TRƯỜNG ĐẠI HỌC BÁCH KHOA – ĐẠI HỌC QUỐC GIA TP.HCM
TRUNG TÂM KỸ THUẬT ĐIỆN TOÁN
268 Lý Thường Kiệt, Phường 14, Quận 10, TP.Hồ Chí Minh
Điện thoại: 84-8-3864 7256 ext 5371 – Fax: 84-8-3865 8687
Website: **www.cce.hcmut.edu.vn**
E-mail: **dientoan@hcmut.edu.vn**

Lab 7 -
Synchronization and Deadlock

Goal
This lab helps student to practice with the synchronization in OS, and understand the reason why we need the synchronization.

Contents
In detail, this lab requires student practice with examples using synchronization techniques to solve the problem called **race condition**. The synchronization is performed on Thread using the following mechanism:

- mutual exclusion (mutex)
- semaphore
- conditional variable

Besides, the practices also introduces and includes some locking variants, i.e. spinlock, read/write spinlock, sequence lock. In addition to using lock, we may reach a **deadlock** state. This lab also covers the experiments and provides practical solutions to deal with a deadlock.

Result
After doing this lab, student can understand the definition of synchronization and write a program which allows to solve the race condition using the techniques above.

Requirements
Student need to review the theory of synchronization

.

1 Background

1.1 Race condition

Race condition is the condition of software where the system'substain behavior is dependent on the sequences of uncontrollable events. Therefore, we need mechanisms that provide mutual exclusion ability.

1.1.1 Race condition caused by atomicity

In high level programming language, we can note that a statement may be implemented in machine language (on a typical machine) as follows:

```
instruction1:= register load
instruction2:= arithmetic operation
instruction3:= register store
```

If there are two or more threads access to a single storage, the data manipulation which is splitable may result an incorrect final state. Such tools provided by POSIX pthread as following:
Mutex Lock The problem with mutex is that the thread is put to sleep state and later is woken to process the task. This sleeping and waking action are expensive operations.
Spin lock In a shorts description, spin lock provides a (CPU) poll waiting until it has got privilege. If the mutex sleep in a very short time then it waste the cost of expensive operations of sleeping and waking up. But if the long time is quite long, CPU polling is a big waste of computation.

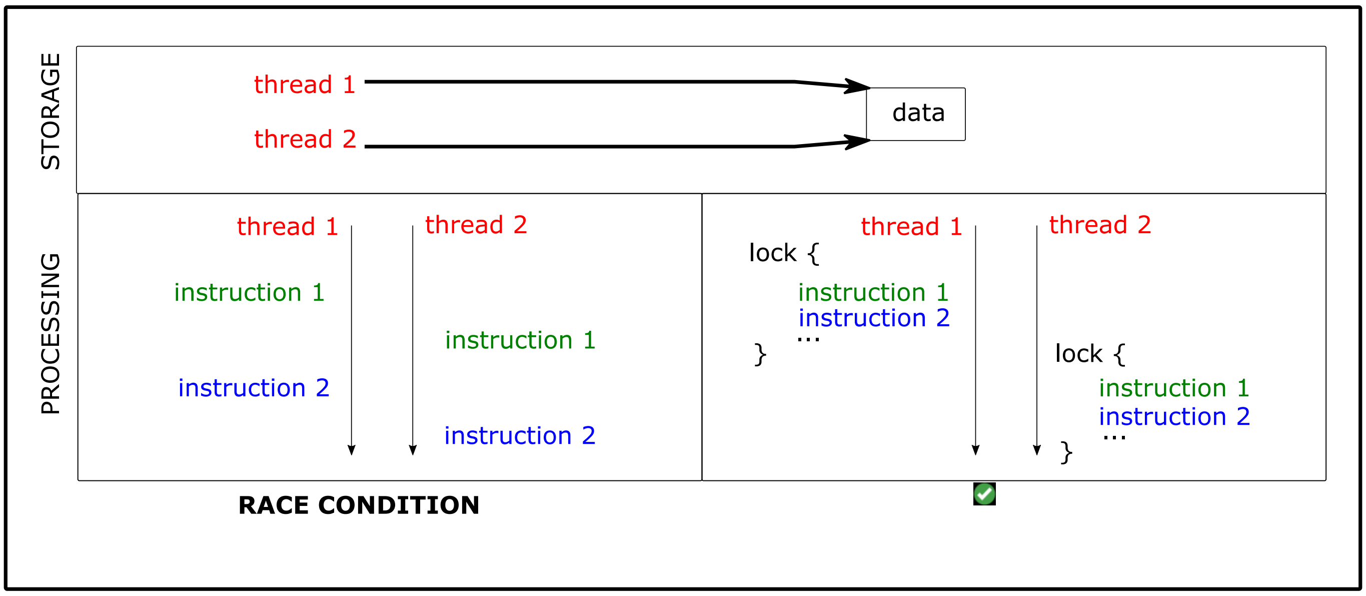

Figure 1The race condition caused by atomicity

1.1.2 Race condition caused by in-correct ordering

In previous chapter of process shared memory, we have an introduction of bounded buffer problem. Event with mutex setting, the race conditional may still happen when a producer fill in a buffer of item before a consumer empties it causing a losing data. The same wrong system behaviors happens when a consumer retrieves a garbage value if the buffer of item is accessed before a producer fill in a meaningful value. These wrong system behaviors caused by the in-correct ordering of the sequence of event.

1.1.2.1 Semaphore

A semaphore is suited cleanly to a producer-consumer model. Semaphore is basically an object that include an internal counter, waiting list of threads and support two different operations i.e., wait and signal. The internal counter with a proper initialization provides a good mechanism to manage the limited number of storage slot in bounded buffer. Some updated model such as few writer many reader is not a good application for semaphore. It relaxes the matching of internal counter with the number of slot then it helps in the unbalancing context between actors who access the buffer.

Although semaphore is perfect fit for bounded buffer problem, it constrains on the equal number of consumers(/readers) and producers(/writer). These following tool provide the ordering mechanism with more relaxation on the number of system actors:

Conditional variable Conditional variable is a synchronization primitive that allows threads to wait until particular condition occur. It just allows a thread to be signaled when something of interested to that thread occurs. It includes two operations wait and signal. The conditional_variable can be used by a thread to block other threads until it notifies the conditional_variable.

1.1.2.2 Read-write spin lock and sequence lock

In previous section, we have seen some locking method like mutex, spinlock, etc. In high speed manner, i.e. kernel driver, high speed/fast communication, when you want to treat both reader and writer equally, then you have to use spin lock. The two following mechanisms provide a different priority policy between reader and writer. We introduce the main characteristic of them and then, we discuss the reader/writer conflict problem later as an exercise flavor. (See more details in Section 4)
Read-write spinlock: In some situation, we may have to give more access frequencies to reader. In such case, the reader-writer spinlock is a suitable solution.
Sequence lock: Reader-writer lock can cause writer starvation. Seqlock give more permission to writer. Sequential lock is a reader-writer mechanism which is given high priority to writer, so this avoids writer starvation problem.

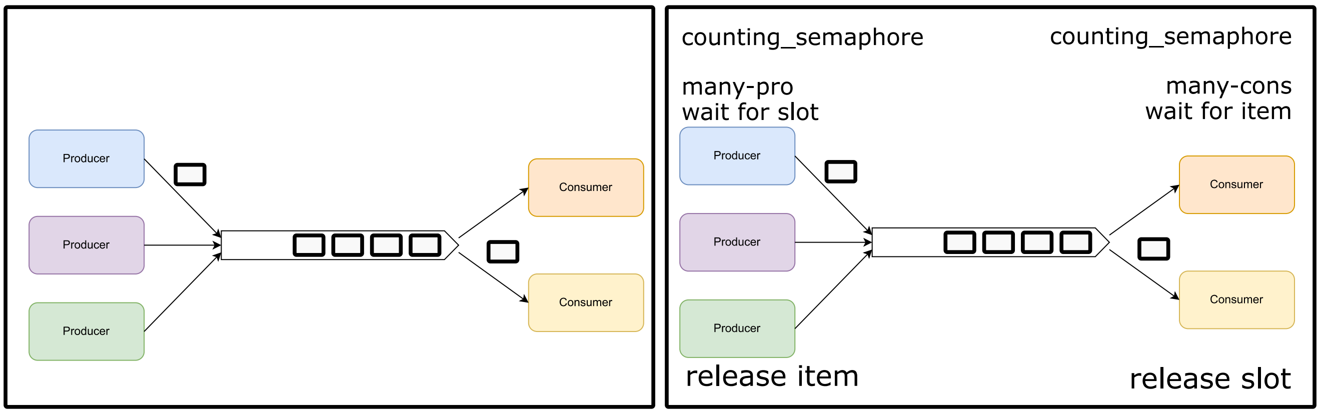

Figure 2 Bounded-Buffer or Producer-Consumer problem

1.2 Deadlock

1.2.1 Dining-Philosophers problem

In short, the problem includes some **philosophers** spend their lives alternating thinking and eating. They don't interact with their neighbors. Occasionally they try to pick up **2 chopsticks** (one at a time) to ***eat***, then they **release** both when done.

```
void philosopher(...){
  while(1)
  {
    /*Philosopher [TAKE] chopstick LEFT and RIGHT */
    wait_to_take_2_chopstick();
    eat();

    /*Philosopher [RELEASE] chopstick LEFT and RIGHT */
    release_2_chopstick();
    think();
  }
}
```

1.2.2 Deadlock

Deadlock can arise if four conditions hold simultaneously

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

In Figure 2, a deadlock state can form based on a circular waiting on chopstick. This state is reached when all philosopher fall into a loop waiting during the wait_to_take_2_chopstick() call.

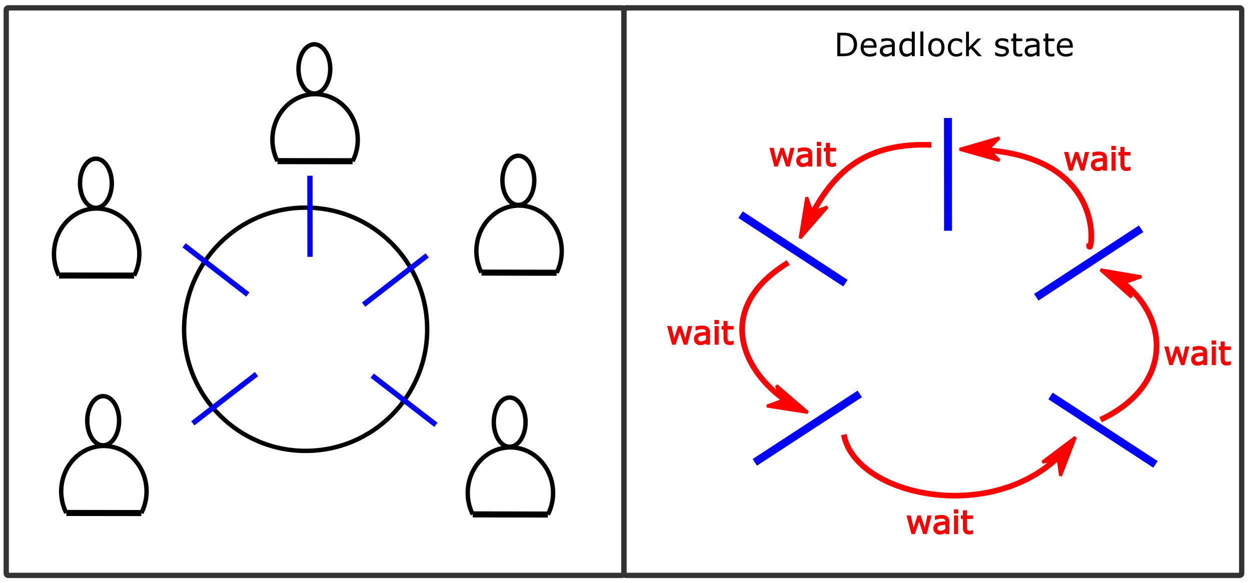

Figure 3 The Dining Philosophers problem and a deadlock state

2 Programming Interface of synchronization tools

POSIX Thread library provided a standard based thread API for C/C++. The function and declaration of many common API in this library conform to POSIX.1 standards through many version (2003, 2017 etc.). In Linux, the library is not integrated by default then it requires an explicit linking declaration with "-pthread". Sequence lock is not implemented in the both library but is added to kernel since Linux 2.5.x.

2.1 Mutex lock

provided in POSIX Thread (pthread) library

```
#include <pthread.h>
pthread_mutex_t lock;

int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
_______________________________________________________________________
int pthread_mutex_lock(pthread_mutex_t *mutex );
int pthread_mutex_unlock(pthread_mutex_t *mutex );
```

Example:

```
pthread_mutex_t lock;
pthread_mutex_init(&lock,NULL);
...
pthread_mutex_lock(&lock);
<CS>

pthread_mutex_unlock(&lock);
<RS>
```

2.2 Spin lock

provided in POSIX Thread (pthread) library

```
#include <pthread.h>

pthread_spinlock_t lock;
int pshare; /* PTHREAD_PROCESS_SHARED
             * or PTHREAD_PROCESS_PRIVATE (creator only)
             */

int pthread_spin_init(pthread_spinlock_t * lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t * lock);
_______________________________________________________________________
int pthread_spin_lock(pthread_spinlock_t * lock);
int pthread_spin_unlock(pthread_spinlock_t * lock);
```

Example:
```
pthread_spinlock_t lock;
pthread_spin_init(&lock,PTHREAD_PROCESS_SHARED);
...
pthread_spin_lock(&lock);
<CS>
pthread_spin_unlock(&lock);
<RS>
```

2.3 Semaphore

provided in POSIX semaphore (not PTHREAD)

```
#include <semaphore.h>
sem_t sem;

int sem_init(sem_t *sem, int pshared, unsigned int value)
int sem_destroy(sem_t *sem);
_______________________________________________________________________
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

Example
```
sem_t sem;
sem_init(&sem,0,5); /* we can use pshared=0
                     for NULL setting or PTHREAD_PROCESS_SHARED */
...
sem_wait(&sem);
<CS>
sem_post(&sem);
<RS>
```

2.4 Conditional Variable

provided in POSIX Thread (pthread) library

```
#include <pthread.h>
pthread_cond_t cv_count;

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
________________________________________________________________________
int pthread_cond_wait (pthread_cond_t *cond , pthread_mutex_t *mutex ) ;
int pthread_cond_signal (pthread_cond_t *cond )
```

Example:
```
pthread_mutex_t mtx;
pthread_cond_t lock;
pthread_mutex_init(&lock,NULL);
```

```
pthread_cond_init(&lock,NULL);
...
pthread_cond_wait(&lock,&mtx);
<CS>
pthread_cond_signal(&lock);
<RS>
```

2.5 Reader-writer lock

provided in POSIX Thread (pthread) library

```
#include <pthread.h>

pthread_rwlock_t lock;

int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t
*attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
______________________________________________________________________________
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Example:

```
pthread_rwlock_t lock;
pthread_rwlock_init(&lock,NULL);
...
pthread_rwlock_rdlock(&lock);
<CS>
pthread_rwlock_unlock(&lock);
<RS>
...
pthread_rwlock_rdlock(&lock);
<CS>
pthread_rwlock_wrunlock(&lock);
<RS>
```

2.6 Sequence lock

have not provided in POSIX Thread library yet. Its implementation existed in kernel (and hence, can be used only in kernel space) since 2.5.60.
Although it lacks of userspace implementation, it is widely used in kernel to protect buffer data in modern programming patterns with many reader, many writer, .i.e. SMP Linux kernel support.

3 Practices

In this section, we work on various “native” problem to recognize the “real” wrong behaviors. All these experiments are derived from theory slides with a minor modification. We practice the provided synchronization mechanisms and see how they work to correct the wrong things.

3.1 Shared buffer problem

Implement the following program.

```
#include <iostream>
#include <pthread.h>

using namespace std;

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

void *f_count(void *_id) {
   int i;
   int id = *(int *)(_id);

   for (i = 0; i < MAX_COUNT; i++) {
         count = count + 1;
   }

   cout << "Thread " << id << ": holding " << count << "\n";
 }

int main() {
  pthread_t thread1, thread2;
  int id1, id2;

  id1=1;
  id2=2;

  /* Create independent threads each of which will execute function */
  pthread_create( &thread1, NULL, &f_count, (void *) &id1);
  pthread_create( &thread2, NULL, &f_count, (void *) &id2);

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

  // Wait for thread th1 finish
  pthread_join( thread2, NULL);

  return 0;
}
```

Step 3.1.1 Compile and execute the program

```
# The program name must be matched your own code,
# copycat may result error gcc: fatal error: no input files
$  g++ -pthread -o mutex_cpp mutex_cpp.cpp
$ ./mutex_cpp
Thread 1: holding 1996747312
Thread 2: holding 1998250555
```

Step 3.1.2 Recognize the wrong issue and proposed a fix mechanism using the provided synchronization tool.

Step 3.1.2 Recognize the wrong issue and proposed a fix mechanism using the provided synchronization tool.

Hint: pthread_mutex_lock() and pthread_mutex_unlock() are useful to protect f_count() thread worker

```
# Implement by your self the fixed pc_sem program (it is not available yet)
# copycat may result error gcc: fatal error: no input files
$  g++ -pthread -o mutex_cpp mutex_cpp.cpp
$ ./mutex_cpp
Thread 2: holding 1002354829
Thread 1: holding 2000000000
```

3.2 Bounded buffer problem

Implement the following program.

```
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

using namespace std;

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

// Initiate shared buffer
int buffer[MAX_ITEMS];
int fillin = 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 */
    put(i); // line P2
    std::cout << "Producer " << tid << " put data " << i <<" \n";
    sleep(1);
    /*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 */
    tmp = get(); // line C2
    std::cout << "Consumer " << tid << " get data " << tmp << "\n";
    sleep(1);
    /*TODO: Fill in the synchronization stuff */
  }
  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 */

  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);
  }

  /*TODO: Fill in the synchronization stuff */

  return 0;
}
```

```
void put(int value) {
  buffer[fillin] = value; // line f1
  fillin = (fillin + 1) % MAX_ITEMS; // line f2
}

int get() {
  int tmp = buffer[use]; // line g1
  use = (use + 1) % MAX_ITEMS; // line g2
  return tmp;
}
```

Step 3.2.1 Compile and execute the program

```
# The program name must be matched your own code
$ g++ -pthread -o pc_cpp pc_cpp.cpp
$ ./pc_cpp
Consumer 0 get data 0
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
...
```

Step 3.2.2 Recognize the wrong issue and proposed a fix mechanism using the provided synchronization tool.
Hint: sem_wait() and sem_signal() are useful to protect consumer() and producer() thread worker

```
# Implement by your self the fixed pc_sem program (it is not available yet)
# copycat may result error gcc: fatal error: no input files
$ g++ -pthread -o pc_cpp pc_cpp.cpp
$ ./pc_cpp
Producer 0 put data 0
Consumer 0 get data 0
Producer 0 put data 1
Consumer 0 get data 1
Producer 0 put data 2
Consumer 0 get data 2
Producer 0 put data 3
Consumer 0 get data 3
...
```

3.3 Dining-Philosopher problem

Implement the following program.

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>
```

```
using namespace std;

#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];

   /* BEGIN PROTECTION MECHANISM */
   //pthread_mutex_init(&mtx, NULL);

   //for (i = 0; i < N; i++)
           //pthread_cond_init(&chopstick[i], NULL);
   /* END PROTECTION MECHANISM */

   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);
}

void *philosopher(void *num)
{
   int phil = *(int*) num;
   std::cout << "Philosopher " << phil << " has entered room\n";

   while (1)
   {
      std::cout << "Philosopher " << phil << " takes fork " << phil
                << " and " << (phil + 1) % N<< " up\n";
      /* PROTECTION MECHANISM */
      //pthread_cond_wait(&chopstick[phil], &mtx);
      //pthread_cond_wait(&chopstick[(phil + 1) % N], &mtx);

      eat(phil);
      sleep(2);

       /* PROTECTION MECHANISM */
      //pthread_cond_signal(&chopstick[phil]);
```

```
        //pthread_cond_signal(&chopstick[(phil + 1) % N]);
        std::cout << "Philosopher " << phil << " puts fork " << phil
                  << " and " << (phil + 1) % N << " down\n";

            think(phil);
            sleep(1);
    }
}

void eat(int phil)
{
    std::cout << "Philosopher " << phil << " is eating\n";
}

void think(int phil)
{
    std::cout<< "Philosopher " << phil << " is thinking\n";
}
```

Step 3.3.1 Compile and execute the program

```
# The program name must be matched your own code,
$ g++ -pthread -o din dinPhl.cpp
$ ./din
Philosopher 4 has entered room
Philosopher 4 takes fork 4 and 0 up
Philosopher 4 is eating
Philosopher 3 has entered room
Philosopher 3 takes fork 3 and 4 up
Philosopher 3 is eating
Philosopher 2 has entered room
Philosopher 2 takes fork 2 and 3 up
Philosopher 2 is eating
Philosopher 1 has entered room
Philosopher 1 takes fork 1 and 2 up
Philosopher 1 is eating
Philosopher 0 has entered room
Philosopher 0 takes fork 0 and 1 up
Philosopher 0 is eating
Philosopher 4 puts fork 4 and 0 down
Philosopher 4 is thinking
Philosopher 3 puts fork 3 and 4 down
Philosopher 3 is thinking
Philosopher 2 puts fork 2 and 3 down
Philosopher 2 is thinking
Philosopher 1 puts fork 1 and 2 down
Philosopher 1 is thinking
Philosopher 0 puts fork 0 and 1 down
Philosopher 0 is thinking
Philosopher 4 takes fork 4 and 0 up
Philosopher 4 is eating
Philosopher 3 takes fork 3 and 4 up
Philosopher 3 is eating
Philosopher 2 takes fork 2 and 3 up
Philosopher 2 is eating
...
```

Step 3.3.2 Analysis the output and figure out the in-correct execution. Enable the PROTECTION MECHANISM and compare the output.

```
# The program name must be matched your own code,
$ g++ -pthread -o din dinPhl.cpp
$ ./din
Philosopher 4 has entered room
Philosopher 4 takes fork 4 and 0 up
Philosopher 3 has entered room
Philosopher 3 takes fork 3 and 4 up
Philosopher 2 has entered room
Philosopher 2 takes fork 2 and 3 up
Philosopher 1 has entered room
Philosopher 1 takes fork 1 and 2 up
Philosopher 0 has entered room
Philosopher 0 takes fork 0 and 1 up
...
```

Step 3.3.3 With the enabled code, the problem still falls into deadlock state, refer the illustration in Figure 3. Use the provided material in theory background section to explain the experimental phenomenon Recall the experiment to manipulate a running process in previous lab. Analysis the status of the working process.

```
$ ps aux  | grep din
oslab     10532  0.0  0.0  47492   800 pts/4    Sl+  13:42   0:00 ./din
oslab     10577  0.0  0.2  11760  2144 pts/0    S+   13:42   0:00 grep --
$ sudo cat /proc/<PID>/status
Name:   din
State:  S (sleeping)
...
$ sudo cat /proc/<PID>/stack
[<ffffffff811011a4>] futex_wait_queue_me+0xc4/0x120
[<ffffffff811020eb>] futex_wait+0x17b/0x270
[<ffffffff811039b6>] do_futex+0xe6/0xbc0
[<ffffffff81104501>] SyS_futex+0x71/0x150
[<ffffffff8182bedb>] entry_SYSCALL_64_fastpath+0x22/0xcb
[<ffffffffffffffff>] 0xffffffffffffffff
```

From kernel.org

```
futex_wait_queue_me:  queue_me and wait for wakeup, timeout, or signal
```

Step 3.3.4 Propose a solution to make it work.

Hint: pthread_cond_signal() is helpful here to provide the trigger to start the whole program work. Try at your own risk

```
# Implement by your self the fixed din program (it is not available yet)
# copycat may result error gcc: fatal error: no input files
$ ./din
Philosopher 4 has entered room
Philosopher 3 has entered room
Philosopher 2 has entered room
Philosopher 1 has entered room
Philosopher 0 has entered room
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
```

Step 3.3.5 Advance step It completely wrong from the scratch with an in-appropriate (or garbage) synchronization mechanism. The last working execution still contains fatal risk. Propose/design and implement an alternative protection mechanism.

Hint: Through this experiment, learn by yourself the important of using the correct synchronization mechanism at the beginning otherwise, we fix something and yield another work of fixing the workaround mechanism. It is important to equip a deep knowledge of synchronization in tackling a real problem.

4 Exercise

PROBLEM 1

Design and implement sequence lock API.

```
#include <seqlock.h> /* TODO: implement this header file */

/*
 *  TODO: Implement these following APIs
 */
pthread_seqlock_t lock;

/* Init with default NULL attribute attr=NULL */
int pthread_seqlock_init(pthread_seqlock_t *seqlock);
int pthread_rwlock_destroy(pthread_seqlock_t *seqlock);
___________________________________________________________________________
int pthread_seqlock_rdlock(pthread_seqlock_t *seqlock);
int pthread_seqlock_rdunlock(pthread_seqlock_t *seqlock);
int pthread_seqlock_wrlock(pthread_seqlock_t *seqlock);
int pthread_seqlock_wrunlock(pthread_seqlock_t *seqlock);
```

The reader/writer conflict resolution following the description:

Reader-writer lock conflict resolution

- When there is no thread in the critical section, any reader or writer can enter into a critical section. But only one thread can enter.
- If reader in critical section, the new reader thread can enter ocassionally, but the writer cannot enter.
- If write in critical section, no other reader or writer can enter.
- If there are some readers in critical section by taking the lock and there is a writer want to enter. That writer has to wait if another reader is coming until all of readers have finish. That why this mechanism is reader preference.

Sequence lock conflict resolution the conflict resolution mechanism in reader-writer lock can cause writer starvation. Sequence lock implement the following policy:

- When no one is at critical section, one writer can enter into critical section and took the lock the sequence number by one to an odd value. When the sequence number is an odd value, the writing is happening. When writing has been done, the sequence is back to even value. Only one writer is allow into critical section.
- When the reader wants to read data, it checks the sequence number which is an odd value, it has to wait until the writer finish.
- When the value is even, many reader can enter the critical section to read the value.
- When there are only reader and no writer in the critical section, a writer want to enter the critical section it can take the lock without blocking.

PROBLEM 2

(***Concurrent PI calculator***) In the previous Lab, we wrote a simple multi-thread program for calculating the value of pi using Monte-Carlo method. In this exercise, we also calculate pi using the same method but with a different implementation. We create a shared (global) count variable and let worker threads update on this variable in each of their iteration instead of on their own local count variable. To make sure the result is correct, remember to avoid race conditions on updates to the shared global variable by using **mutex locks** and **semaphore** (one version of program for each method). Compare the performance of this approach with the previous one in the previous Lab solution.

PROBLEM 3

(***Aggregated Sum***) Implement the thread-safe program to calculate the sum of a given integer array using $< tnum >$ number of thread. The size of the given array $< arrsz >$ and the $< tnum >$ value is provided in program arguments. You are provided a pre-processed argument program interface as the following description.
Design and implement sequence lock API.

```
aggsum, version 0.01

  usage:  aggsum arrsz tnum [seednum]

  Generate randomly integer array size <arrsz> and calculate sum parallelly
  using <tnum> threads. The optional <seednum> value use to control the
  randomization of the generated array.

    Arguments:

      arrsz    specifies the size of array.
      tnum     number of parallel threads.
      seednum  initialize the state of the randomized generator.
```

The last argument $< seednum >$ is an already implemented mechanism. This value is used to generate the integer values in the array and we don't touch it to keep for later validated testcase generating. The data generation mechanism is also provided to fill in the "buf" shared memory buffer.

```
int generate_array_data (int* buf, int arraysize, int seednum);
```

You have to implement the following thread routines.

```
struct _range {
  int start;
  int end;
};

void* sum_worker (struct _range idx_range) {
  //printf("In worker from %d to %d\n", idx_range.start, idx_range.end);

  /*
   * TODO implement a thread safe sum operator works in the range
   *       i.e. for (i=idx_range.start; i<= idx_range.end; i++){}
   *      and write the sum to sumbuff  (in program global data)
   */
}
```

```
int main()
{
   pthread_t tid; /* Sample code is only single thread */
   struct _range thread_idx_range;

   ...
   /* Sample code use full range */
   thread_idx_range.start = 0;
   thread_idx_range.end = arrsz - 1;

   ...
   /* TODO: implement multi-thread mechanism */
   pthread_create(&tid, NULL, sum_worker, thread_idx_range));
}
```

The reader/writer conflict resolution following the description: