Multicore Computing

- Project 2 –

과목명	멀티코어 컴퓨팅
교수명	손봉수 교수님
제출일	2023.05.07
학 번	20200283
학 과	소프트웨어학부
이 름	조범희

INDEX

1. Ex1

- BlockingQueue and ArrayBlockingQueue

Java and C++ provide libraries for queues. These work well in single-threaded environments, but when multiple threads access the queue concurrently, race conditions may occur. For example, if two threads try to push onto the queue at the same time, or if two threads try to pop from an empty queue at the same time, bugs are likely to occur. To solve this problem, condition variables such as wait and notify can be used, but Java provides the BlockingQueue interface as a solution.

This interface resolves the previously identified issue of Race Condition and allows threads to access the queue safely. Internally, if the queue is full and a thread attempts to add more data, the interface will block the thread. When space becomes available in the queue, it will unblock and resume operation. Similarly, if the queue is empty and a thread attempts to retrieve data, it will also block the thread and resume operation once data is available in the queue.

ArrayBlockingQueue is a class that implements the interface. Unlike a typical queue, which does not initially set a capacity, ArrayBlockingQueue is array-based and therefore requires a size to be specified at the outset. If the size is exceeded, threads will be blocked internally. Data can be inserted into the queue using 'put' and values can be removed using 'take'.

- Example

Each thread shares a single ArrayBlockingQueue provided by the main class. They will wait inside the queue for a random amount of time before leaving. This process helps to understand how ArrayBlockingQueue works.

```
public class ex1 {
    public static void main(String[] args) throws InterruptedException {
        ArrayBlockingQueue<Integer> arrayBlockingQueue = new ArrayBlockingQueue<~>( capacity: 3);
        int NUM_THREAD = 6;
        My_Thread[] my_thread = new My_Thread[NUM_THREAD];
        for(int i = 0; i < NUM_THREAD; i++){
            my_thread[i] = new My_Thread(arrayBlockingQueue);
        }
        for(int i = 0; i < NUM_THREAD; i++){
            my_thread[i].join();
        }
        System.out.println("               All Thread Enter and Out.     ");
    }
}
```

```java
public void run() {

    try {
        sleep( millis: (long)(Math.random()) * 1500);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }

    try {
        arrayBlockingQueue.put( e: 0);
        System.out.println("    Hi " + this.getName() + " Enter the Queue. ");

        sleep((int)(Math.random() * 10000));
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }

    try {
        System.out.println("                                        Bye "+ this.getName() +" Out of Queue. ");
        arrayBlockingQueue.take();

        sleep((int)(Math.random() * 10000));
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

The threads access the shared queue through the 'put' and 'take' methods and wait for a random amount of time before exiting. If there is no space available in the queue, they will wait until space becomes available.

```
ex1
"C:\Program Files\Microsoft\jdk-11.0.12.7-hotspot\bin\java.exe" "-javaagent:C
 Files\JetBrains\IntelliJ IDEA 2022.1.1\bin" -Dfile.encoding=UTF-8 -classpath
    Hi Thread-0 Enter the Queue.
    Hi Thread-2 Enter the Queue.
    Hi Thread-5 Enter the Queue.
                                            Bye Thread-5 Out of Queue.
    Hi Thread-4 Enter the Queue.
                                            Bye Thread-4 Out of Queue.
    Hi Thread-3 Enter the Queue.
                                            Bye Thread-2 Out of Queue.
    Hi Thread-1 Enter the Queue.
                                            Bye Thread-3 Out of Queue.
                                            Bye Thread-0 Out of Queue.
                                            Bye Thread-1 Out of Queue.
                All Thread Enter and Out.

Process finished with exit code 0
```

Looking at the output, we can see that with a capacity of 3, threads 0, 2, and 5 enter the queue, and there are no new entries until thread 5 exits, allowing thread 4 to enter. After thread 4 exits, thread 3 enters, followed by thread 1 after threads 3 and 2 exit. The program terminates after all threads have exited the queue.

2. Ex2

- ReadWriteLock

ReadWriteLock is a feature specialized in reading and writing. In general, a normal lock can have only one thread at a time, whether it is read or written. ReadLock and WriteLock classify the situation, and ReadLock can manage the lock efficiently by dividing it into the case of writing and reading variables. WriteLock can have only one thread, but ReadLock can have multiple threads, because in the case of writing, if other threads write at the same time, they affect each other, but in the case of reading, there is no change in the data. So to get a WriteLock, no one else should have a WriteLock and not even try it, and there should be no ReadLock. This is because if someone writes the data they are reading, problems can occur.

As mentioned earlier, ReadLock can be held by multiple threads, but it is not always available to all threads. If a certain thread holds the WriteLock, it means that the data is currently being modified. If a ReadLock is granted at this point, it may read the data that is currently being modified, causing issues. Therefore, to obtain a ReadLock, there should be no threads holding the WriteLock.

- Example

In this example, threads that acquire WriteLock increment the cnt variable, while threads that acquire ReadLock read the current value of cnt. Each thread shares the ReadWriteLock provided by the main class, which manages thread data access through ReadLock and WriteLock. ReadLock is acquired using readLock.lock() and released using readLock.unlock(), while WriteLock is acquired using writeLock.lock() and released using writeLock.unlock().

```
void modify() {
    try {
        sleep((int)(Math.random() * 1000));
        System.out.println(getName() + " : Try Get WriteLock ");
        readWriteLock.writeLock().lock();
        System.out.println(getName() + " : Get the WriteLock, Now count : " + cnt);
        cnt++;
        sleep((int)(Math.random() * 1000));
        System.out.println(getName() + " : Lose the WriteLock");
        readWriteLock.writeLock().unlock();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

Try to get WriteLock and get Lock. If another thread has the Lock here, I'll be on standby. Gets the lock, waits for a while, and returns the lock.

```
void getData() {
    try {
        sleep((int)(Math.random() * 1000));
        System.out.println("                                                    " + getName() + " : Try Get ReadLock ");

        readWriteLock.readLock().lock();
        System.out.println("                                                    " + getName() + "  Get the ReadLock, Now count :" + cnt);
        sleep((int) (Math.random() * 1000));
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    readWriteLock.readLock().unlock();
}
```

Try to obtain Read Lock.Here, you can get Lock, but if other trash can get a Lock, but if you have a Write Lock.

```
Thread-2 : Try Get WriteLock
Thread-1 : Try Get WriteLock
Thread-0 : Try Get WriteLock
Thread-3 : Lose the WriteLock
Thread-2 : Get the WriteLock, Now count : 6
Thread-2 : Lose the WriteLock
Thread-1 : Get the WriteLock, Now count : 7
                                                    Thread-3 : Try Get ReadLock
Thread-1 : Lose the WriteLock
Thread-0 : Get the WriteLock, Now count : 8
Thread-0 : Lose the WriteLock
                                                    Thread-3  Get the ReadLock, Now count :9
                                                    Thread-2 : Try Get ReadLock
                                                    Thread-2  Get the ReadLock, Now count :9
                                                    Thread-1 : Try Get ReadLock
                                                    Thread-1  Get the ReadLock, Now count :9
Thread-3 : Try Get WriteLock
                                                    Thread-0 : Try Get ReadLock
Thread-2 : Try Get WriteLock
Thread-3 : Get the WriteLock, Now count : 9
Thread-1 : Try Get WriteLock
Thread-3 : Lose the WriteLock
                                                    Thread-0  Get the ReadLock, Now count :10
Thread-2 : Get the WriteLock, Now count : 10
Thread-2 : Lose the WriteLock
Thread-1 : Get the WriteLock, Now count : 11
Thread-1 : Lose the WriteLock
                    final Count : 12

Process finished with exit code 0
```

WriteLock can only be acquired by one thread at a time. Therefore, in the output, you can see that after attempting to acquire the WriteLock, the thread waits for a while, and only one thread at a time is allowed to modify the value, resulting in different Now Count values being printed. In the case of ReadLock, if another thread holds the WriteLock, the thread waits, and since multiple threads can hold the ReadLock, the same Now Count value can be printed. Finally, the program prints the final Count and terminates.

3. Ex3

- Atomic Interger

AtomicInteger provides thread-safety for shared variables by ensuring that unexpected changes in values do not occur through the use of locks, similar to how ex2 managed shared variables through ReadWriteLock and individual locks. AtomicInteger guarantees thread-safety through the Compare and Swap approach, which processes operations only if the expected value matches the actual value and fails otherwise. CAS operations are processed by the CPU, making them more efficient than using locks.

The main methods of AtomicInteger are set, get, addAndGet, and getAndAdd. Set initializes the AtomicInteger with a starting value. Get returns the current value of the AtomicInteger. addAndGet and getAndAdd both add a specified value to the variable, but they differ in their return values. addAndGet returns the result of the addition, while getAndAdd returns the original value and then adds the specified value to the variable.

- Exeample

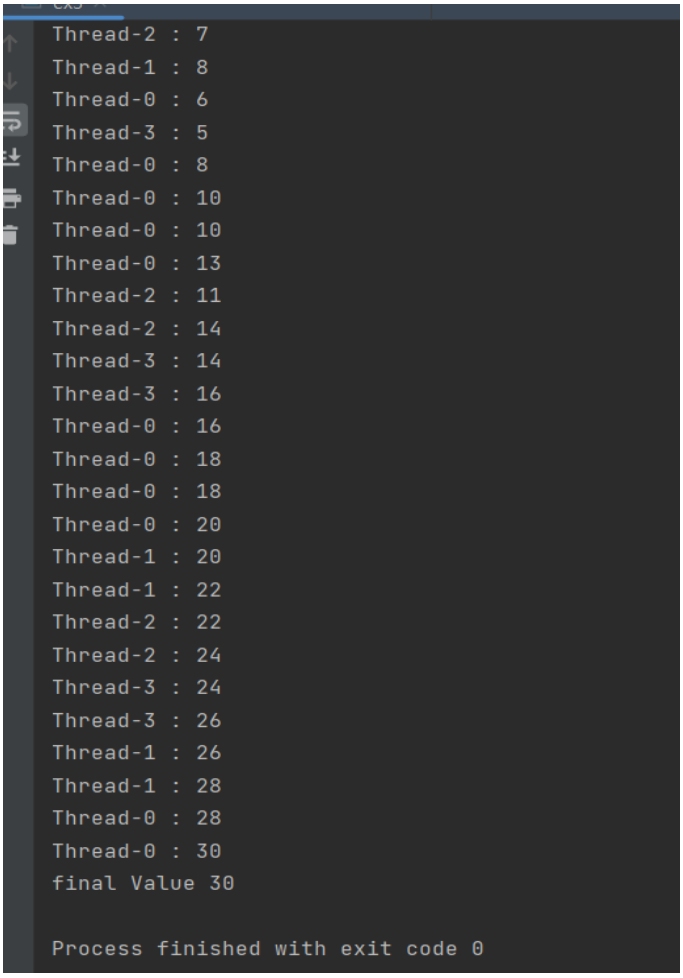
In this example, threads share Atomic Integer and access it at the same time in real time. Each thread checks what is different between getAndAdd, addAndGet methods, and when the number is reached, the program ends.

```
public class ex3 {
    public static void main(String[] args) throws InterruptedException {
        int ATOMIC_NUM = 0;
        AtomicInteger atomicInteger = new AtomicInteger();
        atomicInteger.set(ATOMIC_NUM);

        int NUM_TREAD = 4;
        ex3_Thread[] ex3_threads = new ex3_Thread[NUM_TREAD];

        for(int i = 0; i < NUM_TREAD; i++)
            ex3_threads[i] = new ex3_Thread(atomicInteger);

        for(int i = 0; i < NUM_TREAD; i++)
            ex3_threads[i].join();
        System.out.println("final Value " + atomicInteger.get());
    }
}
```

The thread set the initial value of the four Atomic Integer as a set.

```java
@Override
public void run() {
    while (atomicInteger.get() < 30) {
        try {
            System.out.println(getName() + " : " + atomicInteger.getAndAdd( delta: 1));
            System.out.println(getName() + " : " + atomicInteger.addAndGet( delta: 1));

            sleep((int) (Math.random() * 1000));
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
```

Threads each approach atomic Integer, wait for a while and run to the target number. Looking at the results, there are cases where the same value is printed twice. Because if the current data was 10, thread 1 would output 11 if it did and AndGet. After that, if the second thread immediately gotAndAdd, 11 is outputted twice because 11 is output and the value is added.

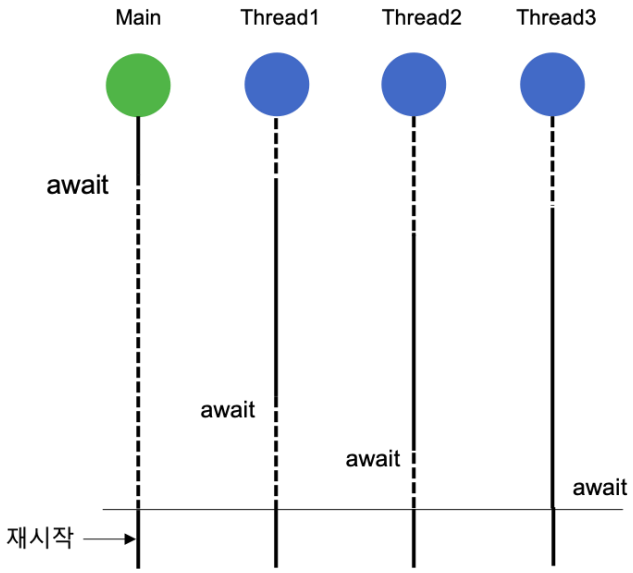


4. Ex4

- CyclicBarrier

CyclicBarrier is used for synchronization in a multi-threaded environment. It uses the await() method to put threads on hold, and unlike other Condition Variables, it is awakened when a specified number of threads are gathered, even without a separate signal. It is used when each task needs to wait for the others to complete, and after use, the CyclicBarrier returns to its initial state, making it reusable. Although CyclicBarrier can manage race conditions in shared variables, it is primarily used for thread management.

CyclicBarrier has one disadvantage: if, for example, four threads need to arrive to be awakened, and one of them encounters a problem and does not arrive, the remaining three threads can wait indefinitely. To prevent this, a timeout value should be set.



- Exeample

The number of threads is eight, and each thread shares the cyclicBarrier declared in the main. The CyclicBarrier parameter was set to 2, which means that two threads must arrive before restarting. Since this example has 8 threads, CyclicBarrier will collect a total of 4 threads.

```java
public class ex4 { // CyclicBarrier
    public static void main(String[] args) throws InterruptedException {
        CyclicBarrier cyclicBarrier = new CyclicBarrier( parties: 2);
        int THREAD_NUM = 8;
        ex4_Thread[] ex4_threads = new ex4_Thread[THREAD_NUM];
        for(int i = 0; i < THREAD_NUM; i++){
            ex4_threads[i] = new ex4_Thread(cyclicBarrier);
        }
        for(int i = 0; i < THREAD_NUM; i++){
            ex4_threads[i].join();
        }
    }
}
```

Declare threads and assign cyclicBarrier

```java
@Override
public void run() {
    try {
        sleep((long)(Math.random() * 1000));
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    try {
        System.out.println(getName() + " CyclicBarrier reach.");
        cyclicBarrier.await();
    } catch (InterruptedException | BrokenBarrierException e) {
        throw new RuntimeException(e);
    }
    System.out.println("                                                    Two Thread's reach, CyclicBarrier Finish!");
}
```

Wait a moment before entering the wait() state. After that, the circulation barrier ends when a fixed number of threads come in.

```
ex4
 "C:\Program Files\Microsoft\jdk-11.0.12.7-hotspot\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\Int
  Files\JetBrains\IntelliJ IDEA 2022.1.1\bin" -Dfile.encoding=UTF-8 -classpath "C:\Users\tiger\OneDrive\바탕
Thread-6 CyclicBarrier reach.
Thread-4 CyclicBarrier reach.
                                                Two Thread's reach, CyclicBarrier Finish!
                                                Two Thread's reach, CyclicBarrier Finish!
Thread-3 CyclicBarrier reach.
Thread-7 CyclicBarrier reach.
                                                Two Thread's reach, CyclicBarrier Finish!
                                                Two Thread's reach, CyclicBarrier Finish!
Thread-1 CyclicBarrier reach.
Thread-5 CyclicBarrier reach.
                                                Two Thread's reach, CyclicBarrier Finish!
                                                Two Thread's reach, CyclicBarrier Finish!
Thread-2 CyclicBarrier reach.
Thread-0 CyclicBarrier reach.
                                                Two Thread's reach, CyclicBarrier Finish!
                                                Two Thread's reach, CyclicBarrier Finish!

Process finished with exit code 0
```

When two threads arrive, a finish print appears for each thread, indicating that the CyclicBarrier is working well.