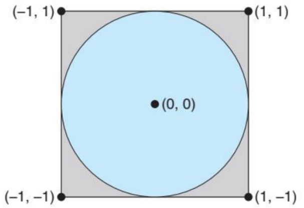
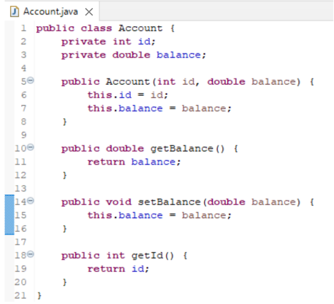
Lab 2 **Process and Thread Management**	**Assignment 1**	Develop a program which has a shared data and two threads. The shared data contains an integer variable which is set to 0 initially. The first thread will increase the variable a number of times (1000000 times) and the second thread will decrease the variable the same number of times (1000000 times).
		a. The program creates two threads and starts the first thread. The program waits for the first thread to terminate and then start the second thread. The program waits for the second thread to terminate and display the variable’s value. Explain the result.
		b. The program creates, starts two threads and waits for the two threads to terminate and display the variable’s value. Explain the result.
	Assignment 2	Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the **average of the numbers**, the second will determine **the maximum value**, and the third will determine the **minimum value**. For example, suppose your program is passed the integers
		90 81 78 95 79 72 85
		The program will report The average value is 82 The minimum value is 72 The maximum value is 95
		The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and **the parent thread will output the values once the workers have exited**. (We could obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)
	Assignment 3	Write a multithreaded program that outputs **prime numbers**. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.
	Assignment 4	An interesting way of calculating π is to use a technique known as Monte Carlo, which involves randomization. This technique works as follows:
		Suppose you have a circle inscribed within a square, as shown in Figure 4.25. (Assume that the radius of this circle is 1.) First, generate a series of random points as simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle. Next, estimate π by performing the following calculation: **π = 4×** (number of points in circle) / (total number of points)
		Write a multithreaded version of this algorithm that creates a separate thread to generate a number of random points. The thread will count the number of points that occur within the circle and store that result in a global variable. When this thread has exited, **the parent thread will calculate and output the estimated value of π**. It is worth experimenting with the number of random points generated. As a general rule, the greater the number of points, the closer the approximation to π.
		 **Figure 4.25** Monte Carlo technique for calculating π.
		The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as: **fib0 = 0** **fib1 = 1** **fibn = fibn−1 + fibn−2**

	Assignment 5	Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, **the parent thread will output the sequence generated by the child thread**. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish.
Lab 3 Scheduling Algorithms	This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:	
	First-come, first-served (FCFS)	which schedules tasks in the order in which they request the CPU.
	Shortest-job-first (SJF)	which schedules tasks in order of the length of the tasks' next CPU burst.
	Priority scheduling	which schedules tasks based on priority.
	Round-robin (RR) scheduling	where each task is run for a time quantum (or for the remainder of its CPU burst).
	Priority with round-robin	which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority.
	Priorities range from 1 to 10, where a lower numeric value indicates a higher relative priority. For round-robin scheduling, the length of a time quantum is 10 milliseconds. The schedule of tasks has the form [task name], [priority], [CPU burst], with the following example format: T1, 4, 20 T2, 2, 25 T3, 3, 25 T4, 3, 15 T5, 10, 10 Thus, task T1 has priority 4 and a CPU burst of 20 milliseconds, and so forth. It is assumed that all tasks arrive at the same time, so your scheduler algorithms do not have to support higher-priority processes preempting processes with lower priorities.	
	File Schedule.txt: T1, 4, 20 T2, 3, 25 T3, 3, 25 T4, 5, 15 T5, 5, 20 T6, 1, 10 T7, 3, 30 T8, 10, 25 Algorithm: FCFS 0 : T1 - Pri: 4 - Burst: 20 - Duration: 20 20 : T2 - Pri: 3 - Burst: 25 - Duration: 25 45 : T3 - Pri: 3 - Burst: 25 - Duration: 25 70 : T4 - Pri: 5 - Burst: 15 - Duration: 15 85 : T5 - Pri: 5 - Burst: 20 - Duration: 20 105 : T6 - Pri: 1 - Burst: 10 - Duration: 10 115 : T7 - Pri: 3 - Burst: 30 - Duration: 30 145 : T8 - Pri: 10 - Burst: 25 - Duration: 25	
		Develop a program which has a shared data and two threads. The shared data contains an integer variable which is set to 0 initially. The first thread will increase the variable a number of times (1000000 times) and the second thread will decrease the variable the same number of times (1000000 times). You should protect critical sections and thus prevent race conditions.

Lab 4.1 Thread Synchronization 1	**Assignment 1**	a. Using synchronized methods b. Using ReentrantLock class c. Using Semaphore class
		The program creates, starts two threads and waits for the two threads to terminate and display the variable's value. Explain the result.
	Assignment 2	Develop **a thread-safe** queue of integer **(TSQueue)**. The thread-safe queue has two methods addLast and removeFirst. The addLast method inserts an integer value into the end of the queue and the removeFirst removes the first element of the queue. Test the class with several threads to insert and remove integer values into the queue.
	Assignment 3 The Bounded-Buffer Problem	A bounded integer buffer is a fixed size FIFO queue and is implemented by an array of integer values. The queue has methods add and remove. The add method adds a new integer value to the end of the queue and the thread that call the add method must wait if the queue is full. The remove method removes the first element of the queue and the thread that call the remove method must wait if the queue is empty. Test the class with several threads to insert and remove integer values into the queue.
	Assignment 4 The Barrier Class	A barrier allows a set of threads to all wait for each other to reach a common barrier point. The situation where the required number of threads have called await(), is called tripping the barrier.
	Assignment 5 – Old Brigde	- Constructor: public Barrier(int parties) o Creates a new Barrier that will trip when the given number of parties (threads) are waiting upon it. o parties: The number of parties (threads) - public void await(): o Waits until all parties have invoked await on this barrier. Write a main method to test the Barrier class.
		An old bridge has only one lane and can only hold at most 3 cars at a time without risking collapse. Create a monitor with methods arriveBridge (int direction) and exitBridge () that controls traffic so that at any given time, there are at most 3 cars on the bridge, and all of them are going the same direction. A car calls arriveBridge when it arrives at the bridge and wants to go in the specified direction (0 or 1); arriveBridge should not return until the car is allowed to get on the bridge. A car calls exitBridge when it gets off the bridge, potentially allowing other cars to get on. Don't worry about starving cars trying to go in one direction; just make sure cars are always on the bridge when they can be.
	(Using ReentrantLock and Condition classes)	
	Assignment 1 Building H2O	Create a monitor with methods hydrogen() and oxygen(), which wait until a water molecule can be formed and then return. Don't worry about explicitly creating the water molecule; just wait until two hydrogen threads and one oxygen thread can be grouped together. For example, if two threads call hydrogen(), and then a third thread calls oxygen(), the third thread should wake up the first two threads and they should then all return.
		The constructor for this H2O class accepts an optional fairness parameter. When set true, the solution should take care the access order of hydrogen or oxygen threads.
	Assignment 2 Project 1-	Thread pools were introduced in Section 4.5.1. When thread pools are used, a task is submitted to the pool and executed by a thread from the pool. Work is submitted to the pool using a queue, and an available thread removes work from the queue. If there are no available threads, the work remains queued until one becomes available. If there is no work, threads await notification until a task becomes available.
		Your thread pool will implement the following API: - ThreadPool() - Create a default-sized thread pool. - ThreadPool(int size) - Create a thread pool of size size. - void add(Runnable task) - Add a task to be performed by a thread in the pool. - void shutdown() - Stop all threads in the pool.

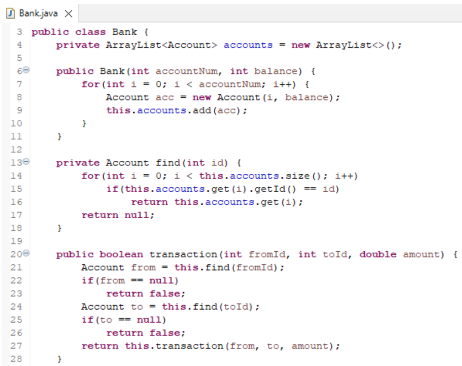
Lab 4.2 Thread Synchronization 2	**Designing a Thread Pool**	Implementation will involve the following activities: 1. The constructor will first create a number of idle threads that await work. 2. Work will be submitted to the pool via the add() method, which adds a task implementing the Runnable interface. The add() method will place the Runnable task into a queue. 3. Once a thread in the pool becomes available for work, it will check the queue for any Runnable tasks. If there is such a task, the idle thread will remove the task from the queue and invoke its run() method. If the queue is empty, the idle thread will wait to be notified when work becomes available. 4. The shutdown() method will stop all threads in the pool by invoking their interrupt() method. This, of course, requires that Runnable tasks being executed by the thread pool check their interruption status.
	Assignment 3 The First Readers -Writers Problem	**The readers–writers problem:** Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database **(readers)**, whereas others may want to update (write) the database **(writers)**. - If two readers access the shared data simultaneously, no adverse affects will result. - However, if a writer and some other thread (either a reader or a writer) accesses the database simultaneously, chaos may ensue. To ensure that these difficulties do not arise, we require that the writers **have exclusive access to the shared database while writing to the database.** **The 1st Readers-Writers Problem:** The 1st readers–writers problem requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object.
	Assignment 4 The Second Readers -Writers Problem	**The 2nd Readers-Writers Problem:** The 2nd readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
	Assignment 1 Dining Philosophers 1	Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The life of a philosopher consists of alternating periods of eating and thinking. When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think. The key question is: Can we write a program for each philosopher that does what it is supposed to do and never gets stuck?
	Assignment 2 Dining Philosophers 2	Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.
		Please show disadvantages the following solution.
		

Account.java

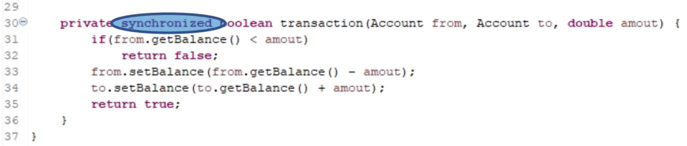
```java
public class Account {
    private int id;
    private double balance;

    public Account(int id, double balance) {
        this.id = id;
        this.balance = balance;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    public int getId() {
        return id;
    }
}
```

Lab 5
Deadlocks

Assignment 3

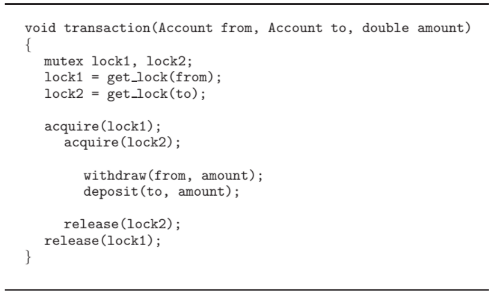


```
public class Bank {
    private ArrayList<Account> accounts = new ArrayList<>();

    public Bank(int accountNum, int balance) {
        for(int i = 0; i < accountNum; i++) {
            Account acc = new Account(i, balance);
            this.accounts.add(acc);
        }
    }

    private Account find(int id) {
        for(int i = 0; i < this.accounts.size(); i++)
            if(this.accounts.get(i).getId() == id)
                return this.accounts.get(i);
        return null;
    }

    public boolean transaction(int fromId, int toId, double amount) {
        Account from = this.find(fromId);
        if(from == null)
            return false;
        Account to = this.find(toId);
        if(to == null)
            return false;
        return this.transaction(from, to, amount);
    }
```



```

    private synchronized boolean transaction(Account from, Account to, double amout) {
        if(from.getBalance() < amout)
            return false;
        from.setBalance(from.getBalance() - amout);
        to.setBalance(to.getBalance() + amout);
        return true;
    }
}
```

In Figure 8.7, we illustrate a transaction() function that dynamically acquires locks. In the text, we describe how this function presents difficulties for acquiring locks in a way that avoids deadlock. Using the Java implementation of transaction() that is provided in the source-code download for this text, modify it using the System.identityHashCode() method so that the locks are acquired in order. You should develop an implementation that each Account instance has a ReentrantLock and these lock objects are ordered using values returned by the System.identityHashCode()method .

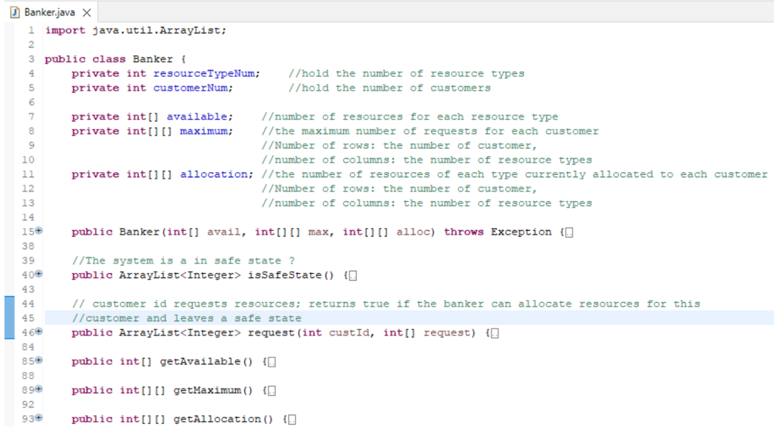


```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);

   acquire(lock1);
      acquire(lock2);

         withdraw(from, amount);
         deposit(to, amount);

      release(lock2);
   release(lock1);
}
```

Figure 8.7 Deadlock example with lock ordering.

For this project, you will write a program that implements the banker’s algorithm used in deadlock avoidance, discussed in Section 8.6.3. Customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied.

Assignment 4
Programing Project – Banker's Algorithm


Banker.java

```
import java.util.ArrayList;

public class Banker {
    private int resourceTypeNum;    //hold the number of resource types
    private int customerNum;        //hold the number of customers

    private int[] available;    //number of resources for each resource type
    private int[][] maximum;    //the maximum number of requests for each customer
                                //Number of rows: the number of customer,
                                //number of columns: the number of resource types
    private int[][] allocation; //the number of resources of each type currently allocated to each customer
                                //Number of rows: the number of customer,
                                //number of columns: the number of resource types

    public Banker(int[] avail, int[][] max, int[][] alloc) throws Exception {□

    //The system is a in safe state ?
    public ArrayList<Integer> isSafeState() {□

    // customer id requests resources; returns true if the banker can allocate resources for this
    //customer and leaves a safe state
    public ArrayList<Integer> request(int custId, int[] request) {□

    public int[] getAvailable() {□

    public int[][] getMaximum() {□

    public int[][] getAllocation() {□
```

Suppose that a snapshot at time T0 is shown below

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

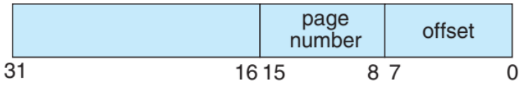
- Run the program and show whether the system is in a safe state or not.
- Suppose that P1 requests (1, 0, 2). Can the request be granted?

Assignment 1

Write a program that implements the FIFO, LRU, and optimal (OPT) page-replacement algorithms presented in Section 10.4. Have your program initially generate a random page-reference string where page numbers range from 0 to 9. Apply the random page-reference string to each algorithm, and record the number of page faults incurred by each algorithm. Pass the number of page frames to the program at startup. You may implement this program in any programming language of your choice.
- Test your program with the page-reference string is 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1, and there are 3 frames.

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size 216 = 65,536 bytes. Your program will read from a file containing logical addresses and, using a TLB and a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. Your learning goal is to use simulation to understand the steps involved in translating logical to physical addresses. This will include resolving page faults using demand paging, managing a TLB, and implementing a page-replacement algorithm.

Specific
Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) an 8-bit page offset. Hence, the addresses are structured as shown as:



Lab 7
Virtual Memory

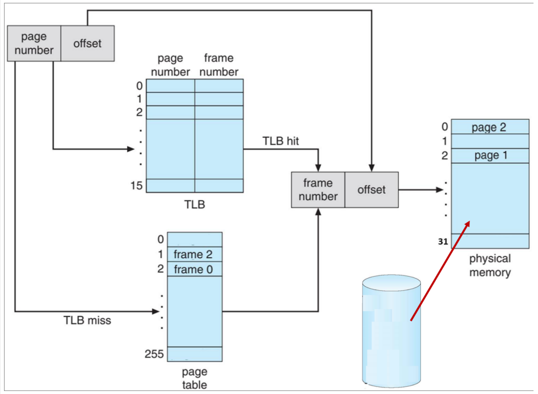
Assignment 2
Designing a Virtual Memory Manager

Other specifics include the following:
• 28
entries in the page table
• Page size of 28
bytes
• 16 entries in the TLB
• Frame size of 28
bytes
• 32 frames
• Physical memory of 8192 bytes (32 frames × 256-byte frame size)

Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

Address Translation

Your program will translate logical to physical addresses using a TLB and page table as outlined in Section 9.3. First, the page number is extracted from the logical address, and the TLB is consulted. In the case of a TLB hit, the frame number is obtained from the TLB. In the case of a TLB miss, the page table must be consulted. In the latter case, either the frame number is obtained from the page table, or a page fault occurs. A visual representation of the address translation process is:



Handling Page Faults

Your program will implement demand paging as described in Section 10.2. The backing store is represented by the file BACKING_STORE.bin, a binary file of size 65,536 bytes. When a page fault occurs, you will read in a 256-byte page from the file BACKING STORE and store it in an available page frame in physical memory. For example, if a logical address with page number 15 resulted in a page fault, your program would read in page 15 from BACKING STORE (remember that pages begin at 0 and are 256 bytes in size) and store it in a page frame in physical memory. Once this frame is stored (and the page table and TLB are updated), subsequent accesses to page 15 will be resolved by either the TLB or the page table.
You will need to treat BACKING_STORE.bin as a random-access file so that you can randomly seek to certain positions of the file for reading. We suggest using the standard C library functions for performing I/O, including fopen(), fread(), fseek(), and fclose().
The size of physical memory, 8192 bytes, is less than the size of the virtual address space—65,536 bytes—so you need to be concerned about page replacements during a page fault.

Test File

We provide the file addresses.txt, which contains integer values representing logical addresses ranging from 0 to 65535 (the size of the virtual address space). Your program will open this file, read each logical address and translate it to its corresponding physical address, and output the value of the signed byte at the physical address.
- Display the number of TLB misses
- Display the number of page faults

Lab 2: Process and Thread Management
Assignment 1:
Mô tả: Phát triển một chương trình có dữ liệu chia sẻ và hai luồng. Dữ liệu chia sẻ là một biến số nguyên khởi tạo bằng 0. Một luồng sẽ tăng giá trị biến lên 1.000.000 lần, trong khi luồng thứ hai sẽ giảm giá trị biến xuống 1.000.000 lần.

a. Chạy luồng đầu tiên, đợi cho đến khi nó kết thúc rồi mới bắt đầu luồng thứ hai. Sau khi cả hai luồng hoàn thành, hiển thị giá trị của biến và giải thích kết quả.

b. Cả hai luồng chạy đồng thời. Chờ cả hai luồng kết thúc và giải thích kết quả.

Assignment 2:
Mô tả: Viết một chương trình đa luồng tính các giá trị thống kê cho một danh sách các số. Tạo ba luồng, một luồng tính giá trị trung bình, một luồng tính giá trị lớn nhất, và một luồng tính giá trị nhỏ nhất.

Chương trình sử dụng các luồng để tính các giá trị thống kê và sau đó in kết quả.

Assignment 3:
Mô tả: Viết chương trình đa luồng để xuất các số nguyên tố. Chương trình nhận một số từ người dùng và tạo một luồng mới để xuất tất cả các số nguyên tố nhỏ hơn hoặc bằng số nhập từ người dùng.

Assignment 4:
Mô tả: Sử dụng kỹ thuật Monte Carlo để ước tính giá trị π. Chương trình sẽ tạo một luồng để sinh ra các điểm ngẫu nhiên và đếm số điểm nằm trong một vòng tròn. Sau khi luồng hoàn thành, chương trình tính toán và in ra ước tính giá trị π.

Assignment 5:
Mô tả: Viết chương trình đa luồng để tạo dãy số Fibonacci. Người dùng nhập số lượng số Fibonacci cần tạo. Chương trình tạo một luồng để tạo ra dãy số và in kết quả sau khi luồng hoàn thành.

Lab 3: Scheduling Algorithms
Mô tả: Triển khai các thuật toán lập lịch tiến trình, bao gồm:

First-Come, First-Served (FCFS)

Shortest Job First (SJF)

Priority Scheduling

Round-Robin (RR) Scheduling

Priority with Round-Robin Scheduling

Các thuật toán này sẽ lên lịch cho các tiến trình dựa trên độ dài CPU burst và ưu tiên.

Lab 4.1: Thread Synchronization 1
Assignment 1:
Mô tả: Phát triển chương trình với dữ liệu chia sẻ và hai luồng. Một luồng tăng và một luồng giảm giá trị biến 1.000.000 lần. Cần bảo vệ các phần chia sẻ để tránh race condition.

a. Sử dụng phương thức synchronized.

b. Sử dụng lớp ReentrantLock.

c. Sử dụng Semaphore.

Assignment 2:
Mô tả: Phát triển một hàng đợi an toàn với các phương thức thêm và xóa phần tử, và kiểm tra với các luồng khác nhau.

Assignment 3:
Mô tả: Giải quyết bài toán "Bounded-Buffer", nơi bạn có một bộ đệm giới hạn và các luồng sẽ thêm và xóa các phần tử khỏi bộ đệm.

Assignment 4:
Mô tả: Phát triển một lớp Barrier cho phép các luồng đồng bộ hóa và chờ nhau đến một điểm chung.

Assignment 5:
Mô tả: Phát triển chương trình để mô phỏng một cây cầu với các phương tiện giao thông. Các luồng mô phỏng các phương tiện sẽ gọi các phương thức để vào và ra khỏi cầu.

Lab 4.2: Thread Synchronization 2
Assignment 1:
Mô tả: Tạo một monitor với các phương thức hydrogen() và oxygen() để tạo ra phân tử nước khi có đủ các luồng hydrogen và oxygen.

Assignment 2:
Mô tả: Thiết kế một Thread Pool, nơi các luồng sẽ thực hiện các công việc trong hàng đợi và dừng khi không có công việc mới.

Assignment 3:
Mô tả: Giải quyết vấn đề "Readers-Writers" để đảm bảo rằng các luồng đọc không bị chặn khi không có luồng viết đang hoạt động.

Assignment 4:
Mô tả: Giải quyết bài toán "Second Readers-Writers", yêu cầu rằng khi có một luồng viết chờ, các luồng đọc không được phép bắt đầu.

Lab 5: Deadlocks
Assignment 1:
Mô tả: Giải quyết vấn đề "Dining Philosophers", nơi các triết gia phải ăn với hai chiếc đũa nhưng có thể gặp deadlock.

Assignment 2:
Mô tả: Sử dụng các chiếc đũa đặt tại trung tâm bàn để tránh deadlock, với một quy tắc đơn giản để đảm bảo không có deadlock.