LCK09-J. Do not perform operations that can block while holding a lock

Holding locks while performing time-consuming or blocking operations can severely degrade system performance and can result in starvation. Furthermore, deadlock can result if interdependent threads block indefinitely. Blocking operations include network, file, and console I/O (for example, `Console.readLine()`) and object serialization. Deferring a thread indefinitely also constitutes a blocking operation. Consequently, programs must not perform blocking operations while holding a lock.

When the Java Virtual Machine (JVM) interacts with a file system that operates over an unreliable network, file I/O might incur a large performance penalty. In such cases, avoid file I/O over the network while holding a lock. File operations (such as logging) that could block while waiting for the output stream lock or for I/O to complete could be performed in a dedicated thread to speed up task processing. Logging requests can be added to a queue, assuming that the queue's `put()` operation incurs little overhead as compared to file I/O [Goetz 2006].

Noncompliant Code Example (Deferring a Thread)

This noncompliant code example defines a utility method that accepts a `time` argument:

```
public synchronized void doSomething(long time)
                         throws InterruptedException {
  // ...
  Thread.sleep(time);
}
```

Because the method is synchronized, when the thread is suspended, other threads cannot use the synchronized methods of the class. The current object's monitor continues to be held because the `Thread.sleep()` method lacks synchronization semantics.

Compliant Solution (Intrinsic Lock)

This compliant solution defines the `doSomething()` method with a `timeout` parameter rather than the `time` value. Using `Object.wait()` instead of `Thread.sleep()` allows setting a timeout period during which a notification may awaken the thread.

```
public synchronized void doSomething(long timeout)
                                     throws InterruptedException {
// ...
  while (<condition does not hold>) {
    wait(timeout); // Immediately releases the current monitor
  }
}
```

The current object's monitor is immediately released upon entering the wait state. When the timeout period elapses, the thread resumes execution after reacquiring the current object's monitor.

According to the Java API Class `Object` documentation [API 2014]

> *Note that the `wait` method, as it places the current thread into the wait set for this object, unlocks only this object; any other objects on which the current thread may be synchronized remain locked while the thread waits.*
>
> *This method should only be called by a thread that is the owner of this object's monitor.*

Programs must ensure that threads that hold locks on other objects release those locks appropriately before entering the wait state. Additional guidance on waiting and notification is available in THI03-J. Always invoke wait() and await() methods inside a loop and THI02-J. Notify all waiting threads rather than a single thread.

Noncompliant Code Example (Network I/O)

This noncompliant code example defines a `sendPage()` method that sends a `Page` object from a server to a client. The method is synchronized to protect the `pageBuff` array when multiple threads request concurrent access.

```
// Class Page is defined separately.
// It stores and returns the Page name via getName()
Page[] pageBuff = new Page[MAX_PAGE_SIZE];

public synchronized boolean sendPage(Socket socket, String pageName)
                                     throws IOException {
  // Get the output stream to write the Page to
  ObjectOutputStream out
      = new ObjectOutputStream(socket.getOutputStream());

  // Find the Page requested by the client
  // (this operation requires synchronization)
  Page targetPage = null;
  for (Page p : pageBuff) {
    if (p.getName().compareTo(pageName) == 0) {
      targetPage = p;
    }
  }

  // Requested Page does not exist
  if (targetPage == null) {
    return false;
  }

  // Send the Page to the client
  // (does not require any synchronization)
  out.writeObject(targetPage);

  out.flush();
  out.close();
  return true;
}
```

Calling `writeObject()` within the synchronized `sendPage()` method can result in delays and deadlock-like conditions in high-latency networks or when network connections are inherently *lossy*.

Compliant Solution

This compliant solution separates the process into a sequence of steps:

1. Perform actions on data structures requiring synchronization.
2. Create copies of the objects to be sent.
3. Perform network calls in a separate unsynchronized method.

In this compliant solution, the unsynchronized `sendPage()` method calls the synchronized `getPage()` method to retrieve the requested `Page` in the `pageBuff` array. After the `Page` is retrieved, `sendPage()` calls the unsynchronized `deliverPage()` method to deliver the `Page` to the client.

```
// No synchronization
public boolean sendPage(Socket socket, String pageName) {
  Page targetPage = getPage(pageName);

  if (targetPage == null){
    return false;
  }
  return deliverPage(socket, targetPage);
}

// Requires synchronization
private synchronized Page getPage(String pageName) {
  Page targetPage = null;

  for (Page p : pageBuff) {
    if (p.getName().equals(pageName)) {
      targetPage = p;
    }
  }
  return targetPage;
}

// Return false if an error occurs, true if successful
public boolean deliverPage(Socket socket, Page page) {
  ObjectOutputStream out = null;
  boolean result = true;
  try {
    // Get the output stream to write the Page to
    out = new ObjectOutputStream(socket.getOutputStream());

    // Send the page to the client
    out.writeObject(page);out.flush();
  } catch (IOException io) {
    result = false;
  } finally {
    if (out != null) {
      try {
        out.close();
      } catch (IOException e) {
        result = false;
      }
    }
  }
  return result;
}
```

Exceptions

LCK09-J-EX0: Classes that provide an appropriate termination mechanism to callers are permitted to violate this rule (see THI04-J. Ensure that threads performing blocking operations can be terminated).

LCK09-J-EX1: Methods that require multiple locks may hold several locks while waiting for the remaining locks to become available. This constitutes a valid exception, although the programmer must follow other applicable rules, especially LCK07-J. Avoid deadlock by requesting and releasing locks in the same order to avoid deadlock .

Risk Assessment

Blocking or lengthy operations performed within synchronized regions could result in a deadlocked or unresponsive system.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
LCK09-J	Low	Probable	High	**P2**	**L3**

Automated Detection

Some static analysis tools are capable of detecting violations of this rule.

Tool	Version	Checker	Description
CodeSonar	5.1p0	**FB.MT_CORRECTNESS.SWL_SLEEP_WITH_LOCK_HELD**	Method calls Thread.sleep() with a lock held
Parasoft Jtest	10.3	**TRS.TSHL, BD.TRS.TSHL**	Implemented
ThreadSafe	1.3	**CCE_LK_LOCKED_BLOCKING_CALLS**	Implemented
SonarQube	6.7	**S2276**	Implemented

Related Guidelines

SEI CERT C Coding Standard	CON05-C. Do not perform operations that can block while holding a lock

Bibliography

[API 2014]	Class `Object`
[Grosso 2001]	Chapter 10, "Serialization"
[JLS 2015]	Chapter 17, "Threads and Locks"
[Rotem 2008]	"Fallacies of Distributed Computing Explained"