

รายงานการออกแบบโปรแกรม
รายวิชา ระบบปฏิบัติการ รหัสวิชา 01204332

เสนอ

อาจารย์ สรยุทธ กลมกล่อม

ชื่อกลุ่ม A4OS

สมาชิกในกลุ่ม

นายพงศกร	สุตมา	6040202998
นางสาวจารุพร	ทัดสี	6040200634
นายกิตติชัย	นิวงษา	6040200278
นางสาวโฉมธิดา	วงศ์ตาแสง	6040200855
นางสาวลดาวัลย์	ใจหมาย	6040204095
นายศิริวิชญ์	คำสงค์	6040204559
นายสาธิต	ทรัพย์เมฆ	6040204923
นางสาววรศรา	จรรยาวัตี	6040206578
นางสาวธิดิสุดา	จิตตะยโสธร	6040202106
นางสาวสุภาวดี	มาลา	6040205211
นายสถาพร	สายืน	5940205046

รายงานฉบับนี้เป็นส่วนหนึ่งของรายวิชา ระบบปฏิบัติการสาขาวิศวกรรมคอมพิวเตอร์

ภาควิชาวิศวกรรมไฟฟ้าและคอมพิวเตอร์

มหาวิทยาลัยเกษตรศาสตร์ วิทยาเขตเฉลิมพระเกียรติ จังหวัดสกลนคร

คำนำ

รายงานเล่มนี้จัดทำขึ้นเพื่อเป็นส่วนหนึ่งของรายวิชา ระบบปฏิบัติการOS รหัส 01204332 เพื่อศึกษาการทำงานของ Mutual Exclusion โดยการเขียนโปรแกรม Producer and Consumer เพื่อศึกษาการออกแบบโปรแกรม เงื่อนไข การทำงานของ Append และRemove และการใช้ Semaphore เพื่อควบคุมการทำงานของโปรแกรม

โดยหวังเป็นอย่างยิ่งว่าการจัดทำรายงานฉบับนี้จะเป็นประโยชน์กับผู้อ่านที่กำลังศึกษาหาข้อมูลเรื่องนี้ อยู่ หากมีข้อผิดพลาดประการใด ผู้จัดทำขออภัยและขออภัยมา ณ ที่นี้ด้วย

สารบัญ

เนื้อหา	หน้า
คำนำ	I
สารบัญ	II
การออกแบบโปรแกรม	1
1.1การออกแบบ Buffer	1
1.2 การออกแบบ Append	2
1.3 การออกแบบ Remove	3
1.4 Flowchart ของโปรแกรม	4
2.เงื่อนไข วิธีการทำงาน และการพิสูจน์คุณสมบัติของ Append	5
2.1 เงื่อนไขของ Append	5
2.2 วิธีการทำงานของ Append	6
2.3 การพิสูจน์คุณสมบัติของ Append	7
3.เงื่อนไข และวิธีการทำงานของ Remove	8
3.1 เงื่อนไขของ Remove	9
3.2 วิธีการทำงานของ Remove	9
4.ผลการ Run & Result	10
5.Sourcecode ของโปรแกรม	10
อ้างอิง	

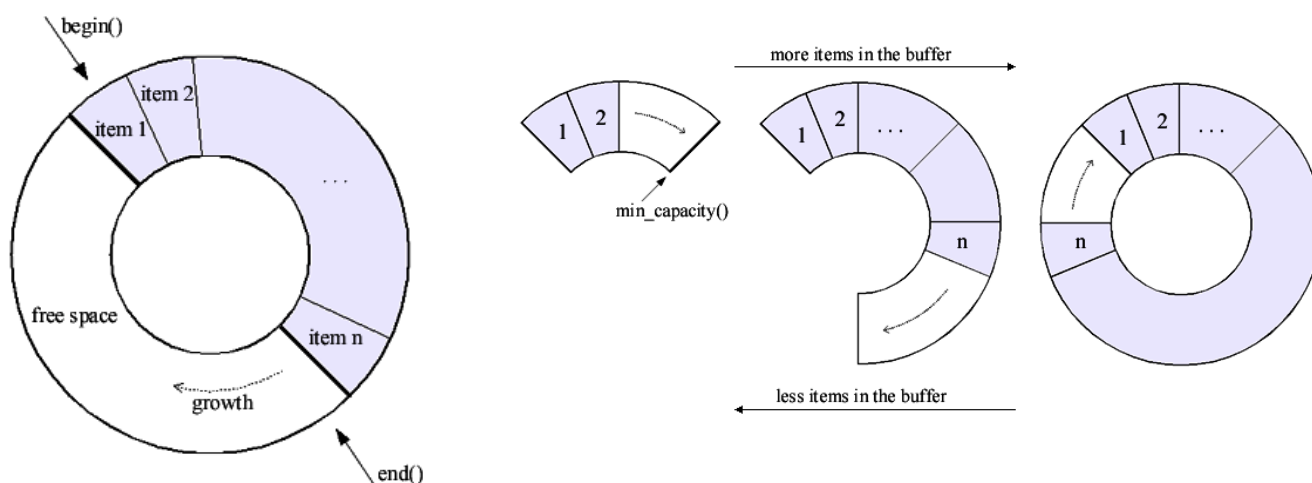
1.การออกแบบโปรแกรม

1.1 การออกแบบ Buffer

```
1. #include <boost/circular_buffer.hpp>
2. boost::circularbuffer<int> buffername(buffer size);
```

บัฟเฟอร์แบบวงกลม โดยใช้ Boost Library

บัฟเฟอร์แบบวงกลม (เรียกอีกอย่างว่าวงแหวนหรือบัฟเฟอร์แบบวงกลม) หมายถึงพื้นที่ในหน่วยความจำซึ่งใช้เพื่อเก็บข้อมูลที่เข้ามา เมื่อบัฟเฟอร์เต็มข้อมูลใหม่จะถูกเขียนเริ่มต้นที่จุดเริ่มต้นของบัฟเฟอร์และเขียนทับเก่า **boost :: circular_buffer** เป็นคอนเทนเนอร์ที่สอดคล้องกับ STL มันเป็นชนิดของลำดับที่คล้ายกับ `std :: list` หรือ `std :: deque` สนับสนุนการเข้าถึงตัววนซ้ำแบบสุ่มแทรกเวลาอย่างต่อเนื่องและลบการดำเนินการที่จุดเริ่มต้นหรือจุดสิ้นสุดของบัฟเฟอร์และการทำงานร่วมกันกับอัลกอริทึม `std Circular_buffer` ได้รับการออกแบบมาเป็นพิเศษเพื่อให้มีความจุคงที่ เมื่อความจุของมันหมดลงองค์ประกอบที่เพิ่งแทรกเข้าไปใหม่จะทำให้องค์ประกอบถูกเขียนทับทั้งที่จุดเริ่มต้นหรือจุดสิ้นสุดของบัฟเฟอร์ (ขึ้นอยู่กับการใช้งานการแทรก) `Circular_buffer` จัดสรรหน่วยความจำเฉพาะเมื่อสร้างขึ้นเมื่อความจุถูกปรับอย่างชัดเจนหรือตามความจำเป็นเพื่อปรับขนาดหรือกำหนดการดำเนินการ



1.2 การออกแบบ Append

Append ฟังก์ชัน รูปร่างหน้าตาของ append

```
void append(int id)
{
    ++num_append_working;
    while(i < REQUEST)
    {
        add_item(id);
        ++i;
    }
    this_thread::sleep_for(chrono::nanoseconds(wait_time));
    --num_append_working;
}
```

ฟังก์ชัน append จะเพิ่มค่าของ num_append_work แล้วทำการเพิ่มค่าลงไปในบัฟเฟอร์คิว ตามจำนวนของการ รีควีส โดยการให้เทรดที่เรียกใช้งาน append ทำงานในฟังก์ชัน add_item และเมื่อทำงานเสร็จสิ้น จะหยุดการทำงานของเทรดที่เรียก append และ ลดค่าของ num_append_working ลง

ฟังก์ชัน add_item

add_item ฟังก์ชัน ที่ถูกเรียกใช้งานโดย เทรดต่าง ๆ ของ append เพื่อนำค่าเข้าไปในคิว

```
void add_item(int append_id)
{
    int random_num = rand()%100;
    unique_lock<mutex> lock(ymutex);
    is_not_full.wait(lock, []{ return myringbuf.size() != BUFFER_SIZE; });
    myringbuf.push_back(random_num);
    //cout << "Append ID : "<< append_id << "add "<< random_num << endl;
    is_not_empty.notify_all();
}
```

add_item() ออกแบบให้สามารถเข้าใช้งานได้เพียงเทรดเดียวเท่านั้นโดยค่าที่ จะถูกส่งเข้ามาเป็นค่า thread id ของ append thread จากนั้นจะใช้ mutex ล็อกทรัพยากรบัฟเฟอร์ให้เทรดนั้น ๆ เพื่อไม่ให้เทรดอื่น ๆ ที่รอคอยอยู่เข้ามาทำงานได้ แล้วจะมีการทำงานอยู่ 2 แบบดังนี้

- 1.) **บัฟเฟอร์เต็ม** ก็จะทำให้การหยุดการทำงานของเทรดนั้นชั่วคราว และส่งสัญญาณว่าคิวไม่ว่างออกไป
- 2.) **บัฟเฟอร์ไม่เต็ม** ก็จะทำให้การเรียกดูขนาดของคิวมาเก็บไว้ในตัวแปร และ เขียนค่าตัวแปรนั้นลง ไปในคิว และเมื่อทำงานเสร็จก็จะส่งสัญญาณออกไปว่าคิวไม่ว่างแล้ว

1.3 การออกแบบ Remove

remove_ ฟังก์ชัน หน้าตาของ remove

```
void remove_(int id)
{
    while(num_append_working == 0)
    {
        this_thread::yield();
    }
    while(num_append_working != 0 || myringbuf.size() > 0)
    {
        remove_item(id);
        this_thread::sleep_for(chrono::nanoseconds(wait_time));
    }
}
```

ฟังก์ชัน remove จะตรวจสอบว่าไม่มีเทรด append ทำงานอยู่หรือบัฟเฟอร์ไม่ว่างก็จะเรียกการทำงานของ remove_item เพื่อนำคิวออกจากบัฟเฟอร์จากนั้นก็หยุดการทำงานของเทรดนั้นลง

ฟังก์ชัน remove_item

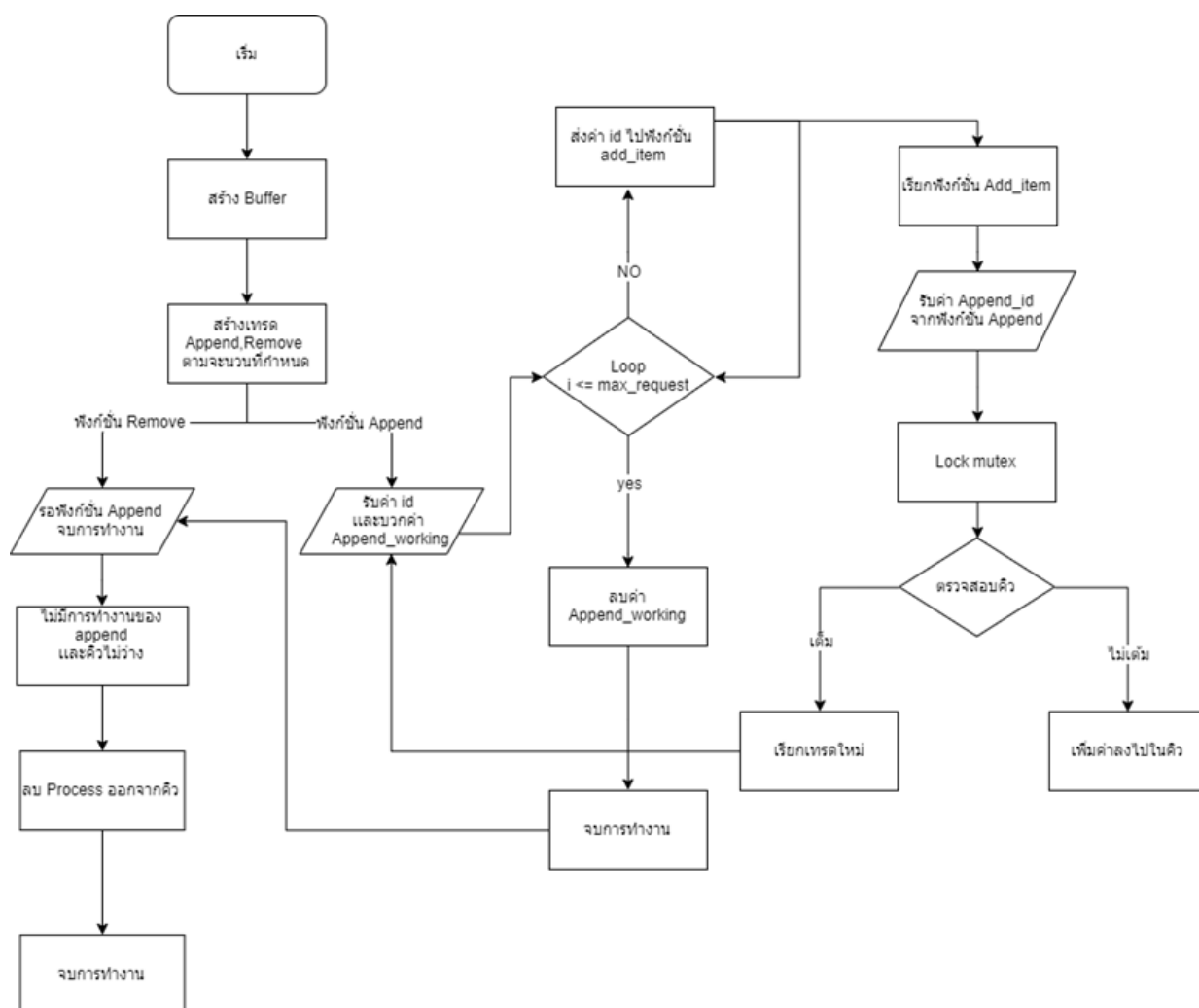
remove_items ฟังก์ชันที่ถูกเรียกใช้งานโดย เทรดต่าง ๆ ของ remove_ เพื่อนำค่าออกจากคิว

```
void remove_item(int remove_id)
{
    unique_lock<mutex> lock(ymutex);
    int product;
    if(is_not_empty.wait_for(lock, chrono::nanoseconds(wait_time),
        []{ return myringbuf.size() > 0; }))
    {
        product = myringbuf.front();
        myringbuf.pop_front();
        ++c_count;
        //cout << "Remove ID : "<< remove_id << "remove "<< product << endl;
        is_not_full.notify_all();
    }
}
```

ฟังก์ชัน remove_item จะมีการเข้าถึงฟังก์ชันนี้ได้เพียงแค่เทรดเดียวเท่านั้น เมื่อทำการเรียกใช้งานฟังก์ชันก็จะใช้ mutex ล็อกทรัพยากรการบัฟเฟอร์ให้เข้าถึงได้เพียงแค่เทรดเดียวและมีการทำงานอยู่ 2 แบบดังนี้

- 1.) **บัฟเฟอร์ว่าง** จะหยุดการทำงานของเทรดนั้น ๆ และส่งสัญญาณออกไปให้เทรดอื่น ๆ ที่รอทำงาน
- 2.) **บัฟเฟอร์ไม่ว่างหรือเต็ม** จะไอเท็มที่หน้าคิวและทำการ dequeue ออกจากบัฟเฟอร์ และส่งสัญญาณให้เทรดอื่น ๆ ที่รอทำงาน

1.4 Flowchart ของโปรแกรม



2.เงื่อนไข วิธีการทำงาน และการพิสูจน์คุณสมบัติของ Append

ฟังก์ชัน Append

```
void append_(int id)
{
    ++num_append_working;
    while(i < REQUEST)
    {
        add_item(id);
        ++i;
    }
    this_thread::sleep_for(chrono::nanoseconds(wait_time));
    --num_append_working;
}
```

2.1 เงื่อนไขของ Append

เงื่อนไขการทำงานของ append

- 1.) ทำงานตามรีเคสที่กำหนด
- 2.) เข้าถึงบัฟเฟอร์ได้เพียงเทรตเดียว
- 3.) ทุกเทรตมีเวลาทำงานจำกัดในการทำงาน

2.2 วิธีการทำงานของ Append

1.) เพิ่มค่าของ num_append_working ขึ้นตามจำนวนเทรตที่ใช้งาน

2.) วนลูปทำงานตามรีเคสที่ได้กำหนดไว้

3.) เรียกใช้งาน add_item โดยส่งค่า Append_id ไปให้

- รับค่าของ append id
- สุ่มตัวเลขและเก็บไว้ในตัวแปร
- ล็อก metex ให้เทรตที่เรียกใช้งาน
- ตรวจสอบบัฟเฟอร์

```
is_not_full.wait(lock, [] { return myringbuf.size() != BUFFER_SIZE; });
```


เมื่อบัฟเฟอร์ไม่เต็มก็就不用เรียกใช้คำสั่ง wait() แต่หากบัฟเฟอร์ เต็มเทรานั้น ๆ ก็จะถูกเปลี่ยนสถานะให้รอจนกว่าจะได้รับสัญญาณ

- เพิ่มค่าของ random_num เข้าไปที่ท้ายคิวของบัฟเฟอร์
- แสดงผลการเพิ่ม
- ส่งสัญญาณให้ทุกเทรตที่รอการทำงานว่าบัฟเฟอร์ไม่ว่าง

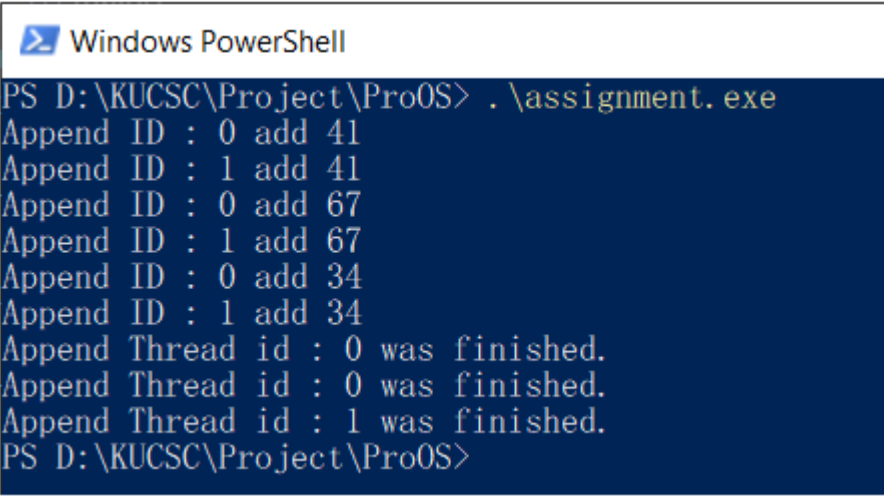
4.) เพิ่มค่า i

5.) เมื่อเทรตใดเทรตหนึ่งทำงานถึงระยะเวลาที่ได้กำหนดไว้ใน wait_time ก็จะหยุดการทำงานของเทรตลงไป

6.) ลดค่าของ num_append_working ลงตามจำนวนเทรตที่ปิดใช้งานไป

การพิสูจน์คุณสมบัติของ Append

ผลการทดสอบคุณสมบัติของ append โดยการขอรหัสไปทำ 6 ตัว และ เทรตของ append 3 เทรต



```

Windows PowerShell
PS D:\KUCSC\Project\ProOS> .\assignment.exe
Append ID : 0 add 41
Append ID : 1 add 41
Append ID : 0 add 67
Append ID : 1 add 67
Append ID : 0 add 34
Append ID : 1 add 34
Append Thread id : 0 was finished.
Append Thread id : 0 was finished.
Append Thread id : 1 was finished.
PS D:\KUCSC\Project\ProOS>
  
```

3. เงื่อนไขการทำงานของ Remove

ฟังก์ชัน remove

```

void remove_(int id)
{
    while(num_append_working == 0)
    {
        this_thread::yield();
    }
    while(num_append_working != 0 || myringbuf.size() > 0)
    {
  
```

```

        remove_item(id);
        this_thread::sleep_for(chrono::nanoseconds(wait_time));
    }
    cout << "Remove thread id : "<< id << "was finished.\n" << endl;
}

```

3.1 เงื่อนไขของ Remove

- 1.) ตรวจสอบการทำงานของ append ว่าไม่มีการทำงานอยู่
- 2.) ตรวจสอบบัฟเฟอร์ว่าไม่ได้ว่างอยู่
- 3.) เข้าถึงบัฟเฟอร์ได้เพียงเทรตเดียว
- 4.) ทุกเทรตที่ทำงานมีเวลาจำกัดในการทำงาน

3.2 วิธีการทำงานของ Remove

- 1.) ตรวจสอบว่าเทรตของ append ไม่ได้มีการทำงานอยู่ โดยตรวจสอบจากตัวแปร num_append_working
- 2.) หากไม่มีเทรตใดของ append ทำงานอยู่แล้ว ให้ทำการรอชั่วขณะ
- 3.) หากบัฟเฟอร์มีขนาดมากกว่า 0 ให้ทำการเรียกใช้งาน remove_item พร้อมส่งค่า id ของเทรตไปด้วย
 - รับค่า thread id
 - ล็อกการทำงานของ mutex ให้เทรตที่เรียกใช้งาน
 - ตรวจสอบว่าบัฟเฟอร์ไม่ว่าง

```

if(is_not_empty.wait_for(lock, chrono::nanoseconds(wait_time),
    []{ return myringbuf.size() > 0; })

```

- เลือกตำแหน่งต้นคิว และ ลบออกจากคิว
 - เพิ่มค่า c_count
 - แสดงผลการทำงาน
 - ส่งสัญญาณว่าบัฟเฟอร์ไม่เต็มออกไปให้ทุกเทรตที่รอการทำงาน
- 4.) หากเทรตทำงานจนถึงเวลาที่กำหนดให้หยุดการทำงานของเทรตนั้น ๆ
 - 5.) แสดงผลการทำงาน

ผลการรันทดสอบ remove

```

Windows PowerShell
PS D:\KUCSC\Project\ProOS> .\assignment.exe
Remove ID : 0 remove 41
Remove ID : 1 remove 41
Remove ID : 1 remove 67
Remove ID : 0 remove 67
Remove ID : 1 remove 34
Remove ID : 0 remove 0
Remove thread id : 0 was finished.
Remove thread id : 1 was finished.
PS D:\KUCSC\Project\ProOS>

```

4.ผลการ Run & Result

```

PS D:\KUCSC\Project\ProOS\src> .\assignment.exe
Producer 20 Consumer 30
Buffer Size 1000
Request 100000
Success fully consume 100036 requests. (100%)
Elapsed Time : 6.537537 Seconds.
Throughput 15301.787 request/s
PS D:\KUCSC\Project\ProOS\src>

```

5. Source code

```

/*
คอมไพล์ไฟล์นี้โดยใช้คำสั่ง : g++ ชื่อไฟล์.cpp -pthread -lpthread -o ชื่อโปรแกรมที่ต้องการ
รันโปรแกรม : ./ชื่อโปรแกรม
คอมไพเลอร์ : MinGW-W64
ลิงก์ดาวน์โหลด : https://sourceforge.net/projects/mingw-w64/
ติดตั้งและเซตพารให้เรียบร้อย
ไลบรารีที่จำเป็น : Boost C++ Libraly
ลิงก์ดาวน์โหลด : https://www.boost.org/
*/

#include <iostream>
#include <mutex>
#include <thread>
#include <condition_variable>
#include <atomic>
#include <vector>
#include <chrono>
#include <bits/stdc++.h>
#include <boost/circular_buffer.hpp>

//or using this include if you don't install boost libraly//

```

```

#include "circular_buffer.hpp" //<- uncomment this line

using namespace std;

#define BUFFER_SIZE 1000
#define REQUEST 100000
#define PROD 20
#define CONS 30

int c_count = 0;
int i = 0;
const int wait_time = 1;
atomic<int> num_append_working(0);
condition_variable is_not_full;
condition_variable is_not_empty;
mutex xmutex;

boost::circular_buffer<int> myringbuf(BUFFER_SIZE);

void add_item(int append_id)
{
    int random_num = rand() % 100;
    unique_lock<mutex> lock(xmutex);
    is_not_full.wait(lock, [] { return myringbuf.size() != BUFFER_SIZE; });
    myringbuf.push_back(random_num);
    // cout << "Append ID : " << append_id << " add " << random_num << endl;
    is_not_empty.notify_all();
}

void remove_item(int remove_id)
{
    unique_lock<mutex> lock(xmutex);
    int product;
    if (is_not_empty.wait_for(lock, chrono::milliseconds(wait_time)),
        [] { return myringbuf.size() > 0; })
    {
        product = myringbuf.front();
        myringbuf.pop_front();
        ++c_count;
        // cout << "Remove ID : " << remove_id << " remove " << product << endl;
        is_not_full.notify_all();
    }
}

```

```

}

void append_(int id)
{
    ++num_append_working;
    while (i < REQUEST)
    {
        add_item(id);
        ++i;
    }
    this_thread::sleep_for(chrono::milliseconds(wait_time));
    // cout << "Append Thread id : " << id << " was finished." << endl; // <- uncomment to display end append_s thread
    --num_append_working;
}

void remove_(int id)
{
    while (num_append_working == 0)
    {
        this_thread::yield();
    }
    while (num_append_working != 0 || myringbuf.size() > 0)
    {
        remove_item(id);
        this_thread::sleep_for(chrono::milliseconds(wait_time));
    }
    // cout << "Remove thread id : " << id << " was finished." << endl;
}

int main()
{
    // Record Start Time //
    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    // Start Producer Consumer Program //
    cout << "Producer " << PROD
        << " Consumer " << CONS
        << "\nBuffer Size " << BUFFER_SIZE
        << "\nRequest " << REQUEST
        << endl;
    vector<thread> vec_of_thread;
    for (int i = 0; i < PROD; i++)
    {

```

```

        vec_of_thread.push_back(thread(append_, i));
    }
    for (int i = 0; i < CONS; i++)
    {
        vec_of_thread.push_back(thread(remove_, i));
    }
    for (auto &t : vec_of_thread)
    {
        t.join();
    }

    auto end = chrono::high_resolution_clock::now();

    // Calculating total time taken by the program.
    double time_taken =
        chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9; // Chang nanosecond to second
    double through_put = c_count / time_taken;
    double percentage = (c_count / REQUEST) * 100.0;
    string c_consume = to_string(c_count);
    cout << "Success fully consume " << c_consume
        << " requests. (" << percentage
        << "%)" << endl;
    cout << "Elapsed Time : " << fixed << time_taken << setpreci
sion(3)
        << " Seconds." << endl;
    cout << "Throughput " << through_put << " request/s" << endl;
    return 0;
}

```


อ้างอิง

std::mutex. cppreference. <https://en.cppreference.com/w/cpp/thread/mutex>

Ankit Lathiya. (2019). **Multithreading In C++ Example | C++ Multithreading Tutorial.**

Appdividend. <https://appdividend.com/2019/08/21/multithreading-in-cpp-example-cpp-multithreading-tutorial/>

Tutorials. cplusplus. <http://www.cplusplus.com/>

producer-consumer problem. wikipedia.

https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem