



# Assignment

Synchronizations

GROUP 3

2/24/17

Operating System



## รายงาน

วิชา 01076259 Operating Systems

### เสนอ

ดร.อัครฤทธิ์      สังข์เพชร

ดร.อรทัย      สังข์เพชร

### จัดทำโดย

1. นาย แทนไท	เอี้ยการนา	56010492
2. นาย พงษ์ศักดิ์	สงวนวงษ์	57010821
3. นาย พรเทพ	แซ่อึ้ง	57010836
4. นาย ภาณุวัฒน์	เอนอ้าไผวงศ์	57010978
5. น.ส. วรัญญา	กิจประไพอำพล	57011116
6. น.ส. วิรชา	เลาหพุนรังษี	57011180
7. นาย ศรัทธาธรรม์	จันท์ชาตรี	57011220
8. นาย ศวีระ	อภินทนาพงศ์	57011229
9. นาย สิริวิชญ์	วนรัฐกาล	57011363
10. นาย อนุรักษ์	จันนวัน	57011470

ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

## Preface

รายงานฉบับนี้จัดทำขึ้นเพื่อศึกษาค้นคว้าในรายวิชา Operating Systems รหัสวิชา 01076259 เกี่ยวกับการทดลองเขียนโปรแกรมโดยใช้มัลติเทรดในการประมวลผล โดยจะทำการเขียนฟังก์ชัน append และ remove เพื่อทำการเปลี่ยนแปลงข้อมูลใน circular queue แล้วทำการเปรียบเทียบเพื่อหาอัลกอริทึมที่ดีที่สุด ในรายงานฉบับนี้จะประกอบไปด้วยเทคนิคต่างๆ ที่ใช้ออกแบบ และคำอธิบายโปรแกรม

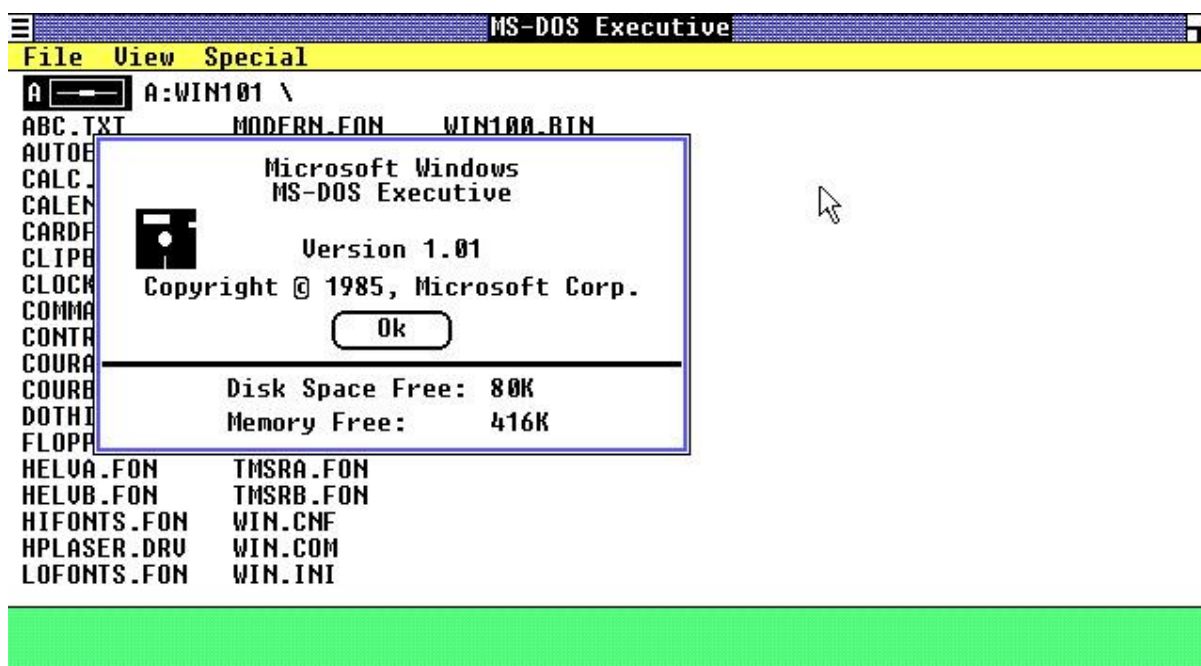
คณะผู้จัดทำ

## Contents

Preface.....	b
Contents.....	c
Chapter 1 History.....	1
Chapter 2 Design Process .....	3
2.1 ปัญหา.....	3
การออกแบบ 2.2.....	3
เงื่อนไขการออกแบบ 3.2.....	3
Chapter 3 Program Design .....	4
Source Code.....	4
Circular Buffer .....	8
Add Function.....	9
Remove Function.....	10
Threads Function .....	11
Function buffer_append.....	12
Function buffer_remove.....	13
ผลการทดลอง.....	14

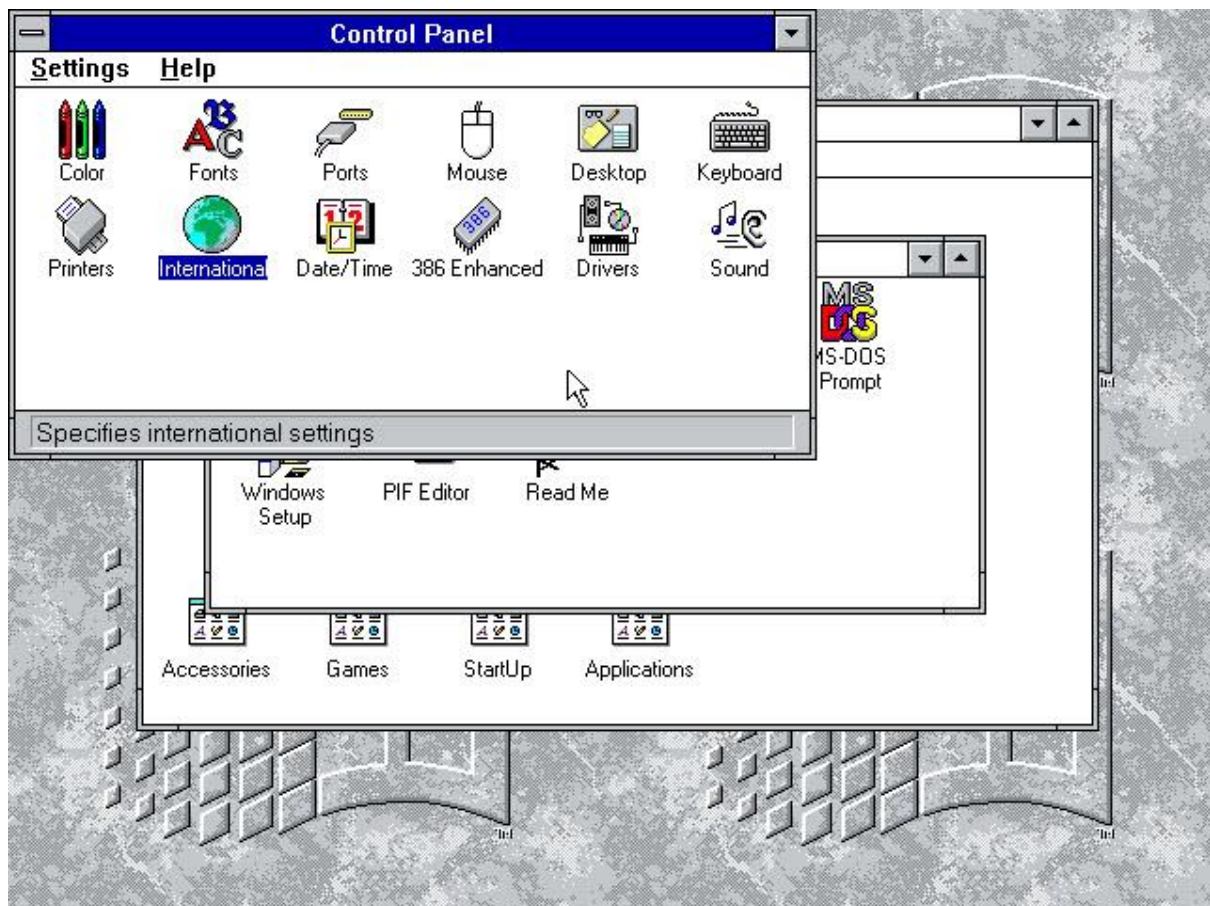
## Chapter 1 History

ในอดีตเครื่องคอมพิวเตอร์นั้นมีระบบการทำงานที่ไม่ซับซ้อนดังเช่นในปัจจุบันนี้ ย้อนกลับไปในสมัยที่ยังเป็น MS-DOS หรือช่วง Windows 1.0 ซึ่งเป็นยุคบุกเบิกในการทำระบบปฏิบัติการที่ใช้ในเครื่องคอมพิวเตอร์ส่วนบุคคลนั้น การทำงานจะใช้ระบบ Mono Thread คือสามารถทำงานได้เพียงอย่างเดียว ระบบการจัดการจึงยังไม่ซับซ้อน



ภาพที่ 1.1 ระบบปฏิบัติการ Microsoft Windows 1.01

ต่อมาเมื่ออุปกรณ์ต่างๆ มีประสิทธิภาพมากขึ้น ระบบปฏิบัติการก็ถูกพัฒนาไปให้สามารถทำงานได้มีประสิทธิภาพมากขึ้นและสามารถใช้ทรัพยากรที่มีได้อย่างเกิดประโยชน์สูงสุด นับตั้งแต่สมัย Microsoft Windows 3.0 ก็เริ่มมีการนำระบบ Multi-Tasking มาใช้งาน การจัดการการใช้ทรัพยากรก็ซับซ้อนมากยิ่งขึ้นและต้องมีการใช้เทคนิคต่างๆ เพื่อจัดการให้เกิดข้อผิดพลาดน้อยที่สุด ซึ่งการจัดการดังกล่าวทางคณะผู้จัดทำได้ทดสอบไว้ในรายงานฉบับนี้ ซึ่งจะกล่าวในบทถัดไป



ภาพที่ 1.2 ระบบปฏิบัติการ Microsoft Windows 3.11

ในบทถัดไปทางคณะผู้จัดทำจะนำเสนอรูปแบบการแก้ปัญหาอันเกิดจากการทำงานแบบ Multi-Threading

## Chapter 2 Design Process

### 2.1 ปัญหา

หากระบบปฏิบัติการหนึ่งมีบัฟเฟอร์แบบวนกลับ (circular buffer) ใช้สำหรับการทำงานแบบคิวหรือ First-in-first-out โดยมีขนาดที่เก็บข้อมูลได้ N รายการ ( $1 \leq N \leq 1000$ ) ที่ถูกใช้งานร่วมกันด้วย thread มากกว่า 1 thread

### 2.2 การออกแบบ

ทางคณะผู้จัดทำมีการนำ Algorithm ต่างๆ เข้ามาใช้เพื่อให้การทำงานมีประสิทธิภาพสูงที่สุดโดย

- ภาษาที่ใช้ในการพัฒนาโปรแกรมคือภาษา C
- ใช้เทคนิค Disabling Interrupt เพื่อปิดการขัดจังหวะการทำงาน
- ใช้เทคนิค Lock เพื่อป้องกันการเข้าถึง Buffer ของ Thread อื่นๆ
- ใช้เทคนิค Mutual Execution เพื่อให้ ณ เวลาใดเวลาหนึ่งมีเพียงแค่ Thread เดียวที่สามารถเข้าถึงและเขียนข้อมูลใน Buffer ได้

### 2.3 เงื่อนไขการออกแบบ

ฟังก์ชัน append มีเอาไว้เพื่อเรียกใช้ฟังก์ชัน add โดยรองรับการทำงานแบบหลายเทรดพร้อม ๆ กัน โดยจะมีคุณสมบัติดังนี้

- Mutually exclusive access to buffer: ณ เวลาใดเวลาหนึ่ง จะมีเพียงเทรดหนึ่งเทรดเท่านั้นที่สามารถเพิ่มหรือลบรายการในบัฟเฟอร์ได้
- No buffer overflow: เทรดสำหรับ append จะสามารถเรียกใช้ add\_item ได้ก็ต่อเมื่อบัฟเฟอร์ไม่เต็ม (ให้รอจนกว่าบัฟเฟอร์จะมีพื้นที่เหลือก่อนที่จะเรียก add\_item)
- No buffer underflow: เทรดสำหรับ remove จะสามารถเรียกใช้ remove\_item ได้ก็ต่อเมื่อบัฟเฟอร์ไม่ว่าง (ถ้าว่างให้รอจนกว่าจะมีรายการในบัฟเฟอร์ก่อนเรียก remove\_item)
- No busy waiting: ห้ามใช้สปินล็อก (spinlock) ในการรอตรวจสอบเงื่อนไขของบัฟเฟอร์
- No producer starvation: เทรดที่จะเรียก append มีเวลารอคอยที่จำกัด
- No consumer starvation: เทรดที่จะเรียก remove มีเวลารอคอยที่จำกัด

## Chapter 3 Program Design

### Source Code

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4  #include<pthread.h>
5  #include<windows.h>
6
7  //Global Variable
8  struct node *head = NULL;
9  struct node *current = NULL;
10 int count = 0;
11 int append_count = 0;
12 int remove_count = 0;
13 int error_count = 0, request_temp;
14 volatile int running_threads = 0;
15 pthread_mutex_t running_mutex = PTHREAD_MUTEX_INITIALIZER;
16 pthread_mutex_t append_mutex = PTHREAD_MUTEX_INITIALIZER;
17 pthread_mutex_t remove_mutex = PTHREAD_MUTEX_INITIALIZER;
18 int temp_consumer_thread[500]={0};
19 int temp_producer_thread[500]={0};
20
21 //Input
22 int producer,consumer,buffer_size,request;
23
24 //Time catcher
25 clock_t initial,final;
26 double temp_cpu;
27
28 pthread_mutex_t lock;
29
30 typedef struct node
31 {
32     int data;
33     struct node* next;
34 }node;
35
36 int isEmpty() {
37     return head == NULL;
38 }
39
40 int isFull() {
41     return count == buffer_size;
42 }
43
44 int isProducerFull(){
45     return append_count == producer;
46 }
47
48 int isConsumerFull(){
49     return remove_count == consumer;
50 }
51
52 node* getNode(int num, node* nxt) {
53     node*q = (node*)malloc(sizeof(node));
54     q->data = num;
55     q->next = nxt;
56     return q;
57 }
58
59 int add(int num) {
60     node *temp = (struct node*) malloc(sizeof(struct node));
61     temp->data = num;
62
63     if (isEmpty()) {
64         head = temp;
65         head->next = head;
66         current = head;
67         count++;
68         return 0;
69     }
70     else {
71         if (count == buffer_size) {
72             return 1;
73         }
74     }
```



```

74         else {
75             current->next = temp;
76             temp->next = head;
77             current = temp;
78             count++;
79             return 0;
80         }
81     }
82 }
83
84 int del() {
85     if (isEmpty()) {
86         //printf("Buffer UNDERFLOW\n");
87         return 1;
88     }
89     else {
90         node* del = head;
91         head=head->next;
92         current->next = head;
93         free(del);
94         count--;
95         if (count == 0) {
96             head = NULL;
97             current = NULL;
98         }
99         return 0;
100     }
101 }
102
103 void *buffer_append(void *vargp) //add when not full
104 {
105
106     clock_t st,en;
107     double diff;
108     int timeout;
109     do{
110         if(request<0)break;
111         pthread_mutex_lock(&append_mutex);
112         append_count++;
113         pthread_mutex_unlock(&append_mutex);
114         st = clock();
115         timeout = ((int)vargp)%100+100;
116
117         if(isFull()==1){
118             while(isFull()==1){
119                 en = clock();
120                 diff = ((double)en-(double)st)/(CLOCKS_PER_SEC/1000);
121                 if((int)diff>=timeout){//Timeout
122                     pthread_mutex_lock(&running_mutex);
123                     error_count++;
124                     request--;
125                     pthread_mutex_unlock(&running_mutex);
126                     break;
127                 }
128             }
129         }
130         else{
131             pthread_mutex_lock(&lock);
132             int tid;
133             tid = (int)vargp;
134
135             if(request<0){
136                 running_threads--; pthread_mutex_unlock(&lock); pthread_exit(NULL);
137             }
138             add(tid);
139             append_count--;
140             request--;
141             pthread_mutex_unlock(&lock);
142             temp_producer_thread[tid]=0;
143         }
144     }while(request>0);
145     running_threads--;
146 }

```

```

147     pthread_exit(NULL);
148 }
149
150 void *buffer_remove(void *vargp)
151 {
152     clock_t st,en;
153     double diff;
154     int timeout;
155     do{
156         if(request<0)break;
157         pthread_mutex_lock(&remove_mutex);
158         remove_count++;
159         pthread_mutex_unlock(&remove_mutex);
160         st = clock();
161         timeout=((int)vargp)%100+100;
162
163         if(isEmpty()==1){
164             while(isEmpty()==1){
165                 en = clock();
166                 diff = ((double)en-(double)st)/(CLOCKS_PER_SEC/1000);
167                 if((int)diff>=timeout){
168                     pthread_mutex_lock(&running_mutex);
169                     error_count++;
170                     request--;
171                     pthread_mutex_unlock(&running_mutex);
172                     break;
173                 }
174             }
175         }
176         else{
177             pthread_mutex_lock(&lock);
178             int tid;
179             tid = (int)vargp;
180             if(request<0){
181                 running_threads--; pthread_mutex_unlock(&lock); pthread_exit(NULL);
182             }
183             del();
184             request--;
185             pthread_mutex_unlock(&lock);
186             temp_consumer_thread[tid]=0;
187         }
188     }while(request>0);
189     running_threads--;
190     pthread_exit(NULL);
191 }
192
193
194 int main(int argc, char *argv[]){
195     if(argc != 5){
196         printf("Parameter not correct\n");
197         return 1;
198     }
199     if (pthread_mutex_init(&lock, NULL) != 0){
200         printf("\n Mutex init failed\n");
201         return 1;
202     }
203
204     // Assign from input
205     producer = atoi(argv[1]);
206     consumer = atoi(argv[2]);
207     buffer_size = atoi(argv[3]);
208     request = atoi(argv[4]);
209
210     printf("Producers %d, Consumers %d\n", producer,consumer);
211     printf("Buffer size %d\n", buffer_size);
212     printf("Requests %d\n", request);
213
214     pthread_t threads[200];
215     int rc;
216     int t,i,j;
217
218     srand(time(NULL));
219     //Start time

```

```

220     initial=clock();
221     request_temp = request;
222
223     for(i=0;i<producer;i++){
224         if(temp_producer_thread[i]==0){
225             pthread_create(&threads[i], NULL, buffer_append, (void *)i);
226             pthread_mutex_lock(&running_mutex);
227             running_threads++;
228             //request--;
229             pthread_mutex_unlock(&running_mutex);
230             temp_producer_thread[i]=1;
231
232         }
233     }
234     for(j=producer;j<(producer+consumer);j++){
235         if(temp_consumer_thread[j]==0){
236             pthread_create(&threads[j], NULL, buffer_remove, (void *)j);
237             pthread_mutex_lock(&running_mutex);
238             running_threads++;
239             pthread_mutex_unlock(&running_mutex);
240             temp_consumer_thread[j]=1;
241
242         }
243         while(request>0){
244             Sleep(1);
245         }
246
247         //Stop time
248         final=clock();
249         temp_cpu = ((double)final-(double)initial) / CLOCKS_PER_SEC ;
250
251         //Outputs
252         //printf("Error count = %d\n",error_count);
253         printf("\nSuccessfully consumed %d requests
254         (%.2f%%)\n",request_temp-error_count,((double)(request_temp-error_count)/(double)r
255         equest_temp)* 100);
256         printf("Elapsed Time: %.2f s\n",temp_cpu);
257         printf("Throughput: %.2f successful
258         requests/s\n", (double)(request_temp-error_count)/temp_cpu);
259
260         //Last thing that main() should do
261         pthread_exit(NULL);
262         return 0;
263     }

```

## Circular Buffer

```
typedef struct node
{
    int data;
    struct node* next;
}node;

int isEmpty() { //empty is 1
    return head == NULL;
}

int isFull() { //full is 1
    return count == max;
}

node* getNode(int num, node* nxt) {
    node*q = (node*)malloc(sizeof(node));
    q->data = num;
    q->next = nxt;
    return q;
}
```

ในส่วนนี้เป็นส่วนของ Buffer โดยมี Data Structure เป็น Linked lists แบบ Circular ซึ่งเป็น Dynamic Memory Allocation จึงไม่มีข้อจำกัดในเรื่องของการใช้พื้นที่จัดเก็บ ในส่วนนี้เปรียบเสมือน Buffer ใน Kernel ของระบบปฏิบัติการซึ่งใช้สำหรับการเก็บค่าของข้อมูลต่างๆ โดยมีส่วนต่างๆ ที่เกี่ยวข้องดังต่อไปนี้

- isEmpty() จะคืนค่าเป็น 1 เมื่อพบว่า Buffer ว่าง
- isFull() จะคืนค่าเป็น 1 เมื่อพบว่า Buffer เต็ม

## Add Function

```
void add(int num) {  
    struct node *temp = (struct node*) malloc(sizeof(struct node));  
    temp->data = num;  
  
    if (isEmpty()) {  
        head = temp;  
        head->next = head;  
        current = head;  
        count++;  
    }  
    else {  
        if (count == max) {  
            printf("It full shit!\n");  
        }  
        else {  
            current->next = temp;  
            temp->next = head;  
            current = temp;  
            count++;  
        }  
    }  
}
```

ฟังก์ชันนี้ใช้เพื่อเพิ่มข้อมูลเข้าสู่ Buffer โดยจะถูกเรียกใช้จากฟังก์ชัน `void *buffer_append()` ซึ่งจะถูก Thread เรียกใช้ ซึ่งฟังก์ชันนี้จะรับค่า `num` เข้ามา โดยฟังก์ชันการทำงานเป็นดังนี้

- สร้างตัวแปรชนิด Pointer of Node ชื่อ `temp` สำหรับเก็บค่าชั่วคราวเพื่อไม่ให้ค่าหายไป
- นำค่าที่ Pass Parameter เข้ามาเก็บไว้ในส่วนของ Data
- ตรวจสอบว่า Buffer ว่าว่างอยู่หรือไม่หากว่างจะทำการเลื่อน pointer ไปยังตำแหน่งถัดไป แล้วเพิ่มค่าตัวนับว่าใช้ Buffer ไปเท่าใดแล้ว
- หาก Buffer ไม่ว่างก็จะไม่เพิ่มค่าเข้าไป

## Remove Function

```
int del() {  
    if (isEmpty()) {  
        printf("Fucking it empty\n");  
        return 1;  
    }  
    else {  
        node* del = head;  
        head = head->next;  
        current->next = head;  
        free(del); //Love leak  
        count--;  
        if (count == 0) {  
            head = NULL;  
            current = NULL;  
        }  
        return 0;  
    }  
}
```

ฟังก์ชันนี้จะทำงานตรงข้ามกับฟังก์ชัน add โดยจะเป็นการลบข้อมูลออกจาก Buffer ซึ่งจะถูกรเรียกใช้ผ่าน `void *buffer_remove()` ซึ่งจะถูก Thread เรียกใช้ โดยฟังก์ชันการทำงานเป็นดังนี้

- ตรวจสอบก่อนว่า Buffer ว่างหรือไม่
- หากไม่ได้ว่างก็จะให้ Pointer ชื่อ del ไปชี้ที่ head ของ buffer ก่อน
- จากนั้นทำการเปลี่ยนตำแหน่งของ Pointer โดยให้ head->next มาชี้ที่ head ตัวที่จะลบออก แล้วให้ current->next มาชี้ที่ head เพื่อทำให้เป็น Circular Linked Lists
- จากนั้นก็คืนค่าพื้นที่ตัวที่ลบโดยใช้ free()
- ลดค่า count ลง
- ตรวจสอบว่า Buffer มีค่า count เป็น 0 หรือไม่ ถ้าใช่แสดงว่า Buffer ว่างอยู่ก็ย้าย pointer มาอยู่ที่ตำแหน่ง NULL (ไม่มี List)

## Threads Function

```
int isProducerFull() {  
    return append_count == producer;  
}  
  
int isConsumerFull() {  
    return remove_count == consumer;  
}
```

ฟังก์ชัน 2 ฟังก์ชันนี้มีหน้าที่ในการจำกัดการสร้าง Thread ทั้ง 2 แบบคือ producer ทำหน้าที่เป็น Thread ผลิตคือเขียนข้อมูลลง buffer และ consumer ทำหน้าที่ลบข้อมูลออกจาก buffer โดยฟังก์ชันจะคืนค่าเป็น 1 หากจำนวน Thread เท่ากับตัวแปร producer และ consumer ซึ่งรับค่ามาในตอนแรกให้เป็นขีดจำกัดของ thread

## buffer\_append

```
void *buffer_append(void *vargp) //add when not full
{
    clock_t st,en;
    double diff;
    int timeout;
    do{
        if(request<0)break;
        pthread_mutex_lock(&append_mutex);
        append_count++;
        pthread_mutex_unlock(&append_mutex);
        st = clock();
        timeout = ((int)vargp)%100+100;

        if(isFull()==1){
            while(isFull()==1){
                en = clock();
                diff = ((double)en-(double)st)/(CLOCKS_PER_SEC/1000);
                if((int)diff>=timeout){//Timeout
                    pthread_mutex_lock(&running_mutex);
                    error_count++;
                    request--;
                    pthread_mutex_unlock(&running_mutex);
                    break;
                }
            }
        }
        else{
            pthread_mutex_lock(&lock);
            int tid;
            tid = (int)vargp;

            if(request<0){
                running_threads--; pthread_mutex_unlock(&lock); pthread_exit(NULL);
            }
            add(tid);
            append_count--;
            request--;
            pthread_mutex_unlock(&lock);
            temp_producer_thread[tid]=0;
        }

    }while(request>0);
    running_threads--;
    pthread_exit(NULL);
}
```

ฟังก์ชันนี้ใช้เพื่อกำหนดเวลาของแต่ละเทรต และเรียกใช้ฟังก์ชัน add เมื่อ Buffer ยังไม่เต็ม โดยฟังก์ชันมีการทำงานดังนี้

- สร้างตัวแปรชนิด clock\_t ชื่อ st,en สำหรับการจับ clock เริ่มต้นและสิ้นสุดของการทำงาน โดยเป็นการเรียกใช้จาก library time.h
- ตัวแปรชนิด double ชื่อ diff เป็นผลลัพธ์การลบของ st และ en เพื่อให้ทราบเวลาที่ใช่ไป
- ตัวแปรชนิด int ชื่อ timeout ไว้สุ่มเวลาสิ้นสุดของเทรตนั้น เป็น static random
- เมื่อกำหนดตัวแปรเสร็จแล้ว จะตรวจสอบว่า Buffer เต็มหรือไม่ ถ้าไม่เต็ม เทรตนั้นจะถูก Lock สแตตัสแล้วเรียกใช้งานฟังก์ชัน add
- หลังจากนั้นจึง Unlock สแตตัสเทรตนั้น และกลับไปทำเทรตต่อไป



## buffer\_remove

```
void *buffer_remove(void *vargp)
{
    clock_t st,en;
    double diff;
    int timeout;
    do{
        if(request<0)break;
        pthread_mutex_lock(&remove_mutex);
        remove_count++;
        pthread_mutex_unlock(&remove_mutex);
        st = clock();
        timeout=((int) vargp)%100+100;

        if(isEmpty()==1){
            while(isEmpty()==1){
                en = clock();
                diff = ((double)en-(double)st)/(CLOCKS_PER_SEC/1000);
                if((int)diff>=timeout){
                    pthread_mutex_lock(&running_mutex);
                    error_count++;
                    request--;
                    pthread_mutex_unlock(&running_mutex);
                    break;
                }
            }
        }
        else{
            pthread_mutex_lock(&lock);
            int tid;
            tid = (int)vargp;
            if(request<0){
                running_threads--; pthread_mutex_unlock(&lock); pthread_exit(NULL);
            }
            del();
            request--;
            pthread_mutex_unlock(&lock);
            temp_consumer_thread[tid]=0;
        }
    }while(request>0);
    running_threads--;
    pthread_exit(NULL);
}
```

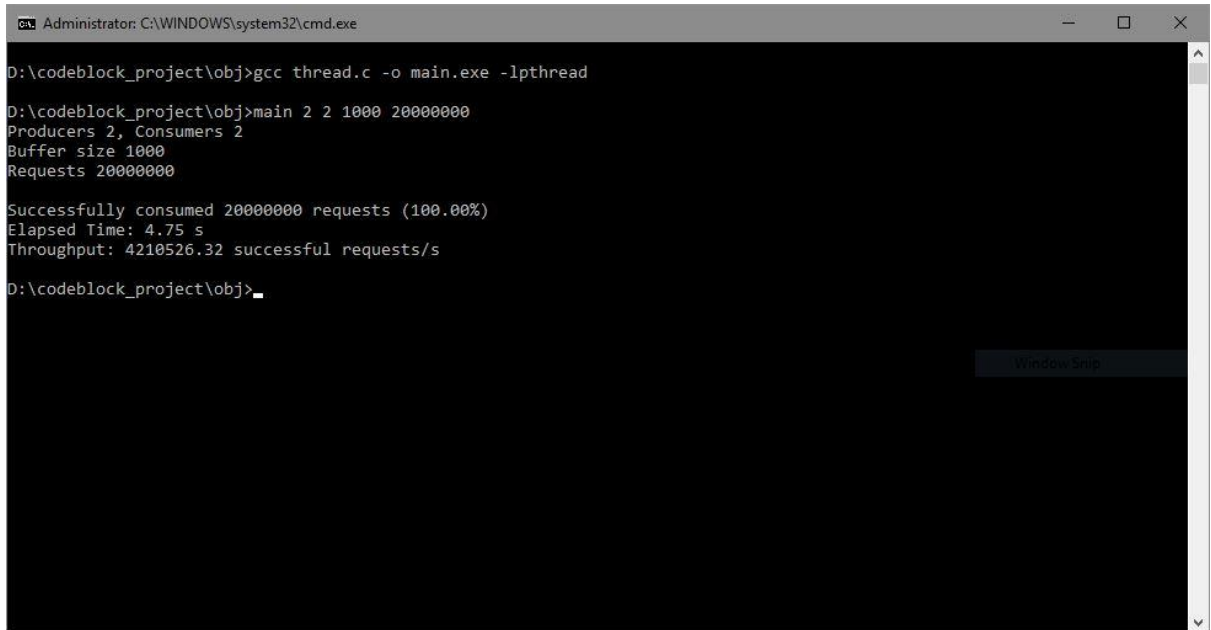
ฟังก์ชันนี้ใช้สำหรับลบข้อมูลใน Buffer ออกเมื่อมีข้อมูลใน Buffer อยู่

- ก่อนอื่นจะทำการตรวจสอบก่อนว่ามีบัฟเฟอร์อยู่หรือไม่
- เมื่อถูกต้องตามเงื่อนไขโดยใช้ while loop เพื่อตรวจสอบก่อนว่า Buffer ว่ามีข้อมูลหรือไม่ ถ้ามีอยู่ก็จะทำการตั้งเวลาก่อนจะใช้ lock แล้วจะทำการลบออก
- มีขั้นตอนการจับเวลาเหมือนกับฟังก์ชัน buffer\_append

\*\* ทั้งฟังก์ชัน buffer\_append และ buffer\_remove นั้นแต่ละ Thread จำเป็นต้องมีเวลารอที่ไม่เท่ากัน เพื่อให้สามารถเข้าถึงทรัพยากรได้ในช่วงเวลาที่แตกต่างกัน ในที่นี้เราใช้ตัวแปรชื่อ vargp เป็นตัวแปรชนิด Pointer of integer ซึ่งไปยัง thread index เพื่อให้ได้เลขที่ต่างกันไปเป็น static random

## ผลการทดลอง

- ผลจากการ execute จำนวน 20,000,000 request



```
Administrator: C:\WINDOWS\system32\cmd.exe

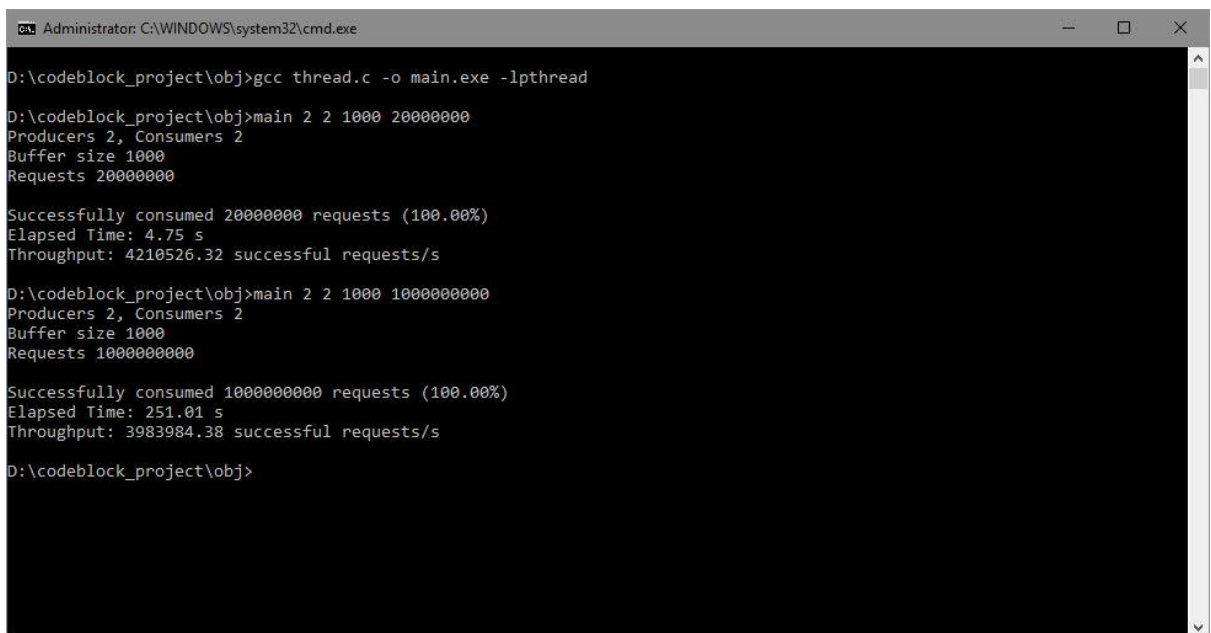
D:\codeblock_project\obj>gcc thread.c -o main.exe -lpthread

D:\codeblock_project\obj>main 2 2 1000 20000000
Producers 2, Consumers 2
Buffer size 1000
Requests 20000000

Successfully consumed 20000000 requests (100.00%)
Elapsed Time: 4.75 s
Throughput: 4210526.32 successful requests/s

D:\codeblock_project\obj>_
```

- ผลจากการ execute จำนวน 1,000,000,000 request



```
Administrator: C:\WINDOWS\system32\cmd.exe

D:\codeblock_project\obj>gcc thread.c -o main.exe -lpthread

D:\codeblock_project\obj>main 2 2 1000 20000000
Producers 2, Consumers 2
Buffer size 1000
Requests 20000000

Successfully consumed 20000000 requests (100.00%)
Elapsed Time: 4.75 s
Throughput: 4210526.32 successful requests/s

D:\codeblock_project\obj>main 2 2 1000 1000000000
Producers 2, Consumers 2
Buffer size 1000
Requests 1000000000

Successfully consumed 1000000000 requests (100.00%)
Elapsed Time: 251.01 s
Throughput: 3983984.38 successful requests/s

D:\codeblock_project\obj>
```

## Reference

Rahul Jain, Multithreading in C. site : <http://www.geeksforgeeks.org/multithreading-c-2/>, date : 19/02/2017

ChristopherWright, Multithreading in C, POSIX style. site : <http://softpixel.com/~cwright/programming/threads/threads.c.php>, date : 19/02/2017

macboypro, Producer-consumer problem in [C] using pthreads/bounded-buffer/semaphores site : <https://macboypro.wordpress.com/2009/05/25/producer-consumer-problem-using-cpthreadsbounded-buffer>, date : 18/02/2017