

Student's name: Nguyễn Thanh Tùng

Class: ICT04-K62

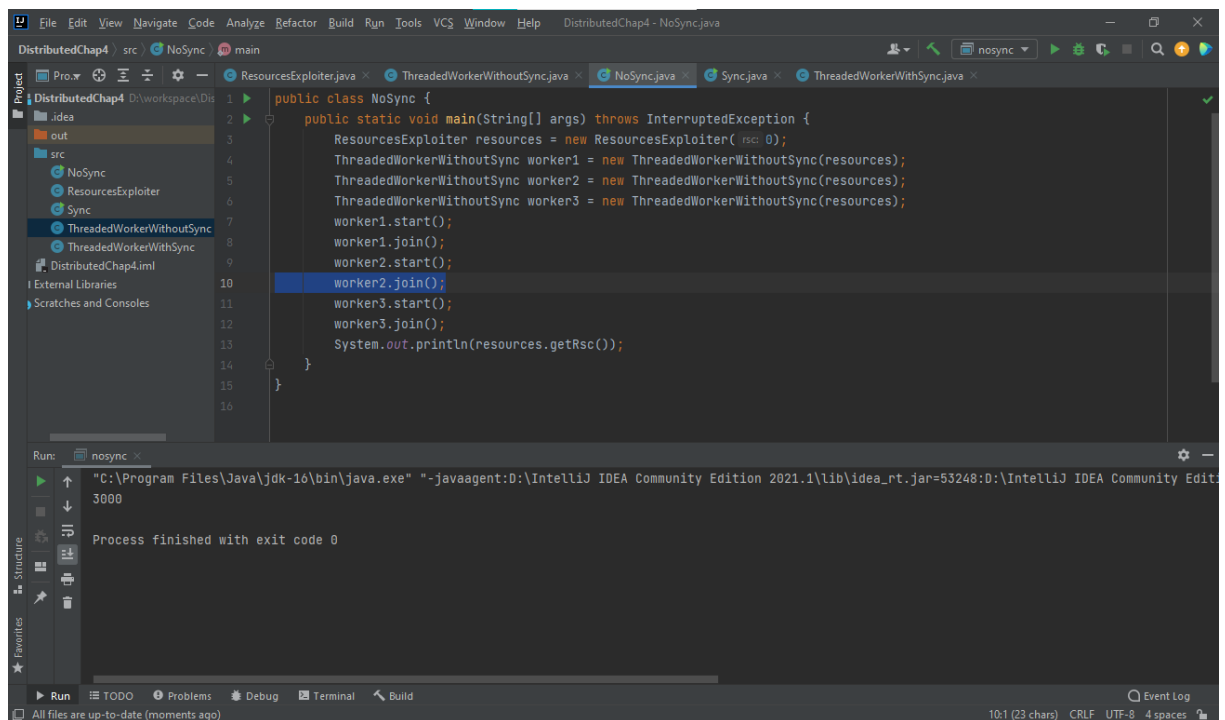
LABWORK

COURSE: DISTRIBUTED SYSTEMS

CHAPTER 4: SYNCHRONIZATION

Question 1:

We can see that if we apply the `join()` function for all three workers, the result of the `rsc` variable will be guaranteed to be 3000. This is because the `join()` function will act as a lock to make sure that the worker 2 only run after the worker 1 has finished its job, same as for the worker 3 and worker 2.



The screenshot shows the IntelliJ IDEA IDE with a project named 'DistributedChap4'. The main file, 'NoSync.java', contains the following code:

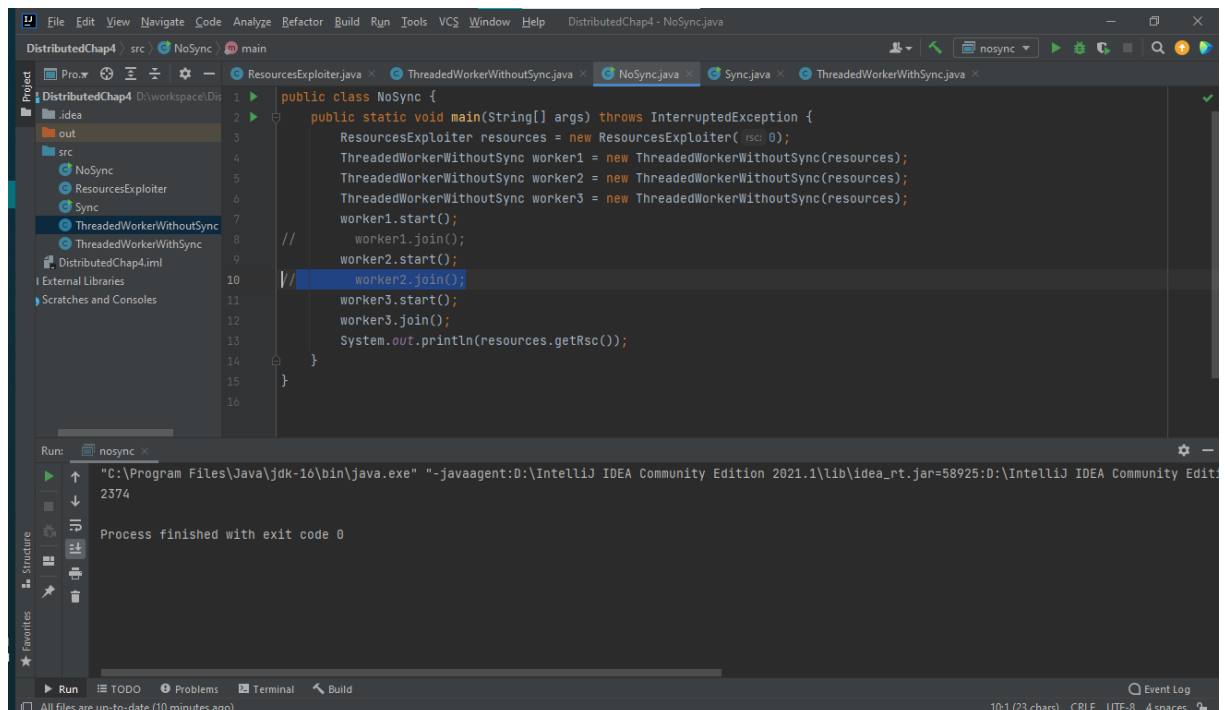
```
1 public class NoSync {
2     public static void main(String[] args) throws InterruptedException {
3         ResourcesExploiter resources = new ResourcesExploiter(1000);
4         ThreadedWorkerWithoutSync worker1 = new ThreadedWorkerWithoutSync(resources);
5         ThreadedWorkerWithoutSync worker2 = new ThreadedWorkerWithoutSync(resources);
6         ThreadedWorkerWithoutSync worker3 = new ThreadedWorkerWithoutSync(resources);
7         worker1.start();
8         worker1.join();
9         worker2.start();
10        worker2.join();
11        worker3.start();
12        worker3.join();
13        System.out.println(resources.getRsc());
14    }
15 }
16 }
```

The 'Run' output window shows the following execution details:

```
Run: nosync
"C:\Program Files\Java\jdk-16\bin\java.exe" "-javaagent:D:\IntelliJ IDEA Community Edition 2021.1\lib\idea_rt.jar=53248:D:\IntelliJ IDEA Community Edition 2021.1\bin" -Dfile.encoding=UTF-8
3000
Process finished with exit code 0
```

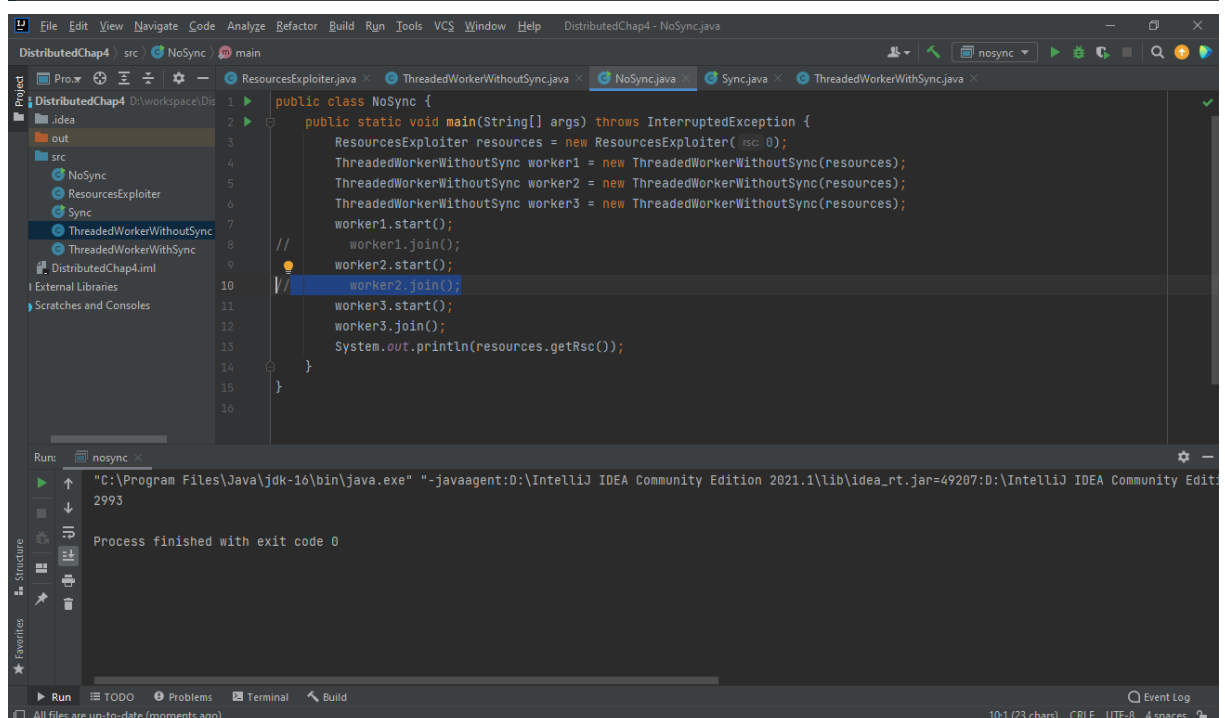
However, if we only make worker 3 join() so that the `System.out.println()` function get the final result of the `rsc` variable, it's not guaranteed that the result we get will be 3000

as expected, this value may vary from time to time depending on each run.



```
public class NoSync {  
    public static void main(String[] args) throws InterruptedException {  
        ResourcesExploiter resources = new ResourcesExploiter(1000);  
        ThreadedWorkerWithoutSync worker1 = new ThreadedWorkerWithoutSync(resources);  
        ThreadedWorkerWithoutSync worker2 = new ThreadedWorkerWithoutSync(resources);  
        ThreadedWorkerWithoutSync worker3 = new ThreadedWorkerWithoutSync(resources);  
        worker1.start();  
        worker1.join();  
        worker2.start();  
        worker2.join();  
        worker3.start();  
        worker3.join();  
        System.out.println(resources.getRsc());  
    }  
}
```

Run: nosync
"C:\Program Files\Java\jdk-16\bin\java.exe" "-javaagent:D:\IntelliJ IDEA Community Edition 2021.1\lib\idea_rt.jar=58925:D:\IntelliJ IDEA Community Edition 2021.1\bin" -Dfile.encoding=UTF-8
2374
Process finished with exit code 0



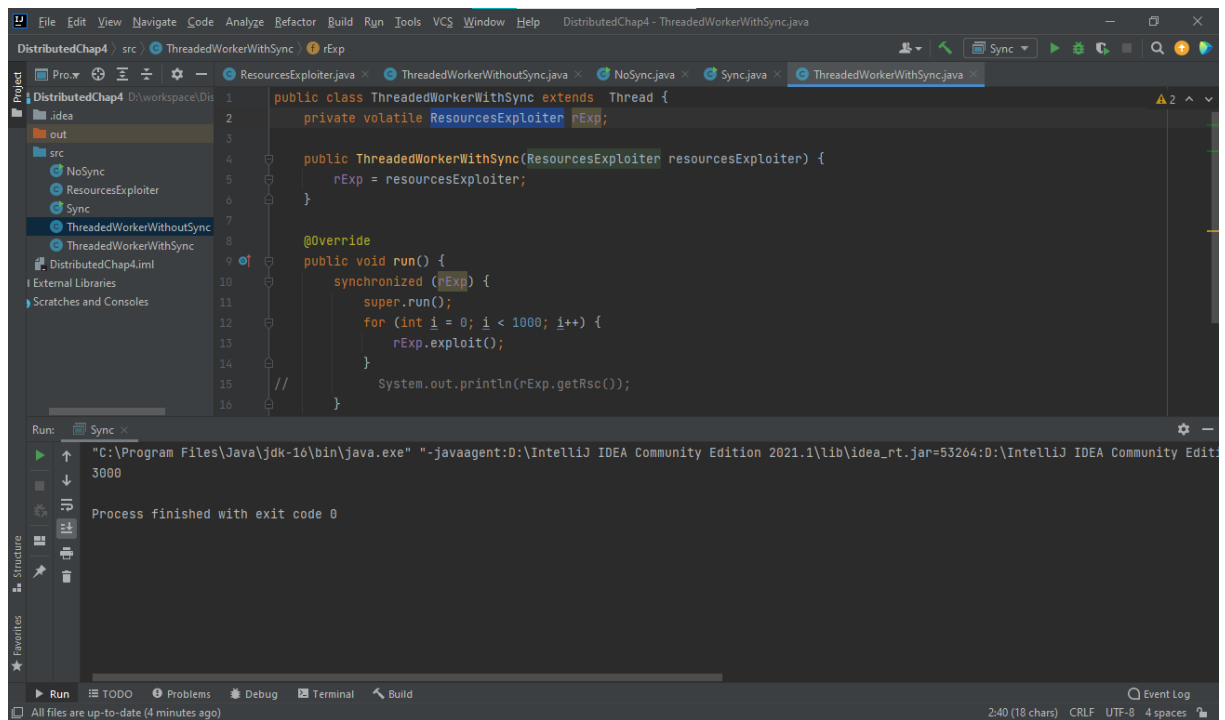
```
public class NoSync {  
    public static void main(String[] args) throws InterruptedException {  
        ResourcesExploiter resources = new ResourcesExploiter(1000);  
        ThreadedWorkerWithoutSync worker1 = new ThreadedWorkerWithoutSync(resources);  
        ThreadedWorkerWithoutSync worker2 = new ThreadedWorkerWithoutSync(resources);  
        ThreadedWorkerWithoutSync worker3 = new ThreadedWorkerWithoutSync(resources);  
        worker1.start();  
        worker1.join();  
        worker2.start();  
        worker2.join();  
        worker3.start();  
        worker3.join();  
        System.out.println(resources.getRsc());  
    }  
}
```

Run: nosync
"C:\Program Files\Java\jdk-16\bin\java.exe" "-javaagent:D:\IntelliJ IDEA Community Edition 2021.1\lib\idea_rt.jar=49207:D:\IntelliJ IDEA Community Edition 2021.1\bin" -Dfile.encoding=UTF-8
2993
Process finished with exit code 0

This is because the rsc is changed by 3 workers at the same time, there are no locking mechanism to protect the rsc to make sure it's working as expected.

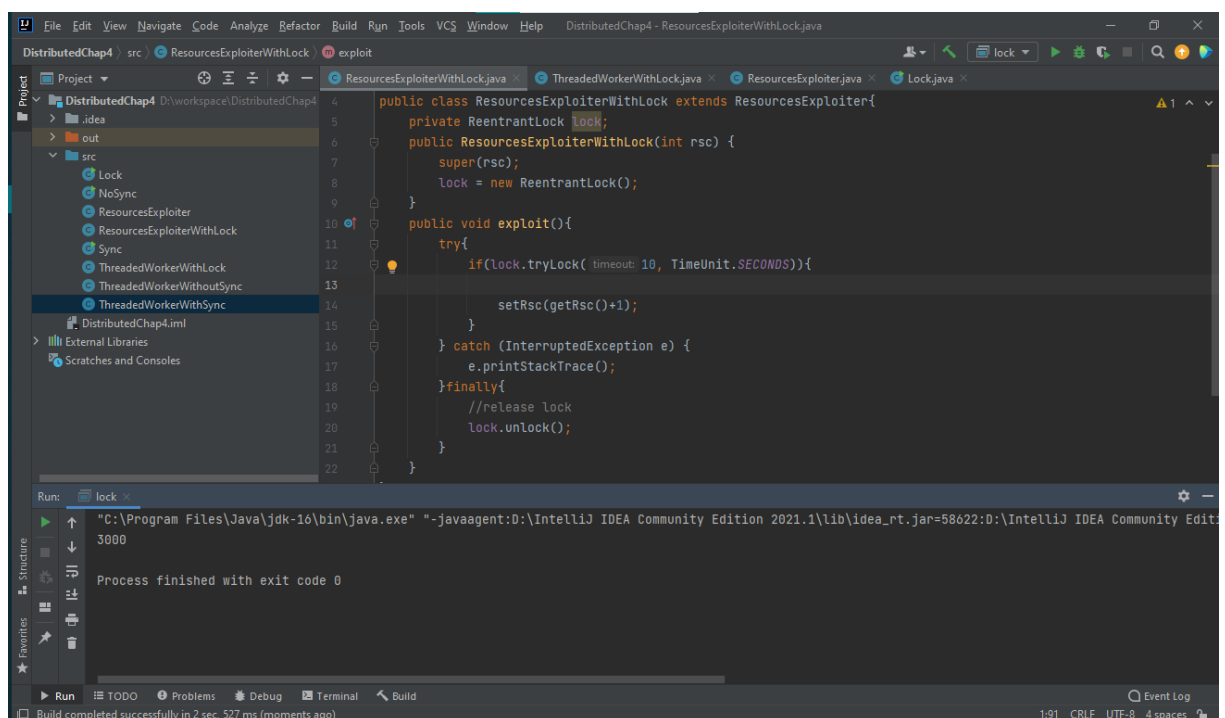
Question 2:

By applying the synchronization mechanism to the program, we can assure that the result we get will be 3000 because now the Resource Exploiter will be accessed synchronously, this means that even without the join() function, no 2 workers can access the rsc at the same time, each worker will have to wait for other until they can change the rsc.



Question 3:

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. As the name says, ReentrantLock allows threads to enter into the lock on a resource more than once. When the thread first enters into the lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlocks request, hold count is decremented by one and when hold count is 0, the resource is unlocked.



tryLock(long timeout, TimeUnit unit): As per the method, the thread waits for a certain time period as defined by arguments of the method to acquire the lock on the resource before exiting. This method helps protecting the rsg of concurrent access from the workers. Because of this locking mechanism, the result we get is guaranteed to be 3000.

Question 4:

```
longt@longt-ThinkPad-T430:~/Documents/DistributedSystem/Lab04$ gcc -pthread simp
le.c && ./a.out
11
12
13
14
15
shared: 15
longt@longt-ThinkPad-T430:~/Documents/DistributedSystem/Lab04$
```

Question 5:

Without increase the number of threads and value of NUM_TRANS, I can already see the different between balance and the equation $INIT_BALANCE + credits - debits$. This occurrence happened because some of the loop function in our 5 threads access the same resource balance or debits, credits concurrently without waiting for other thread to finish its execution on the so-called resources.

```
longt@longt-ThinkPad-T430:~/Documents/DistributedSystem/Lab04$ gcc -pthread without-lock.c && ./a.out 5
Credits:      4624
Debits:      3436

50+4624-3436=  1238
Balance:      1254
```

Question 6:

```
Expect: 10000
longt@longt-ThinkPad-T430:~/Documents/DistributedSystem/Lab04$ gcc -pthread naive-lock.c && ./a.out 100
Shared: 9956
Expect: 10000
longt@longt-ThinkPad-T430:~/Documents/DistributedSystem/Lab04$ gcc -pthread naive-lock.c && ./a.out 100
```

We use lock variable with the hope to identify that only 1 specific thread can access to resources, in this case, shared variable, but the difference still occurred since there will be a time that 2 or more thread that accessed to lock variable when its value equals to 0 and increase shared at the same time, continue the loop, so that the shared cannot exceed number of loop multiply with 100, the difference happened at line `shared++` which is `shared = shared + 1`, the shared on the right hand side will be, for example 2 concurrence equation `shared = shared + 1` and `shared = shared + 1`, if it happened at the same time, shared on the right hand side of the 2 line of code will have the same value thus assign the same value to the new shared, then 2 equations this time will only have effect as 1 equation → shared variable is 9956 when it is expected to be 10000.

Question 7:

```
Balance: 11909
longt@longt-ThinkPad-T430:~/Documents/DistributedSystem/Lab04$ gcc -pthread mutex-lock-banking.c && ./a.out 5
RUN 0:
    Credits:    19407
    Debits:     7548

50+19407-7548= 11909
    Balance:    11909
```

After using mutex for locking critical sections, the experiments show no difference between balance and its expected value → The improvement can be seen.

Question 8:

```
longt@longt-ThinkPad-T430:~/Documents/DistributedSystem/Lab04$ gcc -pthread fine_locking_bank.c && ./a.out 5
RUN 0:
    Credits:    14162
    Debits:     2764

50+14162-2764= 11448
    Balance:    11448
```

Without experiment on large resources of code, we can predict that fine locking strategy give a faster runtime since its decrease waiting time of other thread when access to a specific amount of resource. But in the example code, since there are only 2 type of resources which is balance and credits versus balance and debits, the runtime of fine locking is slightly slower than coarse due to the lock and unlock operations.

Question 9:

```
}
^C
longt@longt-ThinkPad-T430:~/Documents/DistributedSystem/Lab04$ gcc -pthread deadlocks-test.c && ./a.out
```

In the experiment we cannot see the output since the two thread are blocking each other by the mutex lock_a and mutex lock_b, thread 1 lock mutex a and thread 2 lock mutex b cause the other thread cannot keep its operation going since thread 1 need mutex b of thread 2 is locking and thread 2 need mutex a of thread 1 is locking → deadlock.