รายงานการออกแบบโปรแกรม

รายวิชา ระบบปฏิบัติการ รหัสวิชา 01204332

สมาชิกในกลุ่ม

ชื่อกลุ่ม A2-05

นายเมธิส	อักษรวงศ์	รหัสนิสิต	6140200228
นางสาวนลินภัสร์	ภิศักดิ์ธนากูล	รหัสนิสิต	6140203057
นายศุภฤกษ์ชัย	อินธิแสง	รหัสนิสิต	6140205805
นายสนายุทธ	ต้นพนม	รหัสนิสิต	6140205888
นายสุทธิชัย	นพคุณ	รหัสนิสิต	6140206274
นางสาวภัทรวรรณ	โมระดา	รหัสนิสิต	6040203579

เสนอ

อาจารย์ สรยุทธ กลมกล่อม

รายงานฉบับนี้เป็นส่วนหนึ่งของรายวิชา ระบบปฏิบัติการ รหัสวิชา 01204332

สาขาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมไฟฟ้าและคอมพิวเตอร์

มหาวิทยาลัยเกษตรศาสตร์ วิทยาเขตเฉลิมพระเกียรติจังหวัดสกลนคร

คำนำ

รายงานฉบับนี้เป็นส่วนหนึ่งของรายวิชา ระบบปฏิบัติการ รหัสวิชา 01204332 มีจุดประสงค์เพื่อรายงานผลการวิเคราะห์การออกแบบ ซึ่งภายในรายงานฉบับนี้มีเนื้อหาเกี่ยวกับการออกแบบ เงื่อนไข วิธีการทำงานของฟังก์ชันต่างๆ อธิบายความถูกต้องของโปรแกรม ผลลัพธ์ของการรันโปรแกรม ซอร์สโค้ดของโปรแกรม

ผู้จัดทำได้ทำการศึกษาการทำงานของ Mutual Exclution โดยการเขียนโปรแกรม Producer and Consumer เพื่อศึกษาการออกแบบโปรแกรม เงื่อนไข การทำงานของ Append และRemove และการใช้ Semaphore เพื่อแก้ไขปัญหา Race condition , Mutually exclusive access to buffer และใช้ Sleep/Wakeup ในการแก้ปัญหาแก้รอคอยของเทรด

โดยหวังเป็นอย่างยิ่งว่าการจัดทำรายงานฉบับนี้จะเป็นประโยชน์กับผู้อ่านไม่มากก็น้อย ที่กำลังศึกษาหาข้อมูลเรื่องนี้อยู่ หากมีข้อผิดพลาดประการใด ผู้จัดทำขอน้อมรับและขออภัยมา ณ ที่นี้ด้วย

คณะผู้จัดทำ

สารบัญ

1.การออกแบบโปรแกรม

การพัฒนาโปรแกรมในครั้งนี้ ใช้ภาษา Python ในการเขียน Library ที่ใช้ มีดังนี้

- Thread ใช้ในการทำ Multithread
- Event ใช้ในการแก้ปัญหา busy-waiting โดยใช้ฟังก์ชั่น sleep/wake up
- Semaphore ใช้แก้ปัญหา Race condition โดยใช้ฟังก์ชั่น acquire/release

โดยหลักการสร้างโปรแกรม producer and consumer มีดังนี้

(1) สร้าง circular buffer โดยมีฟังก์ชัน add_item กับ remove_item ใช้ เพิ่ม/ลบ ของจาก circular buffer

(2) สร้างฟังก์ชัน append กับ remove ในการใช้เรียก ฟังก์ชัน append กับ remove

(3) การแก้ปัญหา race condition กับการเข้าถึง Critical Region ด้วย Semaphore

(4) การแก้ปัญหา busy waiting ด้วย sleep/wake up

(5) สร้าง Multithreading ในการรันฟังก์ชัน append กับ remove

1.1การออกแบบ Circular Buffer

บัฟเฟอร์แบบวงกลม (Circular Buffer) เป็นโครงสร้างข้อมูลประเภทหนึ่ง ซึ่งเป็นการจัดการข้อมูลในรูปแบบวงกลม แต่ขนาดหรือความจุของข้อมูลจะต้องจำกัดไม่สามารถเพิ่มขึ้นได้ หลักการทำงานส่วนใหญ่จะเป็นการเก็บข้อมูลแบบตามลำดับ จนข้อมูลเต็มก็จะทำการเพิ่มข้อมูลไปใหม่ โดยใช้หลักการ FIFO (FIRST IN FIRST OUT) หรือ มาก่อนก็ออกก่อน

คุณสมบัติพื้นฐานของ Circular Buffer

(1) กำหนด Pointer 2 ตัว คือ Top (ทางออกของ Circular Buffer) กับ Botton (ทางเข้าของ Circular Buffer)

(2) การกำหนดขนาดของ Circular Buffer

หลักการทำงานของบัฟเฟอร์

เริ่มต้นตำแหน่ง Pointer Top และ Botton มีค่าตั้งต้นเป็น 0 (No Item) จะมีการกระทำกับ Circular Buffer อยู่ 2 แบบ คือ

(1) เพิ่มของเข้าไปในบัฟเฟอร์ โดยการเพิ่ม จะเพิ่มของเข้าไปที่ตำแหน่ง Botton และค่า Botton จะเพิ่มขึ้น 1 ตำแหน่ง เมื่อเพิ่มของเข้าไปเรื่อยๆ ถ้าตำแหน่งของ Botton เท่ากับ ขนาดของ Buffer ตำแหน่งของ Botton จะกลับเป็นเป็น 0

(2) ลบของออกจากบัพเฟอร์ เมื่อมีการลบของออกจาก Buffer ตำแหน่ง Top จะเพิ่มขึ้น 1 ตำแหน่ง และถ้าหากทำการลบไปเรื่อยๆ เมื่อตำแหน่งของ Top เท่ากับ ขนาดของ Buffer ตำแหน่งของ Top จะกลับไปเป็น 0

```
class circular_buffer:
    def __init__(self,buffer_size):
        self.top = 0
        self.botton  = 0

        self.buffer_size = buffer_size

        self.buffer = [' '] * buffer_size

    def add_item(self,item):
        if self.buffer[self.botton] == ' ':
            self.buffer[self.botton] = item
            self.botton += 1
            if self.botton >= self.buffer_size:
                self.botton = 0
        else:
            print("over item")
            return None

    def remove_item(self):
        if self.buffer[self.top]  != ' ':
            data = self.buffer[self.top]
            self.buffer[self.top] = ' '
            self.top += 1
            if self.top >= self.buffer_size:
                self.top = 0
            return data
        else:
            print("none item")
            return None

    def display(self):
        return self.buffer
```

Library Circular Buffer (ชื่อไฟล์ Circular_Buffer.py)

ตัวอย่างการใช้งาน Library Circular Buffer

```
1  from circular_buffer import circular_buffer
2  buffer = circular_buffer(10)
3
4  buffer.add_item('x')
5
6  #buffer.remove_item()
7
8  # for i in range(10):
9  #     buffer.add_item('x')
10
11 # for i in range(10):
12 #     buffer.remove_item()
13
14 print(buffer.display())
```

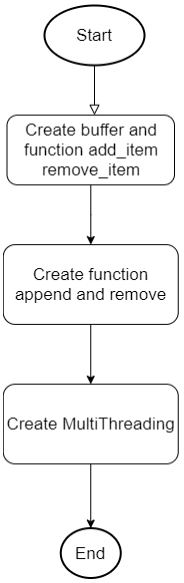
1.2 การออกแบบ Append

- append จะทำงานตามจำนวนรอบของ request ที่กำหนด
- append จะมีการเรียกใช้ฟังก์ชัน add_item
- มีการป้องกันการเข้าถึง Critical Region และแก้ปัญหา race condition ด้วยการครอบ Critical Region ไว้ในข้างใน Semaphore และเมื่อบัพเฟอร์เต็มทำให้เรียกใช้ฟังก์ชัน add_item ไม่ได้ ต้องมีการใช้ฟังก์ชัน sleep มาช่วยในการรอฟังก์ชัน remove ทำงานเสร็จ ถ้ามีของในบัพเฟอร์จะมีเรียกใช้ฟังก์ชัน wake_up ในการปลุก remove ขึ้นมาทำงาน

1.3 การออกแบบ Remove

- remove จะทำงานตามจำนวนรอบของ request ที่กำหนด
- remove จะมีการเรียกใช้ฟังก์ชัน remove_item
- มีการป้องกันการเข้าถึง Critical Region และแก้ปัญหา race condition ด้วยการครอบ Critical Region ไว้ในข้างใน Semaphore และเมื่อบัพเฟอร์มีพื้นที่ว่างทั้งหมด ทำให้เรียกใช้ฟังก์ชัน remove_item ไม่ได้ ต้องมีการใช้ฟังก์ชัน sleep มาช่วยในการรอฟังก์ชัน append ทำงานเสร็จ ถ้ามีพื้นที่ว่างบัพเฟอร์จะมีเรียกใช้ฟังก์ชัน wake_up ในการปลุก append ขึ้นมาทำงาน

1.4 Flowchart ของโปรแกรม



รูปที่ 3 อธิบายการทำงานของโปรแกรม

2. เงื่อนไข วิธีการทำงาน และการพิสูจน์คุณสมบัติของ Append

ฟังก์ชัน Append

```
def append(name,req):
    global count,size
    global count_request_producer
    while req > 0:
        req -=1

        if count > size-1:
            time_out = event_producer.sleep(seconds=5)

            if time_out == False:
                continue

        semaphore.acquire()
        if count > size-1:
            req +=1
            semaphore.release()
            continue
        count_request_producer +=1

        buffer.add_item('x')
        print("request",count_request_producer,f'producer-{name}',buffer.display())
        #print("request",count_request_producer)
        count +=1
        semaphore.release()
        sleep(0.05)
        if count > 0:
            event_consumer.wake_up()
```

2.1 เงื่อนไขของ Append

เงื่อนไขการทำงานของ Append

(1) ทำงานตาม request ที่กำหนด

(2) ณ ช่วงเวลาหนึ่ง จะสามารถเข้าถึงบัพเฟอร์ได้เพียงเทรดเดียวเท่านั้น

(3) ทุกเทรดมีเวลารอที่จำกัดในการทำงาน

2.2 วิธีการทำงานของ Append

(1) Appendมีการรับพารามิเตอร์ 2ตัว คือ name = ชื่อของเทรด,req คือ จำนวนรีเควสที่แบ่งให้เทรดนี้

(2) มีการเรียกใช้ share variable มี count,size,count_request_producer
count กับ size มีความสัมพันธ์กัน count คือ จำนวนของที่อยู่ในบัพเฟอร์ปัจจุบัน size คือ ขนาดของบัพเฟอร์ count_request_producer ใช้นับจำนวนรีเควสของ Producer ณ ปัจจุบัน

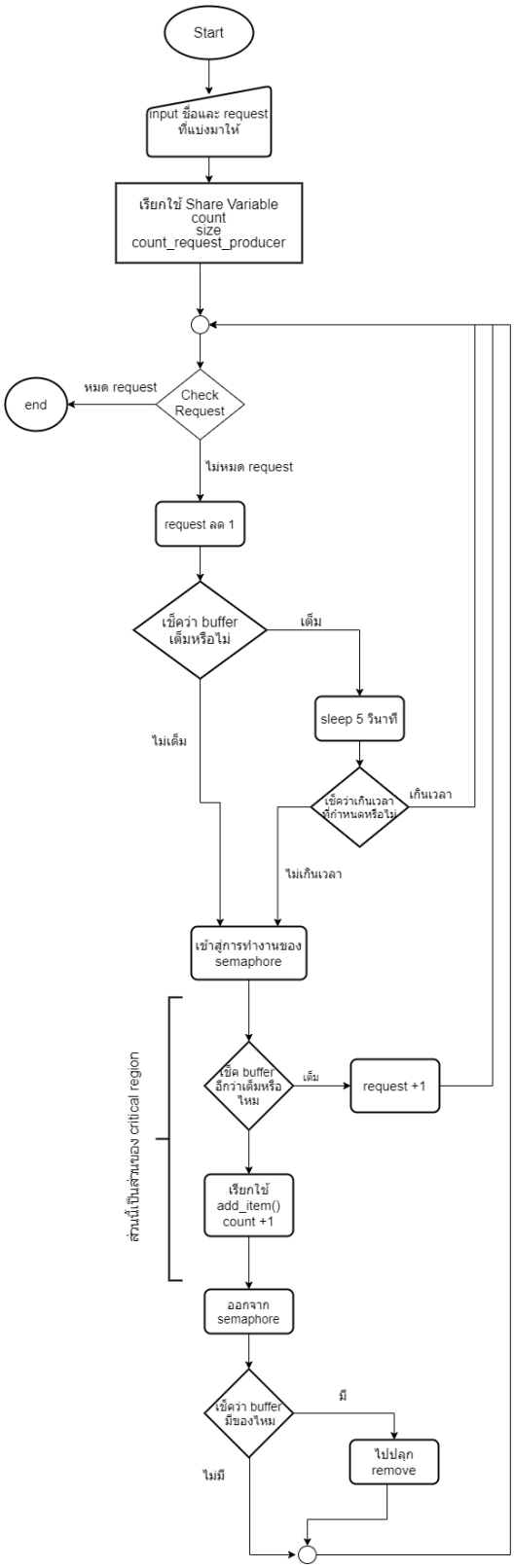
(3) มีการเช็ครอบรีเควสด้วย while loop ทุกครั้งที่วน 1 รอบ จะลดค่ารีเควสลง 1 ค่า

(4) มีการตรวจสอบจำนวนของในบัพเฟอร์ ว่าเกินขนาดที่บรรจุได้หรือไม่ ถ้าเกินเทรดนั้นจะเข้าสู่โหมด sleep เป็นเวลา 5 วินาที ถ้าไม่มีใครมาปลุกภายใน 5 วินาทีจะเริ่มรอบรีเควสใหม่ ถ้าตื่นทันจะเข้าสู่ semaphore เพื่อรอเข้า Critical Region

(5) เมื่อมี Process เข้า Critical Region จะทำการเรียกใช้ add_item และเพิ่มค่า count+1 และออกจาก semaphore

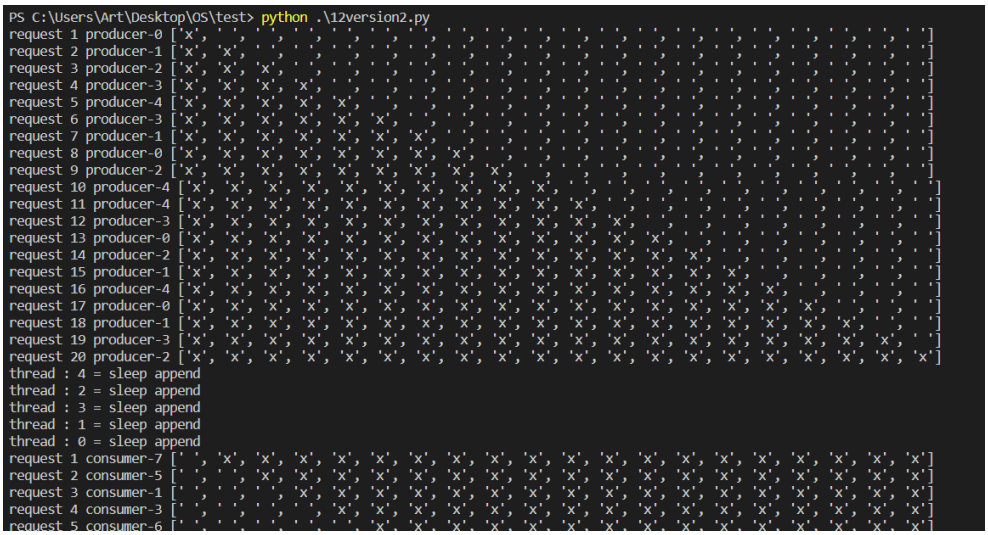
(6) เช็คจำนวนบัพเฟอร์ว่ามีของหรือไม่ ถ้ามี ให้ไปปลุกฟังก์ชัน remove ให้ตื่นมาทำงาน

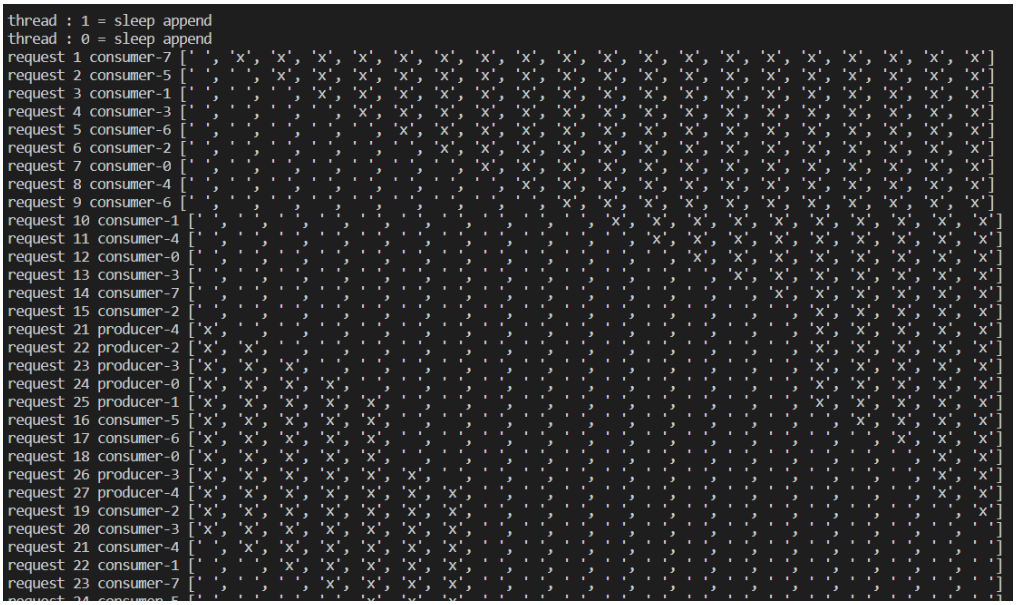
Flowchart การทำงานของ append



2.3 การพิสูจน์คุณสมบัติของ Append

กำหนดให้ Producer มีอยู่ 5 เทรด หลังจากรันครั้งแรก ให้ consumer ทำการ sleep รอ 10 วินาที และกำหนดการ sleep ของ append 15 วินาที ผลที่ได้คือ Producer จะมีการทำงานจนบัฟเฟอร์เต็ม ทำให้ต้อง sleep append รอ หลังจากนั้น 10 วินาที consumer ตื่นขึ้นมา จะมีการเอาของออกจากบัฟเฟอร์ เมื่อของว่างจะทำการปลุก consumer และทำการ context switch กลับไปกลับมา





3.เงื่อนไขการทำงานของ Remove

ฟังก์ชัน remove

```
def remove(name,req):
    global count,size,successes
    global count_request_consumer
    sleep(0.3)
    while req > 0:
        req -=1

        if count < 1:
            time_out = event_consumer.sleep(seconds=5)
            if time_out == False:
                continue

        semaphore.acquire()
        if count < 1:
            req +=1
            semaphore.release()
            continue
        count_request_consumer +=1

        item = buffer.remove_item()
        if item == 'x':
            successes = successes + 1
        print("request",count_request_consumer,f'consumer-{name}',buffer.display())
        #print("request",count_request_consumer)
        count -=1
        semaphore.release()
        sleep(0.05)
        if count > 0:
            event_producer.wake_up()
```

3.1 เงื่อนไขของ Remove

(1) ทำงานตาม request ที่กำหนด

(2) ณ ช่วงเวลาหนึ่ง จะสามารถเข้าถึงบัพเฟอร์ได้เพียงเทรดเดียวเท่านั้น

(3) ทุกเทรดมีเวลารอที่จำกัดในการทำงาน

3.2 วิธีการทำงานของ Remove

(1) Removeมีการรับพารามิเตอร์ 2ตัว คือ name = ชื่อของเทรด,req คือ จำนวนรีเควสที่แบ่งให้เทรดนี้

(2) มีการเรียกใช้ share variable มี count,size,count_request_consumer

-count กับ size มีความสัมพันธ์กัน count คือ จำนวนของที่อยู่ในบัพเฟอร์ปัจจุบัน size คือ ขนาดของบัพเฟอร์ count_request_consumer ใช้นับจำนวนรีเควสของ Consumer ณ ปัจจุบัน

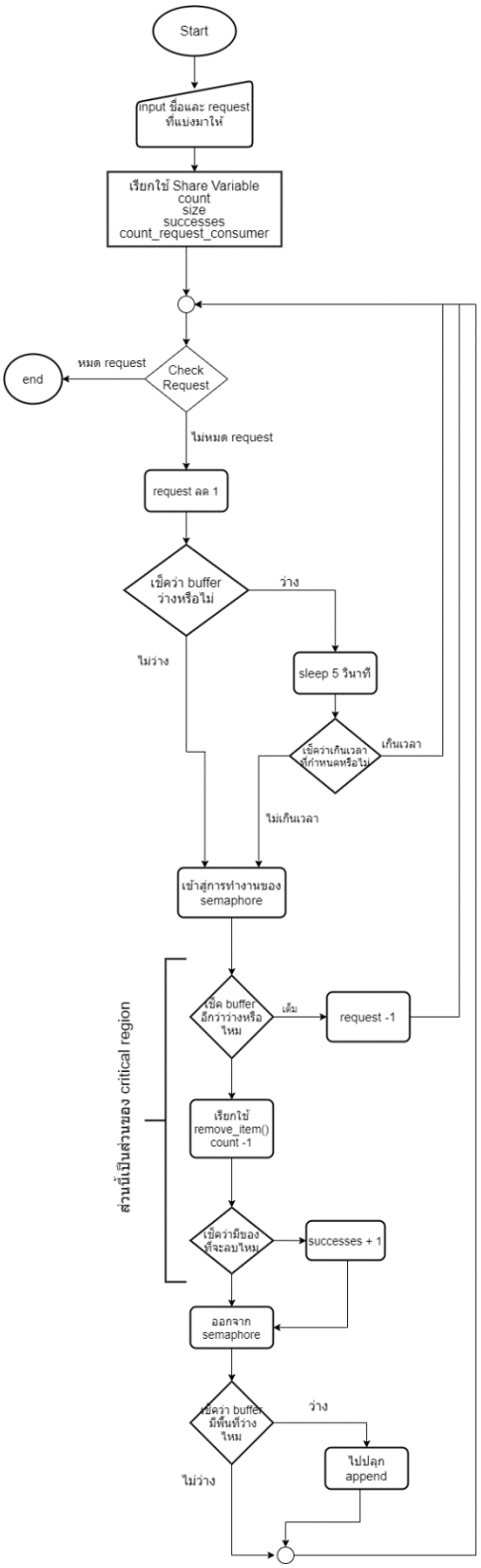
(3) มีการเช็ครอบรีเควสด้วย while loop ทุกครั้งที่วน 1 รอบ จะลดค่ารีเควสลง 1 ค่า

(4) มีการตรวจสอบจำนวนของในบัพเฟอร์ ว่าว่างหรือไม่ ถ้าว่างทั้งหมด เทรดนั้นจะเข้าสู่โหมด sleep เป็นเวลา 5 วินาที ถ้าไม่มีใครมาปลุกภายใน 5 วินาทีจะเริ่มรอบรีเควสใหม่ ถ้าตื่นทันจะเข้าสู่ semaphore เพื่อรอเข้าสู่ Critical Region

(5) เมื่อมี Process เข้า Critical Region จะทำการเรียกใช้ remove_item และเพิ่มค่า count-1 และออกจาก semaphore

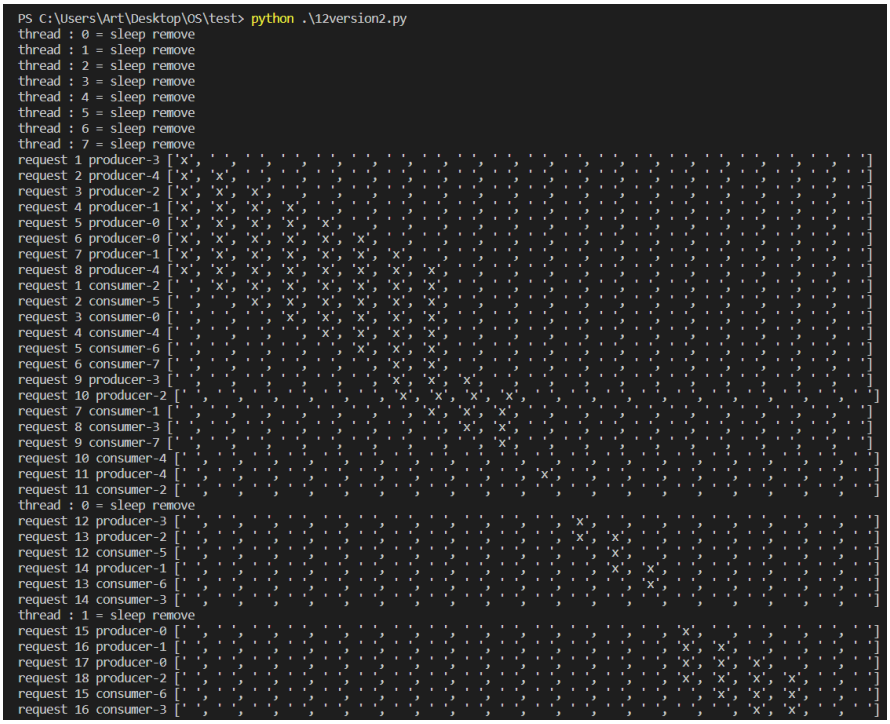
(6) เช็คจำนวนบัพเฟอร์ว่าง ถ้าว่าง ให้ไปปลุกฟังก์ชัน append ให้ตื่นมาทำงาน

Flowchart การทำงานของ remove



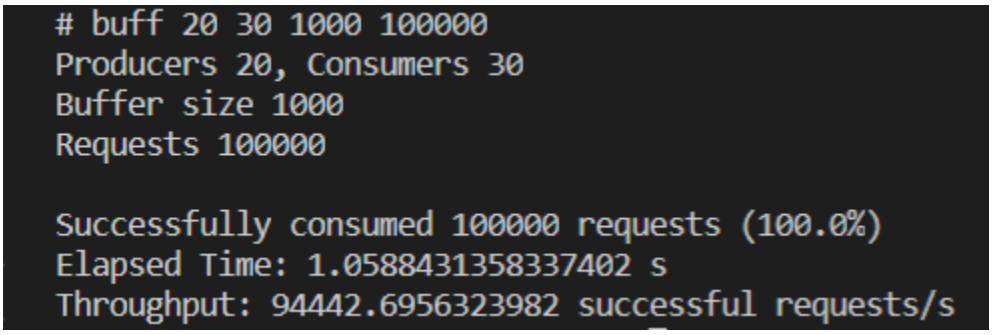
3.3 การพิสูจน์คุณสมบัติของ Remove

กำหนดให้ consumer มีอยู่ 8 เทรด หลังจากรันครั้งแรก ให้ producer ทำการ sleep รอ 10 วินาที และกำหนดการ sleep ของ remove 15 วินาที ผลที่ได้คือ consumer จะมีการ sleep รอ เนื่องจากไม่มีของในบัพเฟอร์ ทำให้ต้อง sleep remove รอ หลังจากนั้น 10 วินาที producer ตื่นขึ้นมา จะมีการเอาของใส่ในบัพเฟอร์ เมื่อมีของในบัพเฟอร์จะทำการปลุก producer และทำการ context switch กลับไปกลับมา



4. ผลการ Run & Result

สามารถอธิบายผลการ Run & Result ได้ดังรูปนี้



5. source code ของโปรแกรมทั้งหมด

```
from circular_buffer import circular_buffer
from threading import Thread,Semaphore
from time import sleep,time
from event_module import event
from divide_req import divide_req

x = input("# buff ").split(" ")

producer_thread = int(x[0])
consumer_thread = int(x[1])
buffer_size = int(x[2])
request_number = int(x[3])

print(f"Producers {producer_thread}, Consumers {consumer_thread}")
print(f"Buffer size {buffer_size}")
print(f"Requests {request_number}\n")

number_producer = producer_thread
number_consumer = consumer_thread
buffer_size = buffer_size
request_number = request_number
global successes
successes = 0
global count,size
count = 0
size = buffer_size

global event_producer,event_consumer
event_producer = event()
event_consumer = event()

global buffer
buffer = circular_buffer(buffer_size)
semaphore = Semaphore()

global count_request_producer
count_request_producer = 0
global count_request_consumer
count_request_consumer = 0

global check_stop_consumer
check_stop_consumer = 0

def append(name,req):
    global count,size
    global count_request_producer
    while req > 0:
        req -=1

        if count > size-1:
            time_out = event_producer.sleep(seconds=5) #seconds = 0.05 may processes
false

            if time_out == False:
                continue

        semaphore.acquire()
        if count > size-1:
            req +=1
            semaphore.release()
```

```
            continue
        count_request_producer +=1

        buffer.add_item('x')
        #print("request",count_request_producer,f'producer-{name}',buffer.display())
        count +=1
        semaphore.release()
        #sleep(0.00002)
        if count > 0:
            event_consumer.wake_up()

def remove(name,req):
    global count,size,successes
    global count_request_consumer
    global check_stop_consumer
    #sleep(1)
    sleep(0.0005)
    while req > 0:
        req -=1

        if count < 1:
            time_out = event_consumer.sleep(seconds=0.05)
            if time_out == False:
                continue

        semaphore.acquire()
        if count < 1:
            req +=1
            semaphore.release()
            continue
        count_request_consumer +=1

        item = buffer.remove_item()
        if item == 'x':
            successes = successes + 1
        #print("request",count_request_consumer,f'consumer-
{name}')#,buffer.display())
        count -=1
        semaphore.release()
        #sleep(0.00002)
        if count > 0:
            event_producer.wake_up()

    semaphore.acquire()
    check_stop_consumer += 1
    #print('count stop =',check_stop_consumer)
    semaphore.release()

list_req_pro = divide_req(request_number,number_producer).value()
list_req_con = divide_req(request_number,number_consumer).value()

#print("pro",list_req_pro,"con",list_req_con)

start = time()
#create thread producer
for i,req in zip(range(number_producer),list_req_pro):
    producer_thread = Thread(target=append,args=(i,req))
    producer_thread.start()

#create thread consumer
```

```
119 for i,req in zip(range(number_consumer),list_req_con):
120     consumer_thread = Thread(target=remove,args=(i,req))
121     consumer_thread.start()
122 
123     if i == number_consumer-1:
124         consumer_thread.join()
125 
126 while check_stop_consumer < number_consumer:
127     pass
128     sleep(0.005)
129     #print('ddddddddd')
130 
131 end = time()
132 timer = end-start
133 print(f'Successfully consumed {successes} requests
    ({(successes/request_number)*100}%)')
134 print(f'Elapsed Time: {timer} s ')
135 print(f'Throughput: {successes/timer} successful requests/s')
```

คอมไพล์เลอร์ที่ใช้ : Python 3.6

รันโปรแกรม : testBuffer.py , 12version2.py , 13version2.py

ไลบรารี่ที่เขียนขึ้น : circular_buffer.py , divide_req.py , event_module.py

ไลบรารี่ที่จำเป็น : Thread , Threading , Event , Semaphore

ลิงก์ดาวน์โหลด : https://www.python.org/ftp/python/3.6.0/python-3.6.0.exe

GitHub: https://github.com/sanayut12345/os_success?fbclid=IwAR1Z_5dlkzC_-zM6nXuDQepcX3ojAyGBtOSYdyDiMg4wEHwqmsYD81mOqOE

6. หลักการทำงานของโปรแกรมทั้งหมด

1.เริ่มต้นจะมีการ Import library ที่จำเป็น

- Library Buffer นำเข้าจากไฟล์ circular_buffer.py
- Library Thread กับ Semaphore จาก Library Threding
- Library sleep และ time จาก Library time
- Library event (sleep/wake up) จากไฟล์ event.py
- Library divide_req (หารrequest/จำนวนเทรด = จำนวนรอบแต่ละเทรด จากไฟล์ divide_req.py

2. มีการรับค่า input จาก keyboard ให้ตัวแปร x เป็นแบบ Array

- producer_thread รับค่า x[0] คือตัวแปร producer_thread ใช้เก็บจำนวนThreadของ producer
- consumer_thread รับค่า x[1] คือตัวแปร consumer_thread ใช้เก็บจำนวนThreadของ producer
-buffer_size รับค่า x[2] คือตัวแปร buffer_size ใช้เก็บค่าขนาดของ buffer
- request_number รับค่า x[3] คือตัวแปร request_number ใช้เก็บค่าจำนวน request ที่ต้องทำงาน

3.ทำการ Report ค่าที่รับมาจาก keyboard มีการแชร์ค่าให้กับตัวแปร ดังนี้

- number_producer = producer_thread
-numcer_consumer = consumer_thread
- buffer_size = buffer_size
- request_number = request_number

4. มีการประกาศตัวแปรแบบ Share variable

-successes ใช้เก็บค่าความสำเร็จจาก remove_item (consumer successes) ค่าเริ่มต้น = 0
-count ใช้นับจำนวนของที่อยู่ใน buffer เริ่มต้น 0
-size ขนาดของ buffer เริ่มต้น size=buffer_size
-event_producer ใช้ในการทำ event sleep/wake up ของ producer เรียกใช้ library event
- event_consumer ใช้ในการทำ event sleep/wake up ของ consumer เรียกใช้ library event
-buffer เรียกใช้ library circular_buffer และกำหนดขนาดของ buffer ตัวแปร buffer_size
-count_request_producer ใช้นับ request ของ producer เริ่มต้น 0
-count_request_consumer ใช้นับ request ของ consumer เริ่มต้น 0
-check_stop_producer ใช้นับ consumer ที่ thread ทำงานเสร็จแล้ว

4. สร้างฟังก์ชัน append และ remove

ประกาศ ตัวแปรชนิด Array

- List_req_pro = เก็บค่าจำนวน request ที่แบ่งให้แต่ละ Thread เรียกใช้จาก library divide_req และใส่ 1 จำนวน req , 2 จำนวน thread ของ producer
- List_req_con = เก็บค่าจำนวน request ที่แบ่งให้แต่ละ Thread เรียกใช้จาก library divide_req และใส่ 1 จำนวน req , 2 จำนวน thread ของ consumer

ตัวอย่างเช่น req = 100

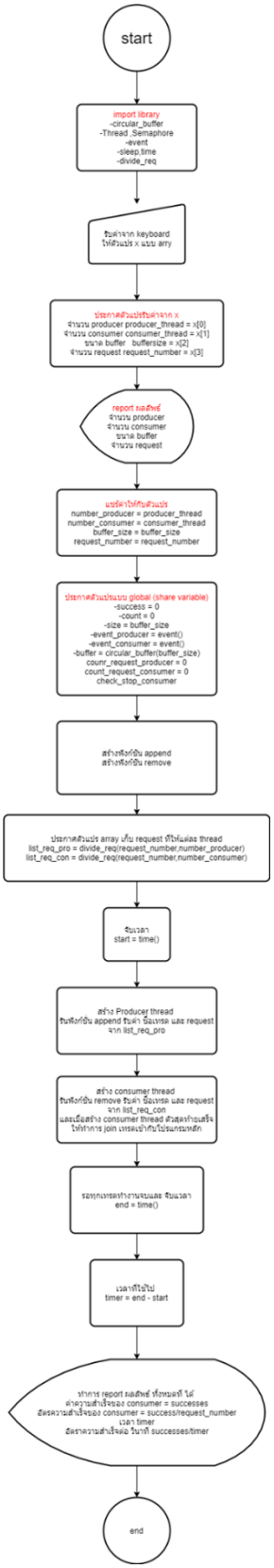
จำนวน producer thread = 8

ผลลัพธ์ [13,13,13,13,12,12,12,12] 4 เทรดแรกจะได้ request 13 ครั้ง 4 เทรดหลังจะได้ request 12 ครั้ง รวม (4*13)+(4*12) = 100

จากนี้เป็นการสั่งให้ Thread ทั้งหมดทำงาน

- ประกาศตัวแปร Start ในการจับเวลา โดย import เวลา จาก library time
- for loop แรก เป็นของการสร้าง producer thread ตัวที่ 2 เป็นของ consumer thread โดย input ค่าชื่อเทรด (ตัวเลข0-n) และ request จาก list_request_producer/consumer หลังจากสร้างเทรดสุดท้ายเสร็จให้ทำการ join เทรดเข้ากับโปรแกรมหลัก หลังจาก join โปรแกรมหลักจะหยุดทำงานจนกว่าเทรดจะหยุดทำงาน
- หลังจาก consumer ทำงานเสร็จ เทรดจะทำการจับเวลาที่ทำงานเสร็จให้กับตัวแปร end ดังนั้น เวลาที่ใช้ในการทำงานทั้งหมดให้ตัวแปร timer = end-start
- ทำการหาค่า เปอร์เซ็นต์ความสำเร็จจาก successes/request_number และอัตราความสำเร็จต่อวินาที จาก successes/timer และทำการ report ผลลัพธ์

Flowchart การทำงานของโปรแกรมทั้งหมด



7. ปัญหาที่พบในการทำงาน

ในการแก้ปัญหา Producer/Consumer Problem ที่ Implement โดยใช้ Sleep/Wake up และ Semaphore มี 2 แบบ ดังนี้

1. ที่ append หลังจากที่เช็คของใน Buffer แล้ว เมื่อเต็ม จะทำการ Sleep (ไปอยู่ที่block state) ถ้าไม่เต็มจะเข้าสู่ Semaphore (Ready state) แต่เราไม่สามารถรู้ได้เลยว่าเหลือพื้นที่เท่าไร ที่จะเอาของใส่ได้ สมมุติว่า ถ้ามีพื้นที่เหลือใน Buffer 2 แต่มี Producer รอที่ Semaphore ต่อคิวอยู่ 3 เทรด ณ ขณะนั้น ใส่เต็มพอดี เทรดที่ 3 จะไม่สามารถเพิ่มของใส่เข้าไปได้
2. ที่ Remove หลังจากเช็คพื้นที่ว่างใน Buffer แล้ว เมื่อ Buffer ว่างทั้งหมดต้องทำการ Sleep(ไปอยู่ที่block state) ถ้าไม่เต็มจะเข้าสู่ Semaphore (Ready state) แต่เราไม่สามารถบอกได้ว่าเหลือของกี่ชิ้นที่สามารถลบได้ สมมุติว่า มีของใน Buffer 1 ชิ้น แต่มี Consumer ในSemaphore 3 เทรด เทรดที่1 สามารถลบได้ (successes) เทรดที่ 2,3 ไม่สามารถลบได้ (failed)

วิธีการแก้ไขปัญหา

1. ที่ append ก่อนที่ Semaphore จะปล่อยเข้าสู่ Critical Region ให้ทำการเช็คว่าสามารถเพิ่มของเข้าไปใน Buffer ได้หรือไม่ ถ้าเพิ่มได้จะเข้าสู่ Critical Region (Running state) ถ้าไม่ได้ ทำการย้าย Process ไปอยู่ที่ (Block state)

```python
def append(name,req):
    global count,size
    global count_request_producer
    while req > 0:
        req -=1

        if count > size-1:
            time_out = event_producer.sleep(seconds=5)

            if time_out == False:
                continue

        semaphore.acquire()

        count_request_producer +=1

        buffer.add_item('x')
        count +=1
        semaphore.release()
        if count > 0:
            event_consumer.wake_up()
```

```python
def append(name,req):
    global count,size
    global count_request_producer
    while req > 0:
        req -=1

        if count > size-1:
            time_out = event_producer.sleep(seconds=5)

            if time_out == False:
                continue

        semaphore.acquire()
        if count > size-1:
            req +=1
            semaphore.release()
            continue
        count_request_producer +=1

        buffer.add_item('x')
        count +=1
        semaphore.release()
        if count > 0:
            event_consumer.wake_up()
```

ตัวอย่าง Code ที่เพิ่มเข้าไปในการแก้ปัญหานี้

2. ที่ Remove ก่อนที่ Semaphore จะปล่อยเข้าสู่ Critical Region ให้ทำการเช็คว่าสามารถลบของออกจาก Buffer ได้หรือไม่ ถ้าลบได้จะเข้าสู่ Critical Region (Running state) ถ้าไม่ได้ ทำการย้าย Process ไปอยู่ที่ (Block state)

```
def remove(name,req):
    global count,size,successes
    global count_request_consumer
    global check_stop_consumer

    sleep(0.0005)
    while req > 0:
        req -=1

        if count < 1:
            time_out = event_consumer.sleep(seconds=5)
            if time_out == False:
                continue

        semaphore.acquire()

        count_request_consumer +=1

        item = buffer.remove_item()
        if item == 'x':
            successes = successes + 1

        count -=1
        semaphore.release()

        if count > 0:
            event_producer.wake_up()

    semaphore.acquire()
    check_stop_consumer += 1
    #print('count stop =',check_stop_consumer)
    semaphore.release()
```

```
def remove(name,req):
    global count,size,successes
    global count_request_consumer
    global check_stop_consumer

    sleep(0.0005)
    while req > 0:
        req -=1

        if count < 1:
            time_out = event_consumer.sleep(seconds=5)
            if time_out == False:
                continue

        semaphore.acquire()
        if count < 1:
            req +=1
            semaphore.release()
            continue
        count_request_consumer +=1

        item = buffer.remove_item()
        if item == 'x':
            successes = successes + 1

        count -=1
        semaphore.release()

        if count > 0:
            event_producer.wake_up()

    semaphore.acquire()
    check_stop_consumer += 1
    #print('count stop =',check_stop_consumer)
    semaphore.release()
```

ตัวอย่าง Code ที่เพิ่มเข้าไปในการแก้ปัญหานี้

บรรณานุกรม

การทำ MutiThreading และ Semaphore (ออนไลน์) แหล่งที่มา :

https://www.geeksforgeeks.org/synchronization-by-using-semaphore-in-python/?fbclid=IwAR0vdScmipvoXYn0H41QkiKHIqiVCrPL-PPo3XbnFi6DunSVKwkjc93SAQg

Threading Event (ออนไลน์) แหล่งที่มา :

https://www.bogotobogo.com/python/Multithread/python_multithreading_Event_Objects_between_Threads.php

multithreading (ออนไลน์) แหล่งที่มา :

https://www.tutorialspoint.com/python/python_multithreading.htm?fbclid=IwAR29iycFsd86uCzI8VrpAQ_12raq36OuXAm-SQczuBzeOQhqiMCpDEa0SmE

Buffer (ออนไลน์) แหล่งที่มา :

https://th.wikipedia.org/wiki/%E0%B8%9A%E0%B8%B1%E0%B8%9E%E0%B9%80%E0%B8%9F%E0%B8%AD%E0%B8%A3%E0%B9%8C%E0%B8%A7%E0%B8%87%E0%B8%81%E0%B8%A5%E0%B8%A1?fbclid=IwAR1BkaadmKN4FhfkIw_TFdQlOuiEk5O4v8wRTJg9E_TNTum7DCdSmbNs-aQ

Lec04 Concurrency and Synchronisation (ออนไลน์) แหล่งที่มา :

https://o365ku.sharepoint.com/sites/OperationSystem631/Class%20Materials/Forms/AllItems.aspx?id=%2Fsites%2FOperationSystem631%2FClass%20Materials%2FLec04%20Concurrency%20and%20%20Synchronisation%2Epdf&parent=%2Fsites%2FOperationSystem631%2FClass%20Materials