

Java ReentrantLock: Usage Considerations



Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

**Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Part of the Lesson

- Understand the concept of mutual exclusion in concurrent programs
- Note a human-known use of mutual exclusion
- Recognize the structure & functionality of Java ReentrantLock
- Be aware of reentrant mutex semantics
- Know the key methods defined by the Java ReentrantLock class
- Master how to use ReentrantLock in practice
- Appreciate Java ReentrantLock usage considerations



ReentrantLock Usage Considerations

ReentrantLock Usage Considerations

- ReentrantLock must be used via a “fully bracketed” protocol



```
void someMethod() {  
    ReentrantLock lock  
        = this.lock;  
    lock.lock();  
    try { ...  
    } finally {  
        lock.unlock();  
    }  
}
```

*The thread that acquires the lock
must be the one to release it*

ReentrantLock Usage Considerations

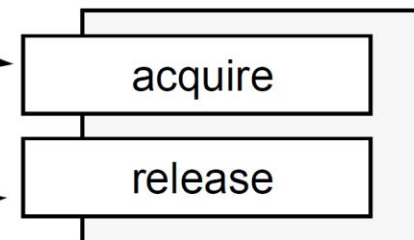
- ReentrantLock must be used via a “fully bracketed” protocol
- This design is known as the “*Scoped Locking*” pattern

The finally clause ensures that the lock is released on all paths out the try clause

```
void someMethod() {  
    ReentrantLock lock  
        = this.lock;  
    lock.lock();  
    try { ...  
    } finally {  
        lock.unlock();  
    }  
}
```

```
begin ## Enter the critical section.  
    ## Acquire the lock automatically.  
  
    ## Execute the critical section.  
    do_something ();  
  
    ## Release the lock automatically.  
end ## Leave the critical section.
```

Lock



ReentrantLock Usage Considerations

- ReentrantLock must be used via a “fully bracketed” protocol
 - This design is known as the “*Scoped Locking*” pattern
 - Implemented implicitly via Java synchronized methods & statements

```
void someMethod() {  
    synchronized (this) {  
        ...  
    }  
}
```

See lesson on “*Java Built-in Monitor Object*”

ReentrantLock Usage Considerations

- ReentrantLock must be used via a “fully bracketed” protocol
 - This design is known as the “*Scoped Locking*” pattern
 - Implemented implicitly via Java synchronized methods & statements
 - This pattern is commonly used in C++ (& C#) via constructors & destructors

```
void write_to_file
    (std::ofstream &file,
     const std::string &msg)
{
    static std::mutex mutex;

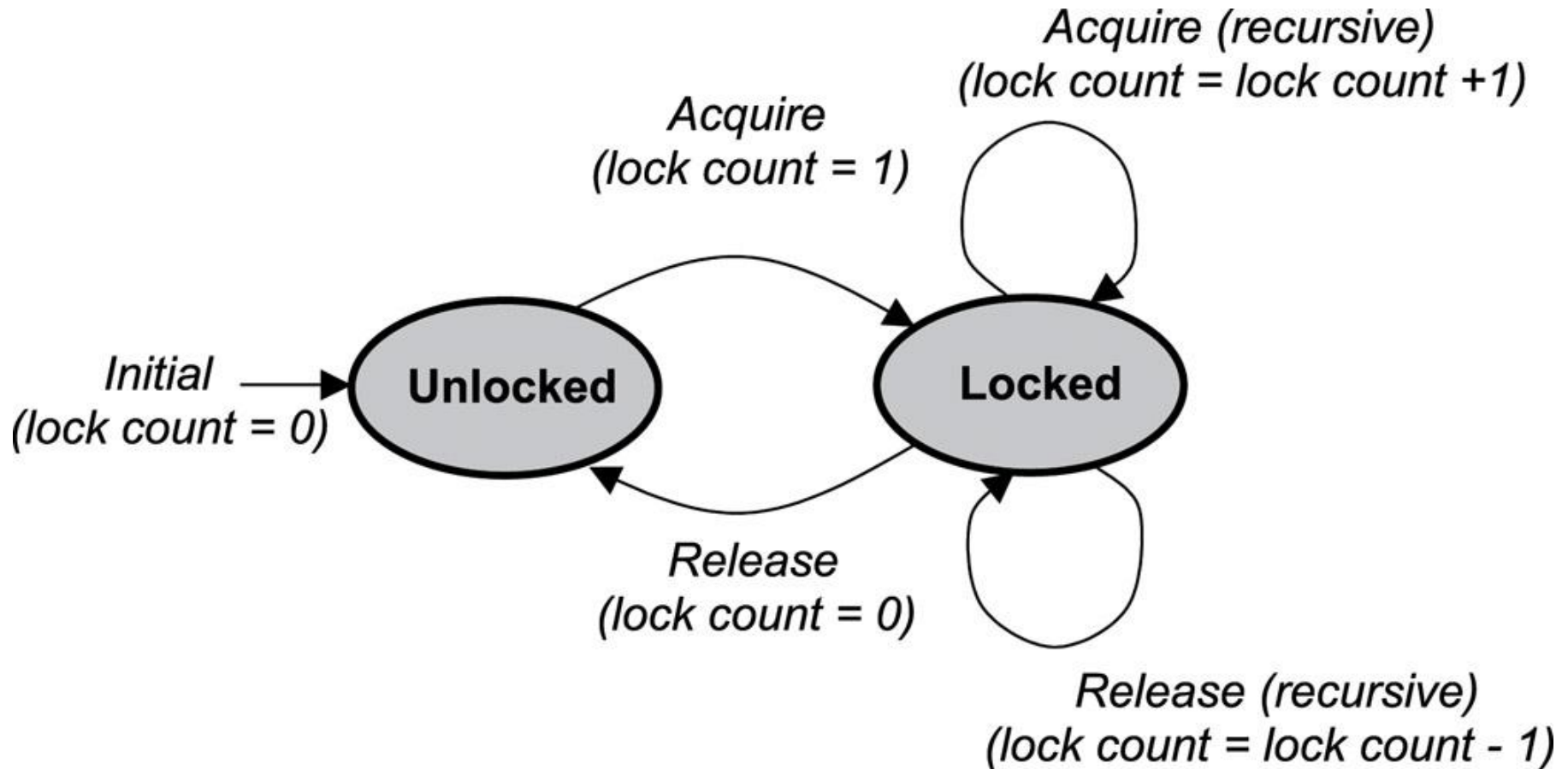
    std::lock_guard<std::mutex>
        lock(mutex) ;

    file << msg << std::endl;
}
```

See en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

ReentrantLock Usage Considerations

- ReentrantLock supports “recursive mutex” semantics where a lock may be acquired multiple times by the same thread, without causing self-deadlock



See en.wikipedia.org/wiki/Reentrant_mutex

ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls



ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls, e.g.
- Holding a lock for a long time without needing it

```
void someMethod() {  
    ReentrantLock lock  
        = this.lock;  
    lock.lock();  
    try {  
        for (;;) {  
            // Do something that  
            // doesn't involve lock  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```

Locked Out?



ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls, e.g.
 - Holding a lock for a long time without needing it
 - Acquiring a lock & forgetting to release it

```
void someMethod() {  
    ReentrantLock lock  
        = this.lock;  
    lock.lock();  
    ... // Critical section  
    return;  
}
```

*This lock may be
locked indefinitely!*



ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls, e.g.
 - Holding a lock for a long time without needing it
 - Acquiring a lock & forgetting to release it

```
void someMethod() {  
    ReentrantLock lock  
        = this.lock;  
    lock.lock();  
    try {  
        ... // Critical section  
        return;  
    } finally {  
        lock.unlock();  
    }  
}
```

Use the try/finally idiom to ensure a fully-bracketed semaphore is always released, even if exceptions occur

See docs.oracle.com/javase/tutorial/essential/exceptions/finally.html

ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls, e.g.
 - Holding a lock for a long time without needing it
 - Acquiring a lock & forgetting to release it
 - Releasing a lock that was never acquired
 - or has already been released

```
void someMethod() {  
    ReentrantLock lock  
        = this.lock;  
    // lock.lock();  
    try {  
        ... // Critical section  
    } finally {  
        lock.unlock();  
    }  
}
```



ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls, e.g.
 - Holding a lock for a long time without needing it
 - Acquiring a lock & forgetting to release it
 - Releasing a lock that was never acquired
 - Accessing a resource without acquiring a lock for it first
 - or after releasing it

```
void someMethod() {  
    ReentrantLock lock  
        = this.lock;  
    // lock.lock();  
    try {  
        ... // Critical section  
    } finally {  
        // lock.unlock();  
    }  
}
```



Compare with lesson on "*Java Built-in Monitor Objects*"

End of Java ReentrantLock: Usage Considerations