

**National University Ho Chi Minh City**  
**HO CHI MINH UNIVERSITY OF TECHNOLOGY**  
**FACULTY OF COMPUTER SCIENCE AND ENGINEERING**



**Operating System - CO2018**

**Instructor:** Assoc. Phạm Hoàng Anh

**Class:** CC04

**Name:** Nguyễn Quang Thiện

**Student ID:** 21522994

*Completion Day: March 29<sup>th</sup>, 2023*

# Table of Contents

<b>I. Shared buffer problem.....</b>	<b>2</b>
1. Requirement .....	2
2. Steps to build .....	2
3. Explain each code paragraph.....	2
4. Implementing .....	3
Source code of Shared buffer problem .....	4
<b>II. Bounded buffer problem .....</b>	<b>5</b>
1. Requirement .....	5
2. Steps to build .....	5
3. Explain the code paragraph .....	6
4. Implementing .....	9
Source code of Bounded buffer problem.....	10
<b>III. Dining-Philosopher problem.....</b>	<b>12</b>
1. Requirement .....	12
2. Steps to build .....	12
3. Explain each code paragraph.....	13
4. Implementing .....	15
Source code of Dining-Philosopher problem .....	17
<b>IV. Aggregated sum .....</b>	<b>19</b>
1. Requirement .....	19
2. Steps to build .....	19
3. Explain each code paragraph.....	19
4. Implementing .....	20
Source code of Aggregated sum .....	21

## I. Shared buffer problem

### 1. Requirement

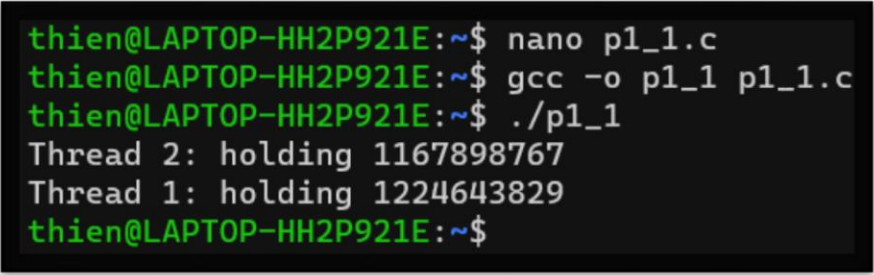
- Recognize the wrong issue and propose a fix mechanism using the provided synchronization tool.

- Using `pthread_mutex_lock()` and `pthread_mutex_unlock()`.

### 2. Steps to build

There is a possible chance that the process will be overwritten if not having a block technique to protect critical sections and thus prevent race conditions when using the initial code.

The program not using block method before. Because there is no way to prevent other processes from changing the "count" concurrently, there might be a loss of data:

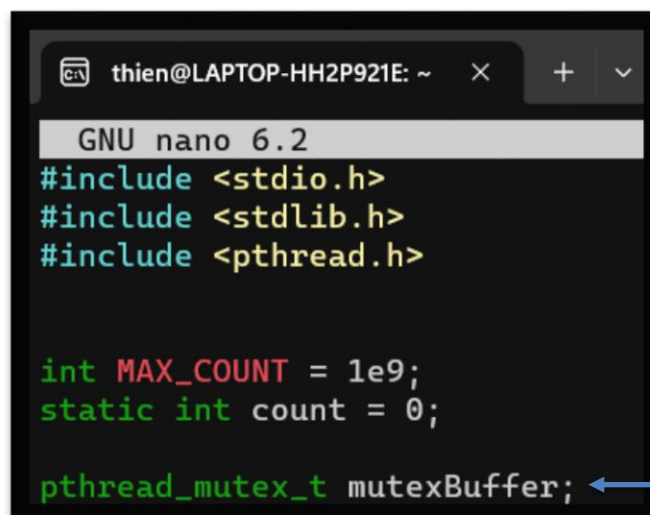


```
thien@LAPTOP-HH2P921E:~$ nano p1_1.c
thien@LAPTOP-HH2P921E:~$ gcc -o p1_1 p1_1.c
thien@LAPTOP-HH2P921E:~$ ./p1_1
Thread 2: holding 1167898767
Thread 1: holding 1224643829
thien@LAPTOP-HH2P921E:~$
```

Implementing a mutex clock so that only one process can access Critical Section (CS) and the other will wait until the process finishes in CS.

### 3. Explain each code paragraph

Creating a mutex variable (mutexBuffer):



```
thien@LAPTOP-HH2P921E: ~  ×  +  ▾
GNU nano 6.2
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int MAX_COUNT = 1e9;
static int count = 0;

pthread_mutex_t mutexBuffer; ←
```

Then, in the `*f_count` function, not only adding the mutex clock so that only one process will in the CS but also change the count value to satisfy the mutual

exclusion. After finishing the modification, implementing the unlock so that the other process can access the CS and set the count value in the CS one by one.

```
void *f_count(void *sid) {
    pthread_mutex_lock(&mutexBuffer); ←
    int i;
    for (i = 0; i < MAX_COUNT; i++) {
        count = count + 1;
    }
    printf("Thread %s: holding %d \n", (char *) sid, count);
    pthread_mutex_unlock(&mutexBuffer); ←
}
```

After that, initializing the mutex variable at the beginning and destroying the mutex variable when the program finish at the int main().

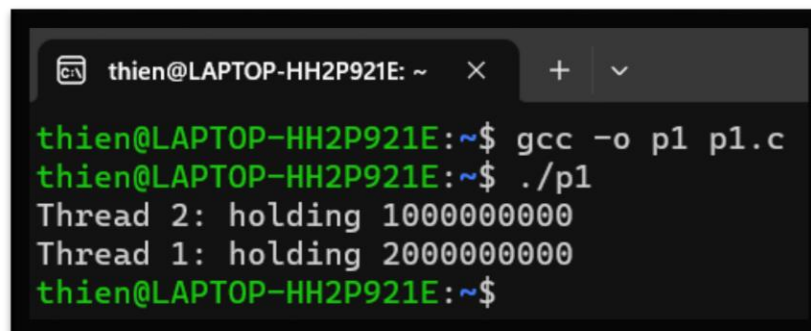
```
int main() {
    pthread_t thread1, thread2;
    pthread_mutex_init(&mutexBuffer, NULL); ←
    /* Create independent threads each of which will execute function */
    pthread_create( &thread1, NULL, &f_count, "1");
    pthread_create( &thread2, NULL, &f_count, "2");

    // Wait for thread th1 finish
    pthread_join( thread1, NULL);

    // Wait for thread th1 finish
    pthread_join( thread2, NULL);
    pthread_mutex_destroy(&mutexBuffer); ←
    return 0;
}
```

#### 4. *Implementing*

Results:



```
thien@LAPTOP-HH2P921E: ~ × + v
thien@LAPTOP-HH2P921E:~$ gcc -o p1 p1.c
thien@LAPTOP-HH2P921E:~$ ./p1
Thread 2: holding 1000000000
Thread 1: holding 2000000000
thien@LAPTOP-HH2P921E:~$
```

The count variable after using mutex clock is corrected as expected

## Source code of Shared buffer problem

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

int MAX_COUNT = 1e9;
static int count = 0;

pthread_mutex_t mutexBuffer;

void *f_count(void *sid) {
    pthread_mutex_lock(&mutexBuffer);
    int i;
    for (i = 0; i < MAX_COUNT; i++) {
        count = count + 1;
    }
    pthread_mutex_unlock(&mutexBuffer);
    printf("Thread %s: holding %d \n", (char *) sid, count);
}

int main() {
    pthread_t thread1, thread2;
    pthread_mutex_init(&mutexBuffer, NULL);
    /* Create independent threads each of which will execute function */
    pthread_create( &thread1, NULL, &f_count, "1");
    pthread_create( &thread2, NULL, &f_count, "2");

    // Wait for thread th1 finish
    pthread_join( thread1, NULL);

    // Wait for thread th1 finish
    pthread_join( thread2, NULL);
    pthread_mutex_destroy(&mutexBuffer);
    return 0;
}
```

## II. Bounded buffer problem

### 1. Requirement

- Recognize the wrong issue and propose a fix mechanism using the provided synchronization tool.

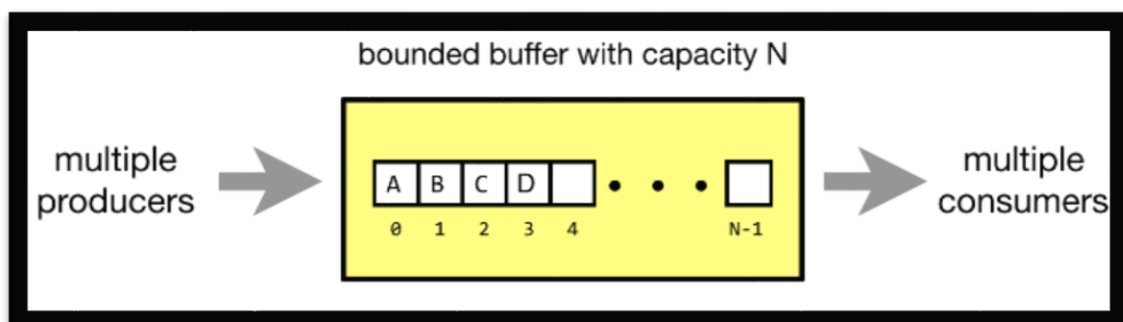
- Using `sem_wait()` and `sem_signal()` to protect `consumer()` and `producer()` thread worker.

### 2. Steps to build

The problem is that when the producer ends the consumer falls into an infinite loop. The program before:

```
thien@LAPTOP-HH2P921E:~$ ./p2p
Producer 0 put data 0
Consumer 0 get data 0
Producer 0 put data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
^C
thien@LAPTOP-HH2P921E:~$
```

A well-known example of accessing data to a shared resource is the producer-consumer problem, commonly known as the bounded-buffer problem. Many producers and consumers can share a single buffer owing to bounded buffers. Consumers read data from the buffer after producers write data there.

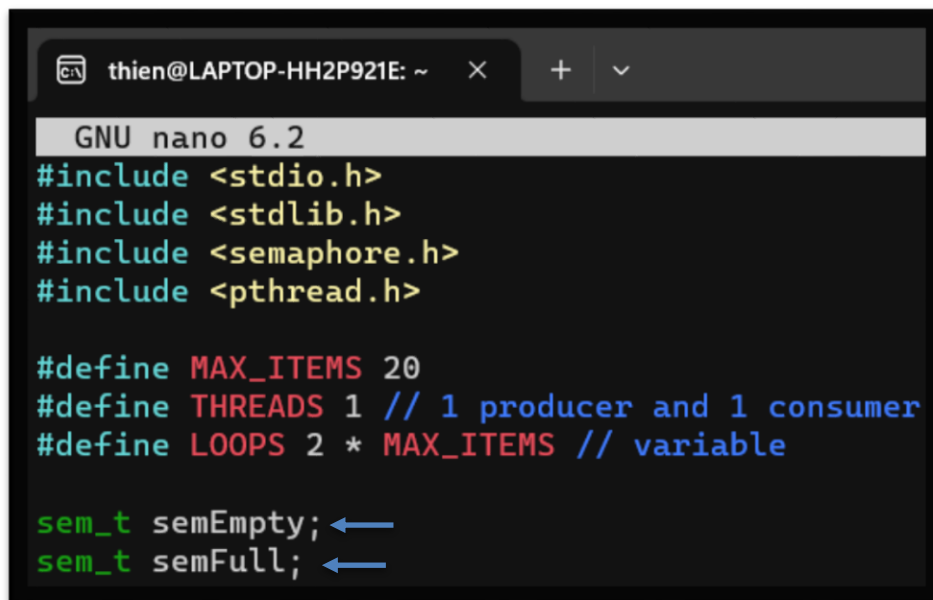


To solving the following problem, using two semaphores variables which namely Full and Empty. Producers must block if the buffer is Full. Moreover,

consumers must block if the block is Empty. And two counting semaphores will be used for this.

### 3. *Explain the code paragraph*

Increasing the MAX\_ITEMS to 20 so that the LOOPS will stop when the consumer get data at 39 (0 to 39). Also, initializing 2 counting semaphores: semEmpty and semFull. We do not need a mutex lock because there are just two threads at first. We will need a technique to block if there are several producers or consumers in order to prevent the loss of data.



```
thien@LAPTOP-HH2P921E: ~  ×  +  ▾
GNU nano 6.2
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

#define MAX_ITEMS 20
#define THREADS 1 // 1 producer and 1 consumer
#define LOOPS 2 * MAX_ITEMS // variable

sem_t semEmpty; ←
sem_t semFull; ←
```

#### 2 counting semaphores:

Using one semaphore named Full to count the number of data item in the buffer. Initialise this semaphore to 0, the buffer starts out with “MAX\_ITEMS” empty spaces and producer will have to wait until there is an empty space if this semaphore decreases to 0, which indicates that the buffer is no longer empty. Producer must wait before writing from the buffer and consumer will post (or signal) after reading from the buffer. The

Using one semaphore named Empty to count the empty slot in the buffer. Initialise this semaphore to MAX\_ITEMS, the complete number must be 0, as the buffer is empty and consumer will need to wait until there is data in the buffer if it is not full. Producer will post (or signal) after writing from the buffer and consumer must wait before reading from the buffer.

Also as mentioned above, producer cannot put data into the buffer if it is Full and so on to consumer is unable to get data if the buffer is Empty. Furthermore, when tmp equals to 39 (LOOPS - 1) then break the while to avoid the case that the consumer waiting infinitely when the Empty variable is 0.

```
void * producer(void * arg) {
    int i;
    int tid = (int) arg;
    for (i = 0; i < LOOPS; i++) {
        /*TODO: Fill in the synchronization stuff */
        sem_wait(&semEmpty); ←

        put(i); // line P2
        printf("Producer %d put data %d\n", tid, i);
        sleep(1);

        sem_post(&semFull); ←
        /*TODO: Fill in the synchronization stuff */
    }
    pthread_exit(NULL);
}
```

```
void * consumer(void * arg) {
    int i, tmp = 0;
    int tid = (int) arg;
    while (tmp != -1) {
        /*TODO: Fill in the synchronization stuff */
        sem_wait(&semFull); ←

        tmp = get(); // line C2
        printf("Consumer %d get data %d\n", tid, tmp);
        sleep(1);

        sem_post(&semEmpty); ←
        /*TODO: Fill in the synchronization stuff */
        if (tmp == LOOPS - 1) ←
            break;
    }
    pthread_exit(NULL);
}
```



Initializing the semaphore variables when starts the process of the int main() and destroying them when the process ends.

```
int main(int argc, char ** argv) {
    int i, j;
    int tid[THREADS];
    pthread_t producers[THREADS];
    pthread_t consumers[THREADS];

    /*TODO: Fill in the synchronization stuff */
    sem_init(&semEmpty, 0, MAX_ITEMS); ←
    sem_init(&semFull, 0, 0); ←

    for (i = 0; i < THREADS; i++) {
        tid[i] = i;
        // Create producer thread
        pthread_create( & producers[i], NULL, producer, (void * ) tid[i]);

        // Create consumer thread
        pthread_create( & consumers[i], NULL, consumer, (void * ) tid[i]);
    }

    for (i = 0; i < THREADS; i++) {
        pthread_join(producers[i], NULL);
        pthread_join(consumers[i], NULL);
    }

    sem_destroy(&semEmpty); ←
    sem_destroy(&semFull); ←
    /*TODO: Fill in the synchronization stuff */

    return 0;
}
```

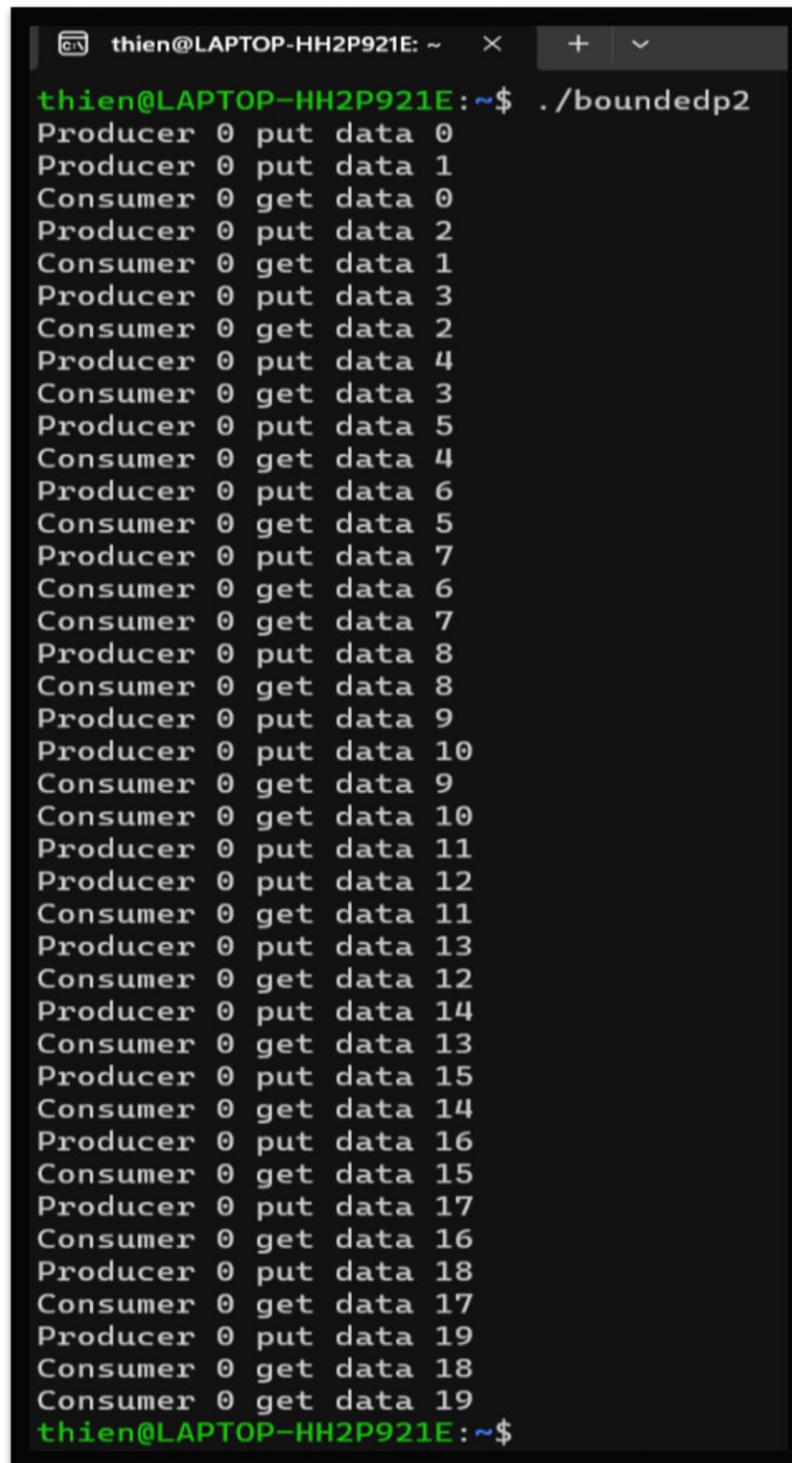
Similarly with producer and consumer function, adding sem\_wait(&semEmpty) and sem\_post(&semFull) for put function (equivalent to the producer function). Also, adding sem\_wait(&semFull) and sem\_post(&semEmpty) for get function (equivalent to the consumer function).

```
void put(int value) {
    sem_wait(&semEmpty); ←
    buffer[fill] = value; // line f1
    fill = (fill + 1) % MAX_ITEMS; // line f2
    sem_post(&semFull); ←
}

int get() {
    sem_wait(&semFull); ←
    int tmp = buffer[use]; // line g1
    use = (use + 1) % MAX_ITEMS; // line g2
    sem_post(&semEmpty); ←
    return tmp;
}
```

#### 4. *Implementing*

Results:

A terminal window titled 'thien@LAPTOP-HH2P921E: ~' with standard window controls. The prompt is 'thien@LAPTOP-HH2P921E: ~\$'. The command './boundedp2' has been executed, resulting in a series of 40 lines of output. Each line shows either a 'Producer' or 'Consumer' process performing a 'put' or 'get' operation on 'data' with an index from 0 to 19. The output demonstrates a correct interleaving of producer and consumer actions, with the consumer always retrieving data that has been previously produced. The terminal ends with the prompt 'thien@LAPTOP-HH2P921E: ~\$' again.

```
thien@LAPTOP-HH2P921E: ~$ ./boundedp2
Producer 0 put data 0
Producer 0 put data 1
Consumer 0 get data 0
Producer 0 put data 2
Consumer 0 get data 1
Producer 0 put data 3
Consumer 0 get data 2
Producer 0 put data 4
Consumer 0 get data 3
Producer 0 put data 5
Consumer 0 get data 4
Producer 0 put data 6
Consumer 0 get data 5
Producer 0 put data 7
Consumer 0 get data 6
Consumer 0 get data 7
Producer 0 put data 8
Consumer 0 get data 8
Producer 0 put data 9
Producer 0 put data 10
Consumer 0 get data 9
Consumer 0 get data 10
Producer 0 put data 11
Producer 0 put data 12
Consumer 0 get data 11
Producer 0 put data 13
Consumer 0 get data 12
Producer 0 put data 14
Consumer 0 get data 13
Producer 0 put data 15
Consumer 0 get data 14
Producer 0 put data 16
Consumer 0 get data 15
Producer 0 put data 17
Consumer 0 get data 16
Producer 0 put data 18
Consumer 0 get data 17
Producer 0 put data 19
Consumer 0 get data 18
Consumer 0 get data 19
thien@LAPTOP-HH2P921E: ~$
```

After adding the two semaphore variables, the process runs correctly as expected.

## Source code of Bounded buffer problem

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#define MAX_ITEMS 20
#define THREADS 1 // 1 producer and 1 consumer
#define LOOPS 2 * MAX_ITEMS // variable

sem_t semEmpty;
sem_t semFull;
// Initiate shared buffer
int buffer[MAX_ITEMS];
int fill = 0;
int use = 0;

/*TODO: Fill in the synchronization stuff */
void put(int value); // put data into buffer
int get(); // get data from buffer

void * producer(void * arg) {
    int i;
    int tid = (int) arg;
    for (i = 0; i < LOOPS; i++) {
        /*TODO: Fill in the synchronization stuff */
        sem_wait(&semEmpty);

        put(i); // line P2
        printf("Producer %d put data %d\n", tid, i);
        sleep(1);

        sem_post(&semFull);
        /*TODO: Fill in the synchronization stuff */
    }
    pthread_exit(NULL);
}

void * consumer(void * arg) {
    int i, tmp = 0;
    int tid = (int) arg;
    while (tmp != -1) {
        /*TODO: Fill in the synchronization stuff */
        sem_wait(&semFull);

        tmp = get(); // line C2
        printf("Consumer %d get data %d\n", tid, tmp);
        sleep(1);

        sem_post(&semEmpty);
        /*TODO: Fill in the synchronization stuff */
    }
}
```

```

    if (tmp == LOOPS - 1)
        break;
    }
    pthread_exit(NULL);
}

int main(int argc, char ** argv) {
    int i, j;
    int tid[THREADS];
    pthread_t producers[THREADS];
    pthread_t consumers[THREADS];

    /*TODO: Fill in the synchronization stuff */
    sem_init(&semEmpty, 0, MAX_ITEMS);
    sem_init(&semFull, 0, 0);

    for (i = 0; i < THREADS; i++) {
        tid[i] = i;
        // Create producer thread
        pthread_create( & producers[i], NULL, producer, (void * ) tid[i]);

        // Create consumer thread
        pthread_create( & consumers[i], NULL, consumer, (void * ) tid[i]);
    }

    for (i = 0; i < THREADS; i++) {
        pthread_join(producers[i], NULL);
        pthread_join(consumers[i], NULL);
    }

    sem_destroy(&semEmpty);
    sem_destroy(&semFull);
    /*TODO: Fill in the synchronization stuff */

    return 0;
}

void put(int value) {
    sem_wait(&semEmpty);
    buffer[fill] = value; // line f1
    fill = (fill + 1) % MAX_ITEMS; // line f2
    sem_post(&semFull);
}

int get() {
    sem_wait(&semFull);
    int tmp = buffer[use]; // line g1
    use = (use + 1) % MAX_ITEMS; // line g2
    sem_post(&semEmpty);
    return tmp;
}

```

### III. Dining-Philosopher problem

#### 1. Requirement

- Analyze the output and figure out the in-correct execution. Enable the PROTECTION MECHANISM and compare the output.

- With the enabled code, the problem still falls into a deadlock state, refers the illustration. Use the provided material in the theory background section to explain the experimental phenomenon.

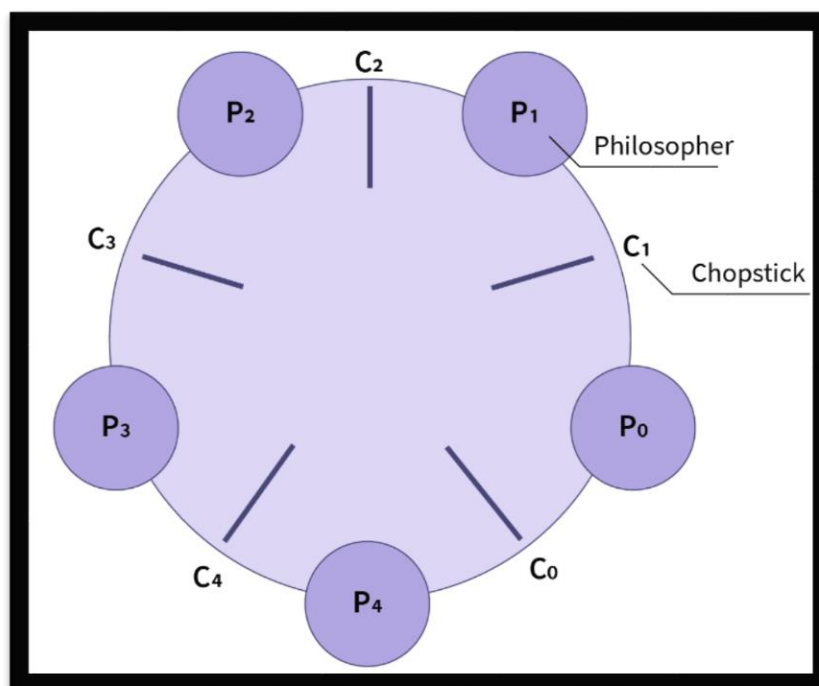
- Recall the experiment to manipulate a running process in the previous lab.

Analyze the status of the working process.

- Propose a solution to make it work.

#### 2. Steps to build

The dining philosopher's issue, also known as the classical synchronization matter, has five philosophers seated around a circular table who must alternate between thinking and eating. At the center of the table sits a bowl of noodles and five chopsticks, one for each of the philosophers. A philosopher requires both a right and a left chopstick in order to eat. Just the immediate left and right chopsticks of the philosopher are available for eating. The philosopher lays down their chopstick (either left or right) and resumes pondering if both of their immediate left and right chopsticks are not accessible.



Let's use P0, P1, P2, P3, and P4 as instances of the philosophers or processes, and C0, C1, C2, C3, and C4 as examples of the five chopsticks or resources that are placed between each philosopher. As P1 and P3 would be left without the resource and the process would not run if P2 wanted to consume, this shows that there are only so many resources (C0, C1, ...) available for use by many processes (P0, P1, ...).

The problem of the initial code (dead clock) when solving the Dining Philosopher issue:

```
thien@LAPTOP-HH2P921E:~$ nano p3_1.c
thien@LAPTOP-HH2P921E:~$ gcc -o p3_1 p3_1.c
thien@LAPTOP-HH2P921E:~$ ./p3_1
Philosopher 0 has entered room
Philosopher 3 has entered room
Philosopher 1 has entered room
Philosopher 4 has entered room
Philosopher 2 has entered room
```

There is a potential that all philosophers would eat at the same moment, which would entail using their left chopsticks first (assumedly) and right chopsticks immediately after. So there isn't a chopstick at their right side since it was seized by another philosopher, which causes everyone to wait indefinitely.

In this problem, a technique can use is mutex (a binary semaphore): using `mutex_clock()` and `mutex_unclock()` to solve the issue which exists deadlock.

### 3. *Explain each code paragraph*

Using `pthread_cond_wait()` and `pthread_cond_signal()` lead to the deadlock concern so that change to using the `pthread_mutex_clock()` and `pthread_unclock()` to solve the problem. The plan is for the odd index philosopher to take the left chopstick first, followed immediately by the right one. The even index philosopher, on the other hand, will use the right and left chopsticks in that order.

```

thien@LAPTOP-HH2P921E: ~  ×  +  ∨
GNU nano 6.2
void *philosopher(void *num)
{
    int phil = *(int*) num;
    printf("Philosopher %d has entered room\n", phil);
    sleep(1);
    while (1)
    {
        pthread_mutex_lock(&mtx); ←
        // Philosopher picks up the left chopstick (wait)
        pthread_mutex_lock(&chopstick[phil]); ←

        // Philosopher picks up the right chopstick (wait)
        pthread_mutex_lock(&chopstick[(phil + 1) % N]); ←

        //pthread_cond_wait(&chopstick[phil], &mtx);
        //pthread_cond_wait(&chopstick[(phil + 1) % N], &mtx);
        printf("Philosopher %d takes fork %d and %d\n",
                phil, phil, (phil + 1) % N);
        eat(phil);
        sleep(2);

        printf("Philosopher %d puts fork %d and %d down\n",
                phil, (phil + 1) % N, phil);
        //pthread_cond_signal(&chopstick[phil]);
        //pthread_cond_signal(&chopstick[(phil + 1) % N]);

        // Philosopher places down the left chopstick (signal)
        pthread_mutex_unlock(&chopstick[phil]); ←

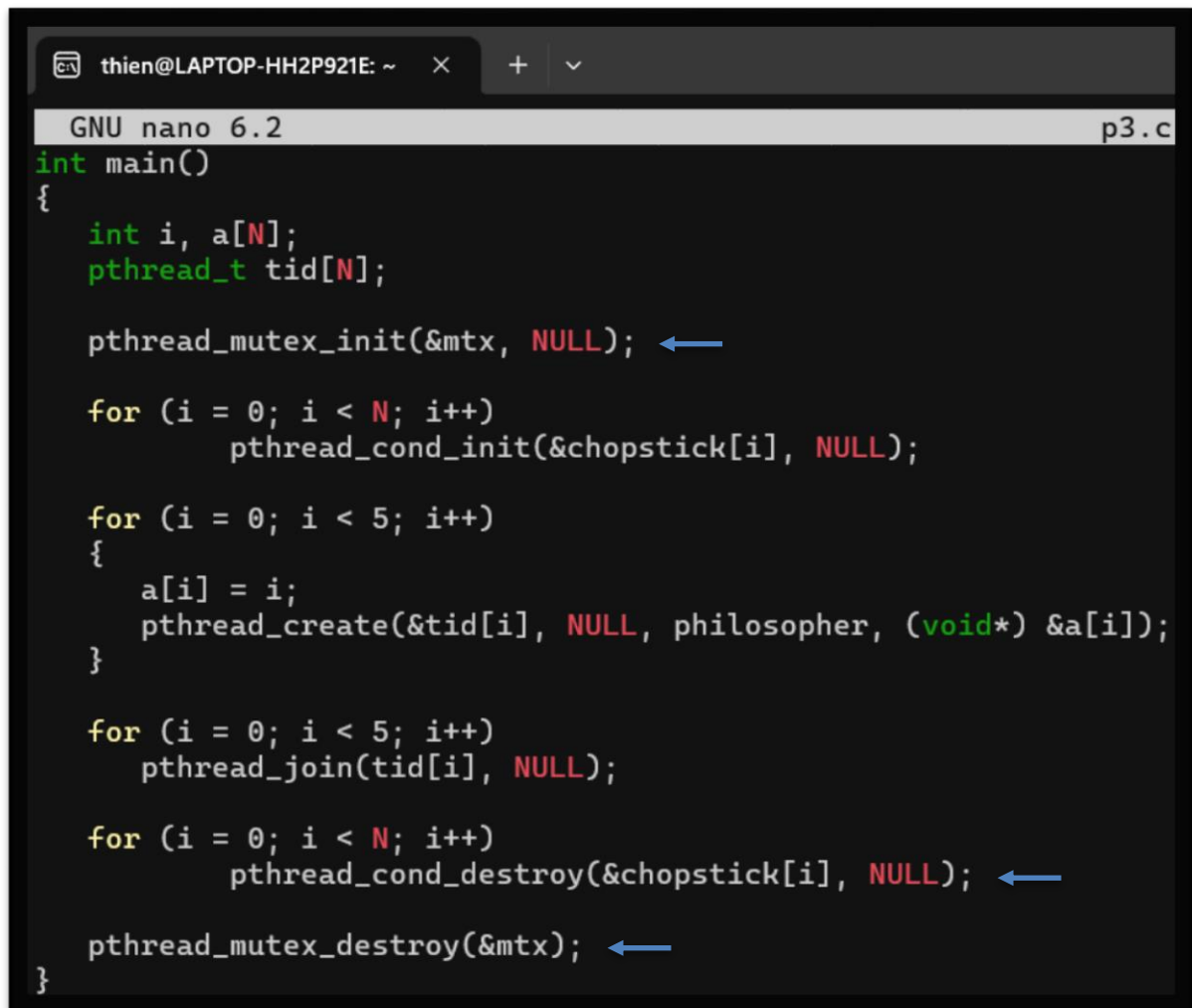
        // Philosopher places down the right chopstick (signal)
        pthread_mutex_unlock(&chopstick[(phil + 1) % N]); ←

        think(phil);
        sleep(1);
        pthread_mutex_unlock(&mtx); ←
    }
}

```



And also initializing the mutex at the beginning of the main function and clearing it, also destroying the pthread\_cond before closing the main function.



```
thien@LAPTOP-HH2P921E: ~  ×  +  ▾
GNU nano 6.2 p3.c
int main()
{
    int i, a[N];
    pthread_t tid[N];

    pthread_mutex_init(&mtx, NULL); ←
    for (i = 0; i < N; i++)
        pthread_cond_init(&chopstick[i], NULL);

    for (i = 0; i < 5; i++)
    {
        a[i] = i;
        pthread_create(&tid[i], NULL, philosopher, (void*) &a[i]);
    }

    for (i = 0; i < 5; i++)
        pthread_join(tid[i], NULL);

    for (i = 0; i < N; i++)
        pthread_cond_destroy(&chopstick[i], NULL); ←
    pthread_mutex_destroy(&mtx); ←
}
```

#### 4. *Implementing*

Result:



```
thien@LAPTOP-HH2P921E:~$ ./p3
Philosopher 0 has entered room
Philosopher 4 has entered room
Philosopher 3 has entered room
Philosopher 1 has entered room
Philosopher 2 has entered room
Philosopher 0 takes fork 0 and 1
Philosopher 0 is eating
Philosopher 0 puts fork 1 and 0 down
Philosopher 0 is thinking
Philosopher 4 takes fork 4 and 0
Philosopher 4 is eating
Philosopher 4 puts fork 0 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 3 and 4
Philosopher 3 is eating
Philosopher 3 puts fork 4 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 2 and 3
Philosopher 2 is eating
Philosopher 2 puts fork 3 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 1 and 2
Philosopher 1 is eating
Philosopher 1 puts fork 2 and 1 down
Philosopher 1 is thinking
Philosopher 0 takes fork 0 and 1
Philosopher 0 is eating
Philosopher 0 puts fork 1 and 0 down
Philosopher 0 is thinking
Philosopher 4 takes fork 4 and 0
Philosopher 4 is eating
Philosopher 4 puts fork 0 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 3 and 4
Philosopher 3 is eating
^C
thien@LAPTOP-HH2P921E:~$
```

The key question is that if the chopsticks are being used by another philosopher, this philosopher will have to wait till the other finishes.

## Source code of Dining-Philosopher problem

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

#define N 5

pthread_mutex_t mtx;
pthread_cond_t chopstick[N];

void *philosopher(void*);
void eat(int);
void think(int);
int main()
{
    int i, a[N];
    pthread_t tid[N];

    pthread_mutex_init(&mtx, NULL);

    for (i = 0; i < N; i++)
        pthread_cond_init(&chopstick[i], NULL);

    for (i = 0; i < 5; i++)
    {
        a[i] = i;
        pthread_create(&tid[i], NULL, philosopher, (void*) &a[i]);
    }

    for (i = 0; i < 5; i++)
        pthread_join(tid[i], NULL);

    for (i = 0; i < N; i++)
        pthread_cond_destroy(&chopstick[i], NULL);

    pthread_mutex_destroy(&mtx);
}

void *philosopher(void *num)
{
    int phil = *(int*) num;
    printf("Philosopher %d has entered room\n", phil);
    sleep(1);
    while (1)
    {
        pthread_mutex_lock(&mtx);
        // Philosopher picks up the left chopstick (wait)
        pthread_mutex_lock(&chopstick[phil]);
```

```

// Philosopher picks up the right chopstick (wait)
pthread_mutex_lock(&chopstick[(n + 1) % N]);

//pthread_cond_wait(&chopstick[phil], &mtx);
//pthread_cond_wait(&chopstick[(phil + 1) % N], &mtx);
printf("Philosopher %d takes fork %d and %d\n",
       phil, phil, (phil + 1) % N);

eat(phil);
sleep(2);

printf("Philosopher %d puts fork %d and %d down\n",
       phil, (phil + 1) % N, phil);
//pthread_cond_signal(&chopstick[phil]);
//pthread_cond_signal(&chopstick[(phil + 1) % N]);

// Philosopher places down the left chopstick (signal)
pthread_mutex_unlock(&chopstick[phil]);

// Philosopher places down the right chopstick (signal)
pthread_mutex_unlock(&chopstick[(n + 1) % N]);

think(phil);
sleep(1);
pthread_mutex_unlock(&mtx);
}
}

void eat(int phil)
{
    printf("Philosopher %d is eating\n", phil);
}

void think(int phil)
{
    printf("Philosopher %d is thinking\n", phil);
}

```

## IV. Aggregated sum

### 1. Requirement

- Implement the thread-safe program to calculate the sum of a given integer array using `< tnum >` number of threads. The size of the given array `< arrsz >` and the `< tnum >` value is provided in the program arguments. You are provided a pre-processed argument program with the usage.

- The last argument `< seednum >` is used in `srand()`, generating a fixed sequence of integer values. Call the following routine to generate a random array `buf` of integer with `arraysize` elements.

- Implementing a single-threaded program to compare the result of aggregated Sum with the one of the multi-threaded program.

### 2. Steps to build

To solve this issue, all that is required is the implementation of the `sum_worker` function and the usage of a mutex lock to prevent concurrent access by all threads to the total sum variable.

### 3. Explain each code paragraph

Using `pthread_mutex_lock(&mtx)` and `pthread_unlock(&mtx)` and adding the `thread_sum` to `global_sum` (`long sumbuf = 0, int* shrdarrbuf[i]: Global sum buffer`).

```
void* sum_worker(struct _range* idx_range) {
    int i;

    printf("In worker from %d to %d\n", idx_range->start, idx_range->end);

    // TODO: implement multi-thread sum-worker
    pthread_mutex_lock(&mtx); ←

    for(int i = idx_range->start; i <= idx_range->end; i++) ←
    {
        sumbuf += shrdarrbuf[i]; ←
    }

    pthread_mutex_unlock(&mtx); ←
    return 0;
}
```

## 4. Implementing

### Results:

```
thien@LAPTOP-HH2P921E:~$ ./4.2 100 5 5
number : 100 valid (and represents all characters read)
number : 5 valid (and represents all characters read)
number : 5 valid (and represents all characters read)
aggsu runs with <arrsz>=100 <tnum>=5 <seednum>=5

[0,19] [20,39] [40,59] [60,79] [80,99]

[ 76, 58, 36, 48, 86, 10, 56, 95, 21, 22, 49, 77, 47, 35, 2,
31, 21, 55, 54, 95, 43, 92, 71, 30, 93, 16, 71, 78, 12, 68,
32, 88, 92, 34, 35, 78, 44, 57, 72, 31, 80, 21, 7, 93, 56,
9, 39, 43, 30, 93, 37, 73, 50, 8, 2, 42, 91, 39, 20, 69,
73, 52, 56, 65, 52, 58, 42, 96, 14, 80, 93, 60, 0, 100, 18,
22, 75, 57, 31, 4, 49, 69, 77, 100, 43, 45, 7, 100, 85, 27,
58, 23, 45, 90, 88, 97, 47, 96, 58, 28, ]

sequence sum results 5353
In worker from 0 to 19
In worker from 20 to 39
In worker from 60 to 79
In worker from 80 to 99
In worker from 40 to 59
aggsu gives sum result 5353
thien@LAPTOP-HH2P921E:~$
```

Adding the library `<time.h>` to print the time taken to execute

```
thien@LAPTOP-HH2P921E:~$ ./4.2 100 1 5
number : 100 valid (and represents all characters read)
number : 1 valid (and represents all characters read)
number : 5 valid (and represents all characters read)
aggsu runs with <arrsz>=100 <tnum>=1 <seednum>=5

[0,99]

[ 76, 58, 36, 48, 86, 10, 56, 95, 21, 22, 49, 77, 47, 35, 2,
31, 21, 55, 54, 95, 43, 92, 71, 30, 93, 16, 71, 78, 12, 68,
32, 88, 92, 34, 35, 78, 44, 57, 72, 31, 80, 21, 7, 93, 56,
9, 39, 43, 30, 93, 37, 73, 50, 8, 2, 42, 91, 39, 20, 69,
73, 52, 56, 65, 52, 58, 42, 96, 14, 80, 93, 60, 0, 100, 18,
22, 75, 57, 31, 4, 49, 69, 77, 100, 43, 45, 7, 100, 85, 27,
58, 23, 45, 90, 88, 97, 47, 96, 58, 28, ]

sequence sum results 5353
In worker from 0 to 99
aggsu gives sum result 5353
Time taken to execute multi-thread in seconds : 0.000328
thien@LAPTOP-HH2P921E:~$
```

Single-thread

```
thien@LAPTOP-HH2P921E:~$ ./4.2 100 5 5
number : 100 valid (and represents all characters read)
number : 5 valid (and represents all characters read)
number : 5 valid (and represents all characters read)
aggsu runs with <arrsz>=100 <tnum>=5 <seednum>=5

[0,19] [20,39] [40,59] [60,79] [80,99]

[ 76, 58, 36, 48, 86, 10, 56, 95, 21, 22, 49, 77, 47, 35, 2,
81, 21, 55, 54, 95, 43, 92, 71, 30, 93, 16, 71, 78, 12, 68,
32, 88, 92, 34, 35, 78, 44, 57, 72, 31, 80, 21, 7, 93, 56,
9, 39, 43, 30, 93, 37, 73, 50, 8, 2, 42, 91, 39, 20, 69,
73, 52, 56, 65, 52, 58, 42, 96, 14, 80, 93, 60, 0, 100, 18,
22, 75, 57, 31, 4, 49, 69, 77, 100, 43, 45, 7, 100, 85, 27,
68, 23, 45, 90, 88, 97, 47, 96, 58, 28, ]

In worker from 0 to 19
sequence sum results 5353
In worker from 40 to 59
In worker from 80 to 99
In worker from 20 to 39
In worker from 60 to 79
aggsu gives sum result 5353
Time taken to execute multi-thread in seconds : 0.000743
thien@LAPTOP-HH2P921E:~$
```

Multi-thread

Due to the necessity to synchronize many threads of computation while dealing with a multi-threaded problem, the time consumption was significantly reduced when only one thread was employed. Other threads will have to wait until the current thread has finished updating the sumBuf variable each time, which will take a long time. The outcome, however, will be what we expected.

### Source code of Aggregated sum

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h> /* for getopt */

#include "utils.h"
#include <errno.h>
#include <limits.h>
#include <pthread.h>
#include <sys/wait.h>

/** process command line argument.
 * values are made available through the 'conf' struct.
 * using the parsed conf to get arguments: the arrsz, tnum, and seednum
 */
extern int processopts (int argc, char **argv, struct _appconf *conf);

/** process string to number.
 * string is stored in 'nptr' char array.
 * 'num' is returned the valid integer number.
 * return 0 valid number stored in num
 *      -1 invalid and num is useless value.
 */
extern int tonum (const char * nptr, int * num);

/** validate the array size argument.
 * the size must be splitable "num_thread".
 */
extern int validate_and_split_argarray (int arraysize, int num_thread, struct
_range* thread_idx_range);

/** generate "arraysize" data for the array "buf"
 * validate the array size argument.
 * the size must be splitable "num_thread".
 */
extern int generate_array_data (int* buf, int arraysize, int seednum);

/** display help */
extern void help (int xcode);
```

```

void* sum_worker (struct _range* idx_range);
long validate_sum(int arraysize);

/* Global sum buffer */
long sumbuf = 0;
int* shrdarrbuf;

pthread_mutex_t mtx;

void* sum_worker(struct _range* idx_range) {
    int i;

    printf("In worker from %d to %d\n", idx_range->start, idx_range->end);

    // TODO: implement multi-thread sum-worker
    pthread_mutex_lock(&mtx);

    for(int i = idx_range->start; i <= idx_range->end; i++)
    {
        sumbuf += shrdarrbuf[i];
    }

    pthread_mutex_unlock(&mtx);
    return 0;
}

int main(int argc, char * argv[]) {
    clock_t start, end;
    double execution_time;
    /* Store start time here */
    start = clock();

    /* Put the main body of your program here */

    int i, arrsz, tnum, seednum;
    char *buf;
    struct _range* thread_idx_range;
    pthread_t* tid;
    int pid;

    if (argc < 3 || argc > 4) /* only accept 2 or 3 arguments */
        help(EXIT_SUCCESS);

#ifdef DBGSTDERR == 1
    freopen("/dev/null", "w", stderr); /* redirect stderr by default */
#endif

    processopts(argc, argv, &appconf); /* process all option and argument */

    fprintf(stdout, "%s runs with %s=%d \t %s=%d \t %s=%d\n", PACKAGE,

```

```

        ARG1, appconf.arsz, ARG2, appconf.tnum, ARG3,
appconf.seednum);

thread_idx_range = malloc(appconf.tnum * sizeof(struct _range));
if(thread_idx_range == NULL)
{
    printf("Error! memory for index storage not allocated.\n");
    exit(-1);
}

if (validate_and_split_argarray(appconf.arsz, appconf.tnum,
thread_idx_range) < 0)
{
    printf("Error! array index not splitable. Each partition need at least
%d item\n", THRSL_MIN);
    exit(-1);
}

/* Generate array data */
shrdarrbuf = malloc(appconf.arsz * sizeof(int));
if(shrdarrbuf == NULL)
{
    printf("Error! memory for array buffer not allocated.\n");
    exit(-1);
}

if(generate_array_data(shrdarrbuf, appconf.arsz, appconf.seednum) < 0)
{
    printf("Error! array index not splitable.\n");
    exit(-1);
}

pid=fork();

if(pid < 0)
{
    printf("Error! fork failed.\n");
    exit(-1);
}

if(pid == 0) { /* child process do a validation sum */
    printf("sequence sum results %ld\n",validate_sum(appconf.arsz));
    exit(0);
} // parent process goes here

/** Create <tnum> thead to calculate partial non-volatile sum
 * the non-volatile mechanism require value added to global sum buffer
 */
tid = malloc (appconf.tnum * sizeof(pthread_t));
pthread_mutex_init(&mtx, NULL);

```



```

    for (i=0; i < appconf.tnum; i++)
        pthread_create(&tid[i], NULL, sum_worker, (
            struct _range *) (&thread_idx_range[i]));
    for (i=0; i < appconf.tnum; i++)
        pthread_join(tid[i], NULL);
    fflush(stdout);

    printf("%s gives sum result %ld\n", PACKAGE, sumbuf);

    /* Program logic ends here */
    end = clock();
    /* Get the time taken by program to execute in seconds */
    execution_time = ((double)(end - start))/CLOCKS_PER_SEC;

    printf("Time taken to execute multi-thread in seconds : %f\n",
execution_time);

    waitpid(pid, NULL, NULL);
    exit(0);
}

long validate_sum(int arraysize)
{
    long validsum = 0;
    int i;

    for (i=0; i < arraysize; i++)
        validsum += shrdarrbuf[i];

    return validsum;
}

```