

Lecture 8.2

Java Synchronization

EN 600.320/420

Instructor: Randal Burns

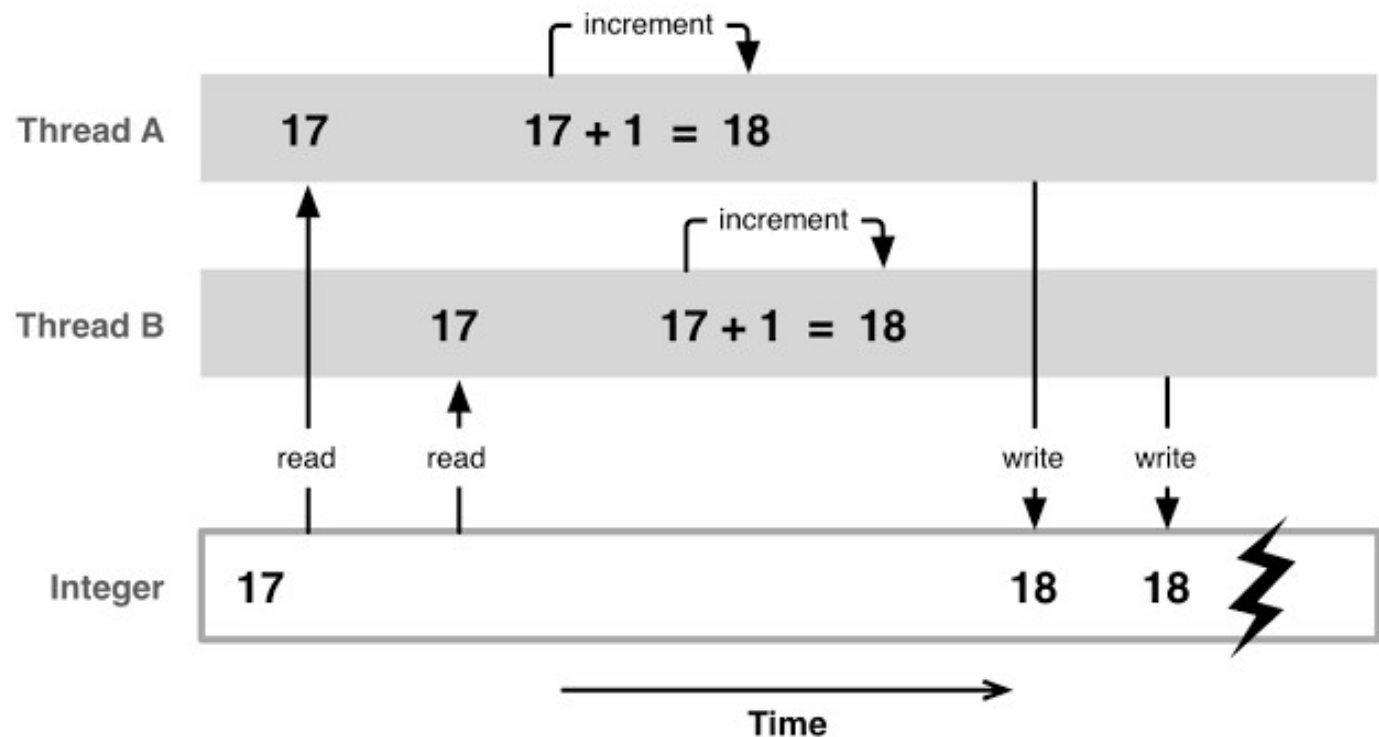
21 February 2017



Department of Computer Science, *Johns Hopkins University*

Communicating Between Threads

- read/write to shared variables
- Uncontrolled sharing leads to unpredictable outcomes
- Race conditions:
 - Conflicting operations from multiple threads
 - Ordered by OS scheduling (not program)



Synchronization Constructs

- A real simple first look at synchronization
 - Some dos and don'ts
- The volatile declaration specifier
- Synchronized blocks
- `java.util.concurrent.atomic`



Some Definitions

- Atomic: the all or nothing property
 - In transactions, either all actions happen or none happen
 - For sequential programs it refers to operations: a sequence of operations is executed by a processor as an indivisible unit that cannot be interrupted.
 - **java.util.concurrent.atomic**: not really atomic, but *lock-free*, *thread safe* encapsulation of fundamental types
- Synchronize: poorly defined, informally used
 - v. To make two or more events happen at exactly the same time or at the same rate
 - In Java, a synchronized block is accessed by only one thread at a time (should be called serialized)
 - Controls access to shared state



volatile: Does it work?

See [VolatileWorks.java](#)

- In Java, a *volatile* variable “is guaranteed to have memory synchronized on each access”
 - Plus atomic reads and writes to long and double
 - All other built-in types are already atomic



volatile: Almost useless

- In Java, a *volatile* variable “is guaranteed to have memory synchronized on each access”
 - Plus atomic reads and writes to long and double
 - All other built-in types are already atomic
- While underlying types are atomic, any operation performed against them is not
 - Increment is not atomic!
 - Any reason to declare variables volatile?

