

# Parallel Programming

## Exercise Session 11

# Outline

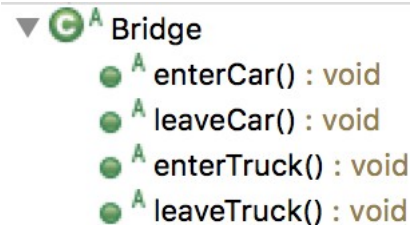
1. Feedback: Assignment 10
2. Assignment 11

# Feedback: Assignment 10


# Task 1 – Monitors, Conditions and Bridges









Only either 3 cars or one truck may be on the bridge at each moment.

Implement Classes BridgeMonitor and BridgeCondition



```
classDiagram
    class Bridge {
        +enterCar() void
        +leaveCar() void
        +enterTruck() void
        +leaveTruck() void
    }
```

▼  Bridge

-   enterCar() : void
-   leaveCar() : void
-   enterTruck() : void
-   leaveTruck() : void

How to Test my Implementation?

Implement method `invariant()` to check if the state is valid: at the end of a method there are never too many cars or trucks on the bridge

## Task 2 – BridgeMonitor

```
private int carCount = 0;
private int truckCount = 0;
private final Object monitor = new Object();
```

```
public void enterCar() throws InterruptedException {
    synchronized(monitor)
    {
        while (carCount >= 3 || truckCount >= 1) {
            monitor.wait();
        }
        carCount++;
    }
}
```

```
public void leaveCar() {
    synchronized (monitor) {
        carCount--;
        monitor.notifyAll();
    }
}
```

```
public void enterTruck() throws InterruptedException {
    synchronized (monitor) {
        while (carCount >= 1 || truckCount >= 1) {
            monitor.wait();
        }
        truckCount++;
    }
}
```

```
public void leaveTruck() {
    synchronized (monitor) {
        truckCount--;
        monitor.notifyAll();
    }
}
```

# Task 2 – BridgeCondition

```
final Lock bridgeLock = new ReentrantLock();
final Condition carCanEnter=bridgeLock.newCondition();
final Condition truckCanEnter=bridgeLock.newCondition();
```

```
public void enterCar() throws InterruptedException {
    bridgeLock.lock();
    try{
        while(numC>=maxC || numT>=maxT){
            carCanEnter.await();
        }
        numC++;
    }
    finally {
        bridgeLock.unlock();
    }
}
```

```
public void leaveCar() {
    bridgeLock.lock();
    try{
        numC--;
        if(numC==0)
            truckCanEnter.signalAll();
        carCanEnter.signalAll();
    }
    finally {
        bridgeLock.unlock();
    }
}
```

```
public void enterTruck() throws InterruptedException {
    bridgeLock.lock();
    try{
        while(numC>0 || numT>0){
            truckCanEnter.await();
        }
        numT++;
    }
    finally {
        bridgeLock.unlock();
    }
}
```

```
public void leaveTruck() {
    bridgeLock.lock();
    try{
        numT--;
        truckCanEnter.signalAll();
        carCanEnter.signalAll();
    }
    finally {
        bridgeLock.unlock();
    }
}
```

# Task 2 – Discussion

```
private AtomicInteger cars = new AtomicInteger();
private AtomicInteger trucks = new AtomicInteger();

public synchronized void enterCar() throws InterruptedException {
    while (cars.get() > 2 || trucks.get() > 0) { wait(); }
    cars.incrementAndGet();
}
```

```
try {
    bridgeLock.lock();
    carsOnBridge--;

    bridgeNotFull.signalAll();
}
finally {
    bridgeLock.unlock();
}
```

```
bridgeLock.lock();
while (carsOnBridge >= 3 || truckOnBridge) {
    condition.await();
}
carsOnBridge++;
bridgeLock.unlock();
```


```
public void leaveCar() {
    // TODO implement rules for car leave
    synchronized (monitor) {
        // while(carsOnBridgeA.get()<=0) {




        while (carsOnBridge <= 0) {
            try {
                monitor.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        // carsOnBridgeA.decrementAndGet();
        carsOnBridge--;
        monitor.notifyAll();
    }
}
```

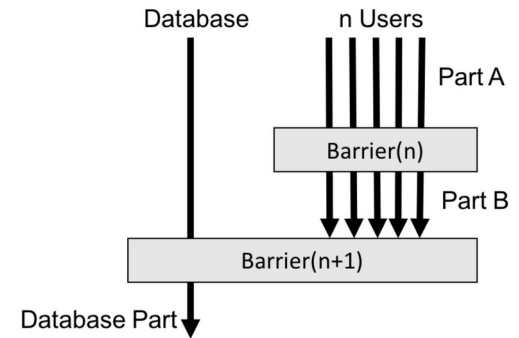
# Task 3 – Semaphores and Databases

Use semaphores to implement login and logout database functionality that supports up to 10 concurrent users

Use barrier to implement 2-phase backup functionality.

▼  Database

- MAX\_USERS : int
- activeUsers : Set<User>
-  login(User) : void
-  logout(User) : void
-  backup() : void





# Task 3 – MySemaphore

```
private volatile int count;

public MySemaphore(int maxCount) {
    count=maxCount;
}

public synchronized void acquire() throws InterruptedException {
    while(!(count>0)){
        wait();
    }
    count--;
}

public synchronized void release() {
    count++;
    notifyAll();
}
```

## Task 3 – MyBarrier

```
int current, max;
MyBarrier(int n){
    max = n;
    current = 0;
}

public synchronized void await() {
    if (++current < max)
        try {
            wait();
        } catch (InterruptedException e) {}
    else{
        notifyAll();
        current = 0;
    }
}
```

# Task 3 – Discussion

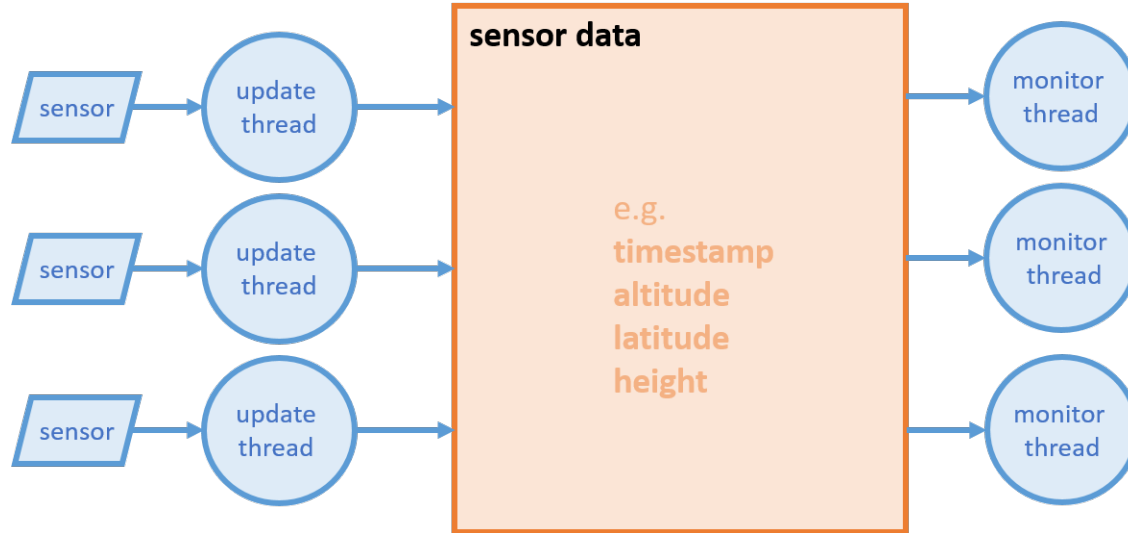
```
public void acquire() throws InterruptedException {  
    while (!(count > 0)) { Thread.sleep(1); }  
    synchronized (this) {  
        count--;  
    }  
}  
  
public void release() {  
    synchronized (this) {  
        count++;  
    }  
}
```

```
private int parties;  
  
MyBarrier(int n){  
    parties = n;  
}  
  
synchronized void await() throws InterruptedException {  
    parties--;  
    while (parties > 0) { wait(); }  
  
    notify();  
}
```

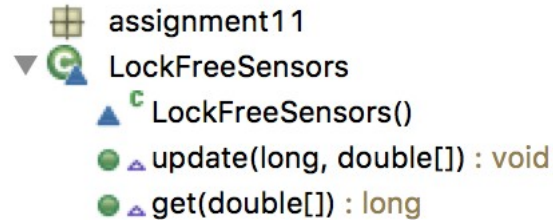
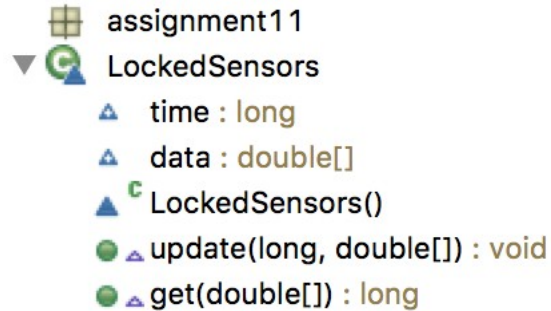
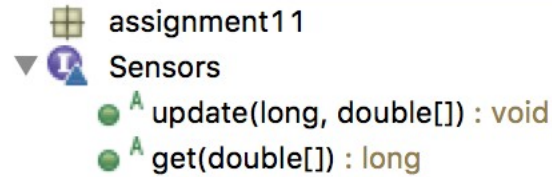
```
void await() throws InterruptedException {  
    // TODO implement the barrier await using Monitors  
    synchronized (monitor) {  
        count++;  
        while (count < n)  
            monitor.wait();  
        monitor.notifyAll();  
    }  
}
```

# Assignment 11

- Multisensor System.



# Multisensor System




# Multisensor System

assignment11

▼ Sensors


- <sup>A</sup> update(long, double[]) : void
- <sup>A</sup> get(double[]) : long

assignment11

▼  SensorData


- ▲ <sup>F</sup> data : double[]
- ▲ <sup>F</sup> timestamp : long
- ▲ <sup>C</sup> SensorData(long, double[])
- ▲ getValues() : double[]
- ▲ getTimestamp() : long

assignment11

▼  LockedSensors

- ▲ time : long
- ▲ data : double[]
- ▲ <sup>C</sup> LockedSensors()
- <sup>A</sup> update(long, double[]) : void
- <sup>A</sup> get(double[]) : long

assignment11

▼  LockFreeSensors

- ▲ <sup>C</sup> LockFreeSensors()
- <sup>A</sup> update(long, double[]) : void
- <sup>A</sup> get(double[]) : long

# Multisensor System

Implement two versions of the sensor data set:

- a) One blocking version based on a readers-writers lock (LockedSensors.java).
- b) A lock-free version (LockFreeSensors.java)

Hints:

- Before you implement the readers-writers lock based version, start with a simple locked version in order to understand. Then try a readers-writers lock but be aware that the Java-implementation does not give fairness guarantees. What can this imply? In any case, you have the code from the lecture slides presenting a fair RW-Lock implementation.
- The lock-free implementation solutions does NOT rely on mechanisms such as Double-Compare- And-Swap. Also it does not rely on a lazy update mechanism. Somehow you have to make sure that with a single reference update you change all data at once. How?

# Readers-writers lock

Recall:

- Multiple concurrent reads of same memory: *Not* a problem
- Multiple concurrent writes of same memory: Problem
- Multiple concurrent read & write of same memory: Problem

So far:

- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But this is unnecessarily conservative:

- Could still allow multiple simultaneous readers!



# Readers-writers lock

A new abstract data type for synchronization : The [readers/writer lock](#)

- A lock's states fall into three categories:
  - “not held”
  - “held for writing” by one thread
  - “held for reading” by *one or more* threads

$$0 \leq \text{writers} \leq 1$$

$$0 \leq \text{readers}$$

$$\text{writers} * \text{readers} == 0$$

# Readers-writers lock

<code>new:</code>	make a new lock, initially “not held”
<code>acquire_write:</code>	block if currently “held for reading” or “held for writing”, else make “held for writing”
<code>release_write:</code>	make “not held”
<code>acquire_read:</code>	block if currently “held for writing”, else make/keep “held for reading” and increment <i>readers count</i>
<code>release_read:</code>	decrement readers count, if 0, make “not held”

# Readers-writers lock in Java

```
double readSomething() {  
    readerWriterLock.readLock().lock();  
    try {  
        double value = retrieveDoubleValue();  
        return value;  
    } finally {  
        readerWriterLock.readLock().unlock();  
    }  
}
```

```
Void writeSomething(double new_value) {  
    readerWriterLock.writeLock().lock();  
    try {  
        storeDoubleValue(new_value);  
    } finally {  
        readerWriterLock.writeLock().unlock();  
    }  
}
```

No fairness guarantees!

# Readers-writers lock with monitors

```
class RWLock {  
    int writers = 0;  
    int readers = 0;  
  
    synchronized void acquire_read() {  
        while (writers > 0)  
            try { wait(); }  
        catch (InterruptedException e) {}  
        readers++;  
    }  
  
    synchronized void release_read() {  
        readers--;  
        notifyAll();  
    }  
}
```

```
    synchronized void acquire_write() {  
        while (writers > 0 || readers > 0)  
            try { wait(); }  
        catch (InterruptedException e) {}  
        writers++;  
    }  
  
    synchronized void release_write() {  
        writers--;  
        notifyAll();  
    }  
}
```

# Readers-writers lock with monitors

```
class RWLock {
    int writers = 0;
    int readers = 0;
    int writersWaiting = 0;

    synchronized void acquire_read() {
        while (writers > 0 || writersWaiting > 0)
            try { wait(); }
        catch (InterruptedException e) {}
        readers++;
    }

    synchronized void release_read() {
        readers--;
        notifyAll();
    }
}
```

```
synchronized void acquire_write() {
    writersWaiting++;
    while (writers > 0 || readers > 0)
        try { wait(); }
    catch (InterruptedException e) {}
    writersWaiting--;
    writers++;
}

synchronized void release_write() {
    writers--;
    notifyAll();
}
}
```

# Readers-writers lock with monitors

```
class RWLock{
    int writers = 0; int readers = 0;
    int writersWaiting = 0; int readersWaiting = 0;
    int writersWait = 0;

    synchronized void acquire_read() {
        readersWaiting++;
        while (writers>0 || (writersWaiting>0 && writersWait <= 0))
            try { wait(); }
        catch (InterruptedException e) {}
        readersWaiting --;
        writersWait--;
        readers++;
    }

    synchronized void release_read() {
        readers--;
        notifyAll();
    }
}
```

```
synchronized void acquire_write() {
    writersWaiting++;
    while (writers > 0 || readers > 0 || writersWait > 0)
        try { wait(); }
    catch (InterruptedException e) {}
    writersWaiting--;
    writers++;
}

synchronized void release_write() {
    writers--;
    writersWait = readersWaiting;
    notifyAll();
}
}
```

# Problems with locks

- **deadlock:** group of two or more competing processes are mutually blocked because each process waits for another blocked process in the group to proceed
- **livelock:** competing processes are able to detect a potential deadlock but make no observable progress while trying to resolve it
- **starvation:** repeated but unsuccessful attempt of a recently unblocked process to continue its execution

# Lock free

- **lock-freedom**: at least one algorithm makes progress even if other algorithms run concurrently.

Implies system-wide progress but not freedom from starvation.



- **wait-freedom**: all algorithms eventually make progress.

Implies freedom from starvation.



# Lock-free

```
Object readSomething() {  
    return atomicReference.get();  
}
```

```
Void writeSomething(Object new_object) {  
    Object old_object;  
    do {  
        old_object = atomicReference.get();  
        // Check if we want to overwrite the latest data (i.e. only write newer or better data)  
        if ( ... ) {  
            return;  
        }  
    } while (!atomicReference.compareAndSet(old_object, new_object));  
}
```