41

CRITICAL SECTION

DESCRIPTION

A Critical Section is a segment of code that must be executed by only one thread at a time to produce the expected results. When more than one thread is allowed to execute this code segment, it could produce unpredictable results. By this definition, a critical section looks very similar to the concept of a Monitor discussed in Section III — Basic Patterns. The following is the list of similarities and differences between Monitors and Critical Sections:

- A Critical Section is a stricter form of a Monitor.
- A Monitor locks a single object whereas a Critical Section requires a lock on an entire class of objects.
- In Java:
  - The implementation of a Monitor on a method requires the method to be declared using the synchronized keyword.
  - A Critical Section can be implemented by using the combination of both the static and the synchronized keywords.
- In the case of a Monitor, no two threads are allowed to execute the synchronized code on the same object. Two threads can execute the same synchronized code on two different objects. In contrast, in the case of a critical section, no two threads are allowed to execute the code on two different objects. This is because the code is locked at the class level, not at the object level.

EXAMPLE

During the discussion of the Singleton pattern, we designed a message logging class `FileLogger` as a singleton. The `FileLogger` class maintains a class variable logger of the `FileLogger` type. This variable is used to hold the singleton `FileLogger` instance. The `FileLogger` class offers a class-level method `getFileLogger` that can be used by different client objects to access the singleton `FileLogger` instance. As part of the `getFileLogger` method implementation, the `FileLogger` checks to see if the singleton instance has already been created. Checking to see if the class variable logger is null does this. If logger is found to be uninitialized, a `FileLogger` instance is created by invoking its private constructor and is assigned to the logger class variable. This implementation of the `getFileLogger` method works fine in a single-threaded environment. In a multithreaded environment, it is possible for two threads to simultaneously execute the `getFileLogger` method to see if the class variable logger is null and, as a result, initialize logger twice. This means that the `FileLogger` private constructor gets invoked twice.

```
public class FileLogger implements Logger {
  private static FileLogger logger;
  private FileLogger() {
  }
  public static FileLogger getFileLogger() {
    if (logger == null) {
      logger = new FileLogger();
    }
    return logger;
  }
  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt",msg, true, true);
  }
}
```

Initializing the logger variable twice in this example does not result in an error. This is because the `FileLogger` private constructor does not do any complex, critical initialization. In contrast, if the singleton constructor method executes such operations as opening a socket connection on a particular port, executing the constructor twice could result in an error.

Let us enhance the design of the `FileLogger` class to make it suitable for use in multithreaded environments. This can be accomplished in two ways.

Approach I (Critical Section)

This involves making the `getFileLogger` method a Critical Section so that only one thread can ever execute it at any given point in time. This can be accomplished by simply declaring the class-level method `getFileLogger` as synchronized.

```
public class FileLogger implements Logger {
  private static FileLogger logger;
  private FileLogger() {
  }
```

```
  public static synchronized FileLogger getFileLogger() {
    if (logger == null) {
      logger = new FileLogger();
    }
    return logger;
  }
  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt”,msg, true, true);
  }
}
```

This simple change turns the getFileLogger method into a Critical Section and guarantees that no two threads ever execute the getFileLogger method at the same time. This completely eliminates the possibility of the FileLogger constructor getting invoked more than once inside the getFileLogger method.

Approach II (Static Early Initialization)

It is to be noted that synchronizing methods can have a significant effect on the overall application performance. In general, synchronized methods run much slower, as much as 100 times slower than their nonsynchronized counterparts. As an alternative to declaring the getFileLogger method as synchronized, the logger variable can be early initialized.

```
public class FileLogger implements Logger {
  //Early Initialization
  private static FileLogger logger = new FileLogger();
  private FileLogger() {
  }
  public static FileLogger getFileLogger() {
    return logger;
  }
  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt”,msg, true, true);
  }
}
```

This eliminates the need for any check or initialization inside the getFileLogger method. As a result, the getFileLogger becomes thread-safe automatically without having to declare it as synchronized.

PRACTICE QUESTIONS

1. Design a database connection class as a thread-safe singleton.
2. Design a printer spooler class as a thread-safe singleton.