LCK07-J. Avoid deadlock by requesting and releasing locks in the same order

To avoid data corruption in multithreaded Java programs, shared data must be protected from concurrent modifications and accesses. Locking can be performed at the object level using synchronized methods, synchronized blocks, or the `java.util.concurrent` dynamic lock objects. However, excessive use of locking can result in deadlocks.

Java neither prevents deadlocks nor requires their detection [JLS 2015]. Deadlock can occur when two or more threads request and release locks in different orders. Consequently, programs are required to avoid deadlock by acquiring and releasing locks in the same order.

Additionally, synchronization should be limited to cases where it is absolutely necessary. For example, the `paint()`, `dispose()`, `stop()`, and `destroy(` methods should never be synchronized in an applet because they are always called and used from dedicated threads. Furthermore, the `Thread.stop()` and `Thread.destroy()` methods are deprecated (see THI05-J. Do not use Thread.stop() to terminate threads for more information).

This rule also applies to programs that need to work with a limited set of resources. For example, liveness issues can arise when two or more threads are waiting for each other to release resources such as database connections. These issues can be resolved by letting each waiting thread retry the operation at random intervals until they successfully acquire the resource.

Noncompliant Code Example (Different Lock Orders)

This noncompliant code example can deadlock because of excessive synchronization. The `balanceAmount` field represents the total balance available for a particular `BankAccount` object. Users are allowed to initiate an operation that atomically transfers a specified amount from one account to another.

```
final class BankAccount {
  private double balanceAmount;  // Total amount in bank account

  BankAccount(double balance) {
    this.balanceAmount = balance;
  }

  // Deposits the amount from this object instance
  // to BankAccount instance argument ba
  private void depositAmount(BankAccount ba, double amount) {
    synchronized (this) {
      synchronized (ba) {
        if (amount > balanceAmount) {
          throw new IllegalArgumentException(
               "Transfer cannot be completed"
          );
        }
        ba.balanceAmount += amount;
        this.balanceAmount -= amount;
      }
    }
  }

  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
        public void run() {
          first.depositAmount(second, amount);
        }
    });
    transfer.start();
  }
}
```

Objects of this class are prone to deadlock. An attacker who has two bank accounts can construct two threads that initiate balance transfers from two different `BankAccount` object instances a and b. For example, consider the following code:

```
BankAccount a = new BankAccount(5000);
BankAccount b = new BankAccount(6000);
BankAccount.initiateTransfer(a, b, 1000); // starts thread 1
BankAccount.initiateTransfer(b, a, 1000); // starts thread 2
```

Each transfer is performed in its own thread. The first thread atomically transfers the amount from a to b by depositing it in account b and then withdrawing the same amount from a. The second thread performs the reverse operation; that is, it transfers the amount from b to a. When executing `depositAmount()`, the first thread acquires a lock on object a. The second thread could acquire a lock on object b before the first thread can. Subsequently, the first thread would request a lock on b, which is already held by the second thread. The second thread would request a lock on a, which is already held by the first thread. This constitutes a deadlock condition because neither thread can proceed.

This noncompliant code example may or may not deadlock, depending on the scheduling details of the platform. Deadlock occurs when (1) two threads request the same two locks in different orders, and (2) each thread obtains a lock that prevents the other thread from completing its transfer. Deadlock is avoided when two threads request the same two locks but one thread completes its transfer before the other thread begins. Similarly, deadlock is avoided if the two threads request the same two locks in the same order (which would happen if they both transfer money from one account to a second account) or if two transfers involving distinct accounts occur concurrently.

Compliant Solution (Private Static Final Lock Object)

This compliant solution avoids deadlock by synchronizing on a private static final lock object before performing any account transfers:

```
final class BankAccount {
  private double balanceAmount;  // Total amount in bank account
  private static final Object lock = new Object();

  BankAccount(double balance) {
    this.balanceAmount = balance;
  }

  // Deposits the amount from this object instance
  // to BankAccount instance argument ba
  private void depositAmount(BankAccount ba, double amount) {
    synchronized (lock) {
      if (amount > balanceAmount) {
        throw new IllegalArgumentException(
            "Transfer cannot be completed");
      }
      ba.balanceAmount += amount;
      this.balanceAmount -= amount;
    }
  }

  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
        @Override public void run() {
          first.depositAmount(second, amount);
        }
    });
    transfer.start();
  }
}
```

In this scenario, deadlock cannot occur when two threads with two different `BankAccount` objects try to transfer to each other's accounts simultaneously. One thread will acquire the private lock, complete its transfer, and release the lock before the other thread can proceed.

This solution imposes a performance penalty because a private static lock restricts the system to performing transfers sequentially. Two transfers involving four distinct accounts (with distinct target accounts) cannot be performed concurrently. This penalty increases considerably as the number of `BankAccount` objects increase. Consequently, this solution fails to scale well.

Compliant Solution (Ordered Locks)

This compliant solution ensures that multiple locks are acquired and released in the same order. It requires a consistent ordering over `BankAccount` objects. Consequently, the `BankAccount` class implements the `java.lang.Comparable` interface and overrides the `compareTo()` method.

```
final class BankAccount implements Comparable<BankAccount> {
  private double balanceAmount;  // Total amount in bank account
  private final Object lock;

  private final long id; // Unique for each BankAccount
  private static final AtomicLong nextID = new AtomicLong(0); // Next unused ID

  BankAccount(double balance) {
    this.balanceAmount = balance;
    this.lock = new Object();
    this.id = nextID.getAndIncrement();
  }

  @Override public int compareTo(BankAccount ba) {
     return (this.id > ba.id) ? 1 : (this.id < ba.id) ? -1 : 0;
  }

  // Deposits the amount from this object instance
  // to BankAccount instance argument ba
  public void depositAmount(BankAccount ba, double amount) {
    BankAccount former, latter;
    if (compareTo(ba) < 0) {
      former = this;
      latter = ba;
    } else {
      former = ba;
      latter = this;
    }
    synchronized (former) {
      synchronized (latter) {
        if (amount > balanceAmount) {
          throw new IllegalArgumentException(
              "Transfer cannot be completed");
        }
        ba.balanceAmount += amount;
        this.balanceAmount -= amount;
      }
    }
  }

  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
        @Override public void run() {
          first.depositAmount(second, amount);
        }
    });
    transfer.start();
  }
}
```

Whenever a transfer occurs, the two `BankAccount` objects are ordered so that the `first` object's lock is acquired before the `second` object's lock. When two threads attempt transfers between the same two accounts, they each try to acquire the first account's lock before acquiring the second account's lock. Consequently, one thread acquires both locks, completes the transfer, and releases both locks before the other thread can proceed.

Unlike the previous compliant solution, this solution permits multiple concurrent transfers as long as the transfers involve distinct accounts.

Compliant Solution (`ReentrantLock`)

In this compliant solution, each `BankAccount` has a `java.util.concurrent.locks.ReentrantLock`. This design permits the `depositAmount()` method to attempt to acquire the locks of both accounts, to release the locks if it fails, and to try again later if necessary.

```
final class BankAccount {
  private double balanceAmount;  // Total amount in bank account
  private final Lock lock = new ReentrantLock();
  private final Random number = new Random(123L);

  BankAccount(double balance) {
    this.balanceAmount = balance;
  }

  // Deposits amount from this object instance
  // to BankAccount instance argument ba
  private void depositAmount(BankAccount ba, double amount)
                             throws InterruptedException {
    while (true) {
      if (this.lock.tryLock()) {
        try {
          if (ba.lock.tryLock()) {
            try {
              if (amount > balanceAmount) {
                throw new IllegalArgumentException(
                    "Transfer cannot be completed");
              }
              ba.balanceAmount += amount;
              this.balanceAmount -= amount;
              break;
            } finally {
              ba.lock.unlock();
            }
          }
        } finally {
          this.lock.unlock();
        }
      }
      int n = number.nextInt(1000);
      int TIME = 1000 + n; // 1 second + random delay to prevent livelock
      Thread.sleep(TIME);
    }
  }

  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
        public void run() {
          try {
            first.depositAmount(second, amount);
          } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); // Reset interrupted status
          }
        }
    });
    transfer.start();
  }
}
```

Deadlock is impossible in this compliant solution because locks are never held indefinitely. If the current object's lock is acquired but the second lock is unavailable, the first lock is released and the thread sleeps for some specified amount of time before attempting to reacquire the lock.

Code that uses this locking strategy has behavior similar to that of synchronized code that uses the traditional monitor lock. `ReentrantLock` also provides several other capabilities. For example, the `tryLock()` method immediately returns false when another thread already holds the lock. Further, the `java.util.concurrent.locks.ReentrantReadWriteLock` class has multiple-readers/single-writer semantics and is useful when some threads require a lock to write information while other threads require the lock to concurrently read the information.

Noncompliant Code Example (Different Lock Orders, Recursive)

The following immutable `WebRequest` class encapsulates a web request received by a server:

```
// Immutable WebRequest
public final class WebRequest {
  private final long bandwidth;
  private final long responseTime;

  public WebRequest(long bandwidth, long responseTime) {
    this.bandwidth = bandwidth;
    this.responseTime = responseTime;
  }

  public long getBandwidth() {
    return bandwidth;
  }

  public long getResponseTime() {
    return responseTime;
  }
}
```

Each request has a response time associated with it, along with a measurement of the network bandwidth required to fulfill the request.

This noncompliant code example monitors web requests and provides routines for calculating the average bandwidth and response time required to serve incoming requests.

```
public final class WebRequestAnalyzer {
  private final Vector<WebRequest> requests = new Vector<WebRequest>();

  public boolean addWebRequest(WebRequest request) {
    return requests.add(new WebRequest(request.getBandwidth(),
                        request.getResponseTime()));
  }

  public double getAverageBandwidth() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageBandwidth(0, 0);
  }

  public double getAverageResponseTime() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageResponseTime(requests.size() - 1, 0);
  }

  private double calculateAverageBandwidth(int i, long bandwidth) {
    if (i == requests.size()) {
      return bandwidth / requests.size();
    }
    synchronized (requests.elementAt(i)) {
      bandwidth += requests.get(i).getBandwidth();
      // Acquires locks in increasing order
      return calculateAverageBandwidth(++i, bandwidth);
    }
  }

  private double calculateAverageResponseTime(int i, long responseTime) {
    if (i <= -1) {
      return responseTime / requests.size();
    }
    synchronized (requests.elementAt(i)) {
      responseTime += requests.get(i).getResponseTime();
      // Acquires locks in decreasing order
      return calculateAverageResponseTime(--i, responseTime);
    }
  }
}
```

The monitoring application is built around the `WebRequestAnalyzer` class, which maintains a list of web requests using the `requests` vector and includes the `addWebRequest()` setter method. Any thread can request the average bandwidth or average response time of all web requests by invoking the `getAverageBandwidth()` and `getAverageResponseTime()` methods.

These methods use fine-grained locking by holding locks on individual elements (web requests) of the vector. These locks permit new requests to be added while the computations are still underway. Consequently, the statistics reported by the methods are accurate when they return the results.

Unfortunately, this noncompliant code example is prone to deadlock because the recursive calls within the synchronized regions of these methods acquire the intrinsic locks in opposite numerical orders. That is, `calculateAverageBandwidth()` requests locks from index 0 up to `requests.size()` 1, whereas `calculateAverageResponseTime()` requests them from index `requests.size()` 1 down to 0. Because of recursion, previously acquired locks are never released by either method. Deadlock occurs when two threads call these methods out of order, because one thread calls `calculateAverageBandwidth()`, while the other calls `calculateAverageResponseTime()` before either method has finished executing.

For example, when there are 20 requests in the vector, and one thread calls `getAverageBandwidth()`, the thread acquires the intrinsic lock of `WebRequest` 0, the first element in the vector. Meanwhile, if a second thread calls `getAverageResponseTime()`, it acquires the intrinsic lock of `WebRequest` 19, the last element in the vector. Consequently, deadlock results because neither thread can acquire all of the locks required to proceed with its calculations.

Note that the `addWebRequest()` method also has a race condition with `calculateAverageResponseTime()`. While iterating over the vector, new elements can be added to the vector, invalidating the results of the previous computation. This race condition can be prevented by locking on the last element of the vector (when it contains at least one element) before inserting the element.

Compliant Solution

In this compliant solution, the two calculation methods acquire and release locks in the same order, beginning with the first web request in the vector.

```
public final class WebRequestAnalyzer {
  private final Vector<WebRequest> requests = new Vector<WebRequest>();

  public boolean addWebRequest(WebRequest request) {
    return requests.add(new WebRequest(request.getBandwidth(),
                        request.getResponseTime()));
  }

  public double getAverageBandwidth() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageBandwidth(0, 0);
  }

  public double getAverageResponseTime() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageResponseTime(0, 0);
  }

  private double calculateAverageBandwidth(int i, long bandwidth) {
    if (i == requests.size()) {
      return bandwidth / requests.size();
    }
    synchronized (requests.elementAt(i)) {
      // Acquires locks in increasing order
      bandwidth += requests.get(i).getBandwidth();
      return calculateAverageBandwidth(++i, bandwidth);
    }
  }

  private double calculateAverageResponseTime(int i, long responseTime) {
    if (i == requests.size()) {
      return responseTime / requests.size();
    }
    synchronized (requests.elementAt(i)) {
      // Acquires locks in increasing order
      responseTime += requests.get(i).getResponseTime();
      return calculateAverageResponseTime(++i, responseTime);
    }
  }
}
```

Consequently, while one thread is calculating the average bandwidth or response time, another thread cannot interfere or induce deadlock. Each thread must first synchronize on the first web request, which cannot happen until any prior calculation completes.

Locking on the last element of the vector in `addWebRequest()` is unnecessary for two reasons. First, the locks are acquired in increasing order in all the methods. Second, updates to the vector are reflected in the results of the computations.

Risk Assessment

Acquiring and releasing locks in the wrong order can result in deadlock.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
LCK07-J	Low	Likely	High	**P3**	**L3**

Automated Detection

Some static analysis tools can detect violations of this rule.

Tool	Version	Checker	Description
Coverity	7.5	**LOCK_INVERSION** **LOCK_ORDERING**	Implemented

Parasoft Jtest	10.3	**TRS.LORD**	Implemented
ThreadSafe	1.3	**CCE_DL_DEADLOCK**	Implemented

Related Guidelines

SEI CERT C Coding Standard	CON35-C. Avoid deadlock by locking in a predefined order
MITRE CWE	CWE-833, Deadlock

Bibliography

[Halloway 2000]	
[JLS 2015]	Chapter 17, "Threads and Locks"