- There is a tension between safety and liveness. Locking is used to ensure safety, but indiscriminate use of locking can cause lock-ordering deadlocks.
- Thread pools and semaphores are used to bound resource consumption, but failure to understand activities being bounded can cause resource deadlocks.

Java applications do not recover from deadlocks, so it’s worthwhile to ensure that your design precludes the conditions that could cause it.

Deadlock

When thread A holds lock L and tries to ackquire lock M, but at the same time thread B holds M and tries to acquire L, both threads will wait forever. If nodes in a directed graph are threads and endges represent “waiting” relation, then a cyclic graph indicates there is a deadlock.

Database systems are designed to recover from deadlocks. A transaction may acquire many locks until it commits, and two transactions may deadlock. When a database detects deadlock (analysing directed graph), it picks a victim and aborts the transaction. The application can then retry the aborted transaction, which may be able to repeat the transaction.

The JVM is not helpful in resolving deadlocks. When a set of Java threads deadlock, they threads are permanently out of comission. The application may stall completely, a particular subsystem may stall, or performance may suffer.

Lock-ordering deadlocks

leftRight method acquires **left** and **right** locks. **rightLeft** methods acquires **right** and **left** locks.

If the **leftRight** methods acquired the **left** lock and then the **rightLeft** acquired **right** lock, they will deadlock.

The deadlock came about because the two threads attempted to acquire the same lock in a different order. A program will be free of lock-ordering deadlocks if all threads acquire the locks in a fixed global order.

Dynamic lock order deadlocks

Locks used may depend on variables values, same method locking on 2 arguments may lead to deadlock when called in two different threads.

- One way to induce an ordering is to use **System.identityHashCode**, which returns the value that would be returned by **Object.hashCode**.
  - If two objects have identical hashes, a third “tie breaking” lock is used. By acquiring the tie-breaking lock before acquiring locks, we ensure that only one thread at a time performs the risky task of acquiring two locks in arbitary order, elimitaring the possibility of deadlock.

    - If hash collisions were common, this technique might become a concurrency bottleneck (== having a single, program-wide lock). Hash collisions with System.identityHashCode are vanishingly infrequent.
- If objects for synchronization has a unique, comparable key, inducing a lock order is even easier: order objects by their key.

Deadlocks between cooperating objects.

No method may explicitly acquire two locks but deadlock may still be possible. Cooperating objects may invoke each other's synchronized methods in a way that leads to monitor locks taken in a wrong order.

In general, invoking an alien method with a lock held is asking for liveness trouble. The alien method might acquire other locks (risking deadlock) or block for unexpectedly long time, stalling other threads that need the lock you hold.

Open calls

Calling a method with no locks held is called an open call

- Classes that rely on open calls are more well-behaved and composable
- The liveness analysis of a program that reles exclusively on open calls is far easier than that of one that does not. Makes it easier to identify the code paths that acquire multiple locks and therefore to ensure that locks are acquired in a constant order

Restructuring a synchronized block to allow open calls can have undesireable consequences, since it takes an operation that was atomic and makes it nonatomic.

- In many cases, the loss of atomicity is perfectly acceptable
- In some cases, the loss of atomicity is a problem, and you will have to use another technique to achieve atomicity. One such technique is to restructure the code around open call so that only one thread can execute that code path. For example, hold the lock to mark the service as shutting down, release the lock (other threads waiting because of status set), make an open call, wait for resources release, mark service as shut down, let other threads access it.

Resource deadlocks

Tasks can deadlock waiting for resources as they can deadlock waiting for locks.

- Thread A could hold a connection to database D1 and wait for a connection to D2, thread B could hold a connection to database D2 and wait for a connection to D1.
- Thread-starvation deadlock: a task executing in a single threaded executor submits another task to it and waits for its completion.

Avoiding and diagnosing deadlocks

- A program that never acquires more than one lock at a time cannot experience lock-ordering deadlock.
- If you must acquire multuple locks, lock ordering must be a part of your design

    - Try to minimize the number of potential lock interactions
    - Follow and document a lock ordering protocol for locks that may be acquired together
- Audit code that uses fine-grained locking
    - Identify where multitple locks could be acquired — try to make this set small. Perform a global analysis of all such instances to ensure that lock ordering is consistent across entire program
    - Using open calls wherever possible simplifies this analysis

Timed lock attempts

Use timed locks for detecting and recovering from deadlocks.

- Use **tryLock** method of explicit **Lock** classes instead of intrinsic locking.
- Use a timeout that is much longer than you expect acquiring the lock will take

When a timed lock attempt failes, there are several possible reasons why:

- Maybe there was a deadlock
- Thread entered infinite loop holding that lock
- Some activity is running a lot slower than you could expect

Using timed lock acquisition to acquire multiple locks can be effective against deadlocks even when timed locking is not used consistently throughout the program. If lock acquisition times out, you can release the locks, back off and wait for a while, possibly clearing the deadlock condition. But this is possible only if two locks are acquired together, not due to the nesting of method calls).

Deadlock analysis with thread dumps

JVM thread dump:

- stack trace for each running thread
- which locks are held by each thread, in which stack frame they were acquired
- which lock a blocked thread is waiting to acquire

JVM searches the is-waiting-for graph for cycles to find deadlocks. If it finds one, it includes deadlock information:

- which threads and locks are involved
- where in the program the offending lock acquisitions are

Triggering a thread dump: **SIGQUIT** request (**kill –3**).

If you are using explicit Lock classes instead of intrinsic locking

- Java 5.0 has no support for associating Lock information with the thread dump, explicit locks do not show up at all in thread dumps

- Java 6 does include thread dump support and deaclock detection with explicit Locks, but the information on where explicit locks are acquired (associated with the acquiring thread) is less precise than for intrinsic locks (associated with the stack frame where they were acquired).

Other liveness hazards

Starvation

Starvation is when a thread is perpetually denied access to resources it needs in order to make progress. The most commonly starved resource is CPU cycles.

Causes:

- nonterminating constructs with a lock held, since other threads will never be able to acquire that lock
- inappropriate use of thread priorities
  - thread priorities are merely a scheduling hint
  - mapped to OS thread priorities as JVM sees fit
  - this mapping is platform specific, due to different number of priorities in different OS.
  - default is **Thread.NORM_PRIORITY**, it's not always obvious what effect bossting it will have: might do nothing or make one thread be scheduled in preference of others, causing starvation

Poor responsiveness

Poor responsiveness (for examples, in GUI applications)

- Latency critical threads compete for CPU resources with heavy background threads. Altering thread priorities can make sense here.
- Poor lock management, if a lock is held for a long time (iterating a large collection, for example), other threads that need that lock and collection may have to wait for a long time

Livelock

Livelock — a thread, while not blocked, still cannot make a progress because it keeps retrying an operation that always fails.

- Transactional messaging applications, from over-eager error recovery code that mistakenly treats an unrecoverable error as recoverable (processing the same message over and over again, the poison message problem)
- Multiple cooperating threads changing their state in response to the others in such a way that no thread can ever make progress. Introduce some randomness into the retry mechanism.