Synchronization: Problems & Solutions

Threads have sharing access to common data . This form of communication is extremely efficient, but makes two kinds of errors possible.

Synchronization Problems

A) Thread Interference Errors:

- Some non-atomic parts of code demand execution by single thread only. Such parts called ***critical sections***

B) Memory Consistency Errors could be result of***:***

- ***Operations Reordering*** by optimizing compilers
- ***CPU Cache Coherency*** problem

Solution for the Problems

A) Enforcing ***exclusive access to critical sections*** of code

B) Establishing ***happens-before dependency***

Mutex

Mutex is a MUTual EXclusion device, which:

- enforces ***exclusive access to critical sections*** (parts of code that cannot be executed concurrently by more than one thread)
- establishes ***happens-before dependency***

Mutex is useful for protecting shared data from concurrent modifications.

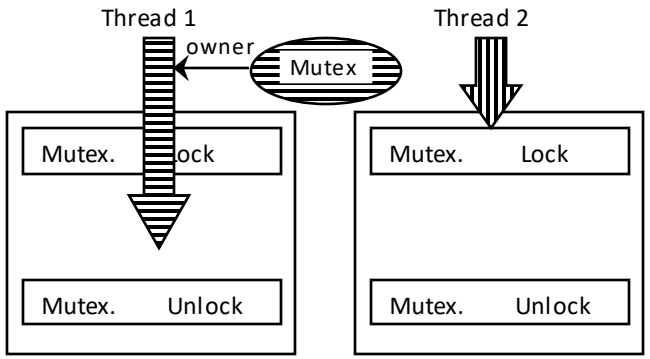
A Mutex has two possible states:

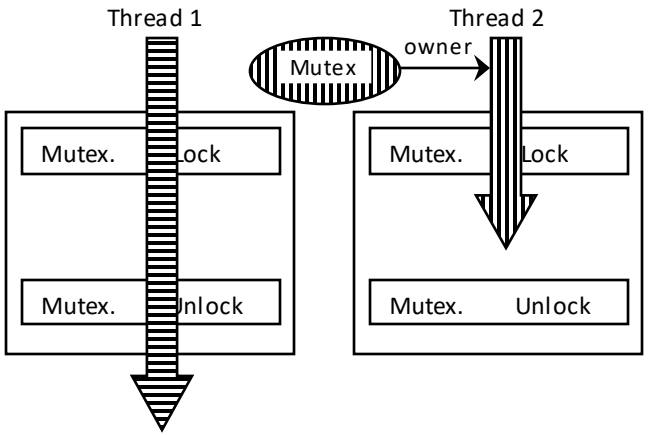
- locked (owned by one thread)
- unlocked (not owned by any thread)

A Mutex could be unlocked only by the same Thread, which locked it.

A Mutex can never be owned by two different threads simultaneously.

A Thread attempting to lock a Mutex that is already locked by another Thread, is suspended until the owning Thread unlocks the Mutex first.

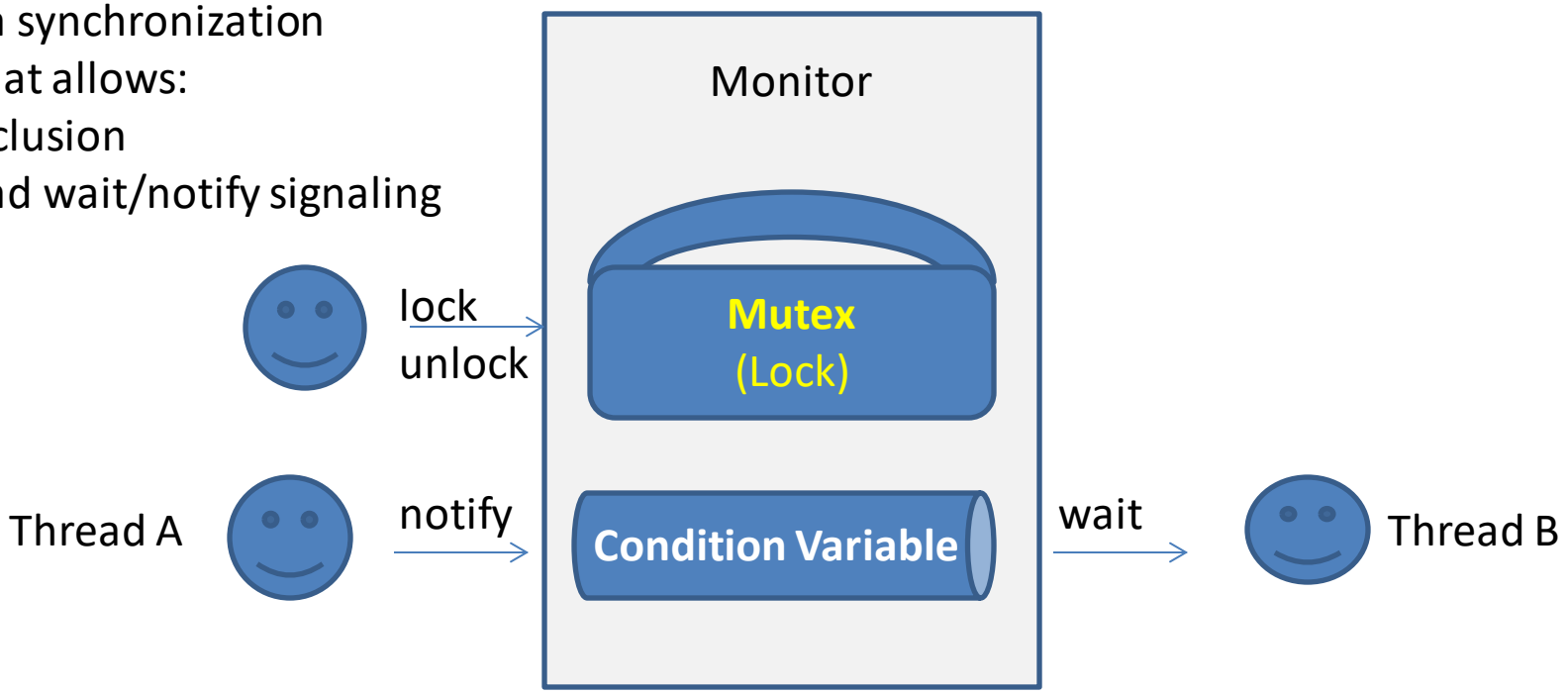




Java Object Monitor

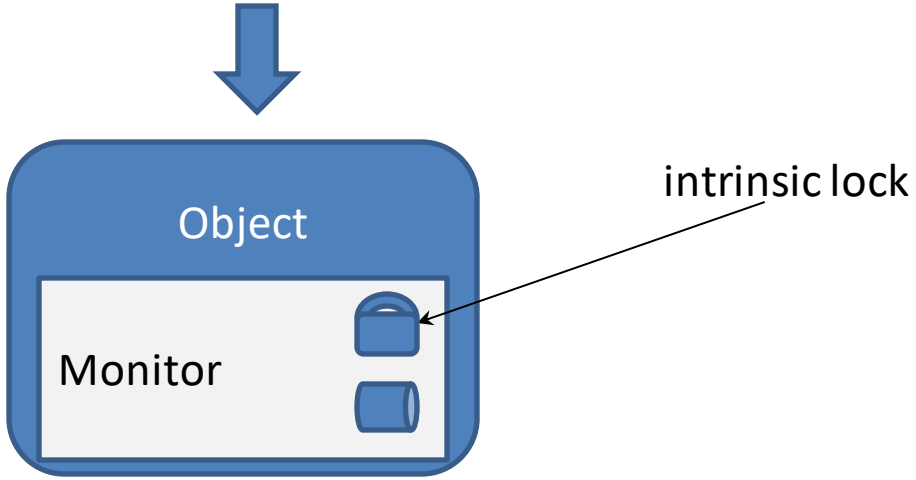
Monitor is a synchronization construct that allows:

- Mutual exclusion
- Inter-thread wait/notify signaling



Java Synchronization is built around an internal entity known as the ***intrinsic lock*** or *monitor lock*. Intrinsic lock is ***mutex*** , implementing both aspects of synchronization:

- enforcing exclusive access to an object's state
- and establishing happens-before relationships that are essential to visibility.



Synchronized

synchronized method:

```
public void synchronized f() {….}

try {
      this.monitor.mutex.lock();       // acquire the lock
      ….// body of sychronized method
} finally {
       this.monitor.mutex.unlock();  // release the lock
}
```

synchronized statement (Monitor Object):

```
Object obj1 = new Object();
synchronized(obj1) {……}

try {
      obj1.monitor.mutex.lock();       // acquire the lock
      …. // body of synchronized method
} finally {
       obj1.monitor.mutex.unlock();  // release the lock
}
```

Synchronized (cont.)

static synchronized method:

```
class X{
    public static  void synchronized f() {….}
}

synchronized(X.class) {
      …. // body of sychronized method
}
                          OR
try {
     X.class.monitor.mutex.lock();       // acquire the lock
     …. // body of sychronized method
} finally {
      X.class.monitor.mutex.unlock();  // release the lock
}
```

Package **java.util.concurrent** and Sub-Packages

java.util.concurrent.locks

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors:

- Interfaces ***Lock, ReadWriteLock*** provide more extensive locking operations than can be obtained using synchronized methods and statements (for example, method **tryLock** or multiple simultaneous read locks versus single write lock)
- Classes ***ReentrantLock***, ***ReentrantReadWriteLock*** – lock implementations
- Interface ***Condition*** provides effect of having multiple wait-sets per object by combining them with the use of arbitrary *Lock* implementations

java.util.concurrent.atomic

The lock-free thread-safe programming on single variables

java.util.concurrent

Utility classes commonly useful in concurrent programming: Synchronized and Lock-free containers, Thread Pools, e.t.c.

Conclusion

	Intrinsic Lock	Reentrant Lock	volatile	Atomic*XXX*
exclusive access to critical sections of code	yes	yes (extended interface)	N/A	yes • lock-free • CPU architecture specific • simple test-and-set scenarios only
establishing happens-before dependency	yes	yes	yes •lock-free •CPU architecture specific	yes •lock-free •CPU architecture specific

The Particularity of Multi-Threaded Programming

All threads within a process share the same global memory. This makes the sharing of information easy between the threads, but along with this simplicity comes *the problem of synchronization*.

Reentrant Functionality

Functionality (procedure, object) is *reentrant* if all its task-unique information (such as local variables) is kept in a separate area of memory that is distinct for each thread (or process), executing this functionality in simultaneous mode.

Thread-Safe Functionality

Functionality is thread-safe if:

- It is reentrant
- All its parts, requiring execution by single thread only, are protected from multiple simultaneous execution. (For this purpose some form of *mutual exclusion* is used.)

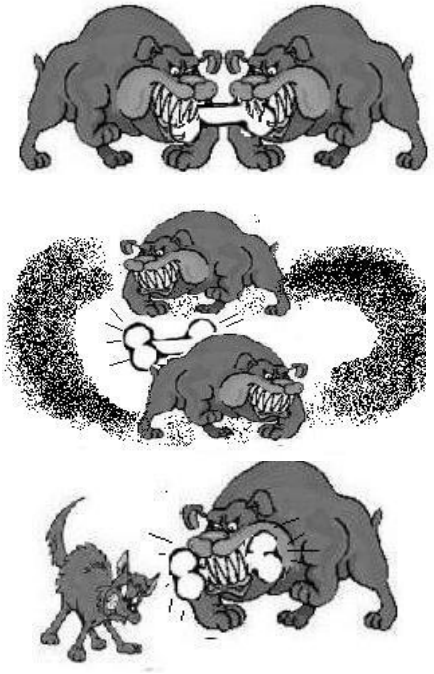
Safety from Deadlocks, Livelocks and Starvation

The following synchronization problems could occur in multi-threaded (multi-process) environment as result of *Race Condition:*

A ***Deadlock*** is a situation in which two or more threads (or processes) sharing the same resource are effectively preventing each other from accessing the resource.

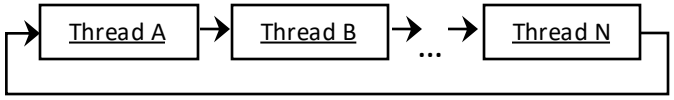
A ***Livelock*** is a situation in which two or more threads (or processes) continually change their state, each in response to state change in the other one. The result is that none of the running threads could make further progress.

A ***Starvation*** is a situation where a thread (or process) is unable to gain regular access to shared resource, because the resource are made unavailable for long periods by "greedy" threads (processes), locking it for a long time.

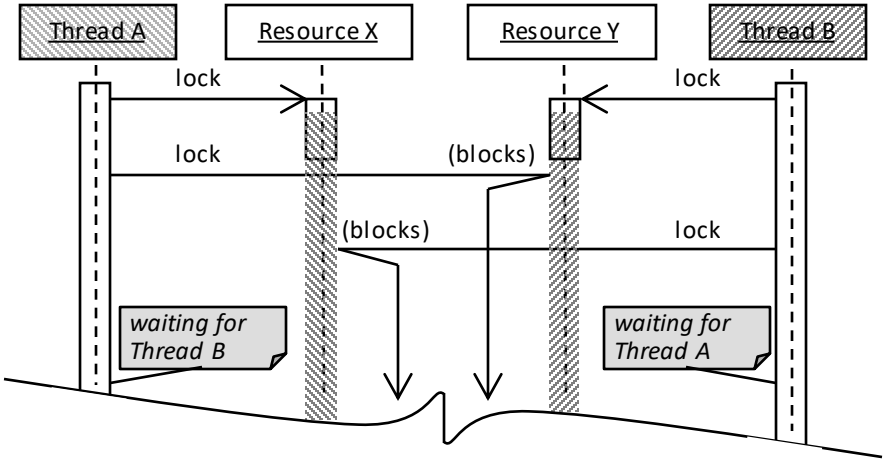


Thread Deadlock and Livelock Examples

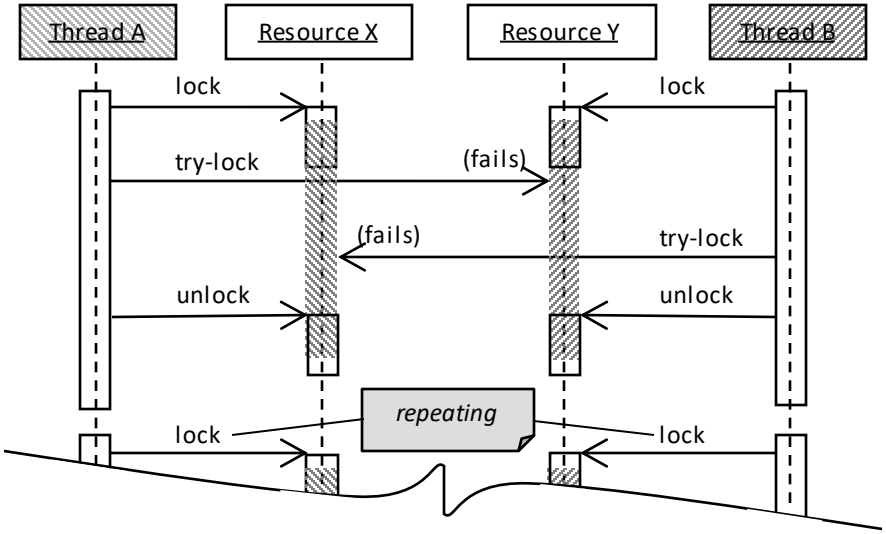
The classic *Deadlock* case is where two Threads both require two shared Resources, and they try to lock them *in opposite order*.



In more sophisticated cases the Deadlock scenario could contain the chain of multiple threads waiting each other:



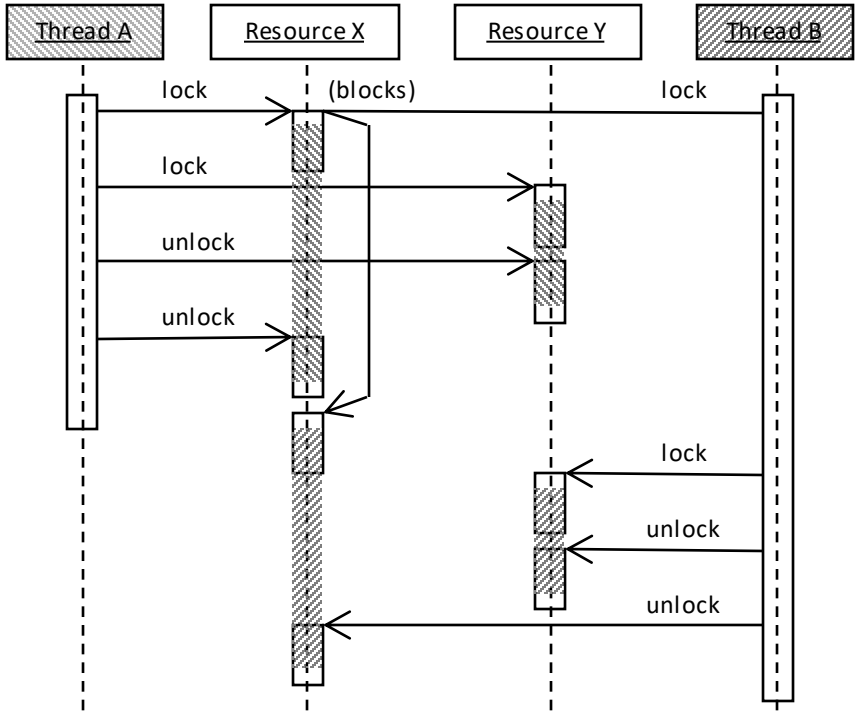
The *Livelock* may arise from attempts to avoid Threads blocking via a *try-lock*.
After the try-lock failure, both threads release their lock and no work is done.
Then the same locking pattern is repeated.



Deadlock Resolution Example

To avoid *Deadlock* in previous example, both Threads have to follow *the same Resource locking order*.

In other words, both Threads have to establish *common Resource locking protocol.*



In more sophisticated systems with big number of Locking Resources, the establishing of *common Resource Locking protocol* could be problematic or impossible.

In this case the system must provide functionality, *prohibiting the specific lock operation, which "fastens" the Deadlock chain.*