HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT REPORT

ILLUSTRATION OF PROCESS SYNCHROMIZATION USING MONITOR IN JAVA

Course	:	**Operating System**
Course ID	:	**IT3070E**
Instructor	:	**Prof. Do Quoc Huy**
Members	:	

Nguyen Tien Doanh	20214881	doanh.nt214881@sis.hust.edu.vn
Le Tuan Anh	20214874	anh.lt214874@sis.hust.edu.vn
Bui Minh Quang	20214925	quang.bm214925@sis.hust.edu.vn
Dinh Nguyen Cong Quy	20214927	quy.dnc214927@sis.hust.edu.vn
Phan Dinh Truong	20214937	truong.pd214937@sis.hust.edu.vn

Hanoi, January 2024

Table of Contents

1. Introduction

A critical resource is a type of resource that is limited in its usage but can be required simultaneously by multiple threads. A physical example of such a resource is a printer where multiple Word processes or threads may send printing requests to this resource at the same time. A critical resource may also be a logical one such as a common file that many processes want to read or modify simultaneously.

In a scenario where multiple processes or threads try to access the same critical resource, a race condition may occur in which the result of the same program depends on the relative timing order of uncontrollable events. A classic example is the Producer-Consumer problem. In this problem, a producer makes a product, then puts it into the buffer and increases the Counter variable by 1. A consumer, in contrast, consumes the product from the buffer and then decreases the Counter variable by 1. Counter variable which indicates the number of products in the buffer is a critical resource as it may be requested by both the producer and consumer simultaneously. Supposed there are 5 products in the buffer, then the producer assumes that Counter = 5, the consumer also assumes Counter = 5, when the producer finishes, it changes Counter to 6 but when the consumer finishes, it does not get the updated value of Counter, which leads to the Counter variable (which is supposed to be 5) is set to 4 by the consumer.

The aforementioned example emphasizes the importance of controlling processes' access to critical resources. Critical Resource Management and Process Synchronization is a traditional challenge every OS designer and software developer must face. Several techniques have been proposed to deal with this challenge, including those without the help of hardware such as Mutex (Lock), or those with the help of hardware (for atomic operation) such as TestAndSet, some version of Semaphore, etc. Among them, Monitor, which is a high-level implementation, has been very popular in different programming languages. It serve as powerful constructs ensuring orderly access to shared resources among concurrent processes. Combining data structures, procedures, and synchronization tools into a cohesive unit, monitors enable controlled access to critical sections of code, safeguarding against race conditions and conflicts. They encapsulate shared variables and methods, allowing processes to interact via well-defined entry and exit protocols. By utilizing mutual exclusion and condition variables, monitors ensure exclusive access to resources when needed while facilitating synchronization through synchronized methods and condition wait-notify mechanisms. This encapsulation and synchronization mechanism simplifies the development of concurrent systems, fostering reliability and integrity in multithreaded environments.

In this project, we are going to implement the Monitor technique in some classic synchronization problems including Dining Philosopher, Reader-Writer, Sleeping Barber and Bathroom.

2. Typical Critical Resource Problem and Monitor Solution

2.1. Dining Philosophers

Problem Description

Five philosophers dine together at the same table. Each philosopher has his own plate at the table. There is a chopstick (critical resource - 5 in total) between each plate. The dish served is a kind of spaghetti which has to be eaten with two chopsticks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat his spaghetti when he has both a left and right chopstick. Thus two chopstick will only be available when his two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, he will put down both chopsticks. The problem is how to design a regimen such that any philosopher will not starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

Algorithm Idea

In tackling the Dining Philosophers problem, we employ a solution where a philosopher is allowed to pick up both chopsticks (left and right) only if both are available. To implement this, we introduce a structure representing the three possible states for each philosopher: **THINKING**, **HUNGRY**, and **EATING**.

A philosopher can transition to the **EATING** state only if both of their neighbors are not currently eating. After finishing a meal, the philosopher puts down the chopsticks and notifies the two neighboring philosophers. The entire process is managed by the DiningPhilosophersMonitor, which controls the distribution of chopsticks. Before commencing a meal, each philosopher must call the **pickup()** operation, potentially leading to the suspension of the philosopher process. Upon successful completion, the philosopher can proceed to eat. Subsequently, the philosopher invokes the **putdown()** operation.

The sequence of actions for philosopher i-th involves thinking, calling **DiningPhilosophersMonitor.pickup(i)** before eating, and finally, calling **DiningPhilosophersMonitor.putdown(i)** after finishing the meal.

This solution effectively ensures that no two neighboring philosophers eat simultaneously, and it prevents deadlocks. However, it's worth noting that there's a possibility of a philosopher starving to death in certain cases, particularly when the two neighboring philosophers continuously take turns eating. This scenario, though rare, is more likely to occur when the thinking time of philosophers is extremely short. Despite this potential issue, the solution remains a viable and practical approach to the Dining Philosophers problem.

2.2. Sleeping Barber

Problem Description

The problem is based on a hypothetical barbershop with a certain number of barbers (critical resources). When there are no customers, the barber sleeps on his chair. If any customer enters, he should wake up the barber to get his haircut. If there are no chairs empty for other customers, they have to wait in the waiting room/chairs.

Algorithm Idea

Java's ReentrantLock and Condition are used for managing critical sections of the code and signaling between threads respectively. This ensures that access to shared resources (like the list of customers and barbers) is synchronized.

Customers are represented as objects. When a customer enters the shop (EnterShop method), they check for available chairs or barbers. If a barber is available, they get a haircut immediately, otherwise, they either take a waiting chair or leave if no chairs are free.Barber. Barbers, in the getHairCut method, check for customers. If there's no customer, the barber sleeps until one arrives. Once a customer is available, the barber performs the haircut, simulating the process with a random delay.

The **signal()** and **await()** methods from Condition are used to notify waiting threads (barbers or customers) about changes in the shop's status (e.g., availability of barbers or customers).

2.3. Readers – Writers

Problem Description

The reader-priority variant of the reader-writer problem addresses the issue of potential writer starvation by favoring readers in accessing a shared resource (critical resource). In this scenario, multiple readers can concurrently read from the resource even if other readers are currently active. Writers, however, have exclusive access and can only write when there are no readers or writers currently using the resource. The solution employs synchronization mechanisms, such as locks and conditional variables, to ensure orderly and fair access to the shared resource while prioritizing reader concurrency.

Algorithm Idea

There are four state variables in this solution: **waitingWriters (wW)**, **activeWriters (aW)**, **waitingReaders (wR)**, and **activeReaders (aR)**. Additionally, there are two condition variables: canRead and canWrite. A writer can only write if there are no active writers and no active readers. If there are active writers or readers, the writer is added to the canWrite queue. After completing the writing task, the writer checks if there are waiting readers (wR). If so, it wakes them up; otherwise, it wakes up the writers in the canWrite queue.

On the other hand, a reader can only read when there are no active writers. If there are active writers, the reader is added to the canRead queue. While a reader is reading, it notifies all readers in the canRead queue to access the database concurrently, including any new readers arriving. When a reader finishes reading, if there is no active readers, it checks if there are waiting writers (wR). If so, it wakes them up; otherwise, it wakes up the writers in the canWrite queue.

The synchronization is implemented using the following patterns:

Reader: monitor.beginread... Read ... Monitor.endread

Writer: monitor.beginwrite... Write ... monitor.endwrite

It is evident that this algorithm avoids deadlocks as the state variables are accessed within the monitor, and there is no scenario where readers and writers wait for each other. However, it has the potential for writer starvation when a continuous influx of readers occurs.

2.4. Bathroom Problem

Problem Description

A mixed bathroom facility is designed for use by both males and females. However, due to the diverse nature of its users, certain constraints must be established to ensure fair and safe usage of the facility. The bathroom has a limited capacity and can accommodate a specific number of users at a given time. The goal is to implement a system that allows fair access to the bathroom while ensuring privacy and security for all users.

Algorithm Idea

Our solution to the problem addresses synchronization among male and female individuals who share a common bathroom, aiming to ensure exclusive use based on gender while efficiently utilizing the bathroom's capacity.

The algorithm employs a monitor-based solution utilizing a shared lock and conditions to coordinate access for males and females. The UnisexBathroom class initializes the bathroom's capacity and maintains counts for male and female users, along with a lock and conditions for synchronization.

The **maleUseBathroom()** and **femaleUseBathroom()** methods serve as entry points for males and females, respectively. Upon invocation, they acquire the lock and check if the bathroom is available based on the current counts of users of the opposite gender and the bathroom's capacity. If the bathroom is not available, the method waits on its associated condition variable.

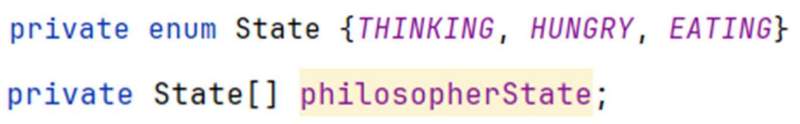
Once the bathroom is available, the respective gender increments its count, simulates bathroom usage, decrements its count upon completion, and signals the waiting condition for the opposite gender to potentially enter.

3. Detailed Solutions

In this section, we will go into details about the implementation of monitor that manage access to the shared resources through the simulation of each

3.1. Dining Philosopher

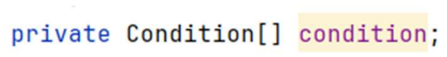
- **State Variables**



```java
private enum State {THINKING, HUNGRY, EATING}
private State[] philosopherState;
```

philosopherState[i]: Represents the state of philosopher i, whether they are THINKING, HUNGRY, or EATING. It gets updated whenever the state of that philosopher changes.

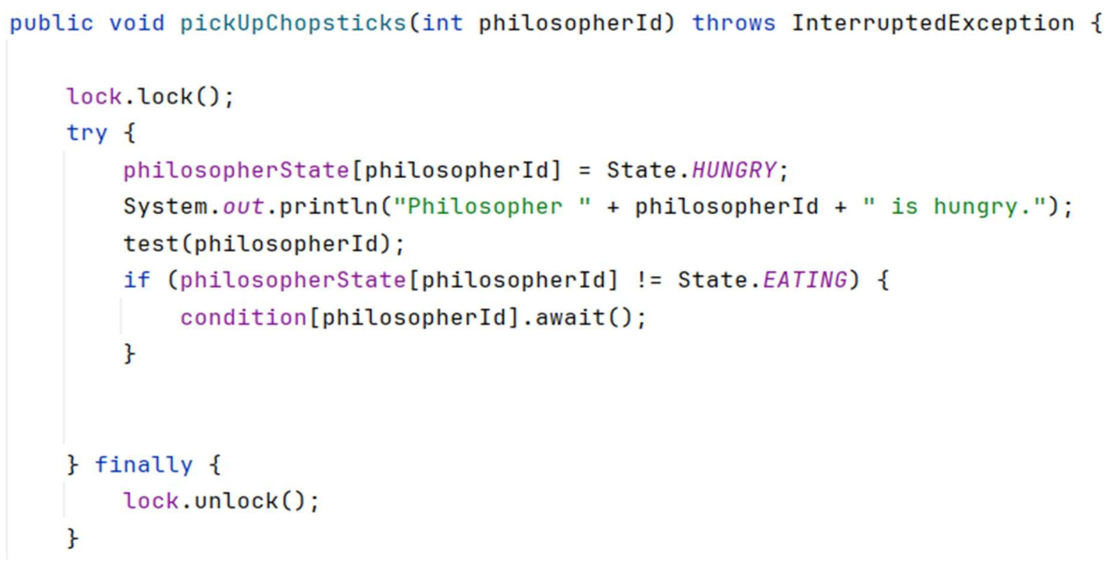
- **Condition Variables**



```java
private Condition[] condition;
```

Condition[i]: The Condition variables play a crucial role in allowing threads to wait for a specific condition to be satisfied using **await()**, and they facilitate signaling other waiting thread when conditions change using **signal()**. This variable allow a philosopher to delay their actions when hungry but unable to obtain the necessary chopsticks.

- **Key Methods**

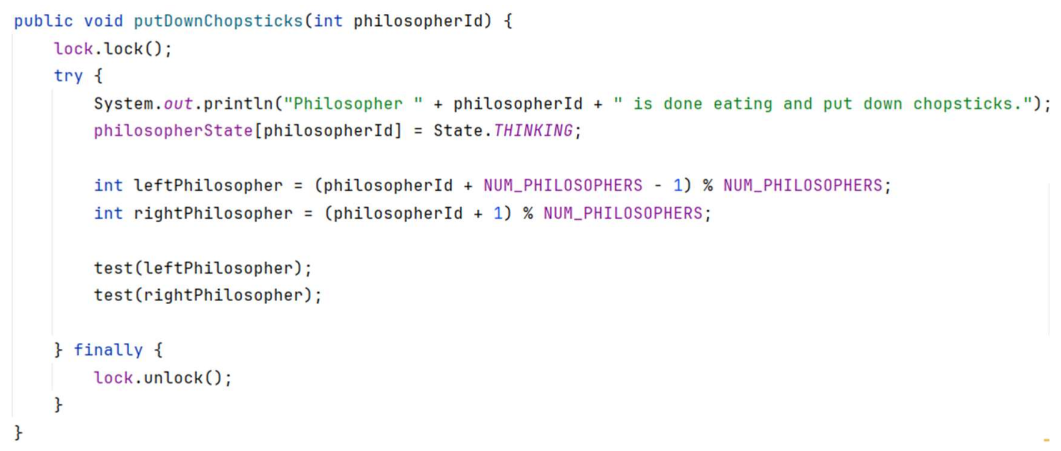


```java
public void pickUpChopsticks(int philosopherId) throws InterruptedException {

    lock.lock();
    try {
        philosopherState[philosopherId] = State.HUNGRY;
        System.out.println("Philosopher " + philosopherId + " is hungry.");
        test(philosopherId);
        if (philosopherState[philosopherId] != State.EATING) {
            condition[philosopherId].await();
        }


    } finally {
        lock.unlock();
    }
```

pickUpChopsticks : This method ensures that a philosopher transitions to the “Hungry” state, tests if they can start eating, and if not, waits for the conditions to be met. The **await()** call releases the lock temporarily, allowing other philosophers to access the critical section and potentially change states. Once the conditions are satisfied, the philosopher proceeds to eat.



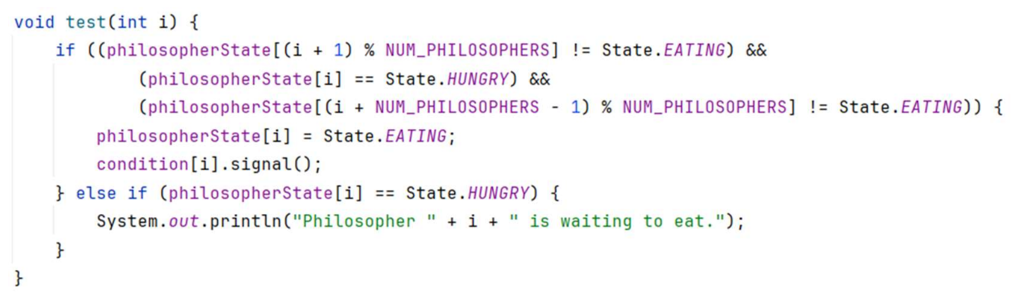
```java
public void putDownChopsticks(int philosopherId) {
    lock.lock();
    try {
        System.out.println("Philosopher " + philosopherId + " is done eating and put down chopsticks.");
        philosopherState[philosopherId] = State.THINKING;

        int leftPhilosopher = (philosopherId + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS;
        int rightPhilosopher = (philosopherId + 1) % NUM_PHILOSOPHERS;

        test(leftPhilosopher);
        test(rightPhilosopher);

    } finally {
        lock.unlock();
    }
}
```

putDownChopsticks : this method ensures that a philosopher transition from the eating to the thinking state, updates the shared state information, and triggers tests for neighboring philosophers to potentially start their own eating phases

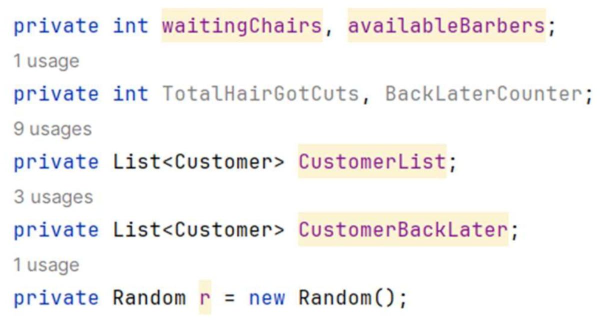


```java
void test(int i) {
    if ((philosopherState[(i + 1) % NUM_PHILOSOPHERS] != State.EATING) &&
            (philosopherState[i] == State.HUNGRY) &&
            (philosopherState[(i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS] != State.EATING)) {
        philosopherState[i] = State.EATING;
        condition[i].signal();
    } else if (philosopherState[i] == State.HUNGRY) {
        System.out.println("Philosopher " + i + " is waiting to eat.");
    }
}
```

test() : This method checks if a philosopher is hungry or not. If he is not hungry, do nothing. Otherwise, check if his neighbors are eating or not. If they are not eating, it means the current philosopher is ready to eat, and his state is set to eating.

3.2. Sleeping Barber

- **State Variables:**



```java
private int waitingChairs, availableBarbers;
1 usage
private int TotalHairGotCuts, BackLaterCounter;
9 usages
private List<Customer> CustomerList;
3 usages
private List<Customer> CustomerBackLater;
1 usage
private Random r = new Random();
```

waitingChairs: Represents the number of chairs available in the waiting room. This variable determines the maximum number of customers that can wait for a haircut before the shop is considered full.

- **availableBarbers:** Denotes the count of available barbers who can provide haircuts to customers. This variable determines how many customers can be served simultaneously.

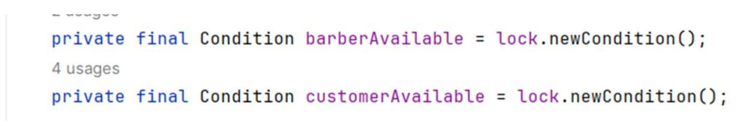
- **totalHairGotCuts:** Tracks the total number of haircuts that have been completed in the shop. It's updated when a haircut is successfully finished by a barber.

- **BackLaterCounter:** Maintains a count of customers who couldn't enter the shop due to a lack of available chairs. When a customer decides to come back later, this counter is incremented.

- **CustomerList**: This is a list or queue representing the customers currently waiting for a haircut. It holds instances of the Customer class and manages the order in which customers will be served.

- **CustomerBackLater**: This list stores customers who attempted to enter the shop but had to leave due to no available chairs. They intend to return later to get a haircut.

- **Condition Variables:**

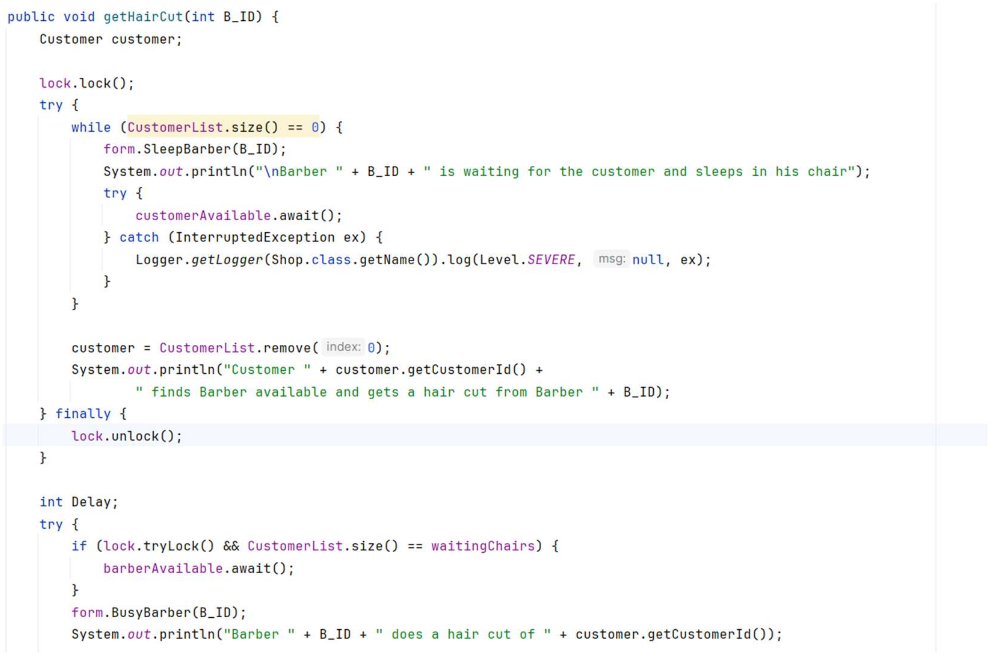


```
private final Condition barberAvailable = lock.newCondition();
4 usages
private final Condition customerAvailable = lock.newCondition();
```

In this implementation, lock variable is a reentrant lock used for synchronization purposes within the barber shop. The Condition variables (**barberAvailable** and **customerAvailable**) play a crucial role in allowing threads to wait for a specific condition to be satisfied using **await()**, and they facilitate signaling other waiting threads when conditions change using **signal()** or **signalAll()**.

- **Key Methods:**

The following methods manage access to shared resources and condition variables to handle waiting and signaling between barbers and customers based on the availability of resources.



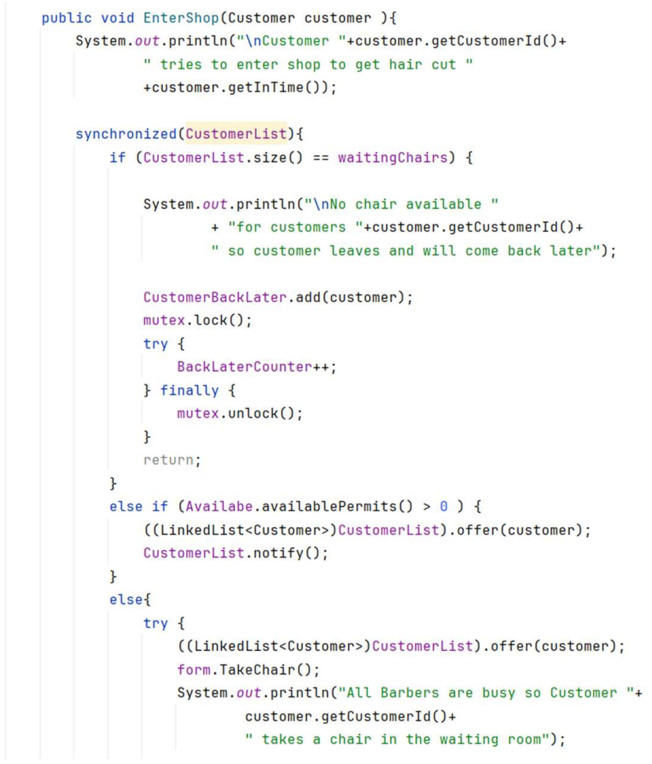
```java
public void getHairCut(int B_ID) {
    Customer customer;

    lock.lock();
    try {
        while (CustomerList.size() == 0) {
            form.SleepBarber(B_ID);
            System.out.println("\nBarber " + B_ID + " is waiting for the customer and sleeps in his chair");
            try {
                customerAvailable.await();
            } catch (InterruptedException ex) {
                Logger.getLogger(Shop.class.getName()).log(Level.SEVERE,  msg: null, ex);
            }
        }

        customer = CustomerList.remove( index: 0);
        System.out.println("Customer " + customer.getCustomerId() +
                " finds Barber available and gets a hair cut from Barber " + B_ID);
    } finally {
        lock.unlock();
    }

    int Delay;
    try {
        if (lock.tryLock() && CustomerList.size() == waitingChairs) {
            barberAvailable.await();
        }
        form.BusyBarber(B_ID);
        System.out.println("Barber " + B_ID + " does a hair cut of " + customer.getCustomerId());
```

getHairCut(int B_ID): This method simulates a customer getting a haircut from a barber. It starts by acquiring a lock on the shared resources to ensure exclusive access. The while loop checks if there are no customers in the shop. If so, the barber waits **(await)** for a customer to arrive, releasing the lock temporarily. Once a customer arrives, the barber removes them from the list and begins the haircut process. After obtaining the lock again, it simulates the haircut duration using a random delay **(Thread.sleep(Delay))** and updates the statistics. If there are more customers waiting, it signals **(signal())** the availability of the barber or customers accordingly.



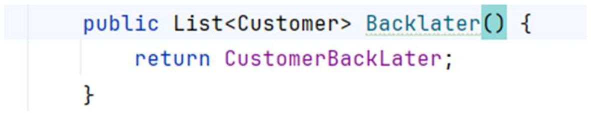
```java
public void EnterShop(Customer customer ){
    System.out.println("\nCustomer "+customer.getCustomerId()+
            " tries to enter shop to get hair cut "
            +customer.getInTime());

    synchronized(CustomerList){
        if (CustomerList.size() == waitingChairs) {

            System.out.println("\nNo chair available "
                    + "for customers "+customer.getCustomerId()+
                    " so customer leaves and will come back later");

            CustomerBackLater.add(customer);
            mutex.lock();
            try {
                BackLaterCounter++;
            } finally {
                mutex.unlock();
            }
            return;
        }
        else if (Availabe.availablePermits() > 0 ) {
            ((LinkedList<Customer>)CustomerList).offer(customer);
            CustomerList.notify();
        }
        else{
            try {
                ((LinkedList<Customer>)CustomerList).offer(customer);
                form.TakeChair();
                System.out.println("All Barbers are busy so Customer "+
                        customer.getCustomerId()+
                        " takes a chair in the waiting room");
```

EnterShop(Customer customer): This method represents a customer entering the shop. It starts by displaying the customer's attempt to enter the shop and then acquires the lock. It checks if all the chairs are occupied, in which case the customer leaves and plans to return later. If there's a barber available, the customer joins the queue for a haircut, and a signal notifies the barber about the customer's arrival. If all barbers are occupied, the customer takes a waiting room chair and waits. If they are the first in line, a signal is sent to notify the barber of their presence.

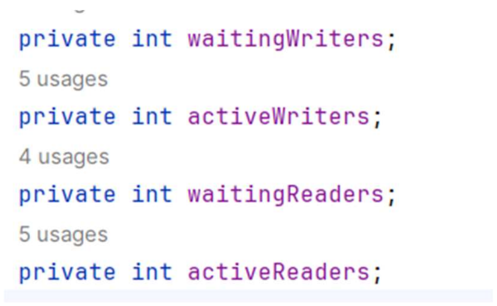


```
public List<Customer> Backlater() {
    return CustomerBackLater;
}
```

Backlater(): This method simply returns the list of customers who opted to come back later because no chairs were available initially. It provides a means to retrieve this list of customers who couldn't immediately enter the shop due to the lack of space.

3.3. Reader-Writer

- **State Variables:**

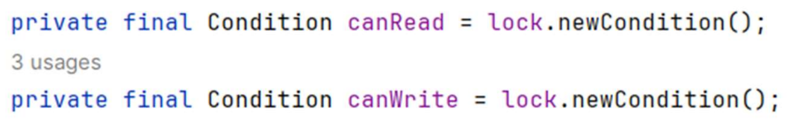


```
private int waitingWriters;
5 usages
private int activeWriters;
4 usages
private int waitingReaders;
5 usages
private int activeReaders;
```

- **waitingWriters**: This variable represents the number of writers that are currently waiting to acquire the lock. Writers have to wait if there is an active writer or if there are active readers. It is incremented when a writer begins waiting and decremented when a writer is allowed to write.
- **activeWriters**: This variable is a binary indicator (0 or 1) representing whether there is an active writer currently writing. It is set to 1 when a writer begins writing and set to 0 when the writer finishes writing.
- **waitingReaders**: Similar to waitingWriters, this variable represents the number of readers that are currently waiting to acquire the lock. Readers have to wait if there is an active writer. It is incremented when a reader begins waiting and decremented when a reader is allowed to read.

- **activeReaders**: This variable represents the number of active readers currently reading. It is incremented when a reader begins reading and decremented when a reader finishes reading. If it becomes 0 and there are waiting writers, it signals the canWrite condition to allow a writer to write.

- **Condition Variables :**

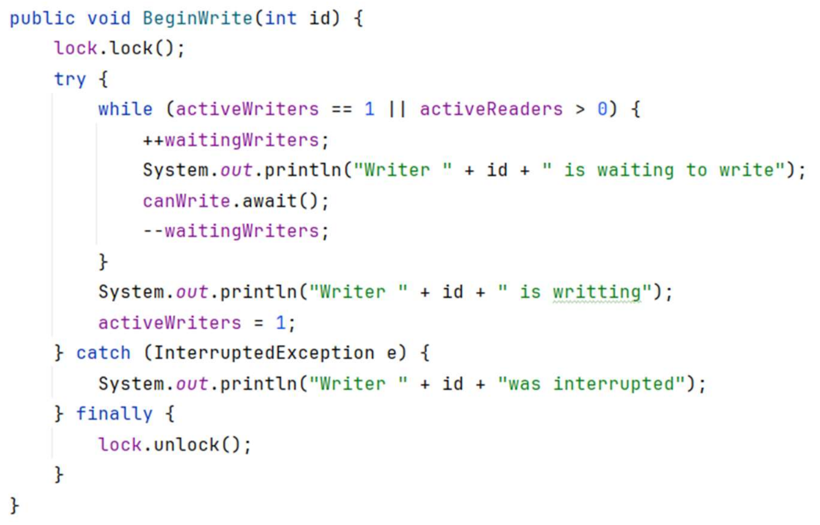


```
private final Condition canRead = lock.newCondition();
3 usages
private final Condition canWrite = lock.newCondition();
```

- **CanRead** : This condition variable is associated with the ability of readers to begin reading. Readers will wait on this condition when there is an active writer.

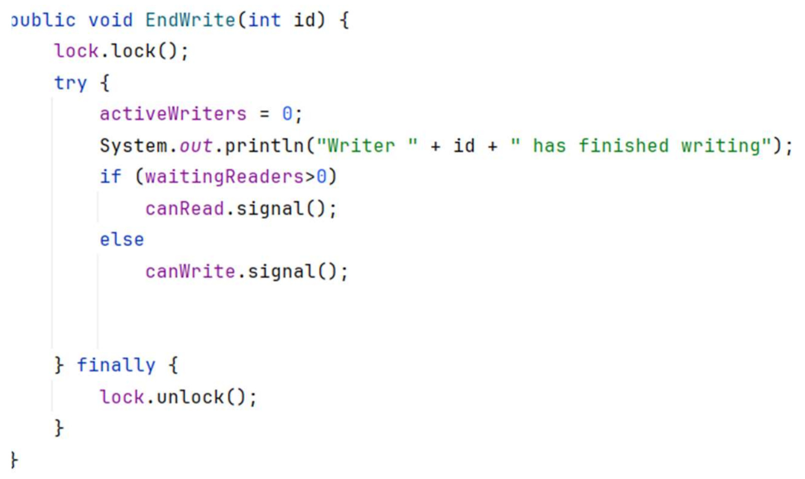
- **CanWrite** : This condition variable is associated with the ability of writers to begin writing. Writers will wait on this condition when there is an active writer or active readers.

- **Key Methods:**



```
public void BeginWrite(int id) {
    lock.lock();
    try {
        while (activeWriters == 1 || activeReaders > 0) {
            ++waitingWriters;
            System.out.println("Writer " + id + " is waiting to write");
            canWrite.await();
            --waitingWriters;
        }
        System.out.println("Writer " + id + " is writting");
        activeWriters = 1;
    } catch (InterruptedException e) {
        System.out.println("Writer " + id + "was interrupted");
    } finally {
        lock.unlock();
    }
}
```

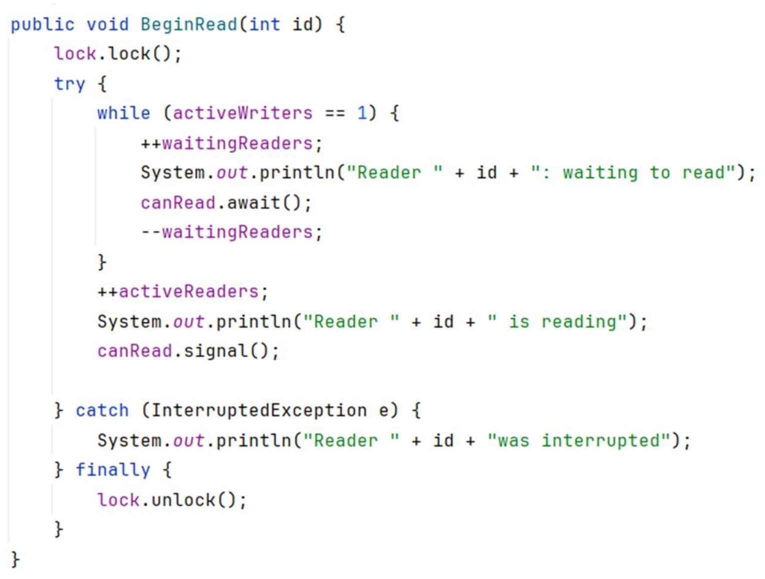
BeginWrite : This method checks whether a writer can write to the database by examining if there is currently any writer writing or any reader reading. If so, it puts the current writer in the queue. Otherwise, it allows this writer to write to the database and set **activeWriter** to 1.



```
public void EndWrite(int id) {
    lock.lock();
    try {
        activeWriters = 0;
        System.out.println("Writer " + id + " has finished writing");
        if (waitingReaders>0)
            canRead.signal();
        else
            canWrite.signal();


    } finally {
        lock.unlock();
    }
}
```

EndWrite :This method is called when a writer completes updating the database. It sets the number of active writers to 0 and checks if there is any waiting reader. If there is, it wakes up that reader and allows them to access the database. If not, it wakes up the canWrite queue so that any waiting writers can proceed with updating the database.



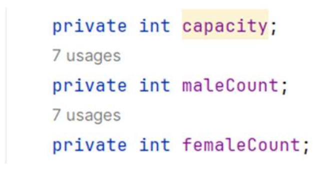
```
public void BeginRead(int id) {
    lock.lock();
    try {
        while (activeWriters == 1) {
            ++waitingReaders;
            System.out.println("Reader " + id + ": waiting to read");
            canRead.await();
            --waitingReaders;
        }
        ++activeReaders;
        System.out.println("Reader " + id + " is reading");
        canRead.signal();

    } catch (InterruptedException e) {
        System.out.println("Reader " + id + "was interrupted");
    } finally {
        lock.unlock();
    }
}
```

BeginRead : This method checks if a reader can start reading by examining if there is any ongoing writing by a writer. If there is, the current reader must wait. If not, he is allowed to access the database and then signal another waiting reader, so they can access the database.

3.4. Bathroom Problem

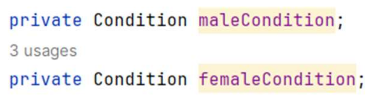
- **State Variables:**



```
private int capacity;
7 usages
private int maleCount;
7 usages
private int femaleCount;
```

- **capacity**: Represents the maximum number of individuals allowed inside the bathroom at a time. It's a fixed value set during the initialization of the bathroom.
- **maleCount**: Tracks the current number of males inside the bathroom.
- **femaleCount**: Tracks the current number of females inside the bathroom.

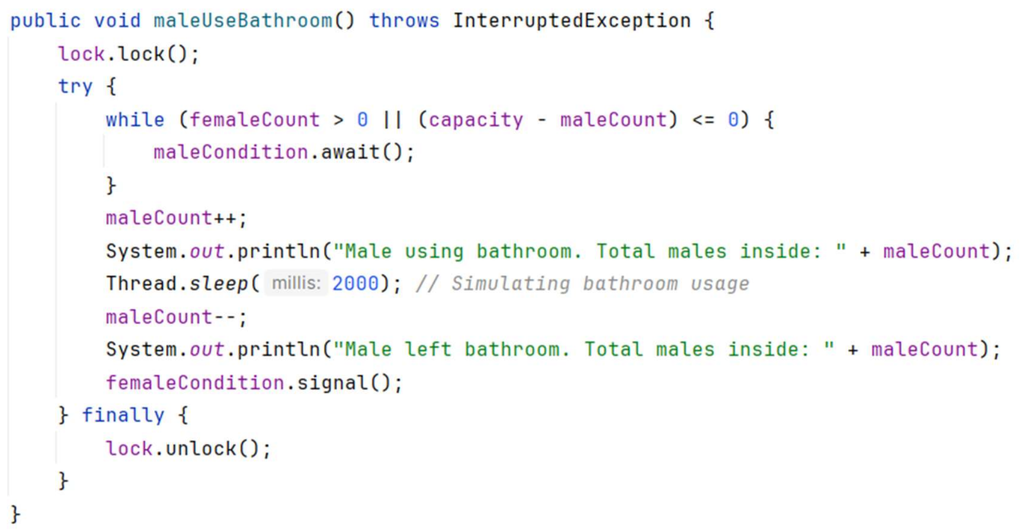
- **Condition Variables:**



```java
private Condition maleCondition;
3 usages
private Condition femaleCondition;
```

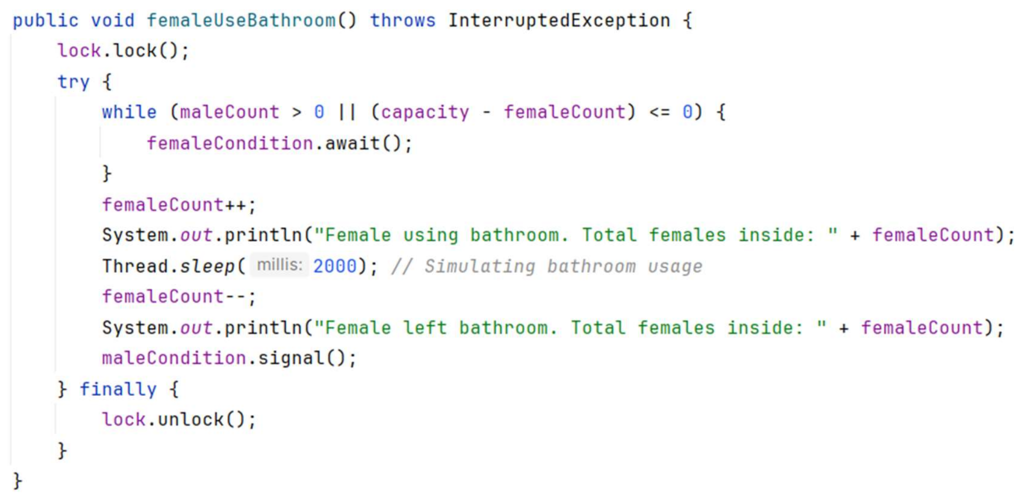
- **maleCondition**: This Condition variable is associated with the male usage of the bathroom. It controls the waiting and signaling for males to use the bathroom based on certain conditions (e.g., availability of the bathroom and absence of females inside).
- **femaleCondition**: Similar to maleCondition, this Condition variable is associated with the female usage of the bathroom. It controls the waiting and signaling for females to use the bathroom based on conditions (e.g., availability of the bathroom and absence of males inside).

- **Key Methods:**



```java
public void maleUseBathroom() throws InterruptedException {
    lock.lock();
    try {
        while (femaleCount > 0 || (capacity - maleCount) <= 0) {
            maleCondition.await();
        }
        maleCount++;
        System.out.println("Male using bathroom. Total males inside: " + maleCount);
        Thread.sleep( millis: 2000); // Simulating bathroom usage
        maleCount--;
        System.out.println("Male left bathroom. Total males inside: " + maleCount);
        femaleCondition.signal();
    } finally {
        lock.unlock();
    }
}
```

maleUseBathroom(): This method represents the logic for a male to use the bathroom. It acquires the lock, checks if there are females in the bathroom or if the maximum capacity for males is reached. If so, it waits using maleCondition.await() until it can proceed. Once inside, it increments maleCount, simulates bathroom usage, decrements maleCount, and signals femaleCondition to notify waiting females.



```
public void femaleUseBathroom() throws InterruptedException {
    lock.lock();
    try {
        while (maleCount > 0 || (capacity - femaleCount) <= 0) {
            femaleCondition.await();
        }
        femaleCount++;
        System.out.println("Female using bathroom. Total females inside: " + femaleCount);
        Thread.sleep( millis: 2000); // Simulating bathroom usage
        femaleCount--;
        System.out.println("Female left bathroom. Total females inside: " + femaleCount);
        maleCondition.signal();
    } finally {
        lock.unlock();
    }
}
```

femaleUseBathroom(): Similar to maleUseBathroom(), this method represents the logic for a female to use the bathroom. It acquires the lock, checks for males in the bathroom or if the maximum capacity for females is reached. If conditions aren't met, it waits using femaleCondition.await() until it can proceed. Once inside, it increments femaleCount, simulates bathroom usage, decrements femaleCount, and signals maleCondition to notify waiting males.

4. Graphical User Interface

For the Graphic User Interface (GUI), we use JavaFX to build GUI for easily illustrate synchronization to user. We only demo **2 Monitor Algorithms** for 2 problems: **Dining Philosophers** and **Sleeping Barber.** Readers – Writers Problem and Bathroom Problem are not having GUI, but the user can try it in the Java console / terminal. The two others may be in the future work (Not having GUI and Source code)

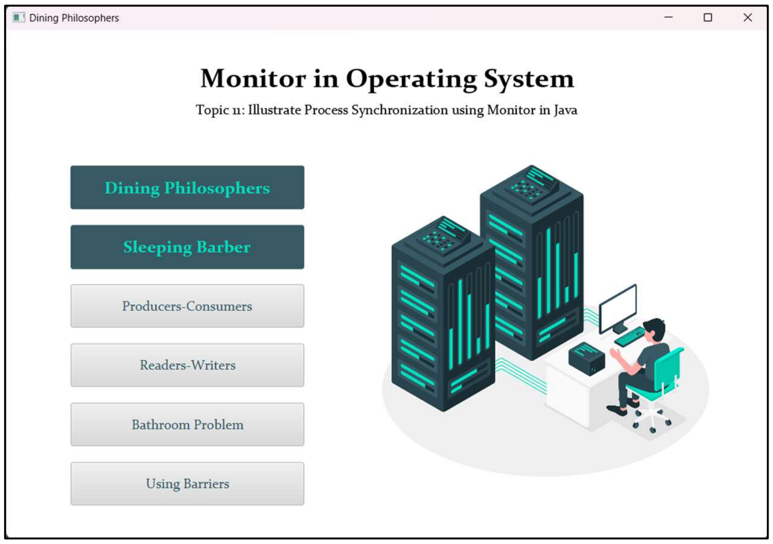


Figure 1. Menu of Monitor in OS

The user can try the monitor to illustrate the process synchronization by clicking to the problem. In **Dining Philosophers Problem**, the GUI will be shown to user:

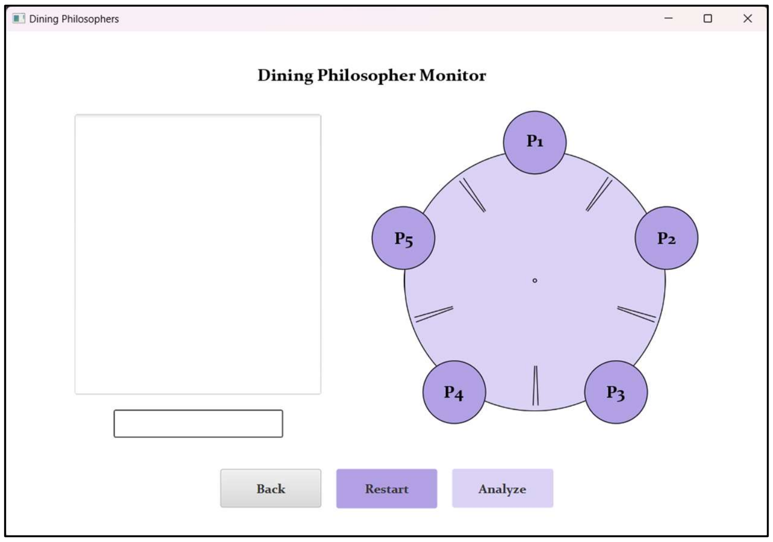
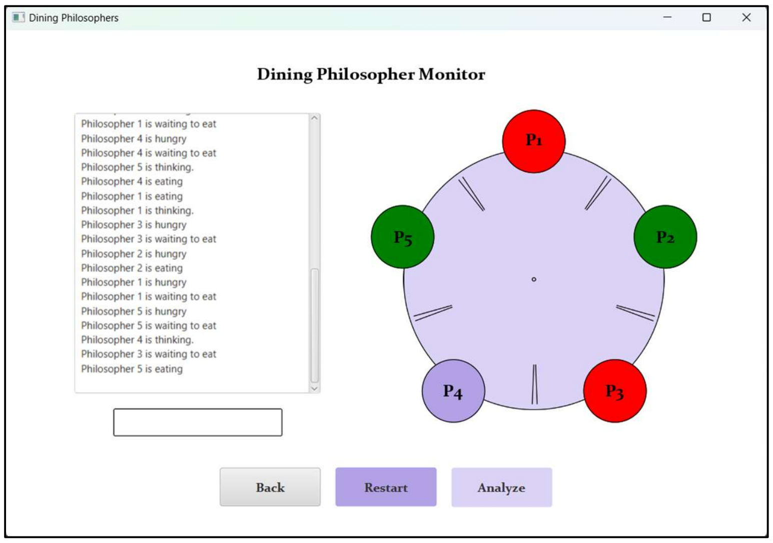


Figure 2. Dining Philosophers Monitor GUI

The user can click **"Restart"** button to start / or restart the program. The philosophers will change the color to in each current state. The color purpul will show that the philosopher is **THINKING.** When they are **EATING,** the color will change to green. And when he is **HUNGRY,** the color is red. The process of synchronization will be logged and printed in the left hand side.



```
Philosopher 1 is waiting to eat
Philosopher 4 is hungry
Philosopher 4 is waiting to eat
Philosopher 5 is thinking.
Philosopher 4 is eating
Philosopher 1 is eating
Philosopher 1 is thinking.
Philosopher 3 is hungry
Philosopher 3 is waiting to eat
Philosopher 2 is hungry
Philosopher 2 is eating
Philosopher 1 is hungry
Philosopher 1 is waiting to eat
Philosopher 5 is hungry
Philosopher 5 is waiting to eat
Philosopher 4 is thinking.
Philosopher 3 is waiting to eat
Philosopher 5 is eating
```

Figure 3. Dining Philosopher Illustration program

When the user does not use the current monitor for **Dining Philosopher Problem,** they can go back to Menu and start new problems.

5. Discussion and Future Work

This project emphasizes the significance of efficiently synchronizing processes and managing critical resources through the implementation of the Monitor technique in Java. Subsequent efforts may concentrate on exploring advanced synchronization methods, optimizing performance, and conducting real-time simulations in similar scenarios.

- Refine the GUI: Enhance graphical representations for better visualization.
- Address CPU starvation: Explore strategies to handle potential CPU starvation issues in the current implementations.
- Continuous improvement: Regularly update and refine the algorithms based on feedback and emerging synchronization challenges.

6. Conclusion

- Monitor is an important tool in multi-threaded programming, helping to ensure synchronized access to shared data.
- Using monitors helps avoid race conditions and deadlocks in multithreaded applications.
- Monitor design and use must comply with safety and efficiency principles to ensure system stability.
- Monitor provides a convenient approach for resource management and concurrency, increasing the scalability of multi-threaded applications.

CONTRIBUTIONS

Nguyễn Tiến Doanh (C)	Implement monitor for “Reader-Writer Problem “
Lê Tuấn Anh	Implement monitor for “Bathroom Problem”
Bùi Minh Quang	Implement monitor for “Dining Philosopher Problem”
Đinh Nguyễn Công Quý	Implement monitor “Sleeping Barber”
Phan Đình Trường	Design and implement GUI

**Note: Report and slides are distributed equally among members*

REFERENCES

[1] Abraham Silberschatz Operating System Concepts 10th 2018

[2] Reader-Writers solution using Monitors