Parallel and Concurrent Programming with Java

## Chapter 3. Mutual Exclusion

# Data Race

Concurrent threads may have dependencies that interfer with one another.

Occurs when

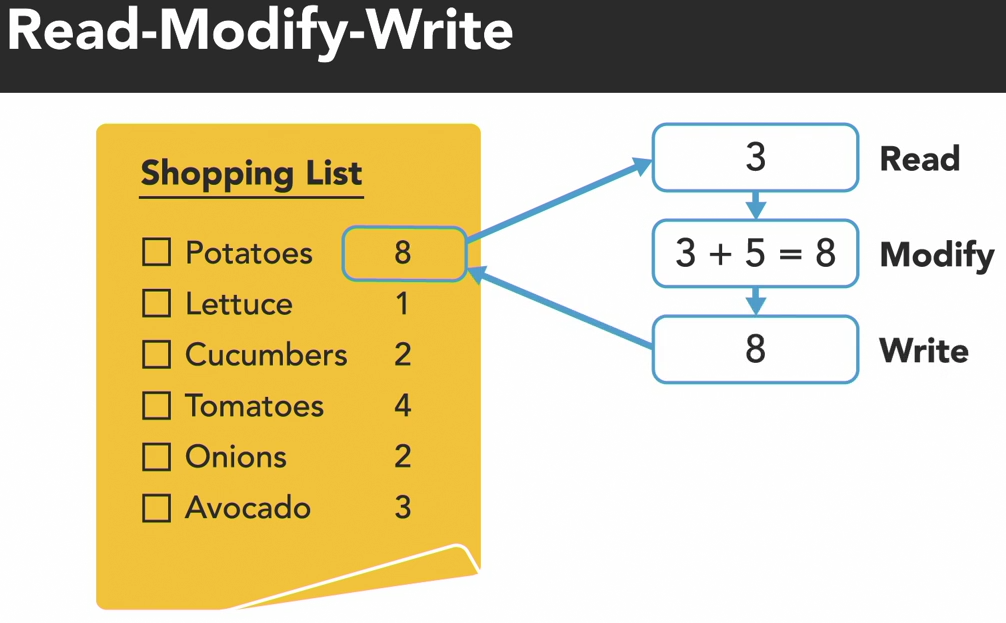
- 2 or more concurrent threads access the same memory location

- at least 1 is writing to it.

Need to recognize when data race can happen.

Protect with synchronisation techniques.

Each operation is a multi-step process.

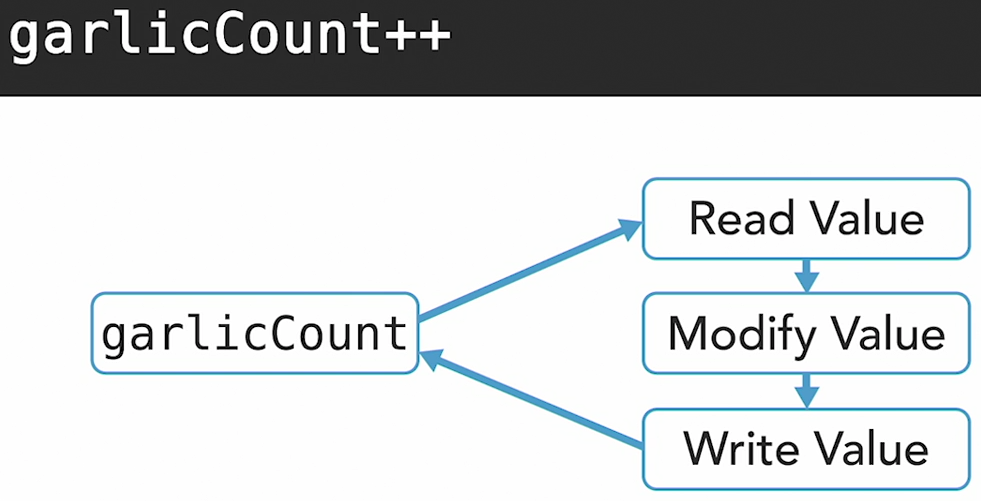
 - Example: Changing value involves read -> modify -> write.

Possible situation:

Barron’s thread is waiting/blocked just after reading value from shared memory. Olivia’s thread read, modify and write new data. When Barron’s thread resumes, it will not read changes made by Olivia’s thead. It will continue modify and write based on outdated data read earlier and stored in local memory.

OS’s scheduler determines which thread gets executed by processor. Unpredictable.

# Data Race: Java Demo

*/src*/DataRaceDemo

2 concurrent threads garlicCount++. Each 10 millions iterations. Output does not add up to 20 millions because of data race.

Chance of data race increases with frequency of data access.

Some programming languages have tools to detect data races. For Java,

- ThreadSafe. Static analysis tool for concurrency bugs. http://www.contemplateltd.com/threadsafe

- JvisualVM. Connect during error state and inspect threads and objects. https://visualvm.github.io/

- FindBugs. Old. http://findbugs.sourceforge.net/

Outcome changes with each test run, so data races are difficult to replicate when testing.

* Log info (system.out.println) for evidence, to prove/disprove theories. Or use in memory traces.
* Deadlocks. Look at stack trace when deadlocked.
  + When testing, hard to acquire same locks in different order.
* Livelocks. Harder. Use one of the tools to observe system when in error state.
* Hardware changes may affect, e.g. number of CPU cores, pipelines, bus bandwidths.

Good code design minimize threading/concurrency problems.

* Use immutable objects.
* Try to isolate mutable objects to a single thread.
* If not, carefully control access to modify/write mutable object.
  + Design based on object hand-over, rather than sharing.
  + If “share” object, use synchronization techniques.
    - Only modify shared state in a critical section of code (mutual exclusion).
    - Acquire locks in a set order, and release them in th eopposite order.
    - Be careful of assignment of shared variables, or operators like ++, where changes are made to same memory location.
    - Avoid having a synchronized object use other objects that must be synchronized too.
* Keep codes simple.
* Use prebuilt abstractions, java.util.concurrent package. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/concurrent/package-summary.html

# Mutual Exclusion

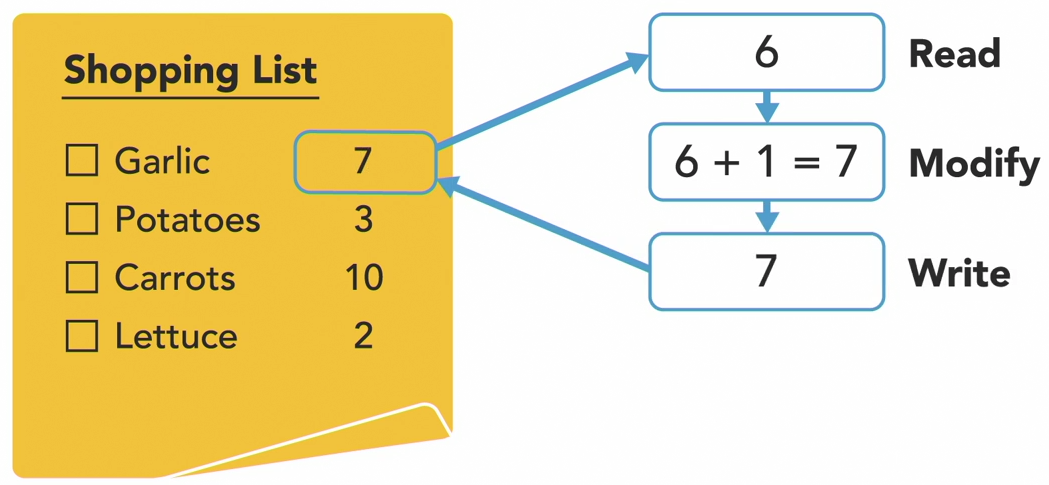
Protect against incorrect behavior of program by limiting access to critical section of code.

Critical section

- code segment that accesses a shared resource

* e.g. data structure memory or I/O device
* will not operate correctly if multiple threads concurrently accessing it

- executed by only 1 thread at a time

- read -> modify -> write are steps in an operation that must run uninterrupted.

Mutex(mutual exclusion, lock)

- analogy: pencil

- provides exclusive access to critical section

- only 1 thread can acquire a lock on the shared resource at a time. Exceptions exist, e.g. read lock of ReadWriteLock.

Atomic operation

- the operation to acquire lock is an atomic operation

- performed as a single unit of work

- not possible to be interrupted by concurrent threads

- may actually involve a few steps

Acquiring a lock

- if lock taken, thread wait/blocked until lock available

Critical section protected by mutex must be as short as possible

- other concurrent threads waiting/blocked while 1 thread executing critical section codes.

- do not have sleep/wait codes within critical section.

# Mutual Exclusion: Java Demo

package java.util.concurrent.locks

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/package-summary.html

Atomic variable: Java Demo

package java.util.concurrent.atomic

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

Java package for a small toolkit of classes, used to create atomic instances.

- examples: AtomicInteger, AtomicBoolean.

Each atomic instance

- provides access and update to a single variable of the same type.

- whose value can only be updated atomically

- have methods for dealing with it. Example: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

- e.g. atomicInstanceName.incrementAndGet(initialValue)

package enables lock-free thread-safe programming.

Synchronized method: Java Demo

Intrinsic lock (aka monitor lock)

Every Java **object** (refering to the **class xxx** extends Thread) has an intrinsic lock associated with it.

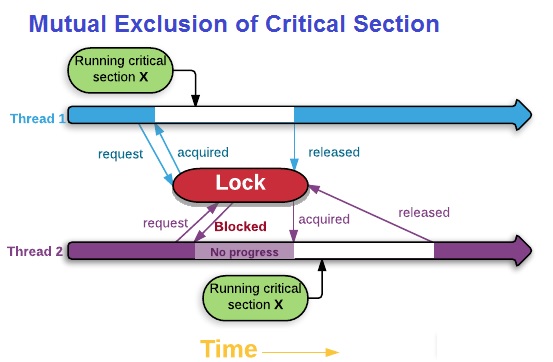
* 2 ways to acquire lock:
  + synchronized method
  + synchronized statement

To make a method “synchronized”, add “synchronized” keyword to method declaration.

* Private static **synchronized** methodName (parameter\_type parameter) { ... body ...}
* any thread that executes synchronized method will invoke intrinsic lock on **object**
* any **other threads** that try to invoke method will be **blocked**
  + until 1st thread terminates, or an exception is thrown and is not caught/handled.
* Note: method can be marked as **static**. Means method belongs to class, not a particular instance. Only 1. Static method is shared by all instances. When a thread acquires lock, will be on this single method shared by all instances. So if one thread locks method, other threads must wait.
* without “static”, each thread will execute method in its own run.
* Usually no need to create instances in Java objects that use “static” method/s.
* Refer to /*src*/SynchronizedMethodDemo.java

Thread.sleep() inside synchronized method does not release the lock.

Intrinsic locks are **reentrant** locks. Once a thread acquired a lock, via a synchronized method in a Java object (refering to **class xxx** extends Thread), it does not need to acquire lock again for other methods in that Java object. The next method in that Java class does not need to use “synchronized” keyword.



A Java class can have 2 or more synchronized methods.

*public class multipleSyncMethods {*

*public static void main (String[] args) throws InterruptedException {*

*Thread thread1 = new Thread( () -> {... body...} );*

*Thread thread2 = new Thread( () -> {... body ...} );*

*thread1.start();*

*thread2.start();*

*}*

*private synchronized syncMethod1 (parameter\_type parameter) {... method body ...}*

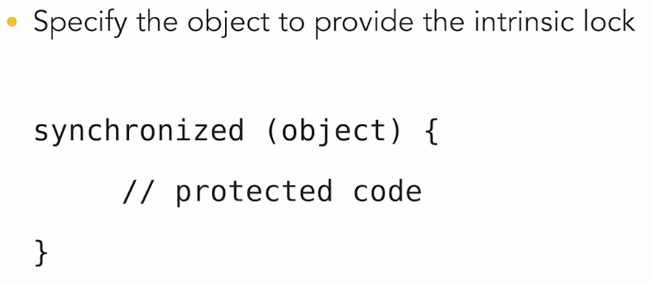
*private synchronized syncMethod2 (parameter\_type parameter) {--- method body ...}*

*}*

If a thread acquires intrinsic lock on Java object via 1 of the synchronized methods, other threads cannot call other synchronized methods in same Java class, will be blocked. Non-synchronized methods will not be blocked.

[https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/java-intrinsic-locks.html#:~:text=An%20intrinsic%20lock%20(aka%20monitor,to%20an%20instance%20method%20call](https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/java-intrinsic-locks.html" \l ":~:text=An intrinsic lock (aka monitor,to an instance method call).

Synchronized statement: Java Demo



A thread must acquire intrinsic lock associated with synchronized Java object before it can execute critical section. Then release lock after finish.

Techniques that implement mutual exclusion

1. intrinsic locks
2. atomic variables
3. synchronized methods
4. synchronized statements

|  |  |
| --- | --- |
| Synchronized | Locks |
| Easier to implement and avoid many of the pitfalls of lock | More flexible structure.  More extensive locking operations possible.  Can support multiple associated Condition objects. |
|  | More functionalities:  - non-blocking tryLock()  - try to acquire lock within timeframe, tryLock(long, TimeUnit)  - lock can be interrupted, lockInterruptibly() |
| Acquisition and release happen in a block.  When multiple locks are acquired, they must be released in the opposite order.  After a thread exit, any other threads can acquire the lock. Some threads may fail to acquire, be starved of resources for a long time. | Fine-grain control. Acquisition and release in a series, chain, “hand-over hand” manner.  Example: 1. acquire A, then B. 2. Release A, then acquire C. 3. Release B, then acquire D.  Can be in any order, or under specific condition. |
|  | Different locks have different properties. |
| Must be acquired and released in the same lexical scope (the context, range that block of code is declared in). | When lock and release in different scopes, must use try-finally or try-catch structures to ensure locks release in case of exceptions. |
| Use as default approach |  |

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/package-summary.html