Operating System

Group 3

Member of Group

❑ 56010492	Tanthai	Earkanna
❑ 57010821	Pongsak	Sanungwong
❑ 57010836	Pornthep	Saoung
❑ 57010978	Phanuwat	Emampaiwong
❑ 57011116	Waranya	Khitpraphiumphol
❑ 57011180	Viracha	Laowhapoonrangsi
❑ 57011220	Sutthathan	Chanchartree
❑ 57011229	Saweera	Apintanapong
❑ 57011363	Sirawit	Wanarattikal
❑ 57011470	Anurak	Jannawan

ASSIGNMENT 1

Synchronization

1.

Problem

Let’s start with
What is the
Probkem?

A PROBLEM IS

- Circular Buffer
  - Size(1 <= N <=1000)
- Basic Operation
  - add / remove item
- Concurrent Operation
  - Mutually exclusive access
  - No buffer overflow /underflow
  - No busy waiting
  - No producer starvation/ consumer starvation
- Buffer Benchmark

What’s your DeSi9N?

What should you THINK about ?

Threads

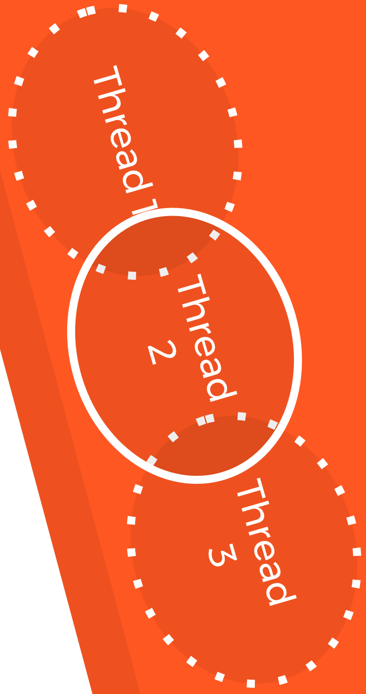
Shared Objects

Problem about lock

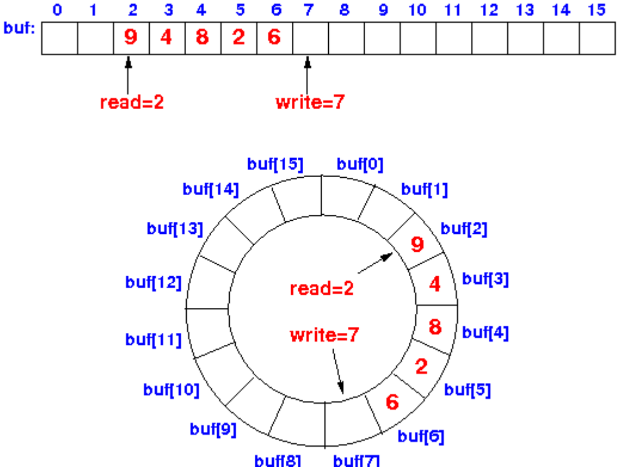
Performance

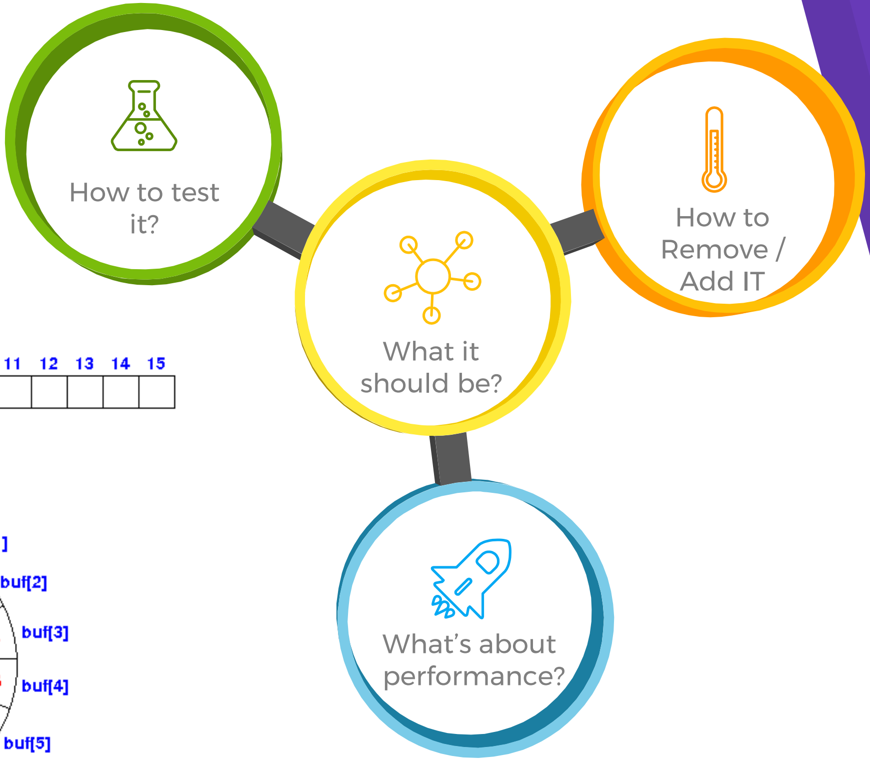
What's your language?

C Language
Use pthread.h library



STRUCTURE of BUFFER





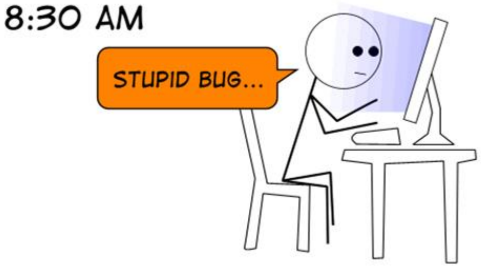
How to test it?

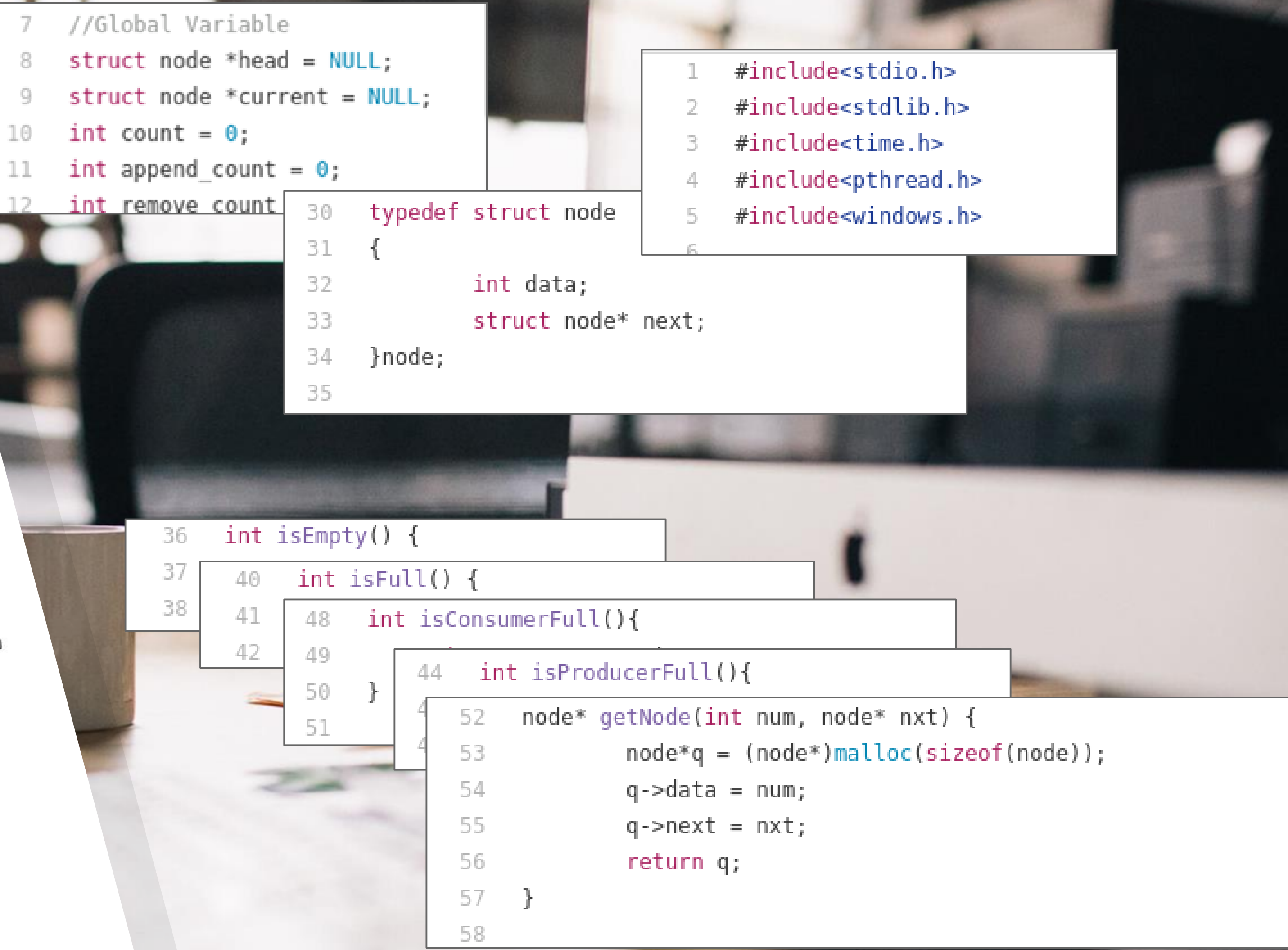
What it should be?

How to Remove / Add IT

What's about performance?

Modify Code.





```
//Global Variable
struct node *head = NULL;
struct node *current = NULL;
int count = 0;
int append_count = 0;
int remove_count
```

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

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

```
int isEmpty() {
```

```
int isFull() {
```

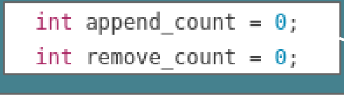
```
int isConsumerFull(){

}
```

```
int isProducerFull(){
```

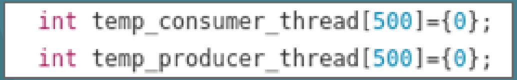
```
node* getNode(int num, node* nxt) {
        node*q = (node*)malloc(sizeof(node));
        q->data = num;
        q->next = nxt;
        return q;
}
```

What's a **SHARED** object?

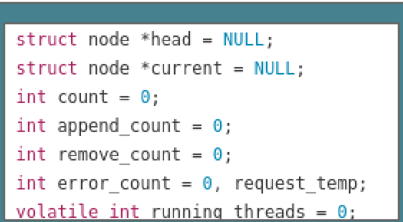
Shared Object



```c
int append_count = 0;
int remove_count = 0;
```

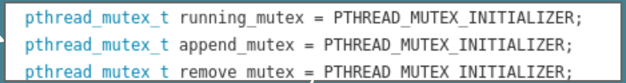


```c
int temp_consumer_thread[500]={0};
int temp_producer_thread[500]={0};
```



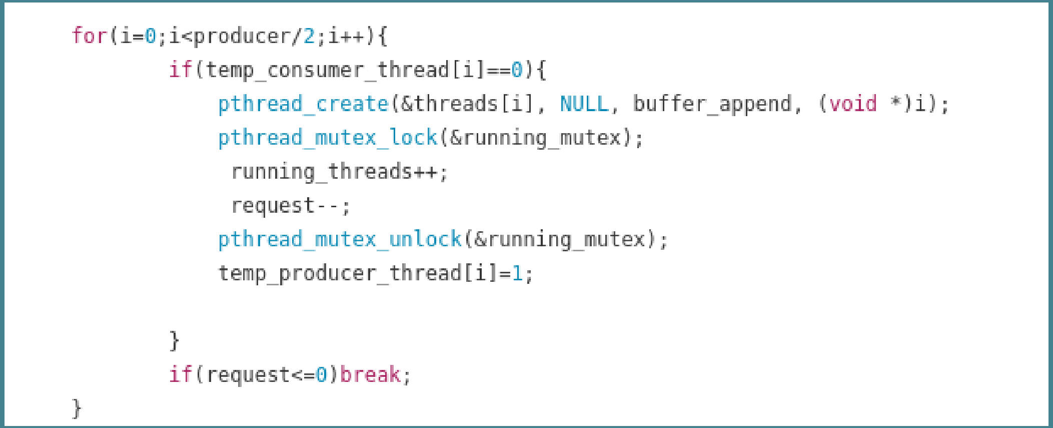
```c
struct node *head = NULL;
struct node *current = NULL;
int count = 0;
int append_count = 0;
int remove_count = 0;
int error_count = 0, request_temp;
volatile int running_threads = 0;
```

Lock



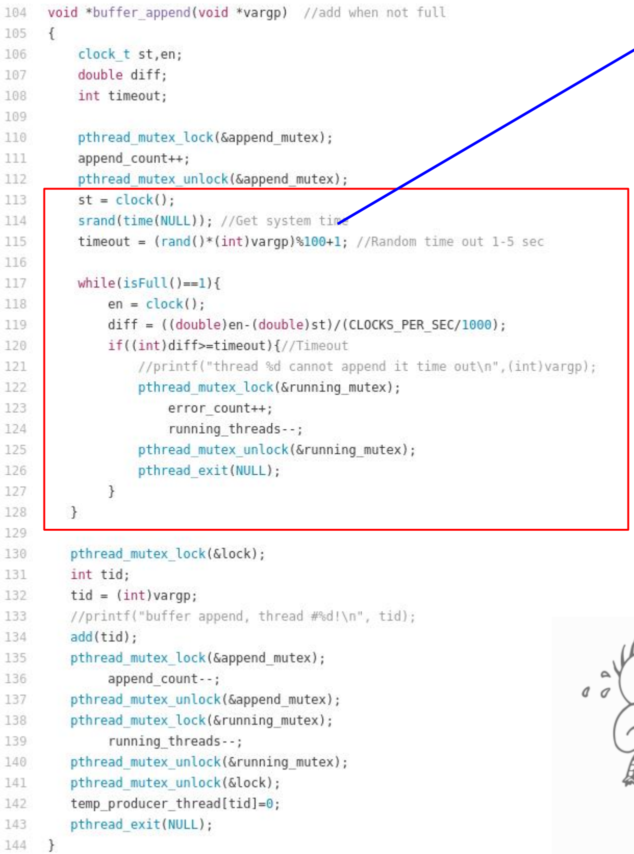
```c
pthread_mutex_t running_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t append_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t remove_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Example



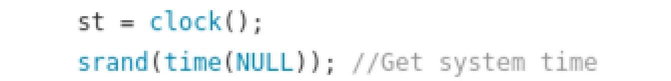
```c
for(i=0;i<producer/2;i++){
        if(temp_consumer_thread[i]==0){
            pthread_create(&threads[i], NULL, buffer_append, (void *)i);
            pthread_mutex_lock(&running_mutex);
             running_threads++;
             request--;
            pthread_mutex_unlock(&running_mutex);
            temp_producer_thread[i]=1;

        }
        if(request<=0)break;
}
```

Avoid problem with lock



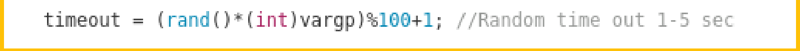
```c
void *buffer_append(void *vargp)  //add when not full
{
    clock_t st,en;
    double diff;
    int timeout;

    pthread_mutex_lock(&append_mutex);
    append_count++;
    pthread_mutex_unlock(&append_mutex);
    st = clock();
    srand(time(NULL)); //Get system time
    timeout = (rand()*(int)vargp)%100+1; //Random time out 1-5 sec

    while(isFull()==1){
        en = clock();
        diff = ((double)en-(double)st)/(CLOCKS_PER_SEC/1000);
        if((int)diff>=timeout){//Timeout
            //printf("thread %d cannot append it time out\n",(int)vargp);
            pthread_mutex_lock(&running_mutex);
                error_count++;
                running_threads--;
            pthread_mutex_unlock(&running_mutex);
            pthread_exit(NULL);
        }
    }

    pthread_mutex_lock(&lock);
    int tid;
    tid = (int)vargp;
    //printf("buffer append, thread #%d!\n", tid);
    add(tid);
    pthread_mutex_lock(&append_mutex);
        append_count--;
    pthread_mutex_unlock(&append_mutex);
    pthread_mutex_lock(&running_mutex);
        running_threads--;
    pthread_mutex_unlock(&running_mutex);
    pthread_mutex_unlock(&lock);
    temp_producer_thread[tid]=0;
    pthread_exit(NULL);
}
```

Initial timeout



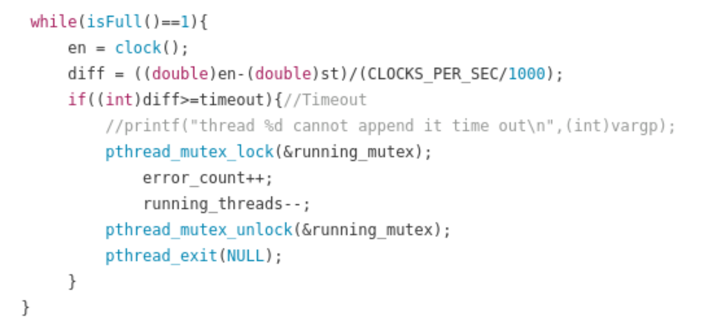
```c
st = clock();
srand(time(NULL)); //Get system time
```

Limit timeout

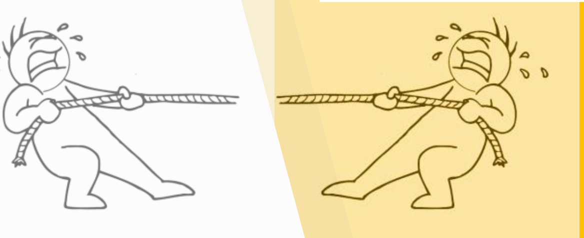


```c
timeout = (rand()*(int)vargp)%100+1; //Random time out 1-5 sec
```

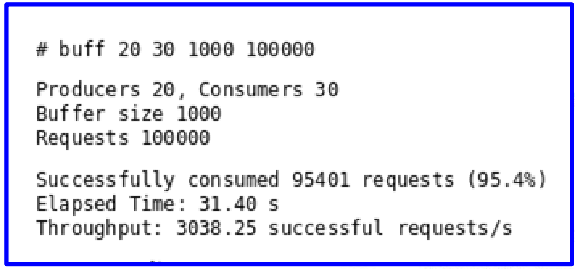
Kill thread



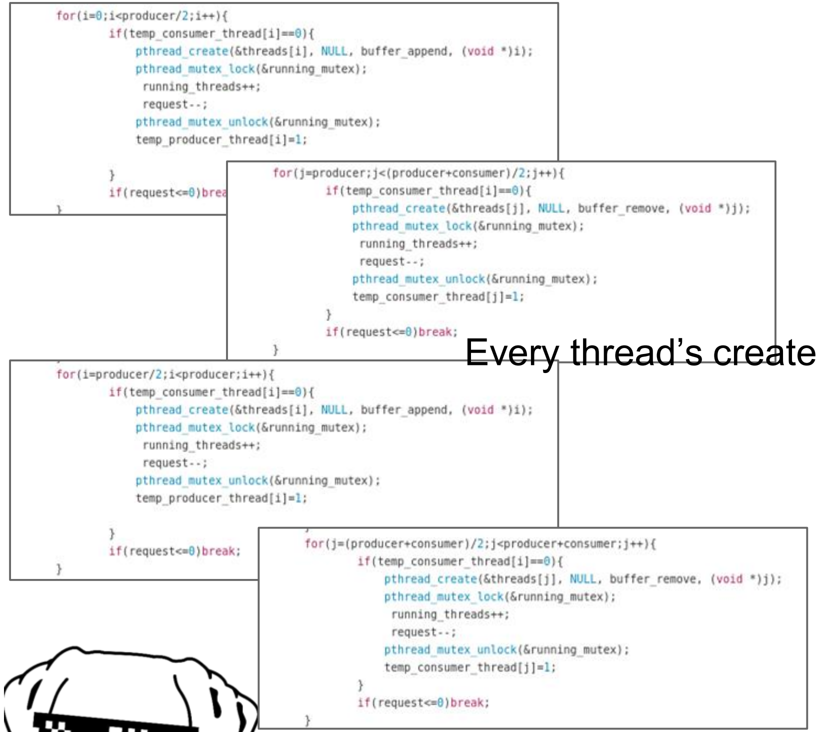
```c
while(isFull()==1){
    en = clock();
    diff = ((double)en-(double)st)/(CLOCKS_PER_SEC/1000);
    if((int)diff>=timeout){//Timeout
        //printf("thread %d cannot append it time out\n",(int)vargp);
        pthread_mutex_lock(&running_mutex);
            error_count++;
            running_threads--;
        pthread_mutex_unlock(&running_mutex);
        pthread_exit(NULL);
    }
}
```



Performance



```
# buff 20 30 1000 100000

Producers 20, Consumers 30
Buffer size 1000
Requests 100000

Successfully consumed 95401 requests (95.4%)
Elapsed Time: 31.40 s
Throughput: 3038.25 successful requests/s
```



```
for(i=0;i<producer/2;i++){
        if(temp_consumer_thread[i]==0){
            pthread_create(&threads[i], NULL, buffer_append, (void *)i);
            pthread_mutex_lock(&running_mutex);
             running_threads++;
             request--;
            pthread_mutex_unlock(&running_mutex);
            temp_producer_thread[i]=1;

        }
        if(request<=0)brea
}
```

```
for(j=producer;j<(producer+consumer)/2;j++){
        if(temp_consumer_thread[i]==0){
            pthread_create(&threads[j], NULL, buffer_remove, (void *)j);
            pthread_mutex_lock(&running_mutex);
             running_threads++;
             request--;
            pthread_mutex_unlock(&running_mutex);
            temp_consumer_thread[j]=1;
        }
        if(request<=0)break;
}
```

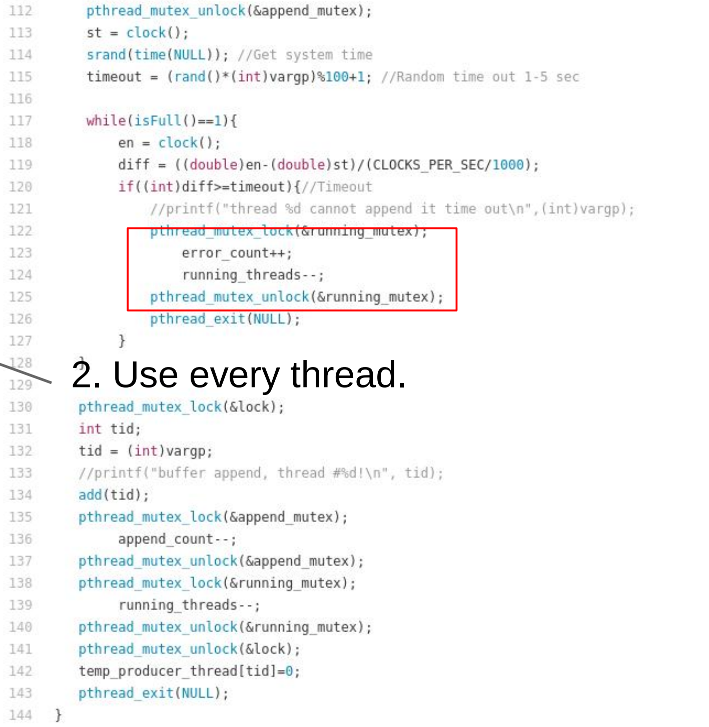
```
for(i=producer/2;i<producer;i++){
        if(temp_consumer_thread[i]==0){
            pthread_create(&threads[i], NULL, buffer_append, (void *)i);
            pthread_mutex_lock(&running_mutex);
             running_threads++;
             request--;
            pthread_mutex_unlock(&running_mutex);
            temp_producer_thread[i]=1;

        }
        if(request<=0)break;
}
```

```
for(j=(producer+consumer)/2;j<producer+consumer;j++){
        if(temp_consumer_thread[i]==0){
            pthread_create(&threads[j], NULL, buffer_remove, (void *)j);
            pthread_mutex_lock(&running_mutex);
             running_threads++;
             request--;
            pthread_mutex_unlock(&running_mutex);
            temp_consumer_thread[j]=1;
        }
        if(request<=0)break;
}
```

Every thread’s created in the same time.

```
d *vargp)  //add when not full
```

```
&append_mutex);
```



```
    pthread_mutex_unlock(&append_mutex);
    st = clock();
    srand(time(NULL)); //Get system time
    timeout = (rand()*(int)vargp)%100+1; //Random time out 1-5 sec

    while(isFull()==1){
        en = clock();
        diff = ((double)en-(double)st)/(CLOCKS_PER_SEC/1000);
        if((int)diff>=timeout){//Timeout
            //printf("thread %d cannot append it time out\n",(int)vargp);
            pthread_mutex_lock(&running_mutex);
                error_count++;
                running_threads--;
            pthread_mutex_unlock(&running_mutex);
            pthread_exit(NULL);
        }
    }

    pthread_mutex_lock(&lock);
    int tid;
    tid = (int)vargp;
    //printf("buffer append, thread #%d!\n", tid);
    add(tid);
    pthread_mutex_lock(&append_mutex);
        append_count--;
    pthread_mutex_unlock(&append_mutex);
    pthread_mutex_lock(&running_mutex);
        running_threads--;
    pthread_mutex_unlock(&running_mutex);
    pthread_mutex_unlock(&lock);
    temp_producer_thread[tid]=0;
    pthread_exit(NULL);
}
```

2. Use every thread.

1.Try to never make NULL circular buffer.