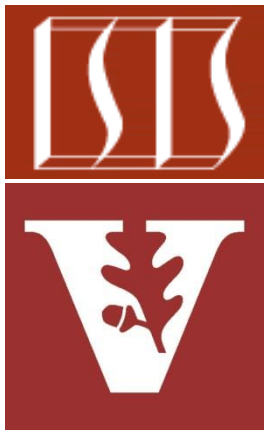
Java ReentrantLock: Introduction

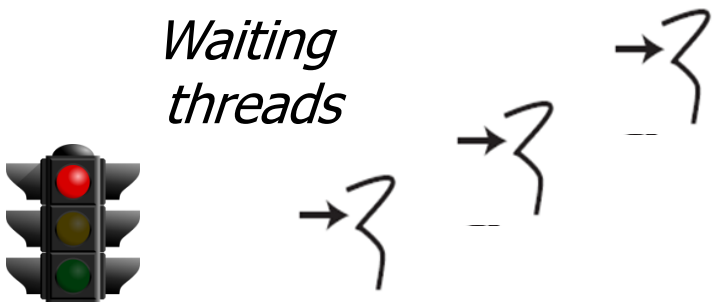


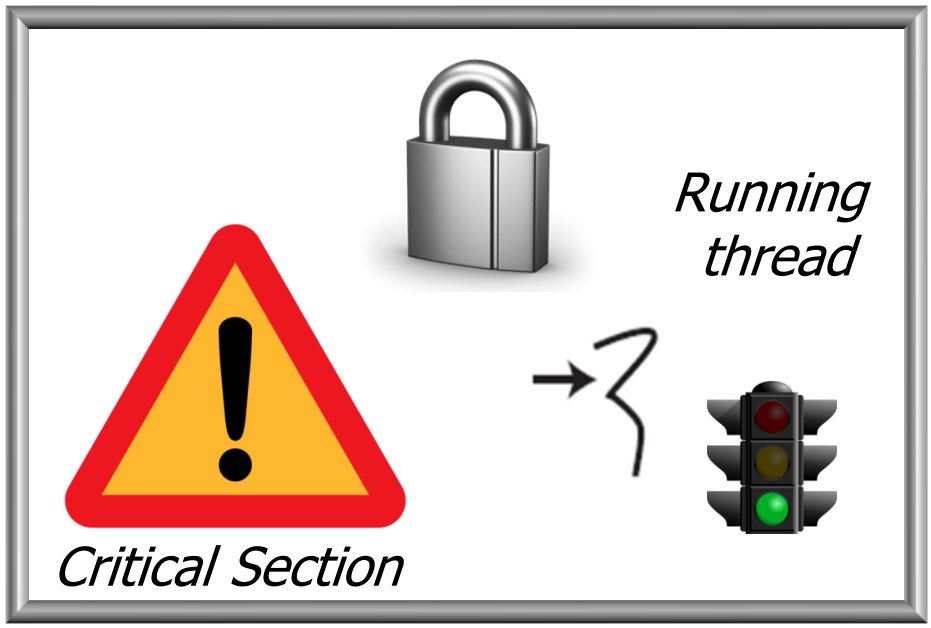
Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

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





See en.wikipedia.org/wiki/Mutual_exclusion

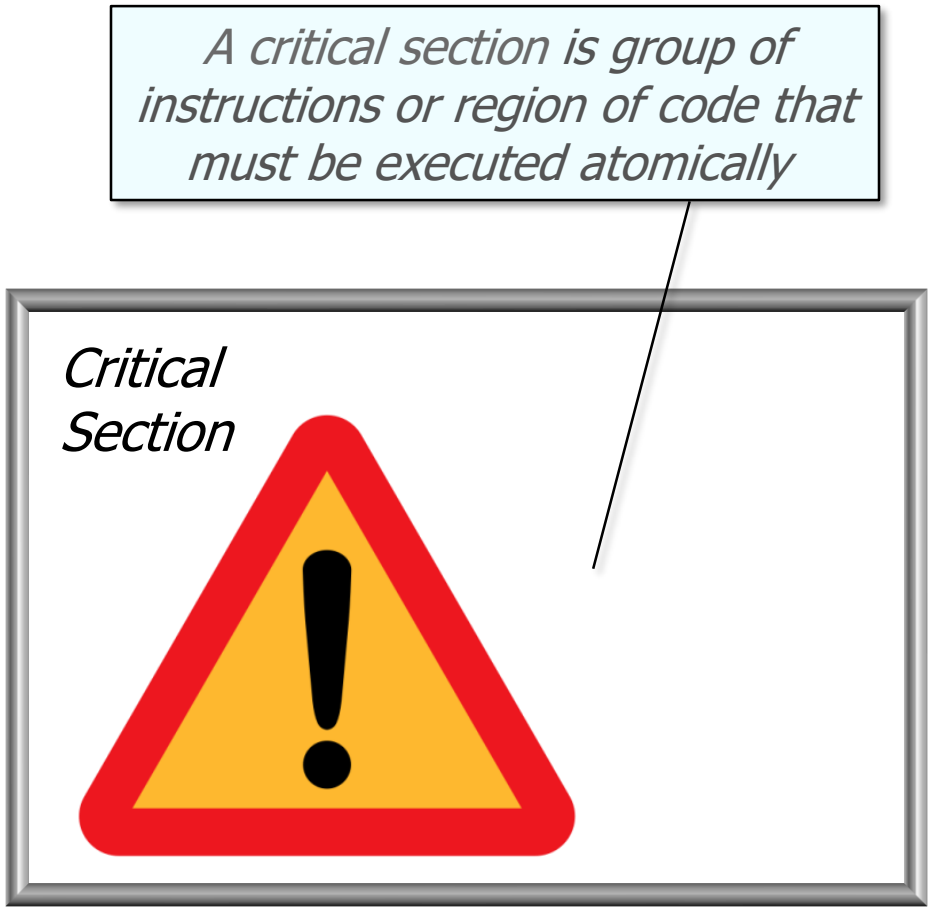
Learning Objectives in this Part of the Lesson

- Understand the concept of mutual exclusion in concurrent programs
- Note a human-known use of mutual exclusion

Overview of Mutual Exclusion Locks

Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"



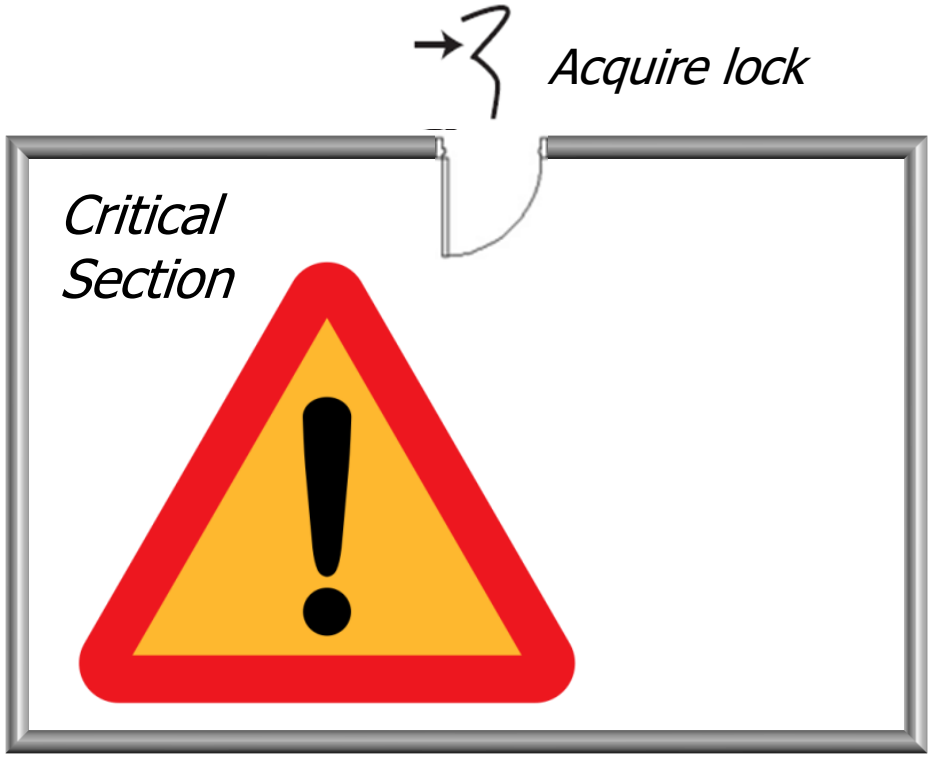
See en.wikipedia.org/wiki/Critical_section

Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time

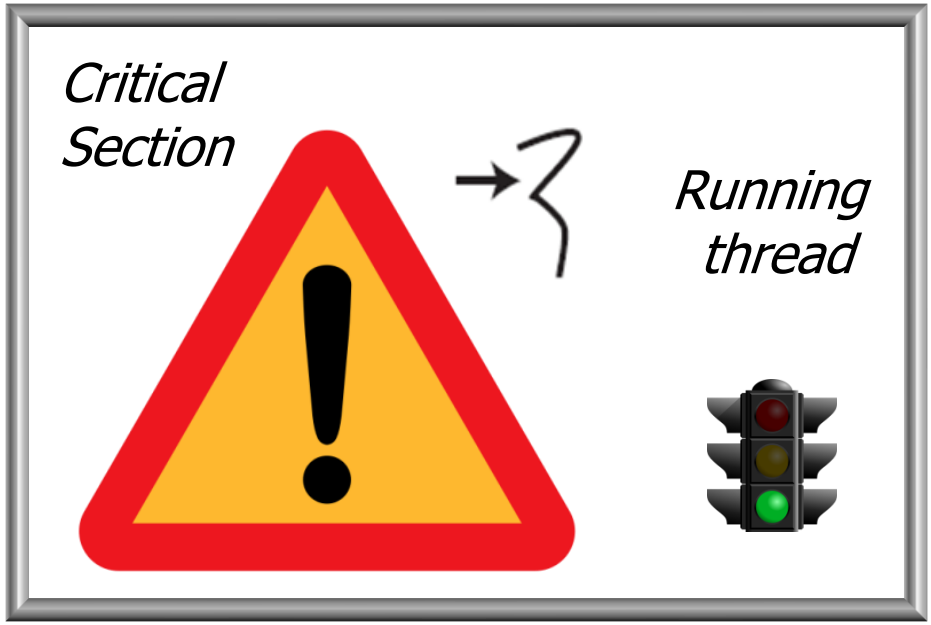
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time



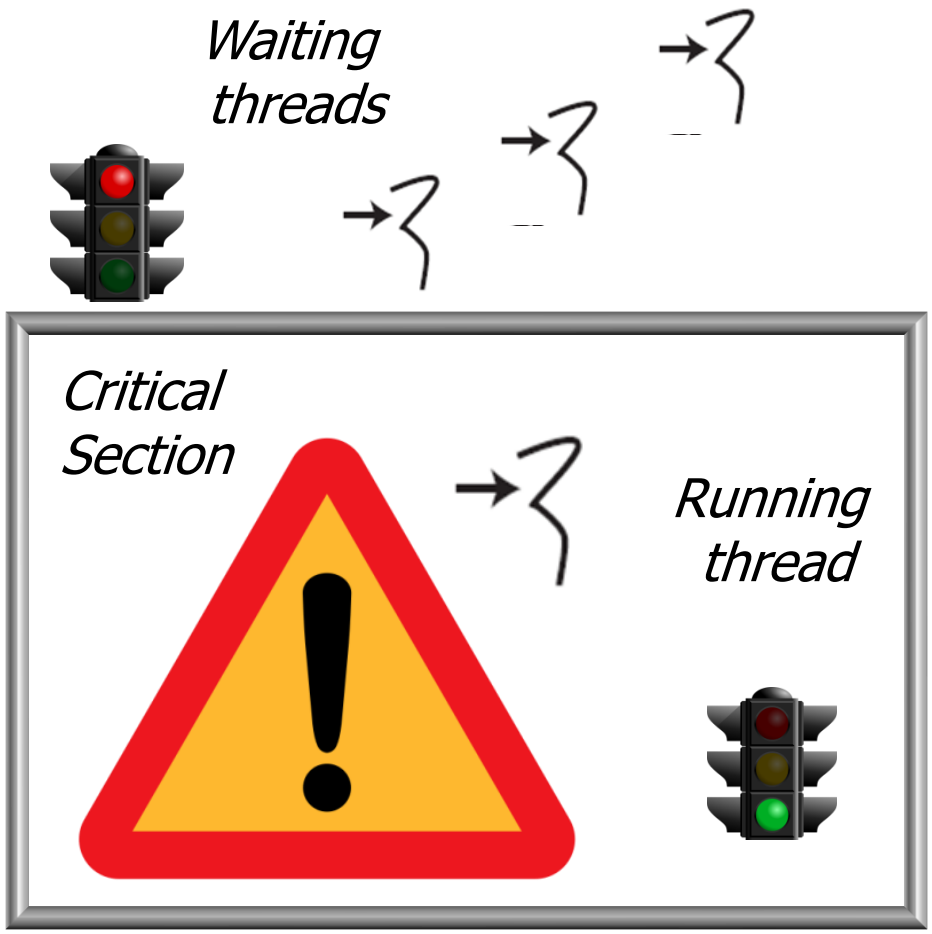
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time



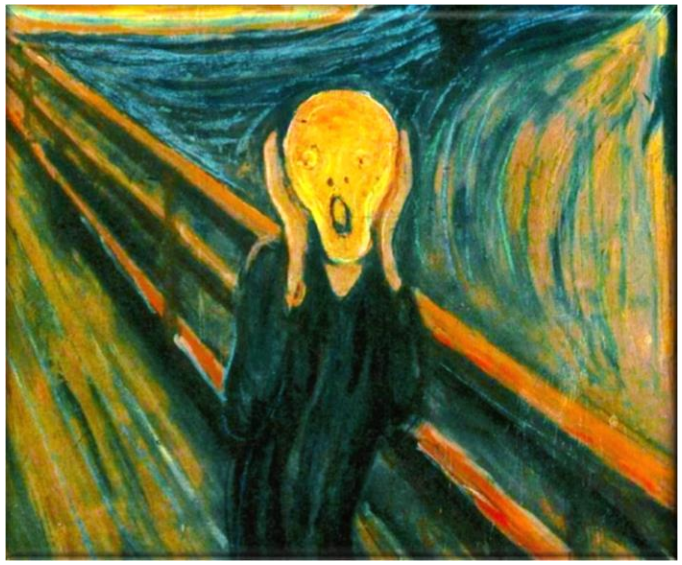
Overview of Mutual Exclusion Locks

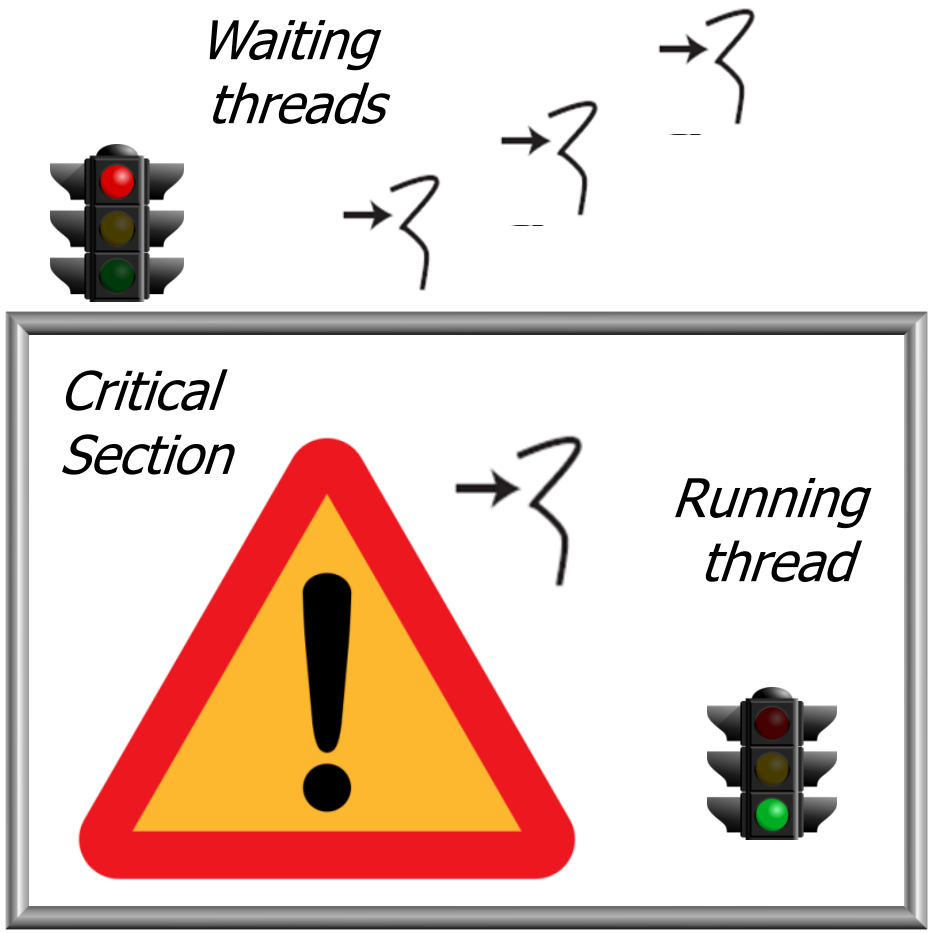
- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time
  - Other threads are kept "at bay"
    - Prevent corruption of shared (mutable) data that can be set/ get by concurrent operations



Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time
  - Other threads are kept "at bay"
    - Prevent corruption of shared (mutable) data that can be set/ get by concurrent operations

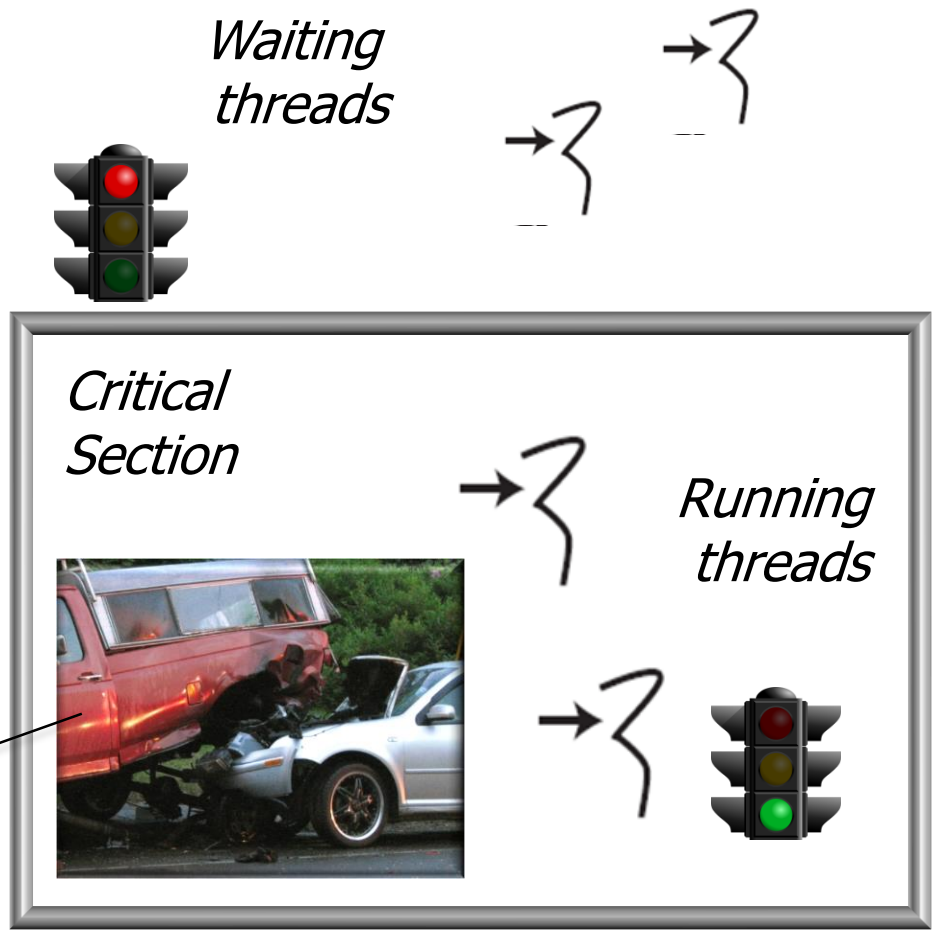




Other threads must obey the locking protocol or chaos will ensue!!

Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time
  - Other threads are kept "at bay"
    - Prevent corruption of shared (mutable) data that can be set/get by concurrent operations
    - Race conditions could occur if multiple threads run within a critical section

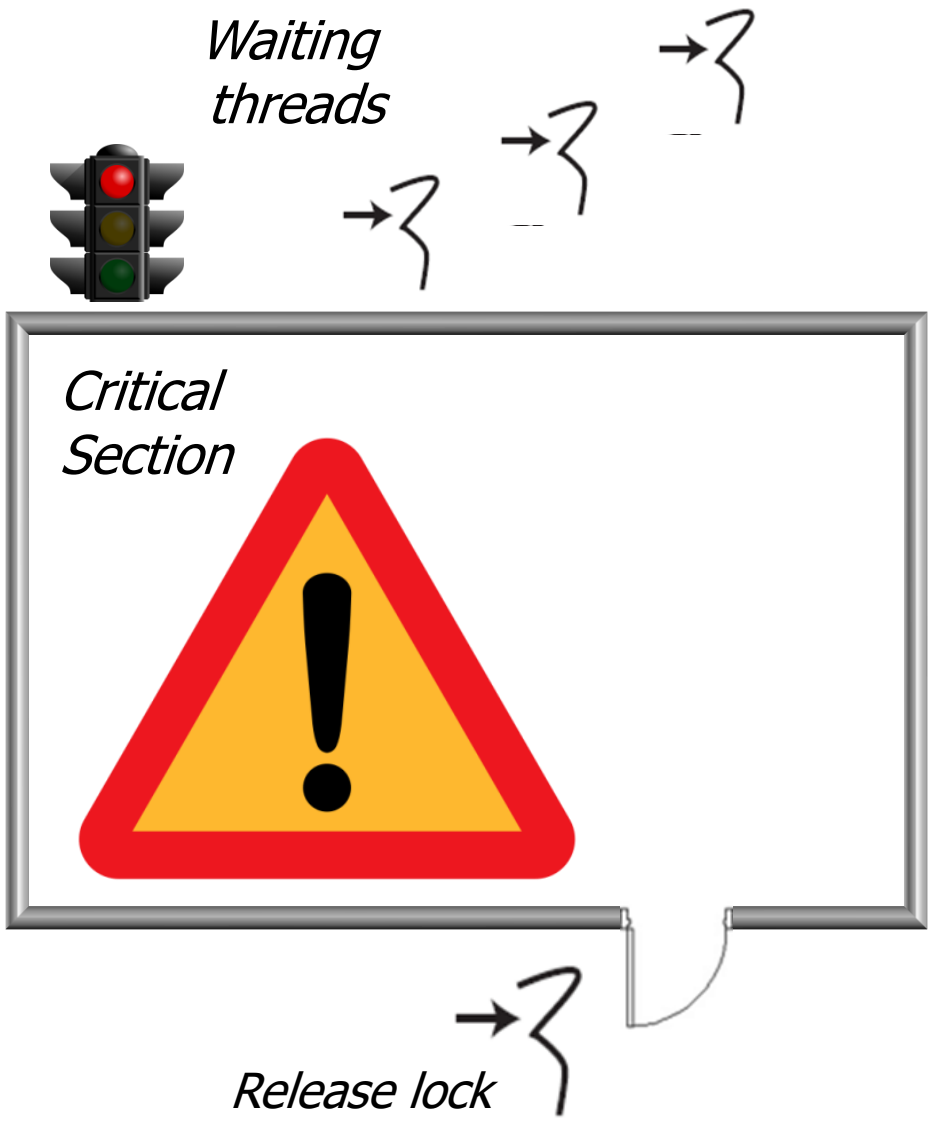


Race conditions arise when a program depends on the sequence or timing of threads for it to operate properly

See en.wikipedia.org/wiki/Race_condition

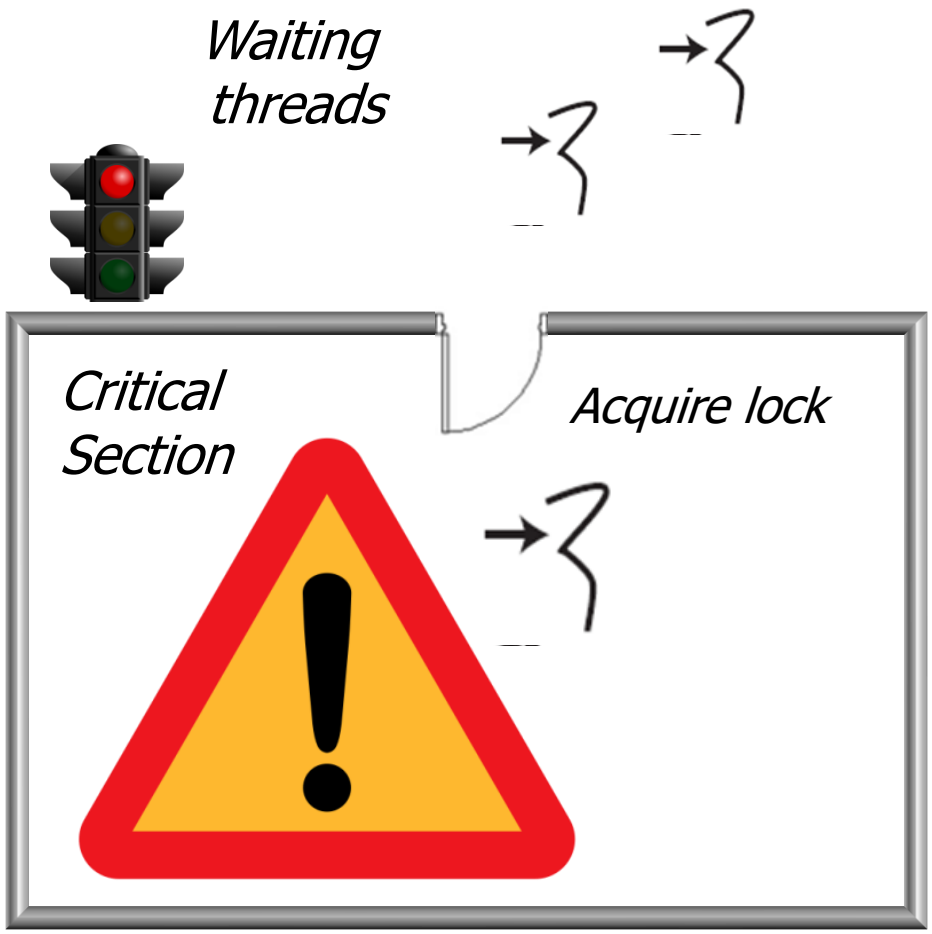
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time
  - Other threads are kept "at bay"
  - After a thread leaves a critical section another thread can enter & start running



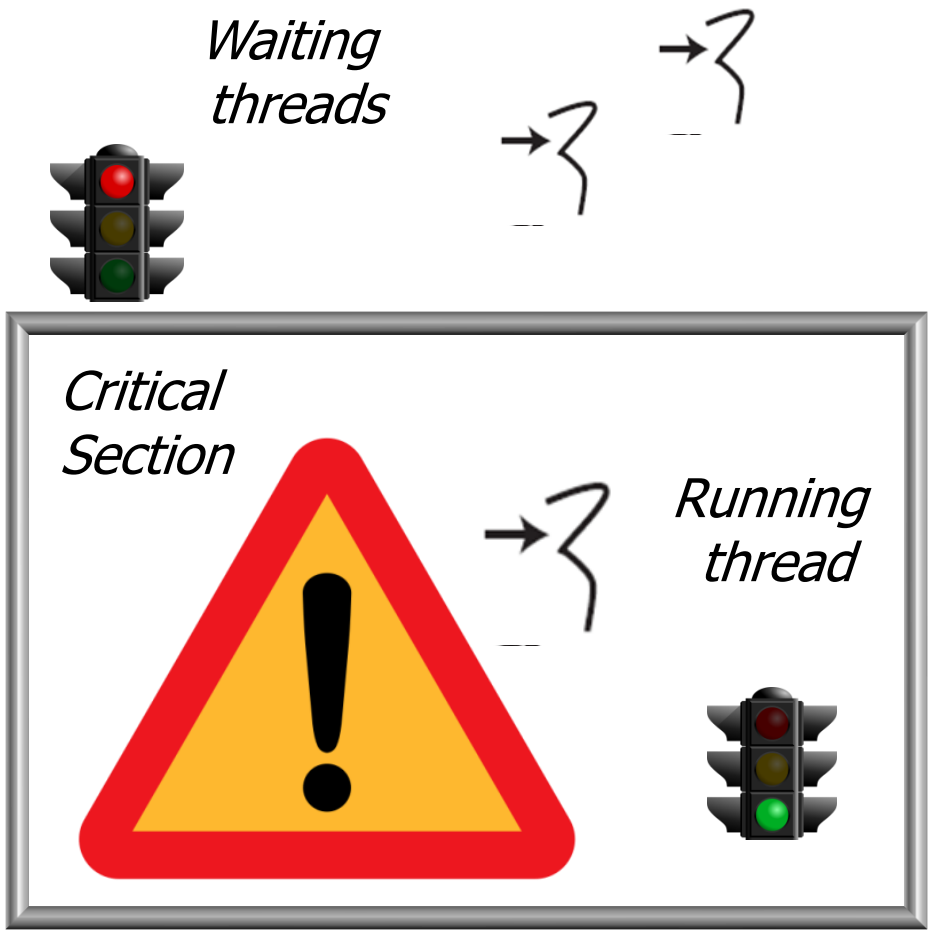
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time
  - Other threads are kept "at bay"
  - After a thread leaves a critical section another thread can enter & start running



Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time
  - Other threads are kept "at bay"
  - After a thread leaves a critical section another thread can enter & start running



Overview of Mutual Exclusion Locks

- A mutex is typically implemented in hardware via atomic operations

Atomic operations appear to occur instantaneously & either change the state of the system successful or have no effect

See en.wikipedia.org/wiki/Linearizability

Overview of Mutual Exclusion Locks

- A mutex is typically implemented in hardware via atomic operations
  - Implemented in Java via the compareAndSwap*() methods in the Unsafe class

Concurrency

And few words about concurrency with `Unsafe`. `compareAndSwap` methods are atomic and can be used to implement high-performance lock-free data structures.

For example, consider the problem to increment value in the shared object using lot of threads.

First we define simple interface `Counter`:

```
interface Counter {
    void increment();
    long getCounter();
}
```

Then we define worker thread `CounterClient`, that uses `Counter`:

```
class CounterClient implements Runnable {
    private Counter c;
    private int num;

    public CounterClient(Counter c, int num) {
        this.c = c;
        this.num = num;
    }

    @Override
    public void run() {
        for (int i = 0; i < num; i++) {
            c.increment();
        }
    }
}
```

See earlier discussion of "*Java Atomic Classes & Operations*"

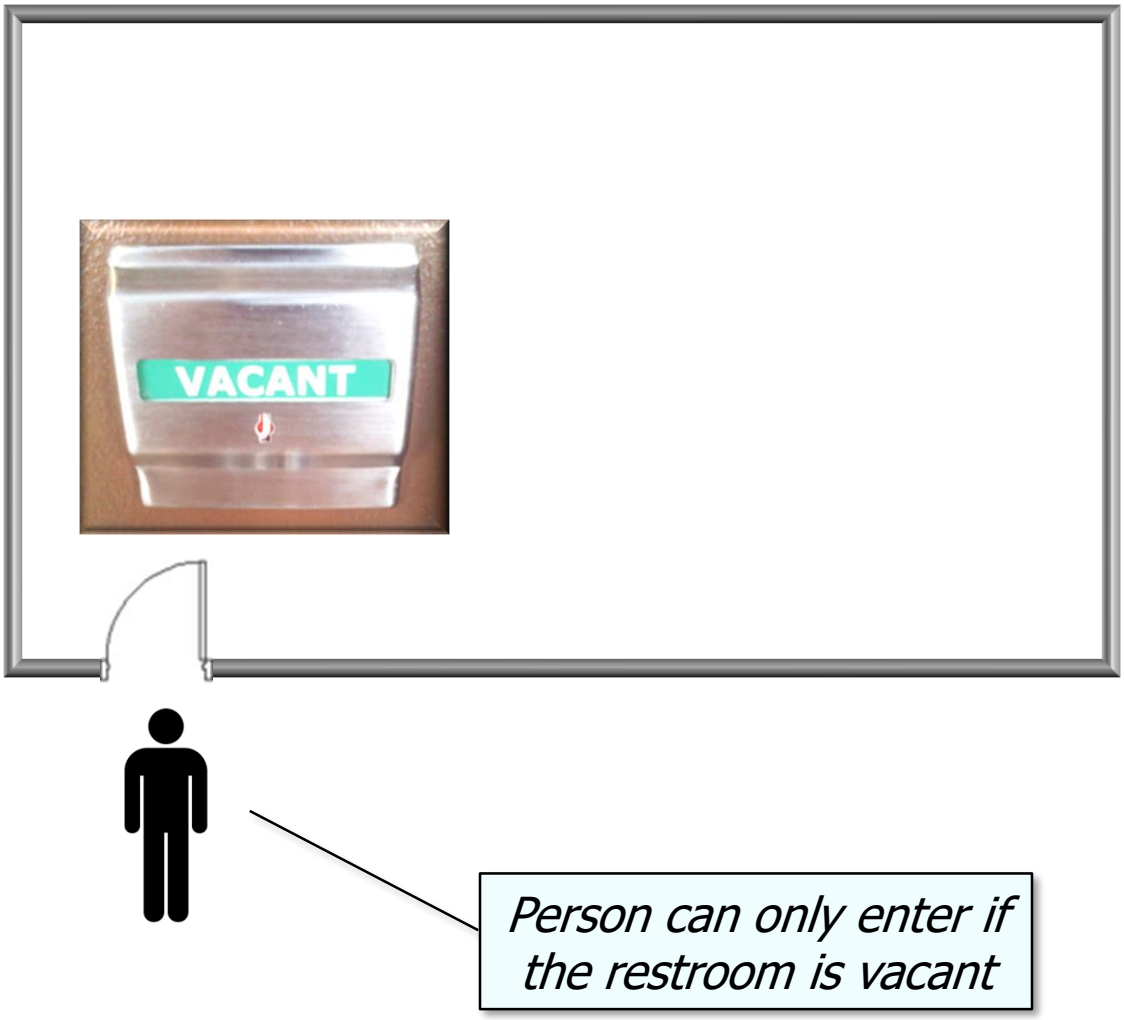
Human Known Use of Mutual Exclusion Locks

Human Known Use of Mutual Exclusion Locks

- A human known use of mutual exclusion locks is an airplane restroom protocol

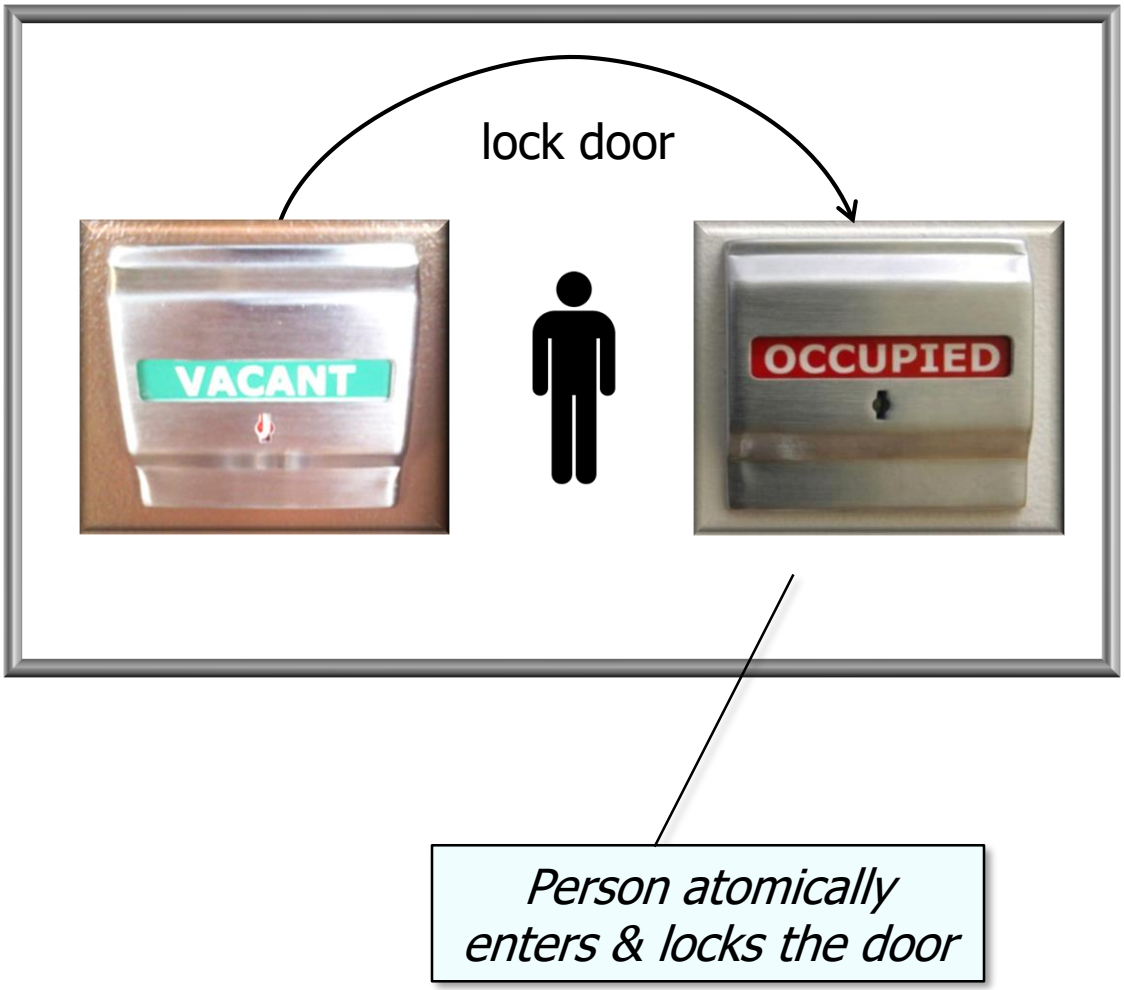
Human Known Use of Mutual Exclusion Locks

- A human known use of mutual exclusion locks is an airplane restroom protocol



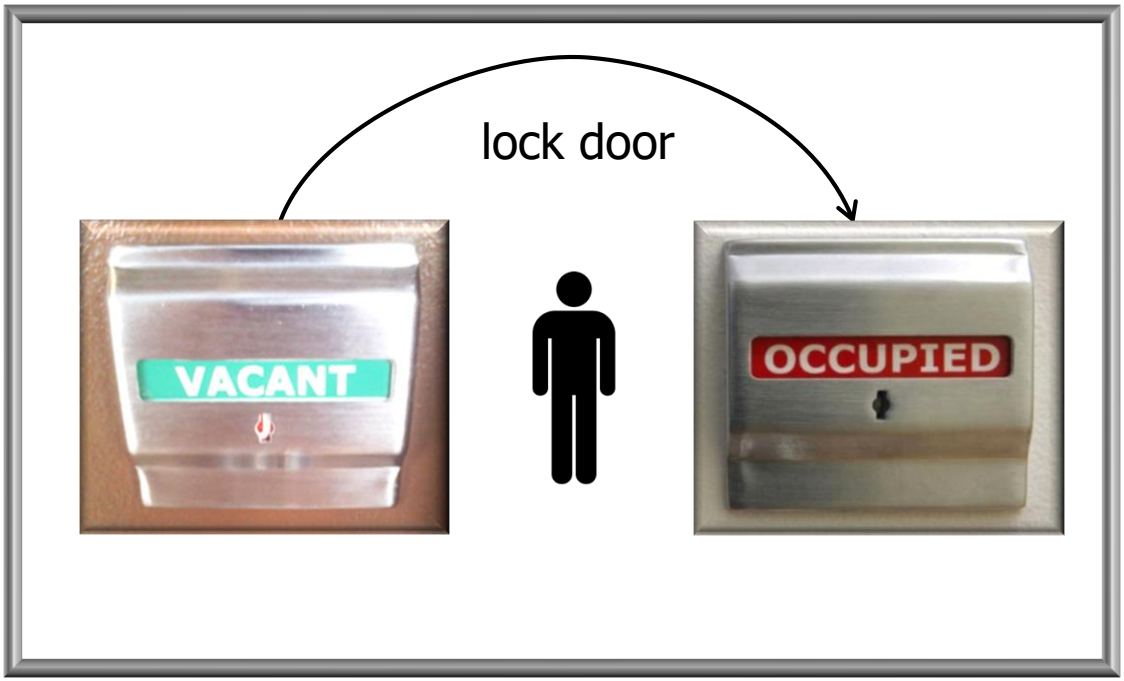
Human Known Use of Mutual Exclusion Locks

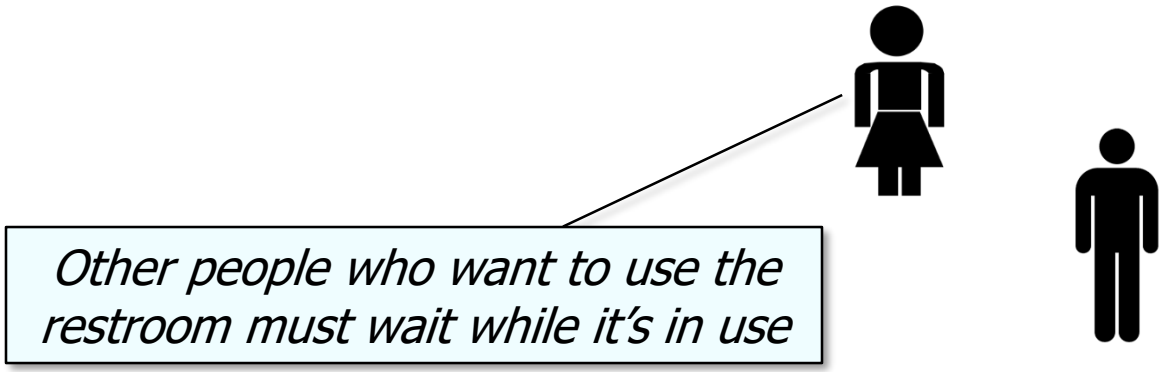
- A human known use of mutual exclusion locks is an airplane restroom protocol



Human Known Use of Mutual Exclusion Locks

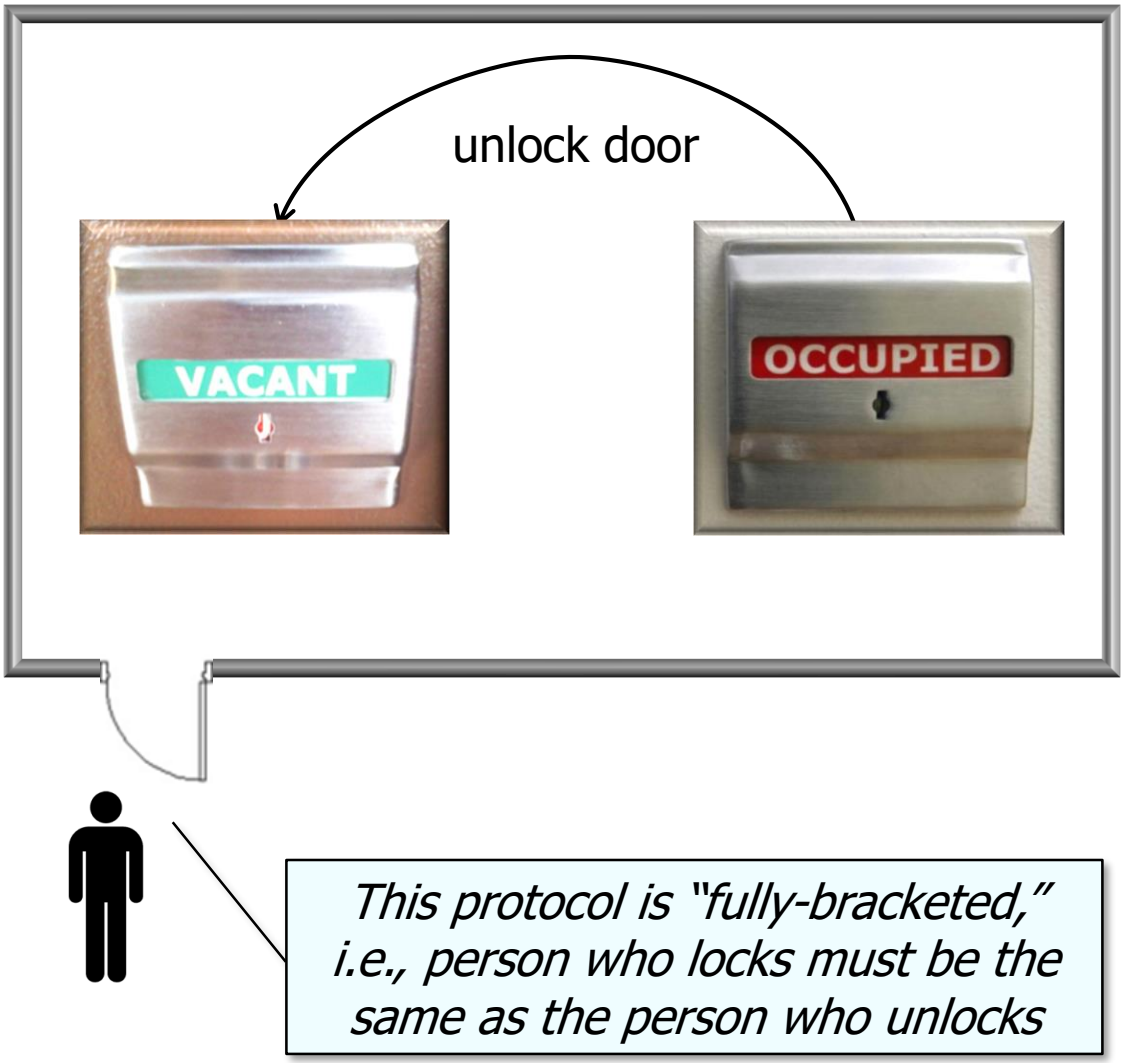
- A human known use of mutual exclusion locks is an airplane restroom protocol





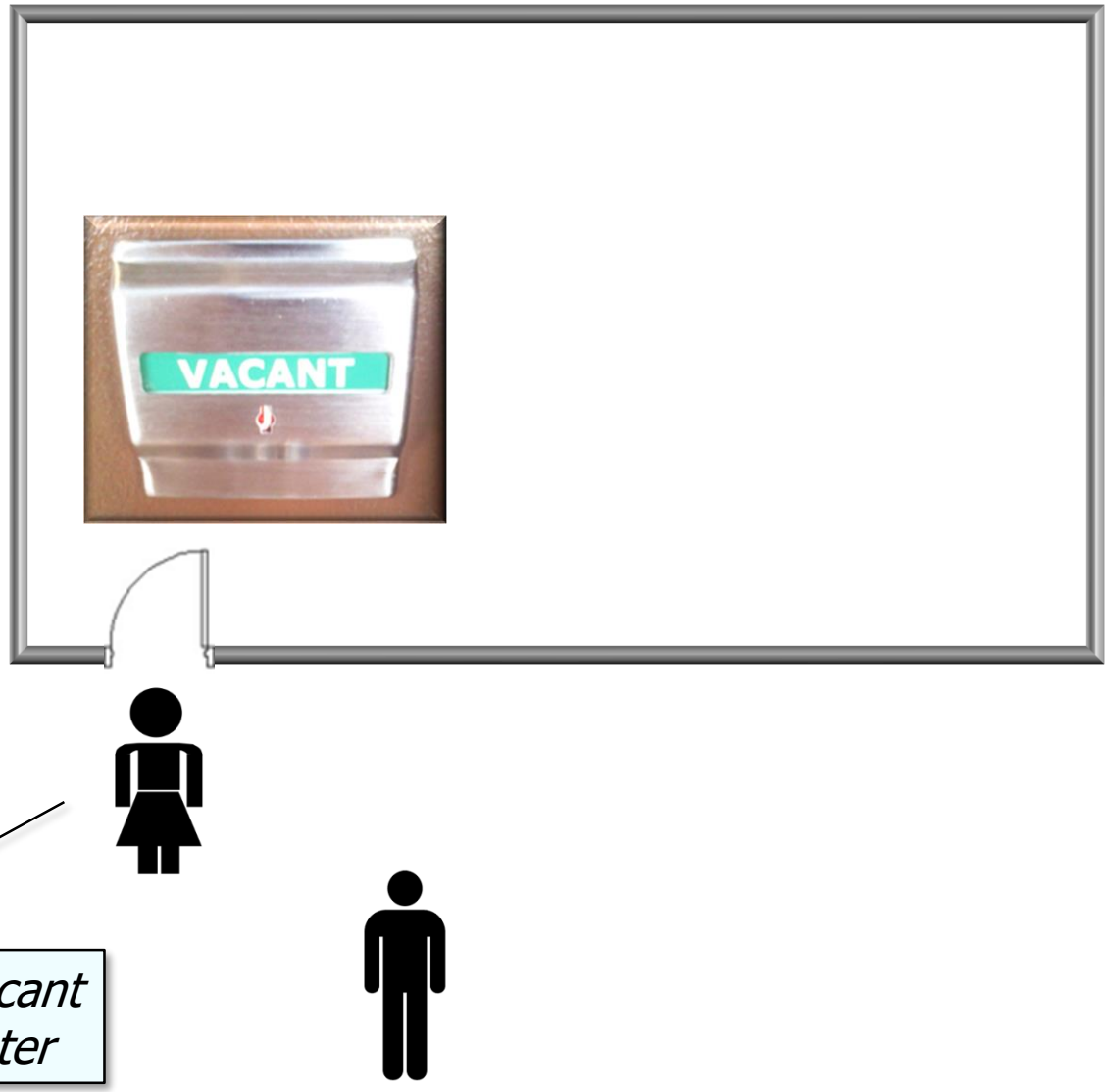
Human Known Use of Mutual Exclusion Locks

- A human known use of mutual exclusion locks is an airplane restroom protocol



Human Known Use of Mutual Exclusion Locks

- A human known use of mutual exclusion locks is an airplane restroom protocol



End of Java ReentrantLock: Introduction