย่างต่อไปนี้

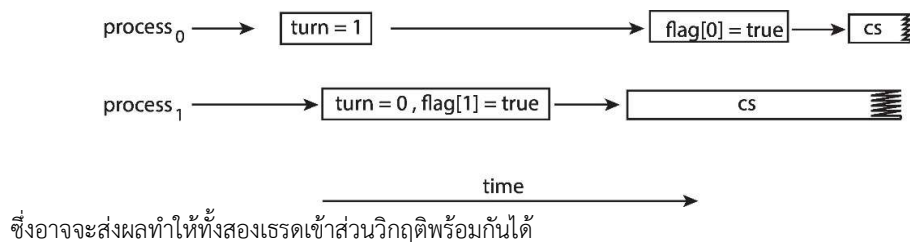
- สมมติว่ามีสองเธรดที่ใช้ตัวแปรร่วมกันคือ
 - `boolean flag = false;`
 - `int x = 0;`
 - `int y = 100;`

เธรด 1	เธรด 2
<code>while (!flag) ;</code>	<code>x = y;</code>
<code>print x;</code>	<code>flag = true;</code>

หากเรดทั้งสองทำงานตามปกติข้างต้น เราจะพบว่า เรด 1 จะหยุดรอที่ข้อความสั่ง while จนกระทั่งเรด 2 กำหนดค่า x ให้เป็น 100 แล้วจึงกำหนดค่า $flag$ ให้เป็นจริง และเมื่อเรด 2 ทำงานถึงจุดดังกล่าวแล้ว เรด 1 จึงจะทำงานต่อไปได้ นั่นหมายความว่า เรด 1 จะแสดง 100 บนจอภาพ

แต่ด้วยกลไกการสลับคำสั่งของคอมไพเลอร์ หรือกลไกการทำ out-of-order execution ของซีพียูสมัยใหม่ อาจส่งผลทำให้ข้อความสั่ง $flag = true$; ถูกดำเนินการก่อน $x = y$; (ด้วยเหตุที่การกำหนดค่า 1 ให้ตัวแปร จะใช้เวลาสั้นกว่าการเข้าถึงหน่วยความจำเพื่ออ่านค่า y มา ดังนั้นซีพียูยุคใหม่จึงเลือกที่จะสลับคำสั่งดังกล่าว) และด้วยเหตุนี้ เราจะพบว่าเรด 1 อาจจะหลุดก่อนที่เรด 2 กำหนดค่า y ให้กับ x และทำให้เรด 1 แสดงค่า 0 บนจอภาพ แทนที่จะเป็น 100

สำหรับกรณีของส่วนวิกฤติของปีเตอร์สัน หากดูลำดับการทำงาน เราอาจจะพบว่า $turn = j$ อาจจะถูกลบกลับกับ $flag[i] = true$ ด้วยความซับซ้อนของการทำงาน(ที่คำนวณตำแหน่งในอะเรย์เพื่อเข้าถึง) และจะทำให้ลำดับการทำงานจริงอาจจะเป็นดังนี้



การแก้ไขปัญหาที่เกิดขึ้นนี้ กระทำได้โดยการใช้กลไกที่เรียกว่าตัวคั่นหน่วยความจำ (memory barrier)

ตัวคั่นหน่วยความจำ (Memory Barrier)

ภายใต้กลไกการทำงานของซีพียูสมัยใหม่ การเข้าถึงหน่วยความจำของโปรเซส/เรด อาจจะสามารถสลับการทำงานได้ หากซีพียูตัดสินใจว่าคำสั่งที่อยู่ในลำดับต่อเนื่องกันนั้น ไม่ได้มีความสัมพันธ์ถึงกัน การจัดการการเข้าถึงหน่วยความจำอาจเป็นในลักษณะของ

- **Strongly ordered (การเรียงลำดับอย่างชัดเจน)** หมายถึงการเปลี่ยนแปลงข้อมูลที่เกิดโดยซีพียูคอร์/เรดหนึ่งๆ จะส่งผลให้ซีพียูคอร์/เรดอื่นๆ ที่อยู่ในระบบเห็นการเปลี่ยนแปลงได้โดยทันที
- **Weakly ordered (การเรียงลำดับอย่างไม่ชัดเจน)** หมายถึงการเปลี่ยนแปลงข้อมูลที่เกิดโดยซีพียูคอร์/เรดหนึ่งๆ ไม่จำเป็นที่จะทำให้ข้อมูลดังกล่าว(ที่เปลี่ยนไป) เห็นโดยซีพียูคอร์/เรด อื่นๆ ได้ในทันที

โดยอาศัยชุดคำสั่งตัวคั่นหน่วยความจำนี้ จะเป็นการสั่งให้ซีพียู(ที่รองรับ) ตรวจจับการเปลี่ยนแปลงนี้เพื่อให้ซีพียูคอร์/เรดอื่นๆ เห็นผลโดยทันที

- เป็นการรับประกันว่า การอ่านค่าหรือกำหนดค่ากับเรจิสเตอร์ หรือกับตัวแปร จะต้องเกิดขึ้นโดยทันที และเกิดก่อนการอ่านค่าหรือกำหนดค่ากับเรจิสเตอร์ หรือกับตัวแปรของชุดคำสั่งที่ตามมา
- เป็นการรับประกันว่า ค่าที่อัปเดตดังกล่าว จะถูกเห็นโดยซีพียูคอร์อื่นๆ เพื่อนำไปใช้ในกระบวนการอ่านค่า หรือกำหนดค่ากับตัวแปรที่เปลี่ยนไปก่อนหน้าตัวคั่นหน่วยความจำได้อย่างถูกต้อง

ซีพียูสมัยใหม่มีกลไกที่เรียกว่า *register renaming* ซึ่งส่งผลให้เมื่อคำนวณเสร็จสิ้นแล้ว ผลลัพธ์จะยังไม่ระบุเป็นคำตอบที่อ่านได้จากเรจิสเตอร์นั้นๆ โดยทันที (ด้วยเหตุที่คำสั่งย่อยนั้นอาจจะถูกประมวลผลก่อนที่จะควรกระทำจริง) ซีพียูจะอัปเดตค่าเรจิสเตอร์ให้เป็นตามจริงก็ต่อเมื่อถึงเวลาที่ควรกระทำ (เช่นถึงเวลาที่ควรทำคำสั่งย่อยนั้นๆ แล้ว) เท่านั้น

ลักษณะของการประยุกต์จะเป็นดังเช่น

เธรด 1 <pre>while (!flag) ; memory_barrier(); print x;</pre>	เธรด 2 <pre>x = y; memory_barrier(); flag = true;</pre>
---	--

```
ระบบปฏิบัติการวินโดวส์
#include <windows.h>
void MemoryBarrier();

ระบบปฏิบัติการลินุกซ์
#include<linux/membarrier.h>
int membarrier(int cmd,int flags);

cmd ที่ใช้บ่อยคือ
MEMBARRIER_CMD_QUERY ทดสอบดูว่ารองรับหรือไม่
MEMBARRIER_CMD_GLOBAL กระทำกับทุกๆเธรดของทุกโปรเซสในระบบ
flags (ยังไม่ใช้งาน) กำหนดให้เป็น 0 เสมอ
```

5.4 การประสานงานโดยใช้กลไกทางฮาร์ดแวร์ (Synchronization Hardware)

ซีพียูปัจจุบันถูกออกแบบมาเพื่อสนับสนุนรองรับการทำงานแบบหลายภารกิจ ดังนั้นจึงเกิดชุดคำสั่งใหม่ๆ เพื่อใช้ประกอบการสร้างส่วนวิกฤติ และอาศัยระบบปฏิบัติการและคอมไพเลอร์ที่รองรับ เราสามารถเข้าถึงชุดคำสั่งเหล่านี้ และนำมาใช้เพื่อสร้างส่วนวิกฤติได้เช่นกัน

สำหรับระบบปฏิบัติการที่มีซีพียูเพียงหน่วยเดียว (uniprocessor) ซึ่ง ณ เวลาใดเวลาหนึ่ง จะมีเพียงโปรเซสเดียว (หรือเธรดเดียว) ที่เข้าครอบครองซีพียูในเวลาใดเวลาหนึ่ง การเปลี่ยนโปรเซสเข้าทำงานนั้น เกิดขึ้นได้ใน 4 สถานะคือ

- 1) โปรเซสดังกล่าวผลการครอบครองซีพียูไปติดต่อกับ I/O
- 2) โปรเซสดังกล่าวถูกอินเทอร์รัปต์
- 3) โปรเซสที่กำลังติดต่อกับ I/O เสร็จสิ้นงานแล้วและพร้อมจะกลับมาทำงานต่อ และ
- 4) เมื่อโปรเซสจบการทำงาน

เราพบว่าในกรณีที่ 1) และ 4) นั้น ชุดคำสั่งของโปรเซสเองนั้นจะเป็นผู้พร้อมจะผลการครอบครองซีพียู (การเปลี่ยนโปรเซสเพียงกรณีที่เกิดขึ้นในจังหวะเหล่านี้จึงเรียกว่า nonpreemptive) แต่ในกรณีที่ 2) และ 3) นั้น ไม่มีองค์ประกอบใดในชุดคำสั่งของโปรเซสที่กำลังทำงาน ณ ปัจจุบันเป็นผู้สั่งการหรือพร้อมจะหยุดครอบครองเวลาซีพียู (การเปลี่ยนโปรเซสในจังหวะนี้จึงเป็นกรณี preemptive = pre-empt) ในทางปฏิบัติ กลไกดังกล่าวเกิดได้โดยใช้อินเทอร์รัปต์ กล่าวคือ เมื่อเกิดอินเทอร์รัปต์ขึ้น ซีพียูจะหยุดการทำงานและหันไปทำงานตามชุดคำสั่งที่ตอบสนองต่อการอินเทอร์รัปต์ ซึ่งชุดคำสั่งอินเทอร์รัปต์นี้เป็นส่วนหนึ่งของตัวระบบปฏิบัติการ การ**แทรกชุดคำสั่งยกเลิกอินเทอร์รัปต์**ในส่วนวิกฤติ จึงเพียงพอที่จะไม่ทำให้โปรเซสอื่นสามารถเข้ามาครอบครองเวลาของซีพียูได้ เป็นหลักประกันว่าโปรเซสที่ยกเลิกอินเทอร์รัปต์จะสามารถครอบครองซีพียูได้ตลอดไปจนกว่าจะอนุญาตให้อินเทอร์รัปต์เกิดขึ้นได้ในภายหลัง

แต่การส่งยกเลิกอินเทอร์รัปต์มีผลเสียหลายประการเช่น

- หากโปรเซสดังกล่าวเกิดทำงานผิดพลาดในช่วงการยกเลิกอินเทอร์รัปต์ คอมพิวเตอร์ทั้งระบบก็จะไม่สามารถทำงานได้ต่อไป โปรเซสอื่นๆ แม้แต่ระบบปฏิบัติการเองก็จะไม่สามารถกู้สภาพดังกล่าวได้ (ต้องบูตเครื่องอย่างดียว)
- การกระทำดังกล่าวส่งผลให้โปรเซสอื่นๆ ที่เกี่ยวข้องกับส่วนวิกฤติดังกล่าวต้องหยุดการทำงานตามไปด้วย และเป็นปัญหาย่างยิ่งกับระบบทันเวลา (real-time system)
- ในระบบคอมพิวเตอร์ที่มีซีพียูมากกว่าหนึ่งตัว การส่งการยกเลิกอินเทอร์รัปต์นั้นจะยุ่งยากมาก เพราะชุดคำสั่งการยกเลิกอินเทอร์รัปต์จะต้องกระจายส่งต่อไปยังซีพียูทุกหน่วยในระบบ และส่งผลให้การออกแบบระบบปฏิบัติการยุ่งยากและอาจจะไม่เหมาะสมต่อการขยายระบบหรือลดขนาดระบบ (เพราะมีจำนวนซีพียูและกลไกการส่งต่องานการเซตและยกเลิกการอินเทอร์รัปต์เข้ามาเกี่ยวข้อง)

ซีพียูที่มีใช้งานในระบบคอมพิวเตอร์ในปัจจุบัน จึงมีการออกแบบแก้ไขสภาพดังกล่าว ด้วยการกำหนดชุดคำสั่ง (instruction set) เรียกว่า **atomic instructions** ชุดคำสั่งเหล่านี้จะทำงานต่อไปแม้หากเกิดอินเทอร์รัปต์ขึ้นในระบบระหว่างทำงานชุดคำสั่งดังกล่าว และในกรณีที่มีซีพียูหลายคอร์ หากซีพียูคอร์หนึ่งกำลังทำ atomic instruction และซีพียูคอร์อื่นๆ จะทำ atomic instruction ด้วยในเวลาเดียวกัน ซีพียูคอร์อื่นๆ ก็จะต้องรอให้ atomic instruction นั้นจบก่อน แล้วจึงค่อยดำเนินการต่อได้

ชุดคำสั่งในกลุ่มนี้อาจแบ่งได้เป็นชุดคำสั่งในกลุ่ม Test-And-Set และ Compare-and-Swap กลไกของ Test-And-Set คือการตรวจสอบค่าในตัวแปรในหน่วยความจำหลักว่าเป็นจริงหรือเท็จ อ่านค่ามาเก็บไว้ และในขั้นตอนเดียวกันนั้นก็จะเซตค่าในตัวแปรให้เป็นจริงไปพร้อมกันเลย จากนั้นจึงส่งค่าที่อ่านได้ก่อนหน้ากลับ ขั้นตอนวิธีของ Test-And-Set เป็นดังนี้

```
bool TestAndSet(bool *target) {
    bool rv = *target;
    *target = TRUE;
    return rv;
}
```

ส่วนขั้นตอนวิธีของ Compare-And-Swap นั้น ตรวจสอบข้อมูลว่าตรงกับที่ต้องการหรือไม่ ดังนี้

```
bool CompareAndSwap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

การควบคุมแม่กุญแจโดยใช้ Test-And-Set

หลักการของการแก้ไขสถานะแย่งชิงเพื่อเข้าสู่ส่วนวิกฤติในลักษณะนี้ เราจะนิยามตัวแปรที่ใช้ร่วมกันระหว่างโปรเซสใดๆ โดยเรากำหนดให้ตัวแปรนี้เป็น**แม่กุญแจ (lock)** สำหรับใช้ไขเข้าสู่ส่วนวิกฤติ โดยในสถานะปกติ lock จะมีสถานะเป็น false เมื่อโปรเซสหนึ่งๆ จะเข้าสู่ส่วนวิกฤติ ก็จะตรวจสอบค่าใน lock พร้อมกับเซตค่าให้เป็นจริงไปพร้อมกันด้วย Test-And-Set ดังนั้นโปรเซสที่ตรวจสอบในจังหวะที่ lock เป็น false อยู่ก่อนแล้วเท่านั้น จึงจะมีโอกาสเข้าสู่ส่วนวิกฤติ แต่ถ้ามีสองโปรเซสหรือมากกว่านั้นพยายามเข้าสู่ส่วนวิกฤติพร้อมๆ กัน แต่ละโปรเซสจะพยายามทำคำสั่ง Test-And-Set ซึ่งมีลักษณะพิเศษที่ไม่มีโอกาสที่โปรเซสใดๆ จะประมวลคำสั่งนี้พร้อมกัน (ในกรณีของระบบที่มีหลายซีพียู และโปรเซสแต่ละตัวที่อยู่บนซีพียูแต่ละหน่วยพยายามทำคำสั่งนี้ ก็จะมีเพียงซีพียูหน่วยเดียว ในเวลาหนึ่งที่จะประมวลคำสั่งนี้ ซีพียูในหน่วยอื่นจะต้องรอ) ดังนั้นจะมีเพียงโปรเซสเดียวเท่านั้นที่

มีโอกาสเดินหน้าเข้าสู่ส่วนวิกฤติได้ และเมื่อโพรเซสจะออกจากส่วนวิกฤติ ก็จะเซตค่าใน lock ให้เป็น false เพื่อเปิดโอกาสให้โพรเซสอื่นๆ ที่รออยู่ หรือยังไม่ได้เข้าสู่ส่วนวิกฤติ สามารถมีโอกาสเข้าสู่ส่วนวิกฤติได้ต่อไป

```
do {
    ทำคำสั่งอื่นๆ ก่อนหน้าเข้าสู่ส่วนวิกฤติ
    while ( TestAndSet (&lock )); // do nothing
    // critical section
    ทำคำสั่งภายในส่วนวิกฤติ
    lock = FALSE;
    // remainder section
    ทำคำสั่งอื่นๆ หลังจากออกจากส่วนวิกฤติแล้ว
} while (TRUE);
```

ฟังก์ชันสำหรับใช้งานในลินุกซ์

```
type __sync_fetch_and_add (type *ptr, type value);
type __sync_fetch_and_sub (type *ptr, type value);
type __sync_fetch_and_or (type *ptr, type value);
type __sync_fetch_and_and (type *ptr, type value);
type __sync_fetch_and_xor (type *ptr, type value);
type __sync_fetch_and_nand (type *ptr, type value);
```

ฟังก์ชันสำหรับใช้งานในวินโดวส์

```
LONG InterlockedExchange(LONG volatile* Target, LONG Value); //แทนค่าด้วยค่าใหม่
LONG InterlockedExchangeAdd(LONG volatile* Addend, LONG Value);
LONG InterlockedAnd(LONG volatile* Destination, LONG Value);
LONG InterlockedOr(LONG volatile* Destination, LONG Value);
LONG InterlockedXor(LONG volatile* Destination, LONG Value);
```

การควบคุมแม่กุญแจโดยใช้ Compare-and-Swap

ในลักษณะทำนองเดียวกันกับการใช้ Test-And-Set แม้เราสามารถกำหนดค่าเริ่มต้นของแม่กุญแจให้เป็นค่าใดๆ และค่าที่เปลี่ยนแปลงให้เป็นค่าใดๆ ก็ได้เนื่องจากชุดคำสั่งมีความอ่อนตัวกว่า แต่ในที่นี้เราจะประยุกต์กลไกที่ใช้กับ Test-And-Set กับ Compare-and-Swap เข้ามาใช้แทน ดังนั้นค่าเริ่มต้นจะเป็น 0 (false) และค่าใหม่จะเป็น 1 (true) ก็จะได้โค้ดดังนี้

```
do {
    ทำคำสั่งอื่นๆ ก่อนหน้าเข้าสู่ส่วนวิกฤติ
    while ( CompareAndSwap (&lock, 0, 1)!=0); // do nothing
    // critical section
    ทำคำสั่งภายในส่วนวิกฤติ
    lock = 0;
    // remainder section
    ทำคำสั่งอื่นๆ หลังจากออกจากส่วนวิกฤติแล้ว
} while (TRUE);
```

ฟังก์ชันสำหรับใช้งานในลินุกซ์

```
type __sync_val_compare_and_swap(type *ptr, type oldval, type newval);
*ptr ตัวชี้รับค่าอ้างอิงของตัวแปรที่ดำเนินการ
oldval ค่าที่จะใช้เทียบว่าเท่ากันหรือไม่
newval ค่าใหม่ที่จะใส่แทนที่ กรณีเท่ากัน
```

ค่ากลับคืนคือค่าก่อนการดำเนินการ

ฟังก์ชันสำหรับใช้งานในวินโดวส์

```
LONG InterLockedCompareExchange(LONG volatile *Destination,  
                                LONG Exchange, LONG Comperand);
```

*Destination ตัวชี้รับค่าอ้างอิงของตัวแปรที่ดำเนินการ

Exchange ค่าใหม่ที่จะใส่แทนที่ กรณีเท่ากัน

Comperand ค่าที่จะใช้เทียบว่าเท่ากันหรือไม่

ค่ากลับคืนคือค่าก่อนการดำเนินการ

การปรับปรุงขั้นตอนวิธีเพื่อให้รองรับ bounded-waiting

การใช้ขั้นตอนวิธีที่กล่าวมาข้างต้นทั้งสองนั้น สามารถควบคุมให้มีเพียงโปรเซสเดียวสามารถเข้าสู่ส่วนวิกฤติได้ก็จริง แต่ยังมีปัญหาหลงเหลืออยู่ กรณีเช่นอาจจะกำลังมีโปรเซสทำงานอยู่สามตัว โปรเซสแรกอาจจะหลุดเข้าไปสู่ส่วนวิกฤติก่อน แล้วโปรเซสที่สองกับที่สามรออยู่ เมื่อโปรเซสแรกออกจากส่วนวิกฤติแล้ว โปรเซสส่วนที่สองอาจจะหลุดเข้าไปในส่วนวิกฤติบ้าง เมื่อโปรเซสที่สองหลุดจากส่วนวิกฤติ โปรเซสแรกอาจจะแย่งชิงโปรเซสที่สามเข้าสู่ส่วนวิกฤติไปอีก นั่นหมายความว่าโปรเซสที่สามอาจจะต้องรอนานกว่าปกติ ซึ่งในความเป็นจริงที่มีหลายโปรเซสแย่งชิงส่วนวิกฤติเดียวกัน อาจจะมีโอกาสที่มีบางโปรเซสไม่มีโอกาสจะได้เข้าไปสู่ส่วนวิกฤติเลย (no progression) หรืออาจจะใช้เวลารอ(โดนโปรเซสอื่นตัดหน้า)ไปโดยไม่สิ้นสุด (unbounded-waiting)

เพื่อแก้ปัญหาดังกล่าว จึงต้องมีการปรับขั้นตอนวิธีข้างต้นเพื่อให้หลักประกันว่า โปรเซสใดๆ จะต้องไม่โดนโปรเซสอื่นตัดหน้าแย่งชิงเข้าสู่ส่วนวิกฤติไม่เกินจำนวนครั้งที่กำหนด (bounded-waiting) ในขั้นตอนวิธีนี้ เรามีตัวแปรร่วมสำหรับใช้กับโปรเซสจำนวน n โปรเซสคือ

```
bool waiting[n];  
bool lock;
```

ในสภาวะเริ่มต้น สมาชิกทุกตัวในอะเรย์และตัวแปรจะถูกกำหนดค่าเริ่มต้นเป็นเท็จ (FALSE) ขั้นตอนวิธีจะมีลักษณะดังนี้

```
do {  
    คำสั่งต่างๆ ก่อนหน้าที่จะเข้าสู่ส่วนวิกฤติ  
  
    waiting[i] = TRUE;  
    key = TRUE;  
    while (waiting[i] && key)  
        key = TestAndSet(&lock);  
    waiting[i] = FALSE;  
    // critical section  
    คำสั่งต่างๆ ภายในส่วนวิกฤติ  
  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
  
    if (j == i)  
        lock = FALSE;  
    else  
        waiting[j] = FALSE;  
    // remainder section  
    คำสั่งต่างๆ หลังจากออกจากส่วนวิกฤติ  
} while (TRUE);
```

จากขั้นตอนวิธีข้างบน จำลองส่วนวิกฤติที่ใช้ร่วมกันระหว่าง n โปรเซส เมื่อโปรเซสใดพยายามจะเข้าสู่ส่วนวิกฤติ โปรเซสนั้นก็จะเช็คค่าสมาชิกประจำโปรเซสในอะเรย์ `waiting` ให้เป็น `TRUE` เป็นการประกาศว่าตนกำลังจะรอเข้าสู่ส่วนวิกฤติ จากนั้นไปเช็คค่า `key` ประจำตนให้เป็น `true` แล้ววนตรวจสอบทั้งค่าในสมาชิกประจำโปรเซสตนใน `waiting` และใน `key` ว่าเป็นจริงทั้งคู่หรือไม่ หากมีค่าใดค่าหนึ่งเป็นเท็จก็จะหลุดเข้าสู่ส่วนวิกฤติได้

ในกรณีปกติที่มีโปรเซสเพียงโปรเซสเดียวเท่านั้นเข้าสู่ส่วนวิกฤติ เราจะพบว่าค่าทั้งสองจะเป็นจริง และสามารถกระทำ `TestAndSet` หนึ่งครั้ง โดยเช็คค่าใน `key` ให้เป็นเท็จ (จากการได้ค่ามาจาก `lock`) และเช็คค่า `lock` ให้เป็นจริง เป็นการป้องกันไม่ให้โปรเซสอื่นๆ สามารถเข้าสู่ส่วนวิกฤติได้ และโปรเซสที่สามารถหลุดเข้าสู่ส่วนวิกฤติได้ จะเช็คค่าประจำสมาชิกในอะเรย์ `waiting` ให้กลับเป็น `false` ตามเดิม (แสดงว่าไม่ได้กำลังหยุดรอส่วนวิกฤติ เพราะเข้ามาได้แล้ว)

หลังจากที่โปรเซสทำงานตามที่ต้องการภายใต้ส่วนวิกฤติเสร็จสิ้น จะตรวจสอบโปรเซสอื่นๆ ที่เหลืออยู่ทั้งหมดว่ามีโปรเซสใดกำลังรออยู่บ้าง โดยสแกนไปในสมาชิกแต่ละตัวของอะเรย์ `waiting` ซึ่งโปรเซสที่กำลังรออยู่เท่านั้นที่จะมีค่าสมาชิกเป็น `true` จะเห็นว่าส่วนการตรวจว่าใครกำลังรออยู่นั้น จะหยุดก็ต่อเมื่อพบตัวที่กำลังรอในลำดับโปรเซสถัดไปทีพบเป็นตัวแรก หรือถ้าไม่พบ การวนเช็คจะจบโดยค่าวนตรวจสอบจะย้อนกลับมายังค่าหมายเลขโปรเซสตนเอง และขั้นตอนสุดท้ายก่อนการออกจากส่วนวิกฤติ จะพิจารณาว่าถ้าไม่มีโปรเซสอื่นใดกำลังรออยู่ ก็จะเช็คค่า `lock` ให้เป็น `FALSE` ตามปกติ (เข้าสู่สภาวะที่ไม่มีโปรเซสใดกำลังหยุดรอเลย) แต่ถ้ามีโปรเซสอื่นใดหยุดรอ (ซึ่งค่า `j` จะเป็นค่าหมายเลขประจำโปรเซสที่หยุดรอที่พบเป็นตัวแรกจากการสแกนไปเรื่อยๆ ทีละสมาชิก) ก็จะเช็คค่าสมาชิกของ `waiting` ประจำโปรเซสนั้นๆ ให้เป็นเท็จ ทำให้โปรเซสดังกล่าวที่หยุดรออยู่ใน `while` ลูปตอนนั้นสามารถหลุดออกจากลูปได้ทันทีเพื่อเข้าสู่ส่วนวิกฤติ

ดังนั้น ในสภาพการทำงานตามขั้นตอนวิธีดังกล่าว หากมีโปรเซสมาหยุดรอพร้อมๆ กันมากกว่าหนึ่งตัว เมื่อโปรเซสแรกที่หลุดเข้ามาได้นั้นหลุดออกจากส่วนวิกฤติ ก็จะส่งผลให้โปรเซสหมายเลขถัดไปเข้าสู่ส่วนวิกฤติได้ ในสภาพที่มีโปรเซสทุกตัวพยายามเข้าสู่ส่วนวิกฤติพร้อมกัน (ในสภาวะที่เลวร้ายสุด) เราจะเห็นว่าแต่ละโปรเซสจะได้รับโอกาสในการเข้าสู่ส่วนวิกฤติต่อเนื่องกันไปเรื่อยๆ ทีละหน่วยจนทุกตัวสามารถผ่านส่วนวิกฤติไปได้ ดังนั้นสภาวะหยุดรอดังกล่าวจึงมีค่าไม่เกิน $n-1$ ครั้ง

5.5 การประสานงานโดยใช้กลไกของระบบปฏิบัติการ

จากการประยุกต์ใช้กลไกของซีพียูที่จะได้เห็นในส่วนปฏิบัติการ จะพบถึงปัญหาการประยุกต์ใช้ในด้านต่างๆ และโดยเฉพาะในกรณีที่ต้องการพัฒนาซอฟต์แวร์โดยไม่ยึดฮาร์ดแวร์เฉพาะเป็นหลักด้วยแล้ว จะพบว่า การเข้าถึงชุดคำสั่งเฉพาะของซีพียูโดยตรงอาจไม่เหมาะสมนัก เนื่องจากซีพียูต่างสถาปัตยกรรมย่อมจะมีชุดคำสั่งที่แตกต่างกันออกไป

ในที่นี้เราจึงหันมาพิจารณาใช้งาน `system call` มาตรฐานที่มีให้ใช้ผ่านทางไลบรารีของคอมพิวเตอร์บนระบบปฏิบัติการต่างๆ แทน ซึ่งจะมีปัญหาในการประยุกต์ใช้งานน้อยกว่าการเข้าถึงชุดคำสั่งเฉพาะโดยตรง

การใช้งาน Mutex Locks

Mutex locks เป็นกลไกการล็อกทรัพยากรที่ถูกออกแบบมาให้ใช้งานได้ง่ายในระบบปฏิบัติการยุคใหม่

- นิยามตัวแปรล็อกเพื่อใช้สำหรับการล็อกทรัพยากรแต่ละตัว
- การจัดการส่วนวิกฤติกระทำโดยการ
 - `acquire()` ล็อก ในส่วนของ `entry section` และ
 - `release()` ล็อก ในส่วนของ `exit section`
- ระบบปฏิบัติการรับประกันกลไก `acquire()` และ `release()` เป็น `atomic` และดูแลจัดการเรื่อง `mutual exclusion`, `progress`, และ `bounded waiting`

```

while (true) {
    acquire lock

    critical section
    release lock

    remainder section
}

```

การใช้งาน Locking mechanism ในลินุกซ์

สำหรับลินุกซ์และยูนิกซ์นั้น กลไกการจัดการล็อกถูกนิยามอยู่ภายใต้ pthread.h และทำงานจัดการล็อกเฉพาะระหว่างเธรดต่างๆ ภายใต้โปรเซสเดียวเท่านั้น โดยมีฟังก์ชันมาตรฐานให้ใช้งานดังนี้

- spinlock คือกลไกการหยุดรอเพื่อเข้าสู่ส่วนวิกฤติ โดยเธรดใดที่รออยู่ จะวนลูปรอไปเรื่อยๆ จนกว่าจะได้ล็อกหรือถูกแฉมาเพื่อเข้าสู่ส่วนวิกฤติ ลักษณะแบบนี้เรียกว่า busy waiting ซึ่งหมายความว่าเธรดดังกล่าวจะวนเช็คสถานะล็อกอยู่ตลอดเวลา
 - pthread_spinlock_lock() ใช้ขาเข้า (entry section)
 - pthread_spinlock_unlock() ใช้ขาออก (exit section)
- Read/Write Lock คือกลไกการหยุดรอเพื่อเข้าสู่ส่วนวิกฤติที่จะแบ่งลักษณะการจัดการกับเธรดสองกลุ่ม ได้แก่ เธรดที่ทำหน้าที่เขียนข้อมูลลงในทรัพยากรร่วม (writer) และเธรดที่ทำหน้าที่อ่านข้อมูลจากทรัพยากรร่วม (reader) ในกรณีที่มีเธรดจำนวนมากเข้าใช้ทรัพยากรร่วมกัน แต่มีเพียงเธรดจำนวนน้อยที่เป็นผู้เขียน การให้ผู้อ่านรอคอยซึ่งกันและกันนั้นไม่จำเป็น เพราะไม่มีการเปลี่ยนแปลงข้อมูล ด้วยเหตุนี้ เธรดผู้อ่านจึงได้รับอนุญาตเข้าสู่ส่วนวิกฤติได้พร้อมๆ กัน แต่เธรดผู้อ่านจะเข้าพร้อมกันกับเธรดผู้เขียนไม่ได้ และเธรดผู้เขียนแต่ละตัวก็ไม่สามารถเข้าสู่ส่วนวิกฤติพร้อมกันได้
 - pthread_rwlock_rdlock() ใช้ขาเข้ากับเธรดอ่าน (read lock)
 - pthread_rwlock_wrlock() ใช้ขาเข้ากับเธรดเขียน (write lock)
 - pthread_rwlock_unlock() ใช้ขาออก
- Mutex Lock เป็นกลไกการหยุดรอที่ถูกพัฒนาขึ้นแทน spinlock โดยในกรณีของ mutex lock นั้น หากไม่สามารถล็อกได้จะ sleep โดยระบบปฏิบัติการจะกลับมาปลุกเธรดที่รอค้างไว้หากเธรดที่เข้าสู่ส่วนวิกฤติออกจากส่วนวิกฤติแล้ว
 - pthread_mutex_lock() ใช้ขาเข้า
 - pthread_mutex_unlock() ใช้ขาออก
- ยังมีฟังก์ชันมาตรฐานอีกตัวหนึ่งในลินุกซ์ ที่ใช้ปล่อยการครอบครองซีพียูเป็นการชั่วคราว
 - int pthread_yield(void);

เราสามารถเรียกฟังก์ชันนี้ร่วมกับกลไกการวนรอบเช็คค่า เพื่อรอเข้าสู่ส่วนวิกฤติที่เขียนโดยขั้นตอนวิธีทางฮาร์ดแวร์หรือซอฟต์แวร์ที่กล่าวมาในหัวข้อก่อนหน้านี้ได้ (ซึ่งจะได้ผลเทียบเท่ากับการใช้ sleep() เป็นเวลาสั้นๆ ที่ปรากฏในตัวอย่างโปรแกรมในปฏิบัติการท้ายบท)

การใช้งาน Locking mechanism ในวินโดวส์

กลไกที่น่าสนใจมีดังนี้

- Mutex lock กลไกการทำ Mutex lock เป็นกลไกที่สามารถใช้งานข้ามโปรเซสได้ (ใช้ named mutex lock) (ตัวอย่างในปฏิบัติการท้ายบทจะนำมาใช้ภายในโปรเซสเท่านั้น ซึ่งจะไม่มีการกำหนดชื่ออ้างอิงไว้)
 - ใช้ WaitForSingleObject() ณ ส่วนขาเข้า
 - ใช้ ReleaseMutex() ณ ส่วนขาออก

-
- criticalsection เป็นกลไกการทำ mutex lock ภายในโปรเซสเดียวกัน (ใช้งานระหว่างเธรด) ซึ่งจะใช้งานได้ง่ายกว่า
 - ใช้ EnterCriticalSection() ณ ส่วนขาเข้า
 - ใช้ LeaveCriticalSection() ณ ส่วนขาออก
 - สำหรับกลไกการปล่อยการครอบครองซีพียูของเธรด/โปรเซสนั้น อาจใช้ทางใดทางหนึ่งดังนี้
 - YieldProcessor()
 - Sleep(0)

ปฏิบัติการ

5.1 ตัวอย่างโปรแกรม Producer-Consumer problem ที่ใช้ตัวแปร counter ร่วมกัน กรณีของลินุกซ์

เพื่อให้เห็นผลของสภาวะแย่งชิงทรัพยากร (ในที่นี้คือตัวแปร counter) ได้ชัดเจนขึ้น ในที่นี้จึงได้สร้างฟังก์ชัน increment() และ decrement() ขึ้นมาแทนการใช้ ++ หรือ -- ซึ่งการดำเนินการทั้งสองเป็นการดำเนินการแบบ atomic (การใช้ตัวดำเนินการดังกล่าวจะทำให้ไม่เห็นผลเสียในกรณีเช่นนี้) และเพื่อเปิดโอกาสให้ระบบปฏิบัติการสามารถเข้าจัดการเปลี่ยนโปรเซสในการเข้าครอบครองซีพียูในจังหวะเวลาดังกล่าว จึงใช้ usleep เข้ามาเพื่อให้รอเวลา (ส่งผลสองประการคือมีโอกาสสูงขึ้นที่จะสลับการเข้าใช้ทรัพยากรระหว่างสองโปรเซส และในระบบปฏิบัติการส่วนมาก หากโปรเซสไม่ครองซีพียูนานเกินไปก็จะไม่เกิด context switch อย่างในกรณีตัวอย่างนี้ซึ่งชุดคำสั่งสั้นมาก)

ให้สังเกตผลการส่งค่าจาก Producer ไปยัง Consumer ดูว่ามีความผิดพลาดอย่างไรเกิดขึ้นได้บ้าง และจงอธิบายว่า ทำไมจึงเป็นเช่นนั้น

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h> // for usleep()
#include <stdlib.h> // for exit() and random generator
#include <wait.h> // for wait()
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <time.h>

#define BUF_SIZE 4

struct shmarea{
    int rp;
    int wp;
    int data[BUF_SIZE];
    int counter;
};

void randomDelay(void);
void consumer(struct shmarea *shmarea);
void producer(struct shmarea *shmarea);
int increment(int *counter);
int decrement(int *counter);

int main(){
    int shmid; // Share memory ID
    struct shmarea *shmarea; // Pointer to the shared segment
    int shmsize; // share segment size
    pid_t pid; // Child Process ID

    shmsize = sizeof(struct shmarea);
    // Allocate the shared block from OS
    shmid = shmget(IPC_PRIVATE, shmsize, 0666 | IPC_CREAT );

    // Attach the shared block to the pointer so we can use it.
    shmarea = (struct shmarea *)shmat(shmid, NULL, 0);

    shmarea->rp = shmarea->wp = shmarea->counter = 0;

    pid = fork(); //Fork a child process

    if(pid < 0){ //Fork error
        fprintf(stderr, "Fork failed.\n");
        exit(-1);
    }
    else if(pid==0){ // This is the path of child process
        consumer(shmarea);
    }
    else { // This is the path of parent process
```



```

        producer(shmarea);
        wait(NULL);
        printf("Child process has terminated\n");

        // Detach the shared memory segment
        shmdt(shmarea);
        // Remove the shared memory segment
        shmctl(shmid, IPC_RMID, NULL);
        exit(0);
    }
}

void consumer(struct shmarea *shmarea){
    int i=0;
    int remaintime;

    while(1) // consume data
    {
        remaintime=5000; // Wait some seconds before quitting
        // Wait 5 seconds if no data in the circular buffer
        while((shmarea->counter==0)&&(remaintime>0)){
            usleep(1000);remaintime--;
            if(!(remaintime%1000))
                printf("        Waiting for %d second(s).\n",remaintime/1000);
        }
        if(remaintime==0) return;

        printf("        Data number:%d = %d\n",i++,shmarea->data[shmarea->rp]);
        shmarea->rp = ((shmarea->rp+1)%BUF_SIZE);
        decrement(&(shmarea->counter));
    }
}

void producer(struct shmarea *shmarea){
    int i;

    for(i=0;i<32;i++) // produce data
    {
        while(shmarea->counter==BUF_SIZE) usleep(1000); //sleep for 1 ms if full
        printf("Enter data %d into share memory...\n",i);
        shmarea->data[shmarea->wp]=i;
        // move the write pointer so that the consumer know when to read.
        shmarea->wp = ((shmarea->wp+1)%BUF_SIZE);
        increment(&(shmarea->counter));
    }
}

void randomDelay(void){
    // This function provides a delay which slows the process down so we can see what happens
    srand(time(NULL));
    int stime = ((rand()%1000)+100)*1000;
    usleep(stime);
}

int increment(int *counter){
    // This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("INC: read\n");
    randomDelay();
    temp = temp +1;printf("INC: ++\n");
    randomDelay();
    *counter = temp;printf("INC: write\n");
    return temp;
}

int decrement(int *counter){
    // This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("        DEC: read\n");
    randomDelay();
    temp = temp -1;printf("        DEC: --\n");
    randomDelay();
    *counter = temp;printf("        DEC: write\n");
    return temp;
}

```

5.2 ตัวอย่างโปรแกรม Producer-Consumer problem ที่ใช้ตัวแปร counter ร่วมกัน กรณีของวินโดวส์

ในการทำงานเดียวกันกับตัวอย่างที่ 5.1 ในกรณีนี้เป็นการพอร์ตตัวอย่าง 5.1 เพื่อมาทำงานในลักษณะเช่นเดียวกันบนวินโดวส์

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#include <stdlib.h>
#include <time.h>

#define BUF_SIZE 4

struct shmarea{
    int rp;
    int wp;
    int data[BUF_SIZE];
    int counter;
};

void randomDelay(void);
void consumer(struct shmarea *shmarea);
void producer(struct shmarea *shmarea);
int increment(int *counter);
int decrement(int *counter);

void main(int argc, char *argv[]){
    TCHAR shmName[] = _T("lab5_2");
    struct shmarea *shmarea; // Pointer to the shared segment
    int shmsize = sizeof(struct shmarea); // share segment size
    HANDLE hMapFile;

    if(argc>1){ // If this is the child process it will run this instead
        hMapFile = OpenFileMapping(
            FILE_MAP_ALL_ACCESS, // read/write access
            FALSE, // do not inherit the name
            shmName); // name of mapping object

        if (hMapFile == NULL) {
            printf("Could not open file mapping object (%d).\n",
                GetLastError());
            return;
        }
        shmarea = (struct shmarea *) MapViewOfFile(hMapFile, // handle to map object
            FILE_MAP_ALL_ACCESS, // read/write permission
            0,
            0,
            shmsize);

        if (shmarea == NULL)
        {
            printf("Could not map view of file (%d).\n",
                GetLastError());
            return;
        }

        consumer(shmarea);

        UnmapViewOfFile(shmarea);
        CloseHandle(hMapFile);
        return;
    }
    // Create Memory Mapping File
    hMapFile = CreateFileMapping(
        INVALID_HANDLE_VALUE, // use paging file
        NULL, // default security
        PAGE_READWRITE, // read/write access
        0, // max. object size
        shmsize, // buffer size
        shmName); // name of mapping object

    if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE){
        printf("Could not create file mapping object...\n");
        return;
    }
}
```

```

STARTUPINFO si;
PROCESS_INFORMATION pi;
TCHAR name[] = _T("lab5_2.exe 1"); // Using the Same code with parameter
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

printf("Before fork a child process...\n");

if(!CreateProcess(NULL, name,
    NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)){
    fprintf(stderr, "Create process failed.\n");
    return ;
}

//Parent Process
// Map share memory
shmarea = (struct shmarea *) MapViewOfFile(hMapFile, // handle to map object
    FILE_MAP_ALL_ACCESS, // read/write permission
    0,
    0,
    shmsize);

if (shmarea == NULL)
{
    printf("Could not map view of file (%d).\n",
        GetLastError());
    return;
}

shmarea->rp = shmarea->wp = shmarea->counter = 0;
producer(shmarea);
printf("Before going into the wait state...\n");
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child process has terminated\n");
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);

UnmapViewOfFile(shmarea);
CloseHandle(hMapFile);
}

void consumer(struct shmarea *shmarea){
    int i=0;
    int remaintime;

    while(1) // consume data
    {
        remaintime = 1000; // Wait a second before quitting
        // Wait 5 millisecs if no data in the circular buffer
        while((shmarea->counter==0) && (remaintime>0)){
            Sleep(1);
            remaintime--;
            if(!(remaintime%1000))
                printf("        Waiting for %d second(s).", remaintime / 1000);
        }
        if(remaintime==0) return;

        printf("        Data number:%d = %d\n", i++, shmarea->data[shmarea->rp]);
        shmarea->rp = ((shmarea->rp+1)%BUF_SIZE);
        decrement(&(shmarea->counter));
    }
}

void producer(struct shmarea *shmarea){
    int i;

    for(i=0; i<32; i++) // produce data
    {
        while(shmarea->counter==BUF_SIZE) Sleep(1);
        printf("Enter data %d into share memory...\n", i);
        shmarea->data[shmarea->wp]=i;
        // move the write pointer so that the consumer know when to read.
        shmarea->wp = ((shmarea->wp+1)%BUF_SIZE);
        increment(&(shmarea->counter));
    }
}

void randomDelay(void){
    // This function provides a delay which slows the process down so we can see what happens

```

```

    srand(time(NULL));
    int stime = (rand()%1000)+100;
    Sleep(stime);
}

int increment(int *counter){
// This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("INC: read\n");
    randomDelay();
    temp = temp +1;printf("INC: ++\n");
    randomDelay();
    *counter = temp;printf("INC: write\n");
    return temp;
}

int decrement(int *counter){
// This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("    DEC: read\n");
    randomDelay();
    temp = temp -1;printf("    DEC: --\n");
    randomDelay();
    *counter = temp;printf("    DEC: write\n");
    return temp;
}

```

5.3 การแก้ไขปัญหาการแย่งชิงทรัพยากรร่วมกันโดยใช้ขั้นตอนวิธีของปีเตอร์สัน กรณีของลินุกซ์

ตัวอย่างต่อไปนี้เป็นการศึกษาการแก้ปัญหาที่นำเสนอโดยปีเตอร์สัน จะเห็นว่าโค้ดบางส่วนจะมีความซับซ้อนขึ้นบ้าง เหตุผลเพื่อแสดงให้เห็นสภาวะการทำงานของโปรแกรม และบางชุดเช่นส่วนของการวนรอก่อนเข้าส่วนวิกฤติ ได้เพิ่มฟังก์ชันวนรอการทำงาน เพื่อเปิดโอกาสให้ระบบปฏิบัติการเข้ามาทำ context switch ในจุดดังกล่าวได้

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h> // for usleep()
#include <stdlib.h> // for exit() and random generator
#include <wait.h> // for wait()
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <time.h>

#define BUF_SIZE 256

struct shmareatype{
    int rp;
    int wp;
    int data[BUF_SIZE];
    int counter;
    // Following section is for Peterson's solution
    int flag[2];
    int turn;
};

void randomDelay(void);
void consumer(struct shmareatype *shmarea);
void producer(struct shmareatype *shmarea);
int increment(int *counter);
int decrement(int *counter);

int main(){
    int shmid; // Share memory ID
    struct shmareatype *shmarea; // Pointer to the shared segment
    int shmsize; // share segment size
    pid_t pid; // Child Process ID

    shmsize = sizeof(struct shmareatype);
    // Allocate the shared block from OS
    shmid = shmget(IPC_PRIVATE, shmsize, 0666 | IPC_CREAT);

    // Attach the shared block to the pointer so we can use it.
    shmarea = (struct shmareatype *)shmat(shmid, NULL, 0);

    shmarea->rp = shmarea->wp = shmarea->counter = 0;

```

```

// Peterson's solution
shmarea->flag[0]=shmarea->flag[1]=0;
shmarea->turn = 0;

pid = fork(); //Fork a child process

if(pid < 0){ //Fork error
    fprintf(stderr,"Fork failed.\n");
    exit(-1);
}
else if(pid==0){ // This is the path of child process
    consumer(shmarea);
}
else { // This is the path of parent process
    producer(shmarea);
    wait(NULL);
    printf("Child process has terminated\n");

    // Detach the shared memory segment
    shmdt(shmarea);
    // Remove the shared memory segment
    shmctl(shmid, IPC_RMID,NULL);
    exit(0);
}
}

void consumer(struct shmarea_t *shmarea){
    int i=0;
    int remaintime;

    while(1) // consume data
    {
        printf("      (Consumer) Before entering critical section\n");
        // Peterson's ENTRY section
        shmarea->flag[0] = true;
        shmarea->turn = 1;
        while(shmarea->flag[1] && shmarea->turn==1) usleep(1000);
        // Critical Section
        remaintime=5000; // Wait some seconds before quitting
        while((shmarea->counter==0)&&(remaintime>0)){
            usleep(1000);remaintime--;
            if(!(remaintime%1000))
                printf("      Waiting for %d second(s).\n",remaintime/1000);
        }
        if(remaintime==0){
            printf("      The consumer process waiting too long, stopping...\n");
            shmarea->flag[0] = false; // EXIT here in case of waiting too long
            return;
        }else{
            printf("      (Consumer) Data number:%d = %d\n",i++,shmarea->data[shmarea->rp]);
            shmarea->rp = ((shmarea->rp+1)%BUF_SIZE);
            decrement(&(shmarea->counter));
        }
        // Peterson's EXIT section
        shmarea->flag[0] = false;
        // Remaining Section
        printf("      (Consumer) After exiting critical section\n");
    }
}

void producer(struct shmarea_t *shmarea){
    int i;

    for(i=0;i<32;i++) // produce data
    {
        printf("(Producer) Before entering critical section\n");
        // Peterson's ENTRY section
        shmarea->flag[1] = true;
        shmarea->turn = 0;
        while(shmarea->flag[0] && shmarea->turn==0) usleep(1000);
        // Critical Section
        while(shmarea->counter==BUF_SIZE) usleep(1000);
        printf("(Producer) Enter data %d into share memory...\n",i);
        shmarea->data[shmarea->wp]=i;
        // move the write pointer so that the consumer know when to read.
        shmarea->wp = ((shmarea->wp+1)%BUF_SIZE);
        increment(&(shmarea->counter));
        // Peterson's EXIT section
    }
}

```

```

        shmarea->flag[1] = false;
// Remaining Section
        printf("(Producer) After exiting critical section\n");
    }
}

void randomDelay(void){
// This function provides a delay which slows the process down so we can see what happens
    srand(time(NULL));
    int stime = ((rand()%1000)+100)*1000;
    usleep(stime);
}

int increment(int *counter){
// This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("INC: read\n");
    randomDelay();
    temp = temp +1;printf("INC: ++\n");
    randomDelay();
    *counter = temp;printf("INC: write\n");
    return temp;
}

int decrement(int *counter){
// This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("    DEC: read\n");
    randomDelay();
    temp = temp -1;printf("    DEC: --\n");
    randomDelay();
    *counter = temp;printf("    DEC: write\n");
    return temp;
}

```

5.4 การแก้ไขปัญหาการแย่งชิงทรัพยากรร่วมกันโดยใช้ขั้นตอนวิธีของปีเตอร์สัน กรณีของวินโดวส์

```

#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#include <stdlib.h>
#include <time.h>

#define BUF_SIZE 4

struct shmareatype{
    int rp;
    int wp;
    int data[BUF_SIZE];
    int counter;
    // Following section is for Peterson's solution
    int flag[2];
    int turn;
};

void randomDelay(void);
void consumer(struct shmareatype *shmarea);
void producer(struct shmareatype *shmarea);
int increment(int *counter);
int decrement(int *counter);

void main(int argc, char *argv[]){
    TCHAR shmName[] = _T("lab5_4");
    struct shmareatype *shmarea; // Pointer to the shared segment
    int shmsize = sizeof(struct shmareatype); // share segment size
    HANDLE hMapFile;

    if(argc>1){ // If this is the child process it will run this instead
        hMapFile = OpenFileMapping(
            FALSE, // read/write access
            FILE_MAP_ALL_ACCESS, // do not inherit the name
            shmName); // name of mapping object

        if (hMapFile == NULL) {
            printf("Could not open file mapping object (%d).\n",
                GetLastError());
            return;
        }
    }
}

```

```

shmarea = (struct shmarea *) MapViewOfFile(hMapFile, // handle to map object
                                           FILE_MAP_ALL_ACCESS, // read/write permission
                                           0,
                                           0,
                                           shmsize);

if (shmarea == NULL)
{
    printf("Could not map view of file (%d).\n",
           GetLastError());
    return;
}

    consumer(shmarea);

    UnmapViewOfFile(shmarea);
    CloseHandle(hMapFile);
    return;
}
// Create Memory Mapping File
hMapFile = CreateFileMapping(
    INVALID_HANDLE_VALUE, // use paging file
    NULL, // default security
    PAGE_READWRITE, // read/write access
    0, // max. object size
    shmsize, // buffer size
    shmName); // name of mapping object

if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE) {
    printf("Could not create file mapping object...\n");
    return;
}

STARTUPINFO si;
PROCESS_INFORMATION pi;
TCHAR name[] = _T("lab5_4.exe 1"); // Using the Same code with parameter
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

printf("Before fork a child process...\n");

if (!CreateProcess(NULL, name,
                  NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {
    fprintf(stderr, "Create process failed.\n");
    return;
}

//Parent Process
// Map share memory
shmarea = (struct shmarea *) MapViewOfFile(hMapFile, // handle to map object
                                           FILE_MAP_ALL_ACCESS, // read/write permission
                                           0,
                                           0,
                                           shmsize);

if (shmarea == NULL)
{
    printf("Could not map view of file (%d).\n",
           GetLastError());
    return;
}

    shmarea->rp = shmarea->wp = shmarea->counter = 0;

    //Peterson's solution
    shmarea->flag[0] = shmarea->flag[1] = 0;
    shmarea->turn = 0;

    producer(shmarea);
    printf("Before going into the wait state...\n");
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child process has terminated\n");
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    UnmapViewOfFile(shmarea);
    CloseHandle(hMapFile);
}

```

```

void consumer(struct shmarea *shmarea){
    int i=0;
    int remaintime;

    while(1) // consume data
    {
        printf("      (Consumer) Before entering critical section\n");
        // Peterson's ENTRY section
        shmarea->flag[0]=true;
        shmarea->turn = 1;
        while(shmarea->flag[1] && shmarea->turn==1) Sleep(1);
        //////////////////////////////////////
        // Critical Section
        remaintime = 1000; // Wait a few seconds before quitting
        while((shmarea->counter==0) && (remaintime>0)){
            Sleep(1);remaintime--;
            if (!(remaintime % 1000))
                printf("      Waiting for %d second(s).\n", remaintime / 1000);
        }
        if(remaintime==0){
            printf("      The consumer process waiting too long, stopping...\n");
            shmarea->flag[0] = false; //EXIT here in case of waiting too long.
            return;
        }else{
            printf("      (Consumer) Data number:%d = %d\n",i++,shmarea->data[shmarea->rp]);
            shmarea->rp = ((shmarea->rp+1)%BUF_SIZE);
            decrement(&(shmarea->counter));
        }
    }
    // Peterson's EXIT section
    shmarea->flag[0] = false;
    // Remaining Section
    printf("      (Consumer) After exiting critical section\n");
}

void producer(struct shmarea *shmarea){
    int i;

    for(i=0;i<32;i++) // produce data
    {
        printf("(Producer) Before entering critical section\n");
        // Peterson's ENTRY section
        shmarea->flag[1] = true;
        shmarea->turn = 0;
        while(shmarea->flag[0] && shmarea->turn==0) Sleep(1);
        // Critical Section
        while(shmarea->counter==BUF_SIZE) Sleep(1);
        printf("(Producer)Enter data %d into share memory...\n",i);
        shmarea->data[shmarea->wp]=i;
        // move the write pointer so that the consumer know when to read.
        shmarea->wp = ((shmarea->wp+1)%BUF_SIZE);
        increment(&(shmarea->counter));
        // Peterson's EXIT section
        shmarea->flag[1] = false;
        // Remaining Section
        printf("(Producer) After exiting critical section\n");
    }
}

void randomDelay(void){
    // This function provides a delay which slows the process down so we can see what happens
    srand(time(NULL));
    int stime = (rand()%1000)+100;
    Sleep(stime);
}

int increment(int *counter){
    // This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("INC: read\n");
    randomDelay();
    temp = temp +1;printf("INC: ++\n");
    randomDelay();
    *counter = temp;printf("INC: write\n");
    return temp;
}

int decrement(int *counter){
    // This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("      DEC: read\n");

```



```

randomDelay();
temp = temp -1;printf("      DEC: --\n");
randomDelay();
*counter = temp;printf("      DEC: write\n");
return temp;
}

```

5.5 การแก้ไขปัญหาการแย่งชิงทรัพยากรร่วมกันโดยใช้ TestAndSet กรณีของลินุกซ์

ตัวอย่างต่อไปนี้จะใช้ฟังก์ชัน(หรือถ้าจะเรียกให้ถูก จะเป็นมาโคร) `__sync_fetch_and_or()` ซึ่งฟังก์ชันนี้จะแตกต่างกันไปขึ้นอยู่กับสถาปัตยกรรมของซีพียู โดยคอมพิวเตอร์จะแปลงฟังก์ชันดังกล่าวเป็นชุดคำสั่งภาษาเครื่องของซีพียูซึ่งจะออกมาเพียงคำสั่งเดียว (instruction) ทำให้มีลักษณะเป็น atomic ตามต้องการ ฟังก์ชันนี้เป็นฟังก์ชันมาตรฐาน (built-in function) ของ GCC ซึ่งมีอยู่หลายตัวด้วยกัน ตัวที่น่าสนใจมีดังนี้

```

type __sync_fetch_and_add (type *ptr, type value);
type __sync_fetch_and_sub (type *ptr, type value);
type __sync_fetch_and_or (type *ptr, type value);
type __sync_fetch_and_and (type *ptr, type value);
type __sync_fetch_and_xor (type *ptr, type value);
type __sync_fetch_and_nand (type *ptr, type value);

```

ชนิดของ type เป็นข้อมูลจำนวนเต็ม ฟังก์ชันที่มีคำว่า fetch นำหน้า หมายความว่า จะอ่านค่าเก่าส่งกลับก่อนแล้วจึงดำเนินการตามการดำเนินการที่แจ้งเช่น `__sync_fetch_and_or` หมายความว่า จะอ่านค่าจากหน่วยความจำในตำแหน่งที่อ้างโดย ptr แล้วนำมา or กับค่า value แต่จะส่งค่าก่อน or กลับจากฟังก์ชัน แล้วนำค่าที่ or แล้วไปใส่ในตำแหน่งหน่วยความจำที่อ้างโดย ptr ส่วน `__sync_orand_fetch` จะทำตรงกันข้าม เป็นต้น

อนึ่ง การใช้งานฟังก์ชันในกลุ่มนี้ จะต้องกำหนดสถาปัตยกรรมในขณะคอมไพล์ด้วย โดยซีพียูที่รองรับในตระกูลอินเทล ทำได้ตั้งแต่ 486 หรือใหม่กว่า ดังนั้นจึงต้องเพิ่มออปชันในการคอมไพล์คือ `-march=i486` (หรือใหม่กว่าเช่น `-march=i686` เป็นต้น) ดังตัวอย่างเช่น

```
gcc -o lab5_5 lab5_5.cpp -march=i686
```

กลไก TestAndSet() เมื่อเทียบกับฟังก์ชันด้านบนนี้แล้ว อาจเทียบได้กับ `__sync_fetch_and_or(ptr,1)` หรือ `__sync_fetch_and_add(ptr,1)` เมื่อค่าเดิมก่อนการเปลี่ยนแปลงคือ 0 (false) ดังนั้นเราสามารถนำเอาโปรแกรมตัวอย่าง 5.3 ที่แก้ไขปัญหาด้วยขั้นตอนวิธีของปีเตอร์สัน มาใช้การ TestAndSet() จะได้ดังนี้

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h> // for usleep()
#include <stdlib.h> // for exit() and random generator
#include <wait.h> // for wait()
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <time.h>

#define BUF_SIZE 4

struct shmareatype{
    int rp;
    int wp;
    int data[BUF_SIZE];
    int counter;
    // Following section is for locking mechanism
    int lock;
};

void randomDelay(void);
void consumer(struct shmareatype *shmarea);
void producer(struct shmareatype *shmarea);
int increment(int *counter);
int decrement(int *counter);

int main(){
    int shmid; // Share memory ID

```

```

struct shmarea_type *shmarea; // Pointer to the shared segment
int shmsize; // share segment size
pid_t pid; // Child Process ID

shmsize = sizeof(struct shmarea_type);
// Allocate the shared block from OS
shmid = shmget(IPC_PRIVATE, shmsize, 0666 | IPC_CREAT);

// Attach the shared block to the pointer so we can use it.
shmarea = (struct shmarea_type *)shmat(shmid, NULL, 0);

shmarea->rp = shmarea->wp = shmarea->counter = 0;
shmarea->lock = false; // Start the lock with noone using it.

pid = fork(); // Fork a child process

if(pid < 0){ // Fork error
    fprintf(stderr, "Fork failed.\n");
    exit(-1);
}
else if(pid==0){ // This is the path of child process
    consumer(shmarea);
}
else { // This is the path of parent process
    producer(shmarea);
    wait(NULL);
    printf("Child process has terminated\n");

    // Detach the shared memory segment
    shmdt(shmarea);
    // Remove the shared memory segment
    shmctl(shmid, IPC_RMID, NULL);
    exit(0);
}

}

void consumer(struct shmarea_type *shmarea){
    int i=0;
    int remaintime;

    while(1) // consume data
    {
        printf("      (Consumer) Before entering critical section\n");
        // Using Test&Set mechanism, in Linux with gcc the function is __sync_fetch_and_add(ptr,1);
        while(__sync_fetch_and_or(&(shmarea->lock),1)) usleep(1000);
        // Critical Section
        printf("      (Consumer) Entered critical section (lock=%d)\n",shmarea->lock);
        remaintime=5000; // Wait a few seconds before quitting
        while((shmarea->counter==0) && (remaintime>0)){
            usleep(1000); remaintime--;
            if(!(remaintime%1000))
                printf("      (Consumer) Waiting for %d\n", remaintime/1000);
        }
        if(remaintime==0){
            printf("      (Consumer) The consumer process waiting too long, stopping...\n");
            shmarea->lock = false; // Consumer releases lock and exits here
            return;
        }else{
            printf("      (Consumer) Data number:%d = %d\n",i++,shmarea->data[shmarea->rp]);
            shmarea->rp = ((shmarea->rp+1)%BUF_SIZE);
            decrement(&(shmarea->counter));
        }
    }
    // Release lock
    shmarea->lock = false;
    // Remaining Section
    printf("      (Consumer) After exiting critical section (lock=%d)\n",shmarea->lock);
    randomDelay();
}

}

void producer(struct shmarea_type *shmarea){
    int i;

    for(i=0;i<32;i++) // produce data
    {

```

```

        printf("(Producer) Before entering critical section\n");
// Using Test&Set mechanism, in Linux with gcc the function is __sync_fetch_and_add(ptr,1);
while(__sync_fetch_and_or(&(shmarea->lock),1)) usleep(1000);
// Critical Section
printf("(Producer) Entered critical section (lock=%d)\n",shmarea->lock);
if(shmarea->counter!=BUF_SIZE){
    printf("(Producer) Enter data %d into share memory...\n",i);
    shmarea->data[shmarea->wp]=i;
    // move the write pointer so that the consumer know when to read.
    shmarea->wp = (shmarea->wp+1)%BUF_SIZE;
    increment (&(shmarea->counter));
}else{
    printf("(Producer) Buffer is full, release lock and try again.\n");
    i--;
    shmarea->lock = false;
    continue;
}
// Release lock
shmarea->lock = false;
// Remaining Section
printf("(Producer) After exiting critical section (lock=%d)\n",shmarea->lock);
randomDelay();
}
}

void randomDelay(void){
// This function provides a delay which slows the process down so we can see what happens
srand(time(NULL));
int stime = ((rand()%1000)+100)*1000;
usleep(stime);
}

int increment(int *counter){
// This function show how non-atomic instruction can cause problem in consumer-producer case
int temp = *counter;printf("INC: read\n");
randomDelay();
temp = temp +1;printf("INC: ++\n");
randomDelay();
*counter = temp;printf("INC: write\n");
return temp;
}

int decrement(int *counter){
// This function show how non-atomic instruction can cause problem in consumer-producer case
int temp = *counter;printf("DEC: read\n");
randomDelay();
temp = temp -1;printf("DEC: --\n");
randomDelay();
*counter = temp;printf("DEC: write\n");
return temp;
}

```

5.6 การแก้ไขปัญหการแย่งชิงทรัพยากรร่วมกันโดยใช้ TestAndSet กรณีของวินโดวส์

ตัวอย่างต่อไปนี้เป็นการประยุกต์ตัวอย่างที่ 5.4 มาใช้หลักการของ TestAndSet โดยในวินโดวส์นั้น ตามปกติแล้ว ตัวแปรชนิดใดๆ ที่มีขนาดไม่เกิน 32 บิตจะเขียนหรืออ่านแบบ Atomic อยู่แล้ว และถึงกระนั้น ระบบปฏิบัติการวินโดวส์ได้ให้ฟังก์ชันจำนวนหนึ่งสำหรับการจัดการแบบ Atomic ซึ่งวินโดวส์จะเรียกตัวแปรที่ถูกจัดการในลักษณะนี้ว่า Interlocked variable โดยการเข้าถึงนี้หากเรดหรือโพเซสมีการใช้พื้นที่หน่วยความจำร่วมกันก็จะสามารถใช้ได้โดยทันที

ในที่นี้เราใช้ฟังก์ชัน InterlockedExchangeAdd() ซึ่งมีกลไกเหมือนกับ __sync_fetch_and_add() ของลินุกซ์ทุกประการ ฟังก์ชันจัดการกับตัวแปรแบบ Interlocked ที่ส่งค่ากลับคืนด้วยค่าก่อนการเปลี่ยนแปลง ที่น่าสนใจดังนี้

```

LONG InterlockedExchange(LONG volatile* Target, LONG Value); // แทนค่าด้วยค่าใหม่
LONG InterlockedExchangeAdd(LONG volatile* Addend, LONG Value);
LONG InterlockedAnd(LONG volatile* Destination, LONG Value);
LONG InterlockedOr(LONG volatile* Destination, LONG Value);
LONG InterlockedXor(LONG volatile* Destination, LONG Value);

```

```

#include <windows.h>
#include <stdio.h>
#include <tchar.h>

```

```

#include <stdlib.h>
#include <time.h>

#define BUF_SIZE 4

struct shmareatype{
    int rp;
    int wp;
    int data[BUF_SIZE];
    int counter;
    // Following section is for locking mechanism
    int lock;
};

void randomDelay(void);
void consumer(struct shmareatype *shmarea);
void producer(struct shmareatype *shmarea);
int increment(int *counter);
int decrement(int *counter);

void main(int argc, char *argv[]){
    TCHAR shmName[] = _T("lab5_6");
    struct shmareatype *shmarea; // Pointer to the shared segment
    int shmsize = sizeof(struct shmareatype); // share segment size
    HANDLE hMapFile;

    if(argc>1){ // If this is the child process it will run this instead
        hMapFile = OpenFileMapping(
            FILE_MAP_ALL_ACCESS, // read/write access
            FALSE, // do not inherit the name
            shmName); // name of mapping object

        if (hMapFile == NULL) {
            printf("Could not open file mapping object (%d).\n",
                GetLastError());
            return;
        }
        shmarea = (struct shmareatype *) MapViewOfFile(hMapFile, // handle to map object
            FILE_MAP_ALL_ACCESS, // read/write permission
            0,
            0,
            shmsize);

        if (shmarea == NULL)
        {
            printf("Could not map view of file (%d).\n",
                GetLastError());
            return;
        }

        consumer(shmarea);

        UnmapViewOfFile(shmarea);
        CloseHandle(hMapFile);
        return;
    }
    // Create Memory Mapping File
    hMapFile = CreateFileMapping(
        INVALID_HANDLE_VALUE, // use paging file
        NULL, // default security
        PAGE_READWRITE, // read/write access
        0, // max. object size
        shmsize, // buffer size
        shmName); // name of mapping object

    if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE){
        printf("Could not create file mapping object...\n");
        return;
    }

    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    TCHAR name[] = _T("lab5_6.exe 1"); // Using the Same code with parameter
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    printf("Before fork a child process...\n");

```

```

    if(!CreateProcess(NULL,name,
        NULL,NULL,FALSE,0,NULL,NULL,&si,&pi)){
        fprintf(stderr,"Create process failed.\n");
        return ;
    }

    //Parent Process
    // Map share memory
    shmarea = (struct shmarea_t *) MapViewOfFile(hMapFile, // handle to map object
        FILE_MAP_ALL_ACCESS, // read/write permission
        0,
        0,
        shmsize);

    if (shmarea == NULL)
    {
        printf("Could not map view of file (%d).\n",
            GetLastError());
        return;
    }
    shmarea->rp = shmarea->wp = shmarea->counter = 0;

    //Start with unlock
    shmarea->lock = false;

    producer(shmarea);
    printf("Before going into the wait state...\n");
    WaitForSingleObject(pi.hProcess,INFINITE);
    printf("Child process has terminated\n");
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    UnmapViewOfFile(shmarea);
    CloseHandle(hMapFile);
}

void consumer(struct shmarea_t *shmarea){
    int i=0;
    int remaintime;

    while(1) // consume data
    {
        printf("          (Consumer) Before entering critical section\n");
        // Using Test&Set mechanism
        while(InterlockedExchangeAdd((volatile LONG *)&(shmarea->lock),1)) Sleep(1);
        // Critical Section
        printf("          (Consumer) Entered critical section (lock=%d)\n",shmarea->lock);

        remaintime = 5000; // Wait a few seconds before quitting
        while((shmarea->counter==0)&& (remaintime>0)){
            Sleep(1); remaintime--;
            if (!(remaintime % 1000))
                printf("          (Consumer) Waiting for %d second(s).\n", remaintime /
1000);
        }
        if(remaintime==0){
            printf("          (Consumer) The consumer process waiting too long,
stopping...\n");
            shmarea->lock = false; // Consumer releases lock and exits here
            return;
        }else{
            printf("          (Consumer) Data number:%d = %d\n",i++,shmarea->data[shmarea-
>rp]);
            shmarea->rp = ((shmarea->rp+1)%BUF_SIZE);
            decrement(&(shmarea->counter));
        }
        // Release lock
        shmarea->lock = false;
        // Remaining Section
        printf("          (Consumer) After exiting critical section (lock=%d)\n",shmarea->lock);
        randomDelay();
    }
}

void producer(struct shmarea_t *shmarea){
    int i;

    for(i=0;i<32;i++) // produce data

```

```

{
    printf("(Producer) Before entering critical section\n");
    // Using Test&Set mechanism
    while(InterlockedExchangeAdd((volatile LONG *)&(shmarea->lock),1)) Sleep(1);
    // Critical Section
    printf("(Producer) Entered critical section (lock=%d)\n",shmarea->lock);
    if (shmarea->counter!=BUF_SIZE){
        printf("(Producer) Enter data %d into share memory...\n",i);
        shmarea->data[shmarea->wp]=i;
        // move the write pointer so that the consumer know when to read.
        shmarea->wp = ((shmarea->wp+1)%BUF_SIZE);
        increment(&(shmarea->counter));
    }else{
        printf("Buffer is full, release lock and try again.\n");
        i--;
        shmarea->lock = false;
        continue;
    }
    // Release lock
    shmarea->lock = false;
    // Remaining Section
    printf("(Producer) After exiting critical section (lock=%d)\n",shmarea->lock);
    randomDelay();
}

void randomDelay(void){
    // This function provides a delay which slows the process down so we can see what happens
    srand(time(NULL));
    int stime = (rand()%1000)+100;
    Sleep(stime);
}

int increment(int *counter){
    // This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("INC: read\n");
    randomDelay();
    temp = temp +1;printf("INC: ++\n");
    randomDelay();
    *counter = temp;printf("INC: write\n");
    return temp;
}

int decrement(int *counter){
    // This function show how non-atomic instruction can cause problem in consumer-producer case
    int temp = *counter;printf("DEC: read\n");
    randomDelay();
    temp = temp -1;printf("DEC: --\n");
    randomDelay();
    *counter = temp;printf("DEC: write\n");
    return temp;
}

```

5.7 ตัวอย่างปัญหาการแย่งชิงทรัพยากรร่วมกัน โปรแกรมทำงานบนลินุกซ์

ตัวอย่างต่อไปนี้เป็นกำลอง Producer-Consumer ที่ซับซ้อนขึ้น ในที่นี้จะมี Producer ทั้งหมด 4 ตัว และมี Consumer 1 ตัว และมีพื้นที่เก็บข้อมูล 3 ส่วน ซึ่งในโปรแกรมตัวอย่างได้จำลองขั้นตอนการเขียนด้วย usleep() เพื่อให้เห็นถึงการเขียน-อ่าน ข้อมูล จำนวนมากบนพื้นที่ร่วมกัน

อนึ่ง โปรแกรมนี้เป็นแบบ multithread (ใช้ทั้งหมด 5 เธรด) ดังนั้นจะต้องเพิ่ม option -pthread ที่ส่วน Linker ด้วย

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
    {"rice with ", "chicken curry, and ", "fish sauce."},
    {"wiskey with ", "lemonade, and ", "soda."},
    {"bread with ", "cheese, and ", "ketchup."},
    {"icecream with ", "banana, and ", "chocolate."}
};

```

```

int  isFull;
int  isReady;

void randomDelay(void);
void serve(char *dest,char *src);
void *chefWork(void *who);
void *customer(void *who);

int main(void){
    int i;
    int param[5]={0,1,2,3,4};
    pthread_t tid[5];          // Thread ID
    pthread_attr_t attr[5];    // Thread attributes

    isFull=false;              // Customer tell all chefs to stop making food
    isReady=false;             // Producer tell customer that food is ready

    for(i=0;i<5;i++){
        pthread_attr_init(&attr[i]); // Get default attributes

        // Create 4 threads for producers
        for(i=0;i<4;i++){
            pthread_create(&tid[i],&attr[i],chefWork,(void *)&param[i]);
        }
        // Create 1 threads for consumer
        pthread_create(&tid[4],&attr[4],customer,(void *)&param[i]);

        // Wait until all threads finish
        for(i=0;i<5;i++){
            pthread_join(tid[i],NULL);
        }

        return 0;
    }

void randomDelay(void){
    int stime = ((rand()%100)+1)*1000;
    usleep(stime);
}

void serve(char *dest,char *src){
    int i;
    srand(time(NULL));

    for(i=0;(i<63)&&(src[i]!=0);i++){
        dest[i]=src[i];
        randomDelay();
    }
    dest[i]=0;
}

void *chefWork(void *who){
    int plateNo,chefNo,i,j;

    chefNo = (int)*((int *)who);

    for(i=0;i<(chefNo+2);i++) printf(" ");
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);

    while(!isFull){

        if(isReady){
            randomDelay(); // Wait for food to be taken
        }

        if((!isFull)&&(!isReady)){
            for(i=0;i<(chefNo+2);i++) printf(" ");
            printf("Chef NO %d is serving the food\n",chefNo+1);
            fflush(stdout);

            for(j=0;j<3;j++){
                serve(plate[j],chef[chefNo][j]);
            }
            isReady=true; //Tel customer that the food is ready

            for(i=0;i<(chefNo+2);i++) printf(" ");
            printf("Chef NO %d has serve the food\n",chefNo+1);
            fflush(stdout);
        }else{
            // for(i=0;i<(chefNo+2);i++) printf(" ");

```

```

//          printf("Chef NO %d has to skip the searving\n",chefNo+1);
//          fflush(stdout);
    }
    if(isFull) break;
    usleep(300000*(chefNo+1)); // This will cause some chef to waiting indefinitely
//    for(i=0;i<(chefNo+2);i++) printf(" ");
//    printf("Chef NO %d has rested\n",chefNo+1);
//    fflush(stdout);
}
pthread_exit(0);
}

void *customer(void *who){
    int i,j;
    char dinner[256]={0};

    for(i=0;i<10;i++){
//        printf("Choochok waits for food to be served\n");
//        fflush(stdout);

        while(!isReady) randomDelay();

        printf("Choochok starts grab a set of dinner\n");
        fflush(stdout);

        serve(dinner,plate[0]); dinner[63]=0;
        serve(dinner + strlen(dinner),plate[1]); dinner[127]=0;
        serve(dinner + strlen(dinner),plate[2]); dinner[191]=0;

        printf("Plate NO:%d Choochok is eating %s\n",i+1,dinner);
        fflush(stdout);

        isReady=false; // Food taken
        // Remaining Section
//        for(j=0;j<10;j++) randomDelay();
//        printf("Choochok is moving to the next table\n");
//        fflush(stdout);
    }
    isFull = true; // Tell other producer threads to stop;
    pthread_exit(0);
}

```

5.8 ตัวอย่างปัญหาการแย่งชิงทรัพยากรร่วมกัน โปรแกรมทำงานบนวินโดวส์

ตัวอย่างต่อไปนี้มีโครงสร้างการทำงานเหมือนกันกับตัวอย่าง 5.7 เพียงแต่ถูกพอร์ตมาบนวินโดวส์ ให้สังเกตผลลัพธ์ของการทำงานว่า consumer ได้ข้อมูลอะไรไปบ้าง

```

#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
    {"rice with ","chicken curry, and ","fish sauce."},
    {"wiskey with ","lemonade, and ","soda."},
    {"bread with ","cheese, and ","ketchup."},
    {"icecream with ","banana, and ","chocolate."}};

int isFull;
int isReady;

void randomDelay(void);
void serve(char *dest,char *src);
DWORD WINAPI chefWork(LPVOID who);
DWORD WINAPI customer(LPVOID who);

int main(void){
    int i;
    DWORD tid[5]; // Thread ID
    HANDLE th[5]; // Thread Handle
    int param[5]={0,1,2,3,4};

    // Create 5 threads
    for(i=0;i<4;i++)
        th[i] = CreateThread(

```

```

        NULL,                // Default security attributes
        0,                  // Default stack size
        chefWork,           // Thread function
        (void *)&param[i],  // Thread function parameter
        0,                  // Default creation flag
        &tid[i]);           // Thread ID returned.

    th[4] = CreateThread(
        NULL,                // Default security attributes
        0,                  // Default stack size
        customer,            // Thread function
        (void *)&param[4],  // Thread function parameter
        0,                  // Default creation flag
        &tid[4]);           // Thread ID returned.

    // Wait until all threads finish
    for(i=0;i<4;i++)
        if(th[i]!=NULL)
            WaitForSingleObject(th[i],INFINITE);

    return 0;
}

void randomDelay(void){
    int stime = ((rand()%100)+1);
    Sleep(stime);
}

void serve(char *dest,char *src){
    int i;
    srand(time(NULL));

    for(i=0;(i<63)&&(src[i]!=0);i++){
        dest[i]=src[i];
        randomDelay();
    }
    dest[i]=0;
}

DWORD WINAPI chefWork(LPVOID who){
    int chefNo,i,j;

    chefNo = (int)*((int *)who);

    for(i=0;i<(chefNo+2);i++) printf(" ");
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);

    while(!isFull){

        if(isReady) {
            randomDelay(); // Wait for food to be taken
        }

        if((!isFull)&&(!isReady)){
            // Critical Section
            for(i=0;i<(chefNo+2);i++) printf(" ");
            printf("Chef NO %d is serving the food\n",chefNo+1);
            fflush(stdout);

            for(j=0;j<3;j++)
                serve(plate[j],chef[chefNo][j]);
            isReady=true; //Tel customer that the food is ready

            for(i=0;i<(chefNo+2);i++) printf(" ");
            printf("Chef NO %d has serve the food\n",chefNo+1);
            fflush(stdout);
        }
        if(isFull) break;
        // Remaining Section
        Sleep(300*(chefNo+1)); // This will cause some chef to waiting indefinitely
    }
    return 0;
}

DWORD WINAPI customer(LPVOID who){
    int i,j;
    char dinner[256]={0};

    for(i=0;i<10;i++){

```

```

while(!isReady) randomDelay();

    // Critical Section
    printf("Choochok starts grab a set of dinner\n");
    fflush(stdout);

    serve(dinner,plate[0]); dinner[63]=0;
    serve(dinner + strlen(dinner),plate[1]); dinner[127]=0;
    serve(dinner + strlen(dinner),plate[2]); dinner[191]=0;

    printf("Plate NO:%d Choochok is eating %s\n",i+1,dinner);
    fflush(stdout);

    isReady=false; // Food taken
    // Remaining Section
}
isFull = true; // Tell other producer threads to stop;
return 0;
}

```

5.9 ตัวอย่างการแก้ไขปัญหการแย่งชิงทรัพยากรร่วมกัน โดยการใช้ locking mechanism โปรแกรมทำงานบนลินุกซ์

ตัวอย่างนี้แก้ไขจากตัวอย่าง 5.7 เพื่อเพิ่มการใช้งานคำสั่ง Test&Set ซึ่งเป็นแบบ atomic ซึ่งในที่นี้ใช้

```

__sync_fetch_and_or()

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
    {"rice with ", "chicken curry, and ", "fish sauce."},
    {"wiskey with ", "lemonade, and ", "soda."},
    {"bread with ", "cheese, and ", "ketchup."},
    {"icecream with ", "banana, and ", "chocolate."};
int isFull;
int isReady;
int lock;

void randomDelay(void);
void serve(char *dest, char *src);
void *chefWork(void *who);
void *customer(void *who);

int main(void){
    int i;
    int param[5]={0,1,2,3,4};
    pthread_t tid[5]; // Thread ID
    pthread_attr_t attr[5]; // Thread attributes

    isFull=false; // Customer tell all chefs to stop making food
    isReady=false; // Producer tell customer that food is ready
    lock = false; // Locking mechanism

    for(i=0;i<5;i++)
        pthread_attr_init(&attr[i]); // Get default attributes

    // Create 4 threads for producers
    for(i=0;i<4;i++)
        pthread_create(&tid[i],&attr[i],chefWork,(void *)&param[i]);
    // Create 1 threads for consumer
    pthread_create(&tid[4],&attr[4],customer,(void *)&param[i]);

    // Wait until all threads finish
    for(i=0;i<5;i++)
        pthread_join(tid[i],NULL);
    return 0;
}

```

```

void randomDelay(void){
    int stime = ((rand()%100)+1)*1000;
    usleep(stime);
}

void serve(char *dest,char *src){
    int i;
    srand(time(NULL));

    for(i=0;(i<63)&&(src[i]!=0);i++){
        dest[i]=src[i];
        randomDelay();
    }
    dest[i]=0;
}

void *chefWork(void *who){
    int plateNo,chefNo,i,j;

    chefNo = (int)*((int *)who);

    for(i=0;i<(chefNo+2);i++) printf(" ");
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);

    while(!isFull){

        if(isReady){
            randomDelay(); // Wait for food to be taken
        }
        while(__sync_fetch_and_or(&lock,1)) usleep(500000);

        if((!isFull)&&(!isReady)){
            // Critical Section
            for(i=0;i<(chefNo+2);i++) printf(" ");
            printf("Chef NO %d is serving the food\n",chefNo+1);
            fflush(stdout);

            for(j=0;j<3;j++)
                serve(plate[j],chef[chefNo][j]);
            isReady=true; //Tel customer that the food is ready

            for(i=0;i<(chefNo+2);i++) printf(" ");
            printf("Chef NO %d has serve the food\n",chefNo+1);
            fflush(stdout);
        }else{
            // for(i=0;i<(chefNo+2);i++) printf(" ");
            // printf("Chef NO %d has to skip the searving\n",chefNo+1);
            // fflush(stdout);
        }
        lock = false;
        if(isFull) break;
        // Remaining Section
        usleep(300000*(chefNo+1)); // This will cause some chef to waiting indefinitely
        for(i=0;i<(chefNo+2);i++) printf(" ");
        printf("Chef NO %d has rested\n",chefNo+1);
        fflush(stdout);
    }
    pthread_exit(0);
}

void *customer(void *who){
    int i,j;
    char dinner[256]={0};

    for(i=0;i<10;i++){
        // printf("Choochok waits for food to be served\n");
        // fflush(stdout);

        while(!isReady) randomDelay();
        while(__sync_fetch_and_or(&lock,1)) usleep(500000);

        // Critical Section
        printf("Choochok starts grab a set of dinner\n");
        fflush(stdout);

        serve(dinner,plate[0]); dinner[63]=0;
        serve(dinner + strlen(dinner),plate[1]); dinner[127]=0;
    }
}

```

```

        serve(dinner + strlen(dinner),plate[2]); dinner[191]=0;

        printf("Plate NO:%d Choochok is eating %s\n",i+1,dinner);
        fflush(stdout);

        isReady=false; // Food taken
        lock=false;
        // Remaining Section
        for(j=0;j<10;j++) randomDelay();
        // printf("Choochok is moving to the next table\n");
        // fflush(stdout);
    }
    isFull = true; // Tell other producer threads to stop;
    pthread_exit(0);
}

```

5.10 ตัวอย่างการแก้ไขปัญหาการแย่งชิงทรัพยากรร่วมกัน โดยใช้ locking mechanism โปรแกรม

ทำงานบนวินโดวส์

ตัวอย่างนี้ใช้หลักการ Interlocking variable และฟังก์ชันที่ทำงานแบบ atomic บนวินโดวส์

```

#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
    {"rice with ", "chicken curry, and ", "fish sauce."},
    {"whiskey with ", "lemonade, and ", "soda."},
    {"bread with ", "cheese, and ", "ketchup."},
    {"icecream with ", "banana, and ", "chocolate."};
};

int isFull;
int isReady;
int lock;

void randomDelay(void);
void serve(char *dest, char *src);
DWORD WINAPI chefWork(LPVOID who);
DWORD WINAPI customer(LPVOID who);

int main(void){
    int i;
    DWORD tid[5]; // Thread ID
    HANDLE th[5]; // Thread Handle
    int param[5]={0,1,2,3,4};

    // Create 5 threads
    for(i=0;i<4;i++)
        th[i] = CreateThread(
            NULL, // Default security attributes
            0, // Default stack size
            chefWork, // Thread function
            (void *)&param[i], // Thread function parameter
            0, // Default creation flag
            &tid[i]); // Thread ID returned.

    th[4] = CreateThread(
        NULL, // Default security attributes
        0, // Default stack size
        customer, // Thread function
        (void *)&param[4], // Thread function parameter
        0, // Default creation flag
        &tid[4]); // Thread ID returned.

    // Wait until all threads finish
    for(i=0;i<5;i++)
        if(th[i]!=NULL)
            WaitForSingleObject(th[i], INFINITE);

    return 0;
}

void randomDelay(void){
    int stime = ((rand()%100)+1);
    Sleep(stime);
}

```

```

void serve(char *dest, char *src) {
    int i;
    srand(time(NULL));

    for (i=0; (i<63)&&(src[i]!=0); i++) {
        dest[i]=src[i];
        randomDelay();
    }
    dest[i]=0;
}

DWORD WINAPI chefWork(LPVOID who) {
    int chefNo, i, j;

    chefNo = (int)*((int *)who);

    for (i=0; i<(chefNo+2); i++) printf(" ");
    printf("Chef NO %d start working...\n", chefNo+1);
    fflush(stdout);

    while(!isFull) {
        if (isReady) {
            randomDelay(); // Wait for food to be taken
        }
        while(InterlockedExchangeAdd((volatile LONG *)&lock, 1)) Sleep(500);

        if ((!isFull) && (!isReady)) {
            // Critical Section
            for (i=0; i<(chefNo+2); i++) printf(" ");
            printf("Chef NO %d is serving the food\n", chefNo+1);
            fflush(stdout);

            for (j=0; j<3; j++)
                serve(plate[j], chef[chefNo][j]);
            isReady=true; // Tel customer that the food is ready

            for (i=0; i<(chefNo+2); i++) printf(" ");
            printf("Chef NO %d has serve the food\n", chefNo+1);
            fflush(stdout);
        } else {
            for (i=0; i<(chefNo+2); i++) printf(" ");
            printf("Chef NO %d has to skip the searving\n", chefNo+1);
            fflush(stdout);
        }
        lock = false;
        if (isFull) break;
        // Remaining Section
        Sleep(300*(chefNo+1)); // This will cause some chef to waiting indefinitely
        for (i=0; i<(chefNo+2); i++) printf(" ");
        printf("Chef NO %d has rested\n", chefNo+1);
        fflush(stdout);
    }
    return 0;
}

DWORD WINAPI customer(LPVOID who) {
    int i, j;
    char dinner[256]={0};

    for (i=0; i<10; i++) {
        printf("Choochok waits for food to be served\n");
        fflush(stdout);

        while(!isReady) randomDelay();
        while(InterlockedExchangeAdd((volatile LONG *)&lock, 1)) Sleep(500);

        // Critical Section
        printf("Choochok starts grab a set of dinner\n");
        fflush(stdout);

        serve(dinner, plate[0]); dinner[63]=0;
        serve(dinner + strlen(dinner), plate[1]); dinner[127]=0;
        serve(dinner + strlen(dinner), plate[2]); dinner[191]=0;

        printf("Plate NO:%d Choochok is eating %s\n", i+1, dinner);
        fflush(stdout);

        isReady=false; // Food taken
    }
}

```

```

        lock=false;
        // Remaining Section
        for(j=0;j<10;j++) randomDelay();
        printf("Choochok is moving to the next table\n");
        fflush(stdout);
    }
    isFull = true; // Tell other producer threads to stop;
    return 0;
}

```

5.11 ตัวอย่างการแก้ไขปัญหาการแย่งชิงทรัพยากรร่วมกัน โดยใช้ locking mechanism และจัดการแบบ Bound-waiting โปรแกรมทำงานบนลินุกซ์

ตัวอย่างนี้เป็นการปรับปรุงมาจากตัวอย่างที่ใช้ locking mechanism แต่มีการจัดการป้องกันไม่ให้บางเธรดรอนานจนเกินไปด้วยการใช้อะเรย์เก็บสถานะของการรอ และหากมีผู้รอหลายคน ก็จะไล่ไปตามลำดับในอะเรย์ที่จัดเก็บโพรเซสที่รออยู่ตามหมายเลขโพรเซส (ดูรายละเอียดจากในเอกสารเนื้อหาข้างต้น)

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
    {"rice with ", "chicken curry, and ", "fish sauce."},
    {"wiskey with ", "lemonade, and ", "soda."},
    {"bread with ", "cheese, and ", "ketchup."},
    {"icecream with ", "banana, and ", "chocolate."};
int isFull;
int lock;
int waiting[4];

void randomDelay(void);
void serve(char *dest, char *src);
void *chefWork(void *who);

int main(void){
    int i;
    int param[5]={0,1,2,3,4};
    pthread_t tid[5]; // Thread ID
    pthread_attr_t attr[5]; // Thread attributes

    isFull=20; // Number of disk remains
    lock = false; // Locking mechanism
    for(i=0;i<5;i++) waiting[i]=false;

    // No consumer!!!
    for(i=0;i<4;i++)
        pthread_attr_init(&attr[i]); // Get default attributes

    // Create 4 threads for producers
    for(i=0;i<4;i++)
        pthread_create(&tid[i], &attr[i], chefWork, (void *) &param[i]);

    // No consumer!!!
    // Wait until all threads finish
    for(i=0;i<4;i++)
        pthread_join(tid[i], NULL);
    return 0;
}

void randomDelay(void){
    int stime = ((rand()%50)+1)*1000;
    usleep(stime);
}

void serve(char *dest, char *src){
    int i;
    srand(time(NULL));

    for(i=0; (i<63) && (src[i] != 0); i++){

```

```

        dest[i]=src[i];
        randomDelay();
    }
    dest[i]=0;
}

void *chefWork(void *who){
    int plateNo, chefNo, i, j;
    int key;

    chefNo = (int)*((int *)who);

    for(i=0; i<(chefNo+2); i++) printf(" ");
    printf("Chef NO %d start working...\n", chefNo+1);
    fflush(stdout);

    while(isFull>0){
        waiting[chefNo]=true;
        key = true;
        while(waiting[chefNo] && key){
            key = __sync_fetch_and_or(&lock, 1);
            usleep(300000);
            if(isFull<=0) break;
        }
        waiting[chefNo]=false;
        // Critical Section
        if(isFull>0){
            for(i=0; i<(chefNo+2); i++) printf(" ");
            printf("Chef NO %d is serving the food\n", chefNo+1);
            fflush(stdout);

            for(j=0; j<3; j++)
                serve(plate[j], chef[chefNo][j]);

            for(i=0; i<(chefNo+2); i++) printf(" ");
            printf("Chef NO %d has serve the food\n", chefNo+1);
            fflush(stdout);
            isFull--;
        }

        j = (chefNo+1)%4;
        while((j!=chefNo)&& !waiting[j])
            j = (j+1)%4;
        if(j==chefNo){
            lock = false;
            printf("Chef No %d has unlocked the
key. [%d] [%d] [%d] [%d]\n", j+1, waiting[0], waiting[1], waiting[2], waiting[3]);
        }
        else{
            waiting[j]=false;
            printf("Passing to chef No %d
[%d] [%d] [%d] [%d]\n", j+1, waiting[0], waiting[1], waiting[2], waiting[3]);
        }
        // Remaining Section
        usleep((1000*(rand()%4000))+1);
    }
    pthread_exit(0);
}

```

5.12 ตัวอย่างการแก้ไขปัญหาการแย่งชิงทรัพยากรร่วมกัน โดยใช้ locking mechanism และจัดการแบบ

Bound-waiting โปรแกรมทำงานบนวินโดวส์

ตัวอย่างนี้ทำงานเหมือน 5.11 แต่เป็นเวอร์ชันบนวินโดวส์

```

#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
    {"rice with ", "chicken curry, and ", "fish sauce."},
    {"wiskey with ", "lemonade, and ", "soda."},
    {"bread with ", "cheese, and ", "ketchup."},

```

```

        {"icecream with ", "banana, and ", "chocolate."});
int  isFull;
int  lock;
int  waiting[4];

void randomDelay(void);
void serve(char *dest, char *src);
DWORD WINAPI chefWork(LPVOID who);

int main(void){
    int i;
    DWORD tid[5];                // Thread ID
    HANDLE th[5];                // Thread Handle
    int param[5]={0,1,2,3,4};

    for(i=0;i<4;i++)
        waiting[i]=false;
    isFull=20;
    lock = false;

    // Create 5 threads
    for(i=0;i<4;i++)
        th[i] = CreateThread(
            NULL,                    // Default security attributes
            0,                        // Default stack size
            chefWork,                // Thread function
            (void *)&param[i],      // Thread function parameter
            0,                        // Default creation flag
            &tid[i]);                // Thread ID returned.

    // Wait until all threads finish
    for(i=0;i<4;i++)
        if(th[i]!=NULL)
            WaitForSingleObject(th[i], INFINITE);

    return 0;
}

void randomDelay(void){
    int stime = ((rand())%100)+1;
    Sleep(stime);
}

void serve(char *dest, char *src){
    int i;
    srand(time(NULL));

    for(i=0;(i<63)&&(src[i]!=0);i++){
        dest[i]=src[i];
        randomDelay();
    }
    dest[i]=0;
}

DWORD WINAPI chefWork(LPVOID who){
    int chefNo, i, j;
    int key;

    chefNo = (int)*((int *)who);

    for(i=0;i<(chefNo+2);i++) printf("      ");
    printf("Chef NO %d start working...\n", chefNo+1);
    fflush(stdout);

    while(isFull>0){
        waiting[chefNo]=true;
        key = true;
        while(waiting[chefNo] && key){
            key = InterlockedExchangeAdd((volatile LONG *)&lock, 1);
            Sleep(500);
            if(isFull<=0) break;
        }
        waiting[chefNo]=false;
        if(isFull>0){
            // Critical Section
            for(i=0;i<(chefNo+2);i++) printf("      ");
            printf("Chef NO %d is serving the food\n", chefNo+1);
            fflush(stdout);
        }
    }
}

```



```

        for(j=0;j<3;j++)
            serve(plate[j],chef[chefNo][j]);

        for(i=0;i<(chefNo+2);i++) printf(" ");
        printf("Chef NO %d has serve the food\n",chefNo+1);
        fflush(stdout);
        isFull--;
    }
    j = (chefNo+1)%4;
    while((j!=chefNo)&& !waiting[j])
        j = (j+1)%4;
    if(j==chefNo){
        lock = false;
        printf("Chef No %d has unlocked the
key.[%d][%d][%d][%d]\n",j+1,waiting[0],waiting[1],waiting[2],waiting[3]);
    }
    else{
        waiting[j]=false;
        printf("Passing to chef No %d
[%d][%d][%d][%d]\n",j+1,waiting[0],waiting[1],waiting[2],waiting[3]);
    }
    // Remaining Section
    Sleep((rand()%8000)+500);
}
return 0;
}

```

5.13 ตัวอย่างการแก้ไขปัญหาการแย่งชิงทรัพยากรร่วมกัน โดยใช้ locking mechanism มาตรฐานของระบบปฏิบัติการลินุกซ์

ตัวอย่างนี้เป็นการแสดงการใช้งาน SpinLock Read/Write Lock และ Mutex ซึ่งเป็นกลไกการจัดสรรการเข้าใช้งานส่วนวิกฤติของลินุกซ์ตามมาตรฐาน POSIX โดยเนื่องจากกลไกการใช้งานของทั้งสามนี้มีความคล้ายคลึงกันมาก เพื่อความสะดวกจึงเขียนโค้ดยู่รวมกันเป็นตัวย่อย และให้นักศึกษาลองใช้งานทีละตัวอย่างโดยปลดคอมเมนต์ตามชื่อที่ต้องการใน #define Directive ที่หัวโปรแกรมแล้วคอมไพล์และลองรันไปทีละครั้งดู

อนึ่ง กลไกในตัวอย่างหัวข้อนี้ เนื่องจากนิยามใช้งานเฉพาะใน PThread ดังนั้นหากต้องการใช้ locking mechanism ที่ทำงานได้ในระดับโพรเซส ให้ใช้ atomic instruction หรือใช้ named semaphore แทน

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

// Copyright 2011 by T.Sripramong. You can use this code for educational purpose only.
// This program shows how to use 3 techniques provided by Linux
// Spinlock is a basic mechanism for critical section where a thread can
// access the critical section while other threads have to be waiting in a loop (spinning)
// Read/Write Lock is another lock mechanism whereby multiple reads from several threads are
// allowed but only one writer allowed.
// Mutex is a general mechanism for critical section which is generally found in other OSes.

// #define USE_SPINLOCK
// #define USE_RWLOCK
#define USE_MUTEX

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
    {"rice with ","chicken curry, and ","fish sauce."},
    {"wiskey with ","lemonade, and ","soda."},
    {"bread with ","cheese, and ","ketchup."},
    {"icecream with ","banana, and ","chocolate."}};

int isFull;
int isReady;

#ifdef USE_SPINLOCK
pthread_spinlock_t spinlock;

```

```

#elif defined(USE_RWLOCK)
pthread_rwlock_t rwlock;
#else
pthread_mutex_t mutex;
#endif

void randomDelay(void);
void serve(char *dest, char *src);
void *chefWork(void *who);
void *customer(void *who);

int main(void){
    int i;
    int param[5]={0,1,2,3,4};
    pthread_t tid[5];      // Thread ID
    pthread_attr_t attr[5]; // Thread attributes

    isFull=false;          // Customer tell all chefs to stop making food
    isReady=false;         // Producer tell customer that food is ready
    // Locking mechanism, the lock starts with false
#ifdef USE_SPINLOCK
    pthread_spin_init(&spinlock,0);
#elif defined(USE_RWLOCK)
    pthread_rwlock_init(&rwlock,NULL);
#else
    pthread_mutex_init(&mutex,NULL);
#endif

    for(i=0;i<5;i++)
        pthread_attr_init(&attr[i]); // Get default attributes

    // Create 4 threads for producers
    for(i=0;i<4;i++)
        pthread_create(&tid[i],&attr[i],chefWork,(void *)&param[i]);
    // Create 1 threads for consumer
    pthread_create(&tid[4],&attr[4],customer,(void *)&param[i]);

    // Wait until all threads finish
    for(i=0;i<5;i++)
        pthread_join(tid[i],NULL);

#ifdef USE_SPINLOCK
    pthread_spin_destroy(&spinlock);
#elif defined(USE_RWLOCK)
    pthread_rwlock_destroy(&rwlock);
#else
    pthread_mutex_destroy(&mutex);
#endif

    return 0;
}

void randomDelay(void){
    int stime = ((rand()%100)+1)*1000;
    usleep(stime);
}

void serve(char *dest, char *src){
    int i;
    srand(time(NULL));

    for(i=0;(i<63)&&(src[i]!=0);i++){
        dest[i]=src[i];
        randomDelay();
    }
    dest[i]=0;
}

void *chefWork(void *who){
    int plateNo, chefNo, i, j;

    chefNo = (int)*((int *)who);

    for(i=0;i<(chefNo+2);i++) printf(" ");
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);

    while(!isFull){

```

```

        if(isReady) {
            randomDelay(); // Wait for food to be taken
        }
    #if defined(USE_SPINLOCK)
        pthread_spin_lock(&spinlock);
    #elif defined(USE_RWLOCK)
        pthread_rwlock_wrlock(&rwlock);
    #else
        pthread_mutex_lock(&mutex);
    #endif

    if((!isFull)&&(!isReady)){
        // Critical Section
        for(i=0;i<(chefNo+2);i++) printf("      ");
        printf("Chef NO %d is serving the food\n",chefNo+1);
        fflush(stdout);

        for(j=0;j<3;j++)
            serve(plate[j],chef[chefNo][j]);
        isReady=true; //Tel customer that the food is ready

        for(i=0;i<(chefNo+2);i++) printf("      ");
        printf("Chef NO %d has serve the food\n",chefNo+1);
        fflush(stdout);
    }else{
        // for(i=0;i<(chefNo+2);i++) printf("      ");
        // printf("Chef NO %d has to skip the searving\n",chefNo+1);
        // fflush(stdout);
    }
    #if defined(USE_SPINLOCK)
        pthread_spin_unlock(&spinlock);
    #elif defined(USE_RWLOCK)
        pthread_rwlock_unlock(&rwlock);
    #else
        pthread_mutex_unlock(&mutex);
    #endif

    if(isFull) break;
    // Remaining Section
    randomDelay();
    // sleep(300000*(chefNo+1)); // This will cause some chef to waiting indefinitely
    // for(i=0;i<(chefNo+2);i++) printf("      ");
    // printf("Chef NO %d has rested\n",chefNo+1);
    // fflush(stdout);
}
pthread_exit(0);
}

void *customer(void *who){
    int i,j;
    char dinner[256]={0};

    for(i=0;i<10;i++){
        // printf("Choochok waits for food to be served\n");
        // fflush(stdout);

        while(!isReady) randomDelay();

    #if defined(USE_SPINLOCK)
        pthread_spin_lock(&spinlock);
    #elif defined(USE_RWLOCK)
        pthread_rwlock_rdlock(&rwlock);
    #else
        pthread_mutex_lock(&mutex);
    #endif

        // Critical Section
        printf("Choochok starts grab a set of dinner\n");
        fflush(stdout);

        serve(dinner,plate[0]); dinner[63]=0;
        serve(dinner + strlen(dinner),plate[1]); dinner[127]=0;
        serve(dinner + strlen(dinner),plate[2]); dinner[191]=0;

        printf("Plate NO:%d Choochok is eating %s\n",i+1,dinner);
        fflush(stdout);

        isReady=false; // Food taken
    #if defined(USE_SPINLOCK)

```

```

        pthread_spin_unlock(&spinlock);
#elif defined(USE_RWLOCK)
        pthread_rwlock_unlock(&rwlock);
#else
        pthread_mutex_unlock(&mutex);
#endif
    // Remaining Section
    for(j=0;j<10;j++) randomDelay();
    // printf("Choochok is moving to the next table\n");
    // fflush(stdout);
}
isFull = true; // Tell other producer threads to stop;
pthread_exit(0);
}

```

5.14 ตัวอย่างการแก้ไขปัญหาการแย่งชิงทรัพยากรร่วมกัน โดยใช้ Mutex โปรแกรมทำงานบนวินโดวส์

ตัวอย่างต่อไปนี้เป็นการใช้ mutex สำหรับการจัดการส่วนวิกฤติ โดยสามารถใช้ได้กับโปรแกรมที่แยกหลายโปรเซสทำงานพร้อมกัน หรือใช้กับโปรเซสที่มีการแตกเธรด แต่ทั้งนี้ในกรณีหลัง ควรไปใช้ critical section ซึ่งเป็นกลไกที่ทำงานได้เร็วกว่า แต่ไม่สามารถทำงานข้ามโปรเซสได้

```

#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
    {"rice with ", "chicken curry, and ", "fish sauce."},
    {"wiskey with ", "lemonade, and ", "soda."},
    {"bread with ", "cheese, and ", "ketchup."},
    {"icecream with ", "banana, and ", "chocolate."};
int isFull;
int isReady;
HANDLE hMutex;

void randomDelay(void);
void serve(char *dest, char *src);
DWORD WINAPI chefWork(LPVOID who);
DWORD WINAPI customer(LPVOID who);

int main(void){
    int i;
    DWORD tid[5]; // Thread ID
    HANDLE th[5]; // Thread Handle
    int param[5]={0,1,2,3,4};

    hMutex = CreateMutex(
        NULL, // default security attributes
        FALSE, // initially not owned
        NULL); // unnamed mutex

    if (hMutex == NULL){
        printf("Can not create a mutex.\n");
        return 255;
    }

    // Create 5 threads
    for(i=0;i<4;i++)
        th[i] = CreateThread(
            NULL, // Default security attributes
            0, // Default stack size
            chefWork, // Thread function
            (void *)&param[i], // Thread function parameter
            0, // Default creation flag
            &tid[i]); // Thread ID returned.

    th[4] = CreateThread(
        NULL, // Default security attributes
        0, // Default stack size
        customer, // Thread function
        (void *)&param[4], // Thread function parameter
        0, // Default creation flag
        &tid[4]); // Thread ID returned.
}

```

```

        // Wait until all threads finish
        for(i=0;i<5;i++)
            if(th[i]!=NULL)
                WaitForSingleObject(th[i],INFINITE);

        CloseHandle(hMutex);
        return 0;
    }

void randomDelay(void){
    int stime = ((rand()%100)+1);
    Sleep(stime);
}

void serve(char *dest,char *src){
    int i;
    srand(time(NULL));

    for(i=0;(i<63)&&(src[i]!=0);i++){
        dest[i]=src[i];
        randomDelay();
    }
    dest[i]=0;
}

DWORD WINAPI chefWork(LPVOID who){
    int chefNo,i,j;
    DWORD dwWaitResult;

    chefNo = (int)*((int *)who);

    for(i=0;i<(chefNo+2);i++) printf(" ");
    printf("Chef NO %d start working...\n",chefNo+1);
    fflush(stdout);

    while(!isFull){

        if(isReady) {
            randomDelay(); // Wait for food to be taken
        }

        dwWaitResult = WaitForSingleObject(
            hMutex, // Mutex handle
            10000L); // 10 Seconds (use INFINITY if no time limited)

        switch(dwWaitResult){
            case WAIT_OBJECT_0: // Can obtain the mutex within time
                __try{
                    // Critical Section
                    if((!isFull)&&(!isReady)){
                        for(i=0;i<(chefNo+2);i++) printf(" ");
                        printf("Chef NO %d is serving the food\n",chefNo+1);
                        fflush(stdout);

                        for(j=0;j<3;j++)
                            serve(plate[j],chef[chefNo][j]);
                        isReady=true; //Tel customer that the food is ready

                        for(i=0;i<(chefNo+2);i++) printf(" ");
                        printf("Chef NO %d has serve the food\n",chefNo+1);
                        fflush(stdout);
                    }
                }__finally{
                    if(!ReleaseMutex(hMutex)){
                        printf("ReleaseMutex error!\n");
                    }
                }
                break;
            case WAIT_TIMEOUT: //For timeout
            case WAIT_ABANDONED: //For ownership of abandoned object
                break;
        }
        // Remaining Section

        if(isFull) break;
        randomDelay(); // This will cause some chef to waiting indefinitely;
    }
}

```

```

    return 0;
}

DWORD WINAPI customer(LPVOID who) {
    int i, j;
    char dinner[256]={0};
    DWORD dwWaitResult;

    for(i=0;i<10;i++){
        // printf("Choochok waits for food to be served\n");
        // fflush(stdout);

        while(!isReady) randomDelay();

        dwWaitResult = WaitForSingleObject(
            hMutex, // Mutex handle
            10000L); // 10 Seconds (use INFINITY if no time limited)

        switch(dwWaitResult){
            case WAIT_OBJECT_0: // Can obtain the mutex within time
                __try{
                    // Critical Section

                    printf("Choochok starts grab a set of dinner\n");
                    fflush(stdout);

                    serve(dinner,plate[0]); dinner[63]=0;
                    serve(dinner + strlen(dinner),plate[1]); dinner[127]=0;
                    serve(dinner + strlen(dinner),plate[2]); dinner[191]=0;

                    printf("Plate NO:%d Choochok is eating %s\n",i+1,dinner);
                    fflush(stdout);

                    isReady=false; // Food taken
                }
                __finally{
                    if(! ReleaseMutex(hMutex)){
                        printf("ReleaseMutex error!\n");
                    }
                }
                break;
            case WAIT_TIMEOUT: //For timeout
            case WAIT_ABANDONED: //For ownership of abandoned object
                break;
        }
        // Remaining Section
    }

    isFull = true; // Tell other producer threads to stop;
    return 0;
}

```

5.15 ตัวอย่างการแก้ไขปัญหาการแย่งชิงทรัพยากรร่วมกัน โดยใช้ Critical Section โปรแกรมทำงานบนวินโดวส์

ตัวอย่างต่อไปนี้เป็นการใช้ข้อจำกัด Critical Section ซึ่งเหมาะสมสำหรับการเขียนโปรแกรมแบบหลายเธรดบนวินโดวส์ เพราะสามารถทำงานได้ไวกว่า และการใช้งานนี้มีความง่ายและสะดวกกว่าการใช้ mutex

```

#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>

char plate[3][64]; // Consumer table
char chef[4][3][64] = {
    {"rice with ", "chicken curry, and ", "fish sauce."},
    {"wiskey with ", "lemonade, and ", "soda."},
    {"bread with ", "cheese, and ", "ketchup."},
    {"icecream with ", "banana, and ", "chocolate."}};

int isFull;
int isReady;

```

```

CRITICAL_SECTION cSection;

void randomDelay(void);
void serve(char *dest, char *src);
DWORD WINAPI chefWork(LPVOID who);
DWORD WINAPI customer(LPVOID who);

int main(void){
    int i;
    DWORD tid[5];           // Thread ID
    HANDLE th[5];           // Thread Handle
    int param[5]={0,1,2,3,4};

    InitializeCriticalSection( &cSection);

    // Create 5 threads
    for(i=0;i<4;i++){
        th[i] = CreateThread(
            NULL,                // Default security attributes
            0,                   // Default stack size
            chefWork,             // Thread function
            (void *)&param[i],   // Thread function parameter
            0,                   // Default creation flag
            &tid[i]);            // Thread ID returned.

        th[4] = CreateThread(
            NULL,                // Default security attributes
            0,                   // Default stack size
            customer,            // Thread function
            (void *)&param[4],   // Thread function parameter
            0,                   // Default creation flag
            &tid[4]);            // Thread ID returned.

        // Wait until all threads finish
        for(i=0;i<5;i++){
            if(th[i]!=NULL)
                WaitForSingleObject(th[i], INFINITE);
        }
        DeleteCriticalSection(&cSection);
        return 0;
    }

    void randomDelay(void){
        int stime = ((rand()%100)+1);
        Sleep(stime);
    }

    void serve(char *dest, char *src){
        int i;
        srand(time(NULL));

        for(i=0; i<63; i++){
            dest[i]=src[i];
            randomDelay();
        }
        dest[i]=0;
    }

    DWORD WINAPI chefWork(LPVOID who){
        int chefNo, i, j;

        chefNo = (int)*((int *)who);

        for(i=0; i<(chefNo+2); i++) printf(" ");
        printf("Chef NO %d start working...\n", chefNo+1);
        fflush(stdout);

        while(!isFull){

            if(isReady) {
                randomDelay(); // Wait for food to be taken
            }

            EnterCriticalSection(&cSection);
            // Critical Section
            if((!isFull)&&(!isReady)){
                for(i=0; i<(chefNo+2); i++) printf(" ");
                printf("Chef NO %d is serving the food\n", chefNo+1);
                fflush(stdout);
            }
        }
    }
}

```

```

        for(j=0;j<3;j++)
        serve(plate[j],chef[chefNo][j]);
        isReady=true; //Tel customer that the food is ready

        for(i=0;i<(chefNo+2);i++) printf("      ");
        printf("Chef NO %d has serve the food\n",chefNo+1);
        fflush(stdout);
    }
    LeaveCriticalSection(&cSection);
    // Remaining Section

    if(isFull) break;
    randomDelay(); // This will cause some chef to waiting indefinitely;
}
return 0;
}

DWORD WINAPI customer(LPVOID who){
    int i,j;
    char dinner[256]={0};

    for(i=0;i<10;i++){
//      printf("Choochok waits for food to be served\n");
//      fflush(stdout);

        while(!isReady) randomDelay();

        EnterCriticalSection(&cSection);
        // Critical Section

        printf("Choochok starts grab a set of dinner\n");
        fflush(stdout);

        serve(dinner,plate[0]); dinner[63]=0;
        serve(dinner + strlen(dinner),plate[1]); dinner[127]=0;
        serve(dinner + strlen(dinner),plate[2]); dinner[191]=0;

        printf("Plate NO:%d Choochok is eating %s\n",i+1,dinner);
        fflush(stdout);

        isReady=false; // Food taken

        LeaveCriticalSection(&cSection);
        // Remaining Section
    }
    isFull = true; // Tell other producer threads to stop;
    return 0;
}

```