Each thread has its own working memory, independent of the main memory.

It is useful to think of each thread as running on a separate CPU, with the working memory being the CPU cache.

Intro to the Java Memory Model
Hursh Jain, mollypages.org

Thread 1 / NYC

Local/working memory

Planet Mars

Main memory

Thread 2 / Tokyo

Local/working memory

NYC and Tokyo **cannot see each other**. They can only see Planet Mars and the main working memory.

Main memory access is much slower than local memory access

1 NYC sets a variable n = 1

Only NYC can see this. This value does not get flushed to Mars (it may in practice, but that is arbitrary and this flush never happens conceptually speaking)

Thread 1 / NYC

Local/working memory

1

n = 1

Planet Mars

Main memory

Thread 2 / Tokyo

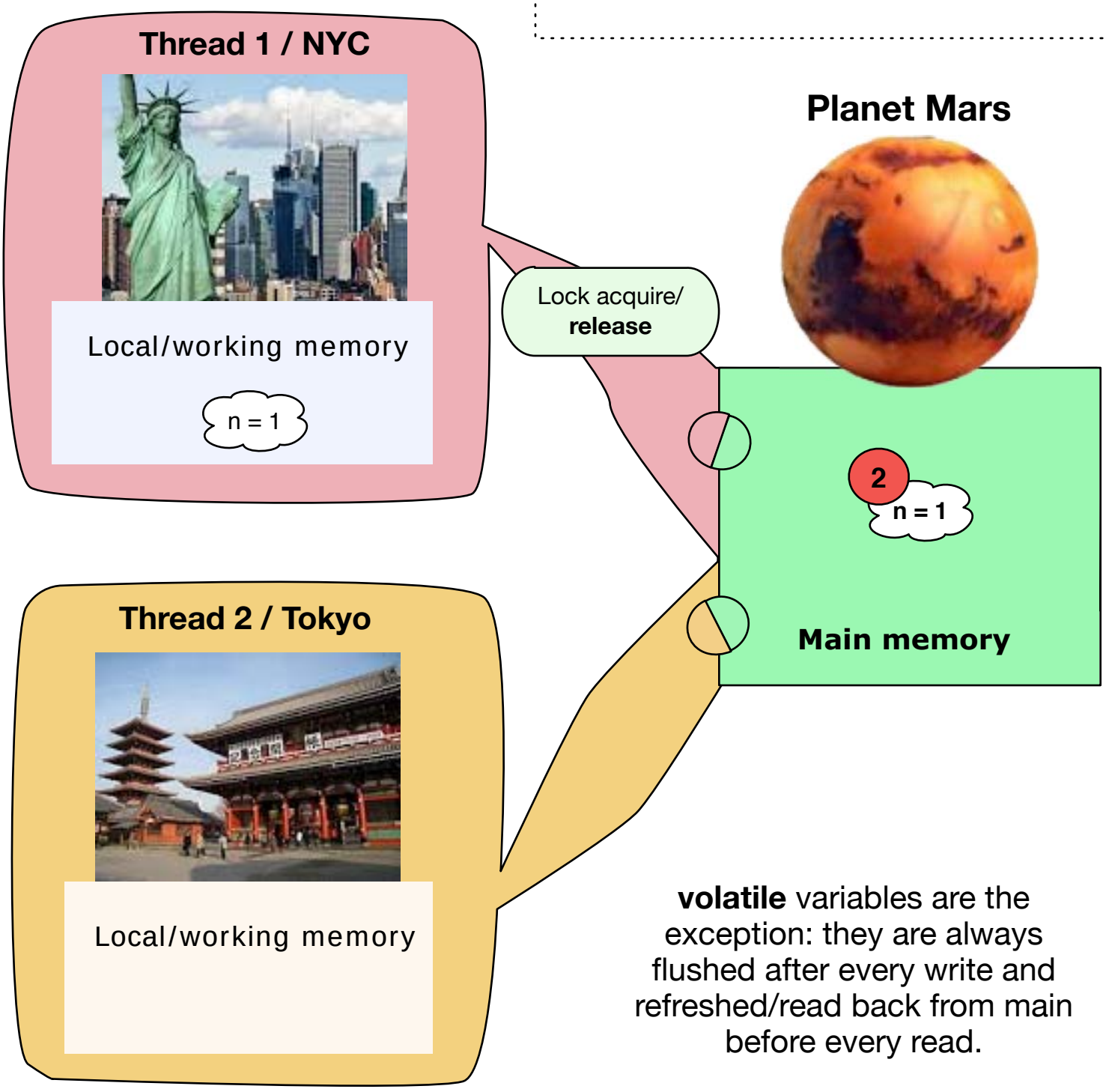
Local/working memory

2 NYC flushes to Mars.

The local/Mars flushing automatically when a lock (any lock on any object) is acquired or released.

Acquiring a lock updates **all** local memory from main memory

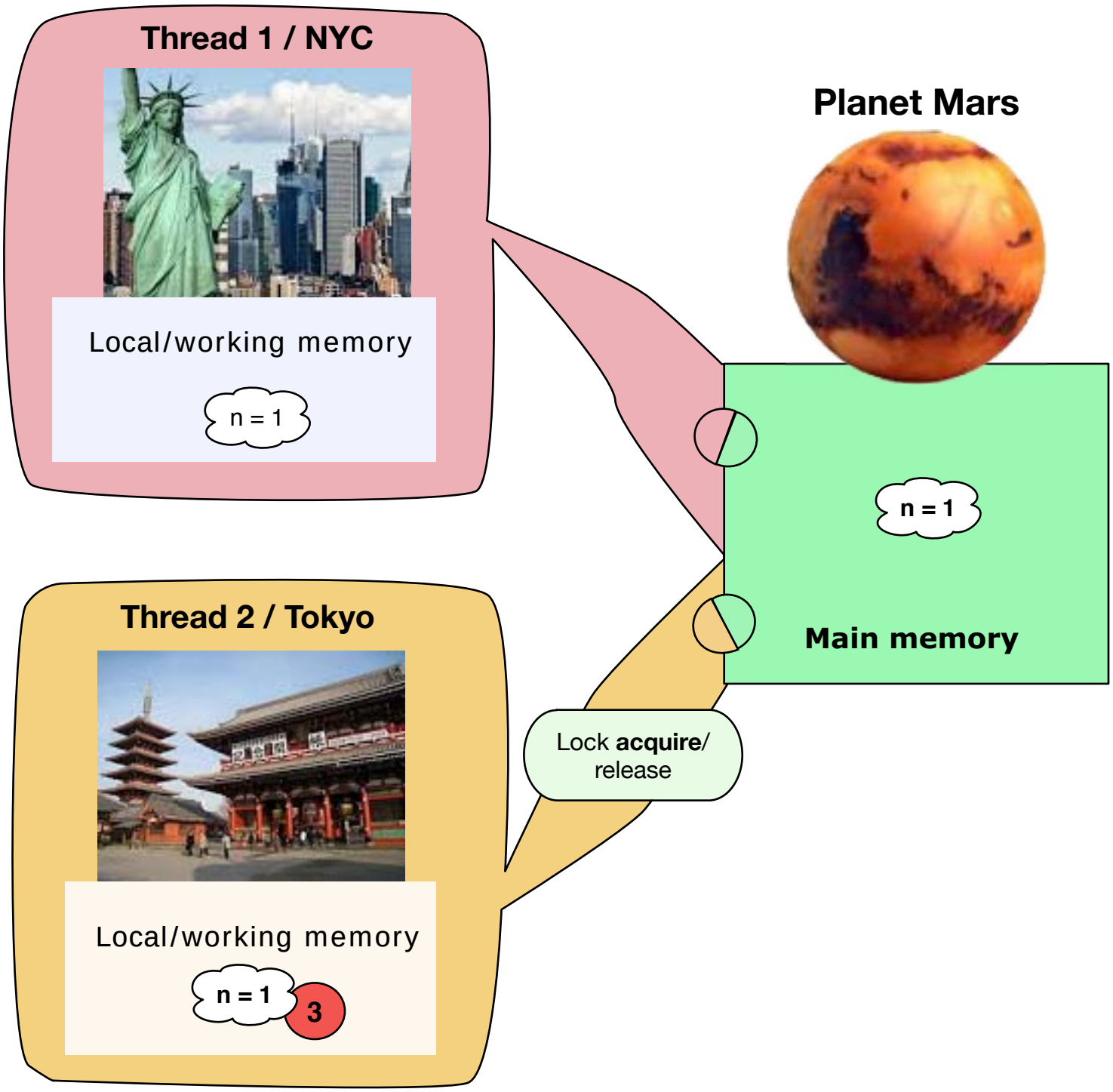
Releasing a lock flushes **all** local memory to main memory

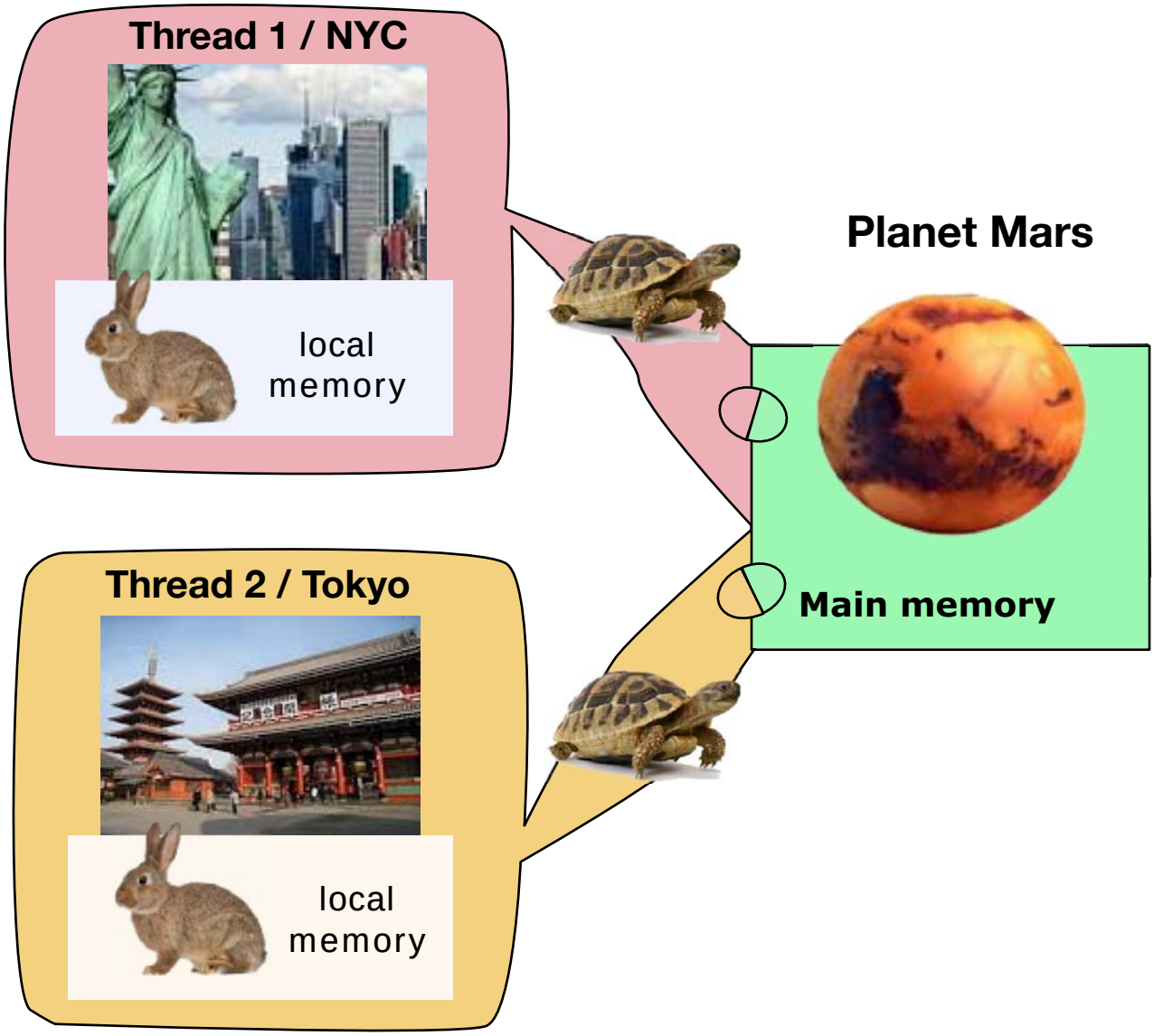


volatile variables are the exception: they are always flushed after every write and refreshed/read back from main before every read.

3 Tokyo acquires a lock

The only way Tokyo can see NYC's update is for NYC to have flushed its memory **and** for Tokyo to then refresh its memory.





- In Java, acquiring locks has a dual *independent* purposes:

  1) exclusion (exclusive access to a block of code)
  2) visibility (memory refresh to-fro from main/local)

- Local working memory is several orders of magnitude **faster** than flushes to main memory

- These are conceptual semantics and guarantees. In practice, flushes may happen at much more optimized times as long as this conceptual order is maintained.