Concurrency and shared state

- 문제가 무엇인가? → 코루틴이 주는 해결책은 무엇인가?

1. Shared state problem : 문제를 보자.
2. **Solution1 : Atomic variables**
3. **Solution2 : Thread confinement**
4. **Solition3 : Mutex**

Shared State

문제는?

코루틴은 공유 상태 변수를 업데이트 할 수 있다. 코루틴은 자체적으로 스레딩을 핸들링 해서 적절하게 값을 업데이트 할 수 있다.

Lost Update 가 있을 수 도 있음. → 두 개의 코루틴이 동시에 값을 증가시키는 등.

```
import kotlinx.coroutines.*
import kotlin.system.measureTimeMillis

fun main() {
    var counter = 0
    runBlocking {
        withContext(
            Dispatchers.Default
        ){
            massiveRun{ counter++ }
        }
    }
    println("Counter = $counter")
}

suspend fun massiveRun(action : suspend () -> Unit){
    val n = 100
    val k = 1000

    val time = measureTimeMillis {
        coroutineScope{
            repeat(n){
                launch {
                    repeat(k){
                        action()
```

```
                    }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

결과:

```
Completed 100000 actions in 12 ms
Counter = 37368
```

→ 100만번 카운터 변수를 증가시켜야 하는데 업데이트가 Conflict 됨. State 를 제대로 매니징 하지 못해서.

Solution1 : Atomic Variables

atomic → 여러개의 코루틴이 동시에 업데이트 할 경우 각각 기다리고 순서에 맞게 업데이트 해야함.

primitive 데이터 타입과 콜렉션에 잘 적용된다.

```
import kotlinx.coroutines.*
import java.util.concurrent.atomic.AtomicInteger
import kotlin.system.measureTimeMillis

fun main() {
    runBlocking {
        var counter = AtomicInteger(0)
        withContext(Dispatchers.Default){
            massiveRunAtomic{counter.incrementAndGet()}
        }
        println("Counter = $counter")
    }
}

suspend fun massiveRunAtomic(action : suspend () -> Unit){
    val n = 100
    val k = 1000

    val time = measureTimeMillis {
        coroutineScope{
            repeat(n){
                launch {
                    repeat(k){
                        action()
```

```
                    }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

결과 :

```
Completed 100000 actions in 12 ms
Counter = 100000
```

- **단점 : 커스텀 클래스의 경우 Atomic implementation 을 제공하지 않는 이상 사용하기 어려울 수 있다.**

Solution 2 : Thread Confinement

- 데이터에 대한 접근은 하나의 스레드로 한다.
- shared state 에 대한 update 는 하나의 스레드로.

1. **Fine grained : 각 increment 는 컨텍스트를 훨씬 느리게 스위칭 한다.**
2. **Coarse-grained : 함수 전체가 하나의 스레드에서 돌아가는 경우. 컨텍스트 스위칭이 없어서 더 빠르다.**

Fine-grained

```
import kotlinx.coroutines.*
import kotlin.system.measureTimeMillis

fun main() {
    runBlocking {
        val counterContext = newSingleThreadContext("CounterContext")
        var counter = 0
        withContext(Dispatchers.Default){
            massiveRunConfined{
                withContext(counterContext){
                    counter++
                }
            }
```

```
        }
        println("Counter = $counter")
    }
}

suspend fun massiveRunConfined(action : suspend () -> Unit){
    val n = 100
    val k = 1000

    val time = measureTimeMillis {
        coroutineScope{
            repeat(n){
                launch {
                    repeat(k){
                        action()
                    }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

결과:

```
Completed 100000 actions in 367 ms
Counter = 100000
```

- **매우 느리다.**

Coarse-grained

```
fun main() {
    runBlocking {
        val counterContext = newSingleThreadContext("CounterContext")
        var counter = 0
        withContext(counterContext){
            massiveRunConfined{
//                withContext(counterContext){
                  counter++
//                }
            }
        }
        println("Counter = $counter")
        //단점 : 모든것이 serial 하게 처리되어서 병렬 처리가 되지 않는다.
    }
}
```

결과:

```
Completed 100000 actions in 17 ms
Counter = 100000
```

- **훨씬 더 빠르지만 병렬처리가 안된다는 단점이 있음.**

Solution 3 : Mutual Exclusion Locks

sensitive part of code 를 lock 한다. 접근하지 못하게 함.

코루틴은 mutex 를 lock, unlock 할 수 있다.

`withLock{ ... }` **메서드는 lock, unlock 기능을 제공한다.**

```
fun main() {
    runBlocking {
        val mutex = Mutex()
        var counter = 0
        withContext(Dispatchers.Default){
            //병렬 처리가 가능하다.
            massiveRunMutex{
                mutex.withLock {
                    //lost update 없음
                    counter++
                }
            }
        }
        println("Counter = $counter")
    }
}
```

결과 :

```
Completed 100000 actions in 161 ms
Counter = 100000
```