นายภฤศ ตัณฑ์วรกุล 6601012610083

Producer-Consumer Problem

หัวข้อ

ส่วนที่ 1 Output ผลการรันโค้ด

1. แสดงให้เห็นการเกิด Race condition

1.1.การทดลองด้วย Process

Output (ผลการรันโค้ด)

```
...
in:3 out:0 | Count:3 Exp:3 | [ 1 1 1 0 0 ]
in:3 out:1 | Count:2 Exp:2 | [ 0 1 1 0 0 ]
in:4 out:2 | Count:3 Exp:2 | [ 0 0 1 1 0 ]

>>> RACE CONDITION DETECTED (Data Corrupted!) <<<
```

```
in:3 out:1 | Count:2 Exp:2 | [ 0 1 1 0 0 ]
in:4 out:2 | Count:3 Exp:2 | [ 0 0 1 1 0 ]
>>> RACE CONDITION DETECTED (Data Corrupted!) <<<
```

คำอธิบายค่าที่แสดง

- in คือ ค่า Index ของ Producer
- out คือ ค่า Index ของ Consumer
- Count คือ Shared value (ค่าที่แบ่งกัน) ใช้บอกจำนวนของ Buffer
- Exp คือ Expect Value (ค่าที่คาดหวัง) ของ Count ใช้เพื่อตรวจสอบ Race condition โดยเฉพาะ

คำอธิบายสิ่งที่เกิดขึ้น

in:4 out:2 | Count:3 Exp:2 | [0 0 1 1 0] ที่ตำแหน่งนี้เกิดการทำงานของ Producer และ Consumer พร้อมๆ กันโดยแบ่งกรณีเป็นดังนี้

เหตุการณ์ก่อน in:3 out:1 | Count:2 Exp:2 | [0 1 1 0 0] จะเห็นได้ว่า Count = 2	
Producer เปลี่ยน Count = 2 + 1 = 3	Consumer เปลี่ยน Count = 2 – 1 = 1
จริงๆ ผลลัพธ์เกิดได้ทั้ง 3, 1 แต่จากการรันดังกล่าวได้ผลเป็น 3	

ถ้าทำงานแบบ Serial (อนุกรม) หรือไม่เกิดไม่ทำงานพร้อมกันจะได้ผลลัพธ์ที่ถูกต้องดังนี้

A. Consumer เปลี่ยน Count = 2 – 1 = 1
B. Producer เปลี่ยน Count = 1 + 1 = 2

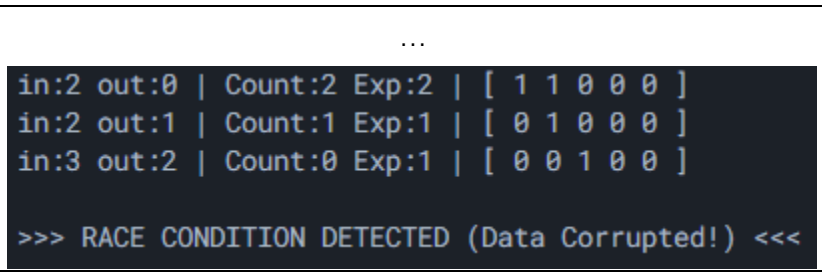
หรือ

A. Producer เปลี่ยน Count = 2 + 1 = 3
B. Consumer เปลี่ยน Count = 2 - 1 = 2

แบบนี้จึงจะได้ผลลัพธ์เป็น 2 แบบที่ถูกต้อง

1.2.การทดลองด้วย Thread

Output (ผลการรันโค้ด)



in:2 out:1 | Count:1 Exp:1 | [0 1 0 0 0]

in:3 out:2 | Count:0 Exp:1 | [0 0 1 0 0]

>>> RACE CONDITION DETECTED (Data Corrupted!) <<<

คำอธิบายค่าที่แสดง

เหมือนตอนทดลองด้วย Process 1.1.

คำอธิบายสิ่งที่เกิดขึ้น

in:3 out:2 | Count:0 Exp:1 | [0 0 1 0 0] ที่ตำแหน่งนี้เกิดการทำงานของ Producer และ Consumer พร้อมๆ กันโดยแบ่งกรณีเป็นดังนี้

เหตุการณ์ก่อน in:2 out:1 | Count:1 Exp:1 | [0 1 0 0 0] จะเห็นได้ว่า Count = 1	
Producer เปลี่ยน Count = 1 + 1 = 2	Consumer เปลี่ยน Count = 1 – 1 = 0
จริงๆ ผลลัพธ์เกิดได้ทั้ง 2, 0 แต่จากการรันดังกล่าวได้ผลเป็น 0	

ถ้าทำงานแบบ Serial (อนุกรม) หรือไม่เกิดไม่ทำงานพร้อมกันจะได้ผลลัพธ์ที่ถูกต้องดังนี้

A. Consumer เปลี่ยน Count = 1 – 1 = 0
B. Producer เปลี่ยน Count = 0 + 1 = 1

หรือ

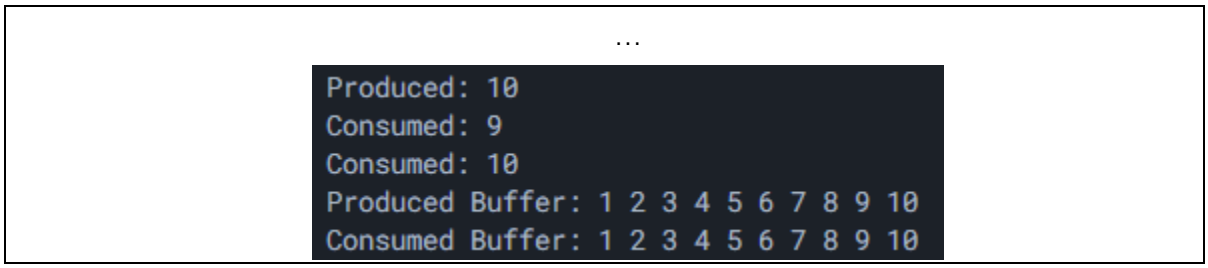
A. Producer เปลี่ยน Count = 1 + 1 = 2
B. Consumer เปลี่ยน Count = 2 – 1 = 1

แบบนี้จึงจะได้ผลลัพธ์เป็น 1 แบบที่ถูกต้อง

2. แสดงการใช้ Semaphores

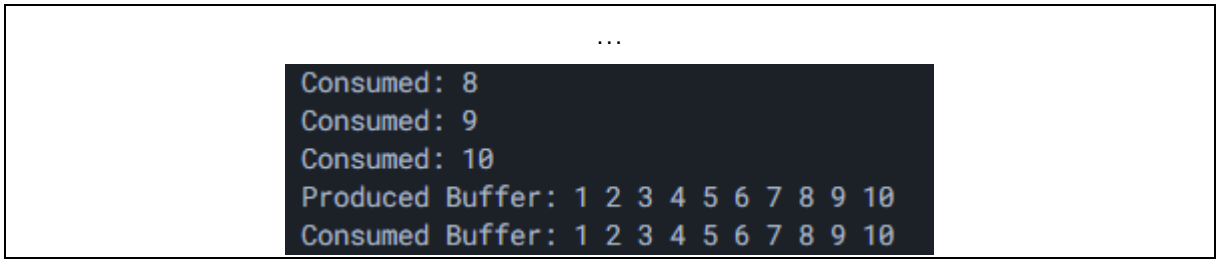
2.1. การทดลองแบบ Process

Output (**ผลการรันโค้ด**)



2.2. การทดลองแบบ Thread

Output (**ผลการรันโค้ด**)

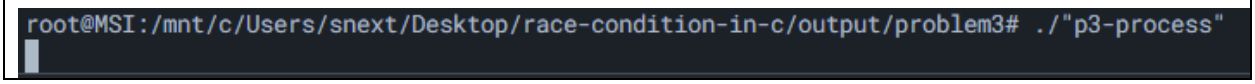


3. ใช้ code จาก midterm ข้อ 6b. เพื่อแสดงการเกิด Deadlock

3.1. การทดลองแบบ Process

Output (**ผลการรันโค้ด**)

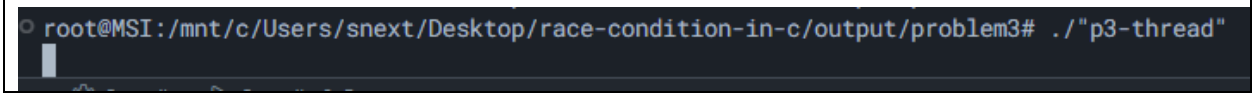
โปรแกรมไม่ Print อะไรออกมาเลย



3.2. การทดลองแบบ Thread

Output (**ผลการรันโค้ด**)

โปรแกรมไม่ Print อะไรออกมาเลย



ส่วนที่ 2 Source code และการอธิบาย

1. Race Condition ในปัญหา Producer-Consumer

```
#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE]={0};
int count =0;
int in=0;
int out=0;

long total_produced =0;
long total_consumed =0;
```

เนื่องจากเราต้องการจะจับผิด Count ว่าผิดหรือไม่? เราจึงมีค่าที่ไม่ได้แชร์กันได้แก่

```
long total_produced =0;
long total_consumed =0;
```

โดยการตรวจสอบค่าพวกนี้เราจะเพิ่มทุกครั้งที่ Producer, Consumer อัพเดตค่า

```
void *producer(void *arg){
    while(1){
        while(count == BUFFER_SIZE) usleep(100);

        buffer[in]=1;
        in=(in+1)% BUFFER_SIZE;
        total_produced++;
        count++;

. . .
    }
}

```

```
void *consumer(void *arg){
    while(1){
        while(count ==0) usleep(100);

        buffer[out]=0;
        out=(out+1)% BUFFER_SIZE;
        total_consumed++;

        count--;
. . .
    }
}

```

การตรวจสอบให้พบค่า Count ที่ผิดพลาด เนื่องจากค่า `total_produced, total_consumed` เป็นค่าที่ไม่ได้แชร์กัน จึงถือว่าเชื่อถือได้ว่าจะเป็นค่า Count ที่แม่นยำเพราะไม่มีทางเกิด Race Condition ที่ค่า `total_produced, total_consumed`

```
int expected_count = total_produced - total_consumed;
```

การตรวจสอบต้องประกอบด้วย

- เช็คว่า count = expected_count ไหม?
- เช็คว่า count > 0 ไหม?
- เช็คว่า count เกิน Buffer ไหม?

```
if(count != expected_count || count < 0 || count > BUFFER_SIZE){
        exit(1);
}
```

```
void *producer(void *arg){
    while(1){
        while(count == BUFFER_SIZE) usleep(100);

        buffer[in]=1;
        in=(in+1)% BUFFER_SIZE;
        total_produced++;
        count++;

        // ตรวจสอบ
    }
}

```

```
void *consumer(void *arg){
    while(1){
        while(count ==0) usleep(100);

        buffer[out]=0;
        out=(out+1)% BUFFER_SIZE;
        total_consumed++;

        count--;
      // ตรวจสอบ
    }
}
```

```
19.
```

2. การใช้ Semaphore เขียนตามไสลด์เรียน

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5
#define ITERATIONS 10

struct shared_data {
    int buffer[BUFFER_SIZE];
    int in;
    int out;
    bool produce_sucess;

    sem_t mutex;
    sem_t empty;
    sem_t full;
};

struct shared_data shared;

void* producer(void* arg){

    int produced_buffer[ITERATIONS];

    for(int i =0; i < ITERATIONS; i++){

        int next_produced = i +1;

        //entry section
        sem_wait(&shared.empty);
        sem_wait(&shared.mutex);

        shared.buffer[shared.in] = next_produced;
        shared.in =(shared.in +1)% BUFFER_SIZE;

        //exit section
        sem_post(&shared.mutex);
        sem_post(&shared.full);

        produced_buffer[i] = next_produced;
        printf("Produced: %d\n", next_produced);
    }

    sleep(1);

    printf("Produced Buffer: ");
    for(int i =0; i <ITERATIONS; i++){
        printf("%d ", produced_buffer[i]);
    }
```

```
53.     printf("\n");
54. 
55.     shared.produce_sucess =true;
56. 
57.     pthread_exit(NULL);
58. }
59. 
60. void* consumer(void* arg){
61. 
62.     int consumed_buffer[ITERATIONS];
63. 
64.     for(int i =0; i < ITERATIONS; i++){
65. 
66.         //entry section
67.         sem_wait(&shared.full);
68.         sem_wait(&shared.mutex);
69. 
70.         int next_consumed = shared.buffer[shared.out];
71.         shared.out=(shared.out+1)% BUFFER_SIZE;
72. 
73.         //exit section
74.         sem_post(&shared.mutex);
75.         sem_post(&shared.empty);
76. 
77.         consumed_buffer[i]= next_consumed;
78.         printf("Consumed:%d\n", next_consumed);
79.     }
80. 
81.     while(!shared.produce_sucess){
82.         usleep(1000);
83.     }
84. 
85.     printf("Consumed Buffer:");
86.     for(int i =0; i < ITERATIONS; i++){
87.         printf("%d ", consumed_buffer[i]);
88.     }
89.     printf("\n");
90. 
91.     pthread_exit(NULL);
92. }
```

การเขียนนี้รับประกันว่าจะไม่เกิด Race condition โดยรับประกันว่า

- มีแค่ 1 Process/Thread ใน Critical section
- Producer จะเขียนไม่เกิน Buffer และรอ Buffer ว่างก่อนเขียน
- Consumer จะไม่อ่านตอน Buffer ว่าง

3. การเกิด Dead lock ของโจทย์ Midterm

```
1. void* producer(void* arg){
2.
3.     int i =0;
4.     while(1){
5.
6.         int next_produced = i +1;
7.
8.         //entry section
9.         sem_wait(&shared.mutex);
10.
11.        shared.buffer[shared.in]=next_produced;
12.        shared.in=(shared.in+1)% BUFFER_SIZE;
13.
14.        //exit section
15.        sem_wait(&shared.empty);
16.        sem_post(&shared.mutex);
17.
18.        i++;
19.    }
20.
21.    shared.produce_sucess =true;
22.    pthread_exit(NULL);
23. }
24.
25. void* consumer(void* arg){
26.
27.    int consumed_count =0;
28.
29.    while(1){
30.        sleep(1);
31.        sem_wait(&shared.mutex);
32.
33.        int next_consumed = shared.buffer[shared.out];
34.        shared.out=(shared.out+1)% BUFFER_SIZE;
35.
36.        sem_post(&shared.mutex);
37.        sem_post(&shared.empty);
38.
39.        printf("Consumed:%d\n", next_consumed);
40.    }
41.
42.    pthread_exit(NULL);
43. }
44.
```

ล็อคผลการเกิด Deadlock ด้วยการหน่วงเวลา เพื่อให้ Consumer เริ่มช้ามากๆ จน Producer ผลิต Buffer จนเต็ม แต่ปัญหาคือ Producer จะต้องรอ empty จาก Consumer แต่ Consumer ก็รอ mutex