Documentation for OS ch1 tasks

1. Test and Set Code

```
Import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

- These statements import the Lock interface and the ReentrantLock class from the java.util.concurrent.locks package. These are used for implementing locks to control access to the shared resource.

```
class Process extends Thread {
private static Lock resourceLock = new ReentrantLock();
private static boolean resourcesOpen = true;
```

- This class represents a process that attempts to access the shared resource. It extends the Thread class to be executed concurrently.
- resourceLock is a static instance of ReentrantLock used to control access to the shared resource.
- resourcesOpen is a static boolean variable indicating whether the resources are open or closed.

```
public void run() {
try {
Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() + " trying to access
resources.");
boolean acquired = false;
while (!acquired) {
if (resourceLock.tryLock()) {
acquired = true;
System.out.println(Thread.currentThread().getName() + " acquired
resources.");
Thread.sleep(2000);
System.out.println(Thread.currentThread().getName() + " releasing
resources.");
resourceLock.unlock();
} else {
System.out.println(Thread.currentThread().getName() + " couldn't acquire
resources. Busy waiting.");
Thread.sleep(500); // Simulate busy waiting
}
}
} catch (InterruptedException e) {
e.printStackTrace();
}
}
```

- The run() method overrides the run() method of the Thread class.
- It simulates a process attempting to access the resources.
- The process attempts to acquire the resourceLock using tryLock() which returns true if the lock is acquired successfully.
- If the lock is acquired, the process performs its work (simulated by sleeping for 2 seconds) and then releases the lock.
- If the lock is not acquired, the process enters a busy waiting loop where it periodically checks for the availability of the lock every 500 milliseconds.

```
public class Main {
public static void main(String[] args) {
Process[] processes = new Process[5];

for (int i = 0; i < processes.length; i++) {
processes[i] = new Process();
processes[i].start();
}

Thread closer = new Thread(() -> {
try {
Thread.sleep(1500); // Wait for other processes to start
System.out.println("Closing resources...");
Process.resourceLock.lock();
Process.resourcesOpen = false;
Process.resourceLock.unlock();
} catch (InterruptedException e) {
e.printStackTrace();
```

```
        }
        });
        closer.start();
        }
}
```

- The Main class contains the main() method, which is the entry point of the program.
- It creates an array of Process objects and starts them as threads.
- It also creates a separate thread closer to simulate the process responsible for closing the resources. This thread waits for 1.5 seconds (1500 milliseconds) and then acquires the resourceLock to close the resources by setting resourcesOpen to false.

2. Wait and Signal Code

Resource Class:

- This class manages the shared resource.
- It contains a boolean variable available to indicate whether the resource is available or not.
- The acquire() method is synchronized, which means only one thread can execute it at a time. It waits until the resource becomes available by calling wait() inside a loop.
- The release() method is synchronized as well. It marks the resource as available and notifies all waiting threads by calling notifyAll().

Process Class:

- This class represents a process that wants to use the resource.
- It takes a Resource object in its constructor.
- The run() method is overridden from the Thread class. Inside this method, the process acquires the resource by calling resource.acquire(). It then prints a message indicating that it has acquired the resource and simulates some work by sleeping for 2 seconds.
- After completing the work, the process releases the resource by calling resource.release() and prints a message indicating that it has released the resource.

Main Method:

- The main() method creates a single instance of the Resource class.
- It starts multiple Process threads, each with a reference to the same Resource object.
- The threads compete for access to the shared resource. When one thread acquires the resource, the other threads are blocked until the resource is released.

Thread Synchronization:

- By using the synchronized keyword in the acquire() and release() methods of the Resource class, thread synchronization is achieved. This ensures that only one thread can execute the critical section of code (acquiring or releasing the resource) at a time, preventing race conditions.
- The use of wait() and notifyAll() allows threads to efficiently wait for a condition to be satisfied (resource availability) and to be notified when the condition changes (resource release).

Output:

- When a thread acquires the resource, it prints a message indicating that it has acquired the resource.
- After completing its work and releasing the resource, it prints a message indicating that it has released the resource.

3. Producer and Consumer using Semaphore technique

Buffer Class:

- The Buffer class represents a shared buffer that stores data produced by the producer and consumed by the consumer.
- It contains the following instance variables:
- BUFFER_SIZE: An integer constant representing the size of the buffer.
- data: An integer array to store the data in the buffer.

- in and out: Pointers to keep track of the position for producing and consuming data in the buffer.
- mutex, empty, and full: Semaphores for mutual exclusion, signaling empty slots in the buffer, and signaling filled slots in the buffer, respectively.
- The constructor initializes the buffer and semaphores.

produce(int item) Method:

- The produce method is called by the producer to produce and add an item to the buffer.
- It acquires the empty semaphore to wait for an empty slot in the buffer.
- It then acquires the mutex semaphore to enter the critical section and modify the buffer safely.
- The item is added to the buffer, and the in pointer is updated.
- After updating the buffer, it releases the mutex semaphore to exit the critical section and releases the full semaphore to signal that a slot has been filled.

consume() Method:

- The consume method is called by the consumer to consume and remove an item from the buffer.
- It acquires the full semaphore to wait for a filled slot in the buffer.
- It then acquires the mutex semaphore to enter the critical section and modify the buffer safely.
- The item is consumed from the buffer, and the out pointer is updated.
- After updating the buffer, it releases the mutex semaphore to exit the critical section and releases the empty semaphore to signal that a slot has been emptied.
- The consumed item is returned.

Producer and Consumer Threads:

- The Producer and Consumer classes extend the Thread class and override the run() method.
- The Producer produces items by calling the produce method of the Buffer class.

- The Consumer consumes items by calling the consume method of the Buffer class.
- Each thread simulates a delay between producing or consuming items using Thread.sleep().

main() Method:

- The main method creates an instance of the Buffer class and Producer and Consumer threads.
- It starts the producer and consumer threads, allowing them to produce and consume items concurrently.

4. Reader and Writer using Semaphore technique

import java.util.concurrent.Semaphore;

- **This statement imports the Semaphore class from the java.util.concurrent package. Semaphores are synchronization primitives used to control access to resources in concurrent programming.**

class ReaderWriter {

- **This declares a class named ReaderWriter. This class is responsible for managing access to a shared resource by multiple readers and writers.**

private int readersCount;
private Semaphore mutex;
private Semaphore readerSem;
private Semaphore writerSem;

- **readersCount: An integer variable to keep track of the number of active readers.**
- **mutex: A semaphore for mutual exclusion to ensure that only one thread accesses shared data at a time.**
- **readerSem: A semaphore for controlling access by readers.**
- **writerSem: A semaphore for controlling access by writers.**

```
public ReaderWriter() {
    readersCount = 0;
    mutex = new Semaphore(1); // Semaphore for mutual exclusion
    readerSem = new Semaphore(1); // Semaphore for reader access
control
    writerSem = new Semaphore(1); // Semaphore for writer access control
}
```

- **This is the constructor of the ReaderWriter class. It initializes the instance variables:**
- **readersCount is set to 0.**
- **mutex, readerSem, and writerSem are initialized with semaphores, each with an initial permit count of 1.**

```
public void startRead() throws InterruptedException {
    readerSem.acquire(); // Wait for reader access
    mutex.acquire(); // Enter critical section
    readersCount++;
    if (readersCount == 1) {
        writerSem.acquire(); // Prevent writers from entering
    }
    mutex.release(); // Exit critical section
    readerSem.release(); // Allow other readers to access
}
```

- **This method is called by a reader thread to start reading.**
- **readerSem.acquire() blocks if there is a writer currently writing. Once the semaphore is acquired, it decrements the semaphore's permit count.**
- **mutex.acquire() ensures mutual exclusion, allowing only one thread at a time to increment readersCount.**
- **If the reader is the first reader (readersCount == 1), writerSem.acquire() is called to prevent writers from entering while readers are reading.**
- **The method releases the semaphores after updating readersCount.**

```
public void endRead() throws InterruptedException {
    mutex.acquire(); // Enter critical section
    readersCount--;
    if (readersCount == 0) {
        writerSem.release(); // Allow writers to enter
    }
    mutex.release(); // Exit critical section
}
```

- **This method is called by a reader thread to end reading.**
- **mutex.acquire() ensures mutual exclusion while updating readersCount.**
- **If there are no more active readers (readersCount == 0), writerSem.release() is called to allow writers to enter.**
- **The method releases the mutex semaphore after updating readersCount.**

```
public void startWrite() throws InterruptedException {
    writerSem.acquire(); // Wait for writer access
}
```

- **This method is called by a writer thread to start writing.**
- **writerSem.acquire() blocks if there are any readers or writers currently accessing the resource. Once the semaphore is acquired, it decrements the semaphore's permit count.**

```
public void endWrite() {
    writerSem.release(); // Release writer access
}
```

- **This method is called by a writer thread to end writing.**
- **writerSem.release() releases the semaphore, allowing other writers or readers to access the resource.**

Reader and Writer Threads:

- **These classes represent reader and writer threads, respectively. They override the run() method to perform their respective operations (reading or writing) using the methods provided by the ReaderWriter class.**

```
public static void main(String[] args) {
    ReaderWriter rw = new ReaderWriter();
    for (int i = 0; i < 5; i++) {
        new Reader(rw).start();
        new Writer(rw).start();
    }
}
```

- **This is the entry point of the program.**
- **It creates an instance of ReaderWriter.**
- **It starts multiple reader and writer threads to demonstrate concurrent access to the shared resource managed by the ReaderWriter class.**