COS40003 Concurrent Programming

Lecture 5: Lock (I)

CRICOS 00111D
TOID 3059
KNOW
ING

Review

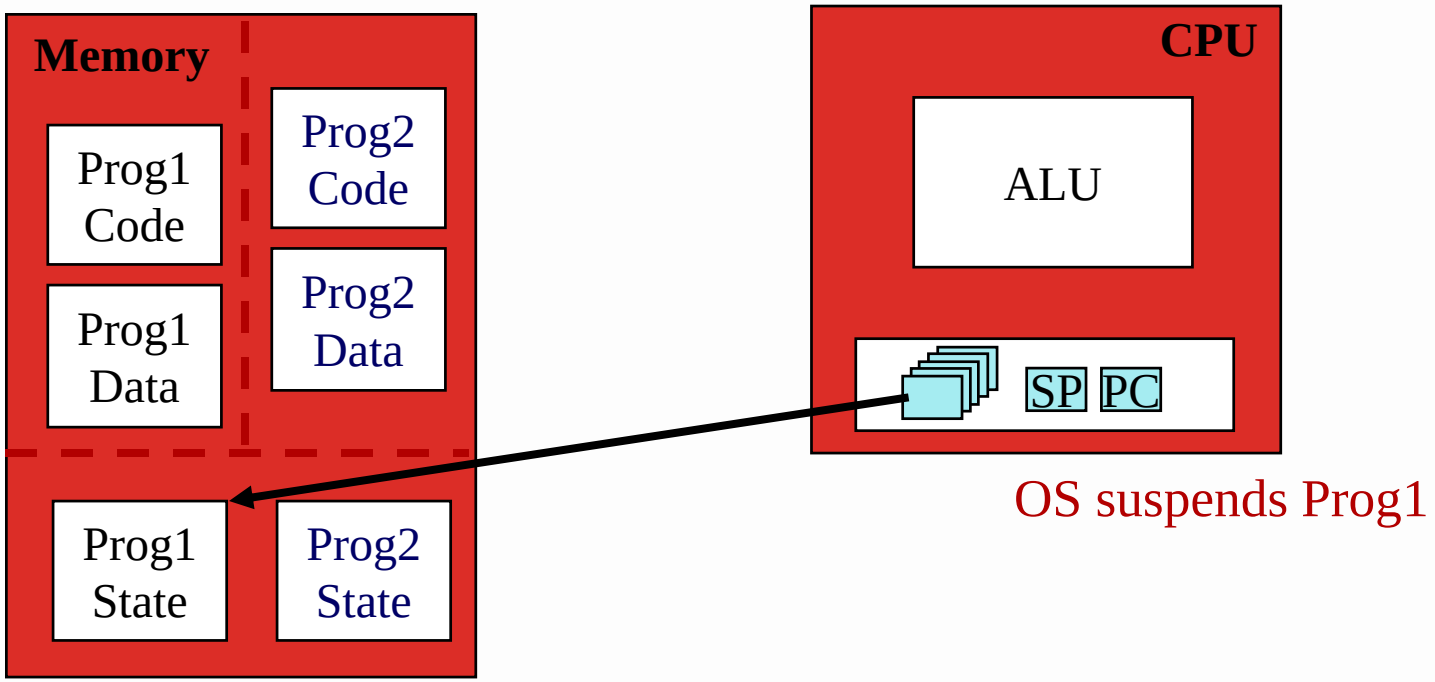
- Question: why process context switch is more expensive than thread context switch?

- Process switch:
  - CPU state and memory working space
- Thread switch
  - CPU state

KNOW ING

Recall: process context switch

- Save registers, program counter, stack pointer of Prog1



Outline

- Why there is concurrency problem?
- Key concurrency terms
- Lock implementation
  - Using Lock object
  - Using Synchronized keyword

Example 3 (Recall)

```
3  public class IncrementTest implements Runnable{
4
5      static int classData = 0;
6      int        instanceData = 0;
7
8      @Override
9      public void run() {
10         int localData = 0;
11
12         while (localData < 10000000) {
13             localData++;
14             instanceData++;
15             classData++;
16         }
17
18         System.out.println("localData:   " + localData +
19                            "\tinstanceData:   " + instanceData +
20                            "\tclassData:   " + classData);
21     }
22
23
24     public static void main(String[] args) {
25         // TODO Auto-generated method stub
26         IncrementTest instance = new IncrementTest();
27
28         Thread t1 = new Thread(instance);
29         Thread t2 = new Thread(instance);
30
31         t1.start();
32         t2.start();
33
34     }
35
36 }
37
```

is shared between all instances of this clas

If done in a single procedure

{
IncrementTest t1 =
new IncrementTest()

IncrementTest t2 =
new IncrementTest()

t1.classData ++;
t2.classData ++;

// classData increased twice
}

Example 3 (Recall)

```
3  public class IncrementTest implements Runnable{
4
5      static int classData = 0;
6      int        instanceData = 0;
7
8      @Override
9      public void run() {
10         int localData = 0;
11
12         while (localData < 10000000) {
13             localData++;
14             instanceData++;
15             classData++;
16         }
17
18         System.out.println("localData:   " + localData +
19                            "\tinstanceData:   " + instanceData +
20                            "\tclassData:   " + classData);
21     }
22
23
24     public static void main(String[] args) {
25         // TODO Auto-generated method stub
26         IncrementTest instance = new IncrementTest();
27
28         Thread t1 = new Thread(instance);
29         Thread t2 = new Thread(instance);
30
31         t1.start();
32         t2.start();
33
34     }
35
36 }
37
```

Each object has a copy of instanceData

```
{
IncrementTest t1 =
new IncrementTest()

IncrementTest t2 =
new IncrementTest()

t1.instanceData ++;
t2.instanceData ++;

// t1's instanceData
increased once
// t2's instanceData
increased once
}
```

Example 3 (Recall)

```
3  public class IncrementTest implements Runnable{
4
5      static int classData = 0;
6      int        instanceData = 0;
7
8      @Override
9      public void run() {
10         int localData = 0;
11
12         while (localData < 10000000) {
13             localData++;
14             instanceData++;
15             classData++;
16         }
17
18         System.out.println("localData:   " + localData +
19                            "\tinstanceData:   " + instanceData +
20                            "\tclassData:   " + classData);
21     }
22
23
24     public static void main(String[] args) {
25         // TODO Auto-generated method stub
26         IncrementTest instance = new IncrementTest();
27
28         Thread t1 = new Thread(instance);
29         Thread t2 = new Thread(instance);
30
31         t1.start();
32         t2.start();
33
34     }
35
36 }
37
```

not shared, used within the calling function

```
{
IncrementTest t1 =
new IncrementTest()

IncrementTest t2 =
new IncrementTest()

t1.start();
t1.start();

// localData only
available in the function

}
```

Results

- localData = 10,000,000
- instanceData != 20,000,000
- classData != 20,000,000
- Why?

- Try: create thread t2 with another instance and see the result:

```
IncrementTest instance2 = new IncrementTest();
Thread t2 = new Thread(instance2);
```

Example 3 – Shared variables in concurrency

`i++;`	is actually	1. Load data from variable `i` 2. Increment data by `1` 3. Store data to variable `i`

- Decomposed in assembly code:
  1. Mov Addr1 R1 (mov value at Addr1 to R1)
  2. Add 1 R1 (increase Register R1 by 1)
  3. Mov R1 Addr1 (store value at R1 to Addr1)

  Addr1: value in memory
  R1: value in CPU register

Beginning value at Addr1 is 50

- Thread1
  1. Mov Addr1 R1
  2. Add 1 R1

  (context switch to Thread 2, R1 saved)

  (context switch back to Thread1, R1 resored)

  6. Mov R1 Addr1

- Thread2

  3. Mov Addr1 R1
  4. Add 1 R1
  5. Mov R1 Addr1

R1	Addr1
50	50
51	50
50	50
51	50
51	51
51	51

Outline

- Why there is concurrency problem?
- Key concurrency terms
- Lock implementation
  - Using Lock object
  - Using Synchronized keyword

Key Concurrency Terms

- Critical section
- Race condition
- Indeterminate
- Mutual exclusion
- Atomicity

Critical section

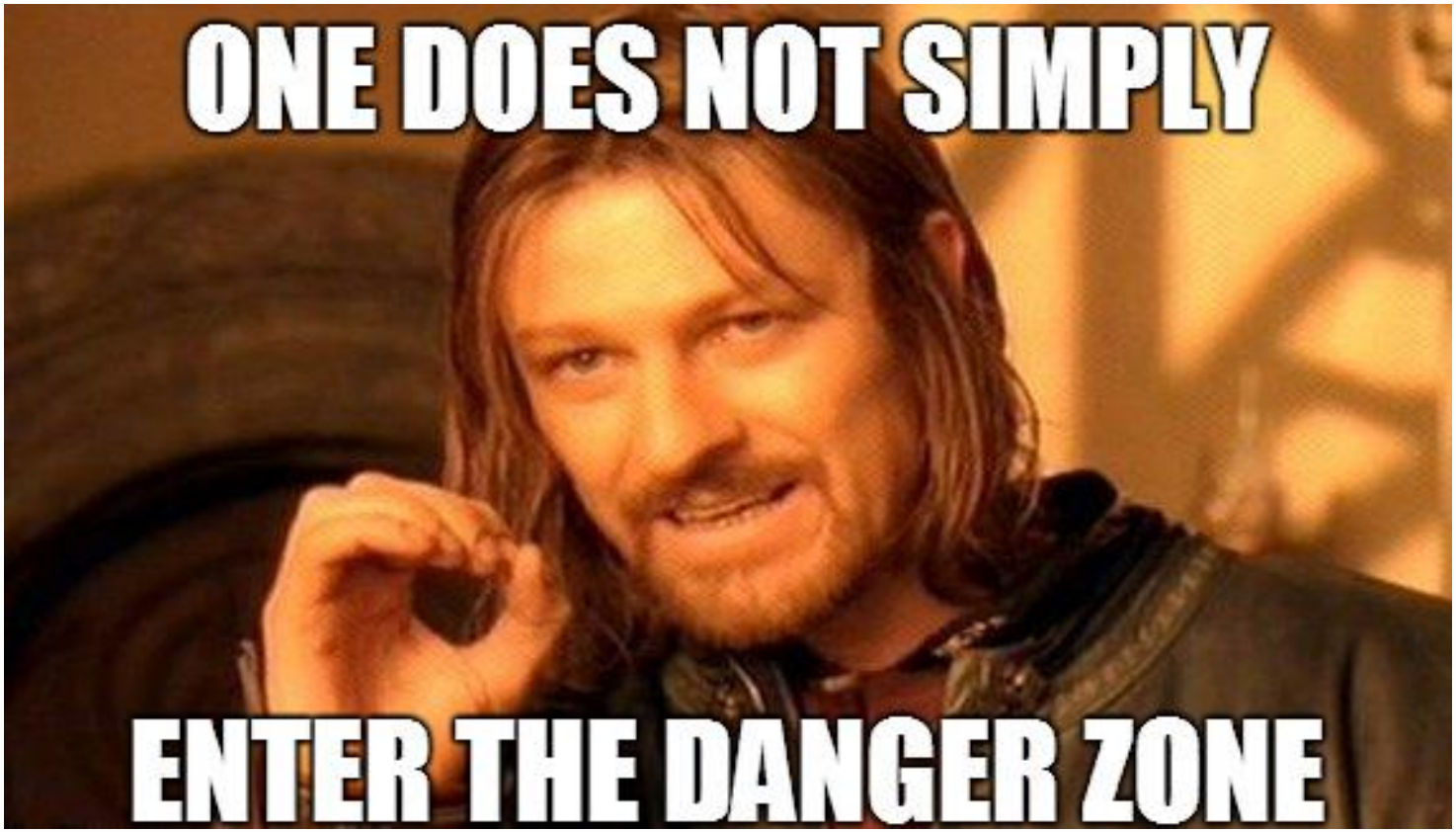
- A critical section is a piece of code that accesses a shared resource, usually a variable or data structure.

```
public void run() {
    int localData = 0;

    while (localData < 10000000) {
        localData++;

        instanceData++;
        classData++;

    }
```



Attribute to: The lord of the rings

Race condition

- A race condition arises if multiple threads of execution enter the critical section at roughly the same time; eg., both attempt to update the shared data structure, leading to an undesirable outcome.

- Example:
  - Both threads want to run
    instanceData ++
    classData++

Indeterminate

- An indeterminate program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when.

- Example:
  - Run example3 multiple times, get different results

Mutual exclusion

- To avoid these problems, threads should use some kind of mutual exclusion primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

- Example
  - Lock (introduced later)

Atomicity

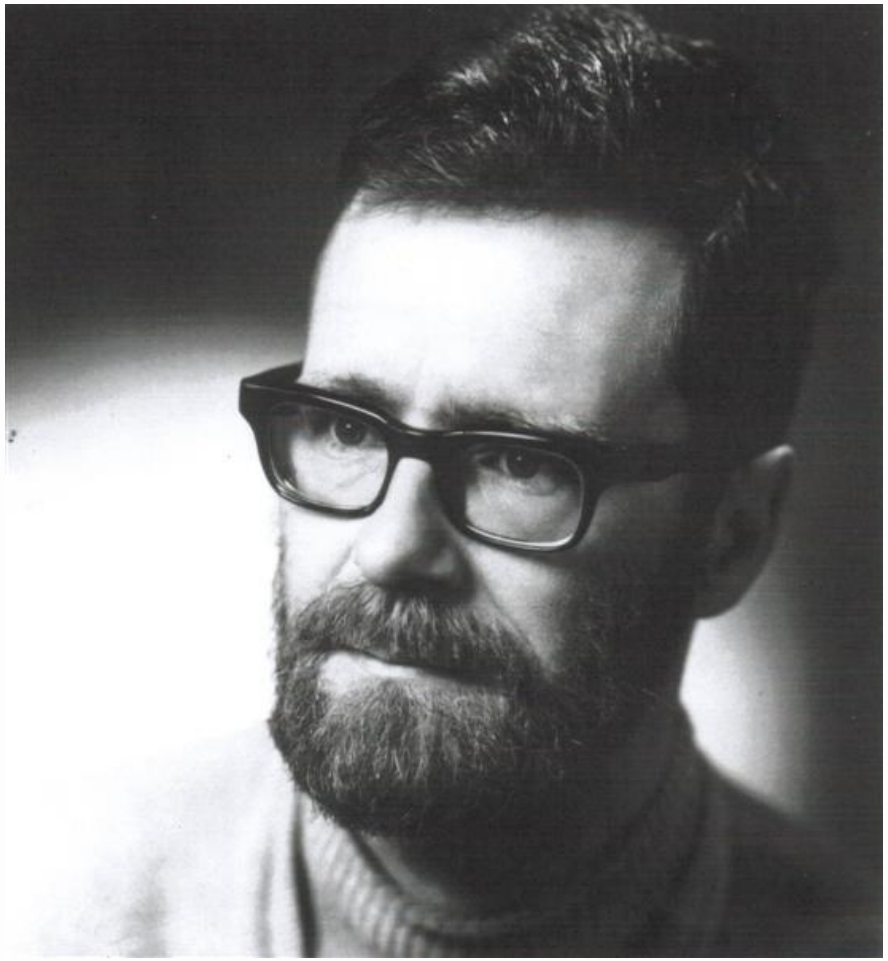
- Means: all or none

  either it has not run at all, or it has run to completion

Similar idea applied in database:

- Transaction: grouping of many actions into a single atomic action
  - Eg., A transfers $50 to B

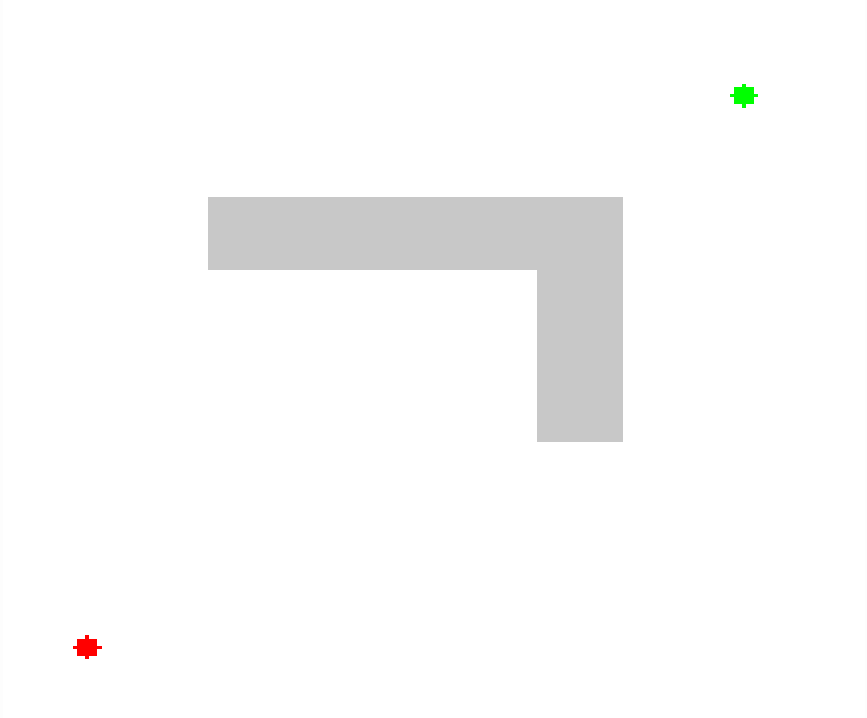
Edsger W. Dijkstra



- Turing Award (1972)
- “No other individual has had a larger influence on research in principles of distributed computing”
- https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

Dijkstra's algorithm

- is an algorithm for finding the shortest paths between nodes in a graph



Key Concurrency Terms Summary

- A critical section is a piece of code that accesses a *shared* resource, usually a variable or data structure.

- A race condition arises if multiple threads of execution enter the critical section at roughly the same time; eg., both attempt to update the shared data structure, leading to an undesirable outcome.

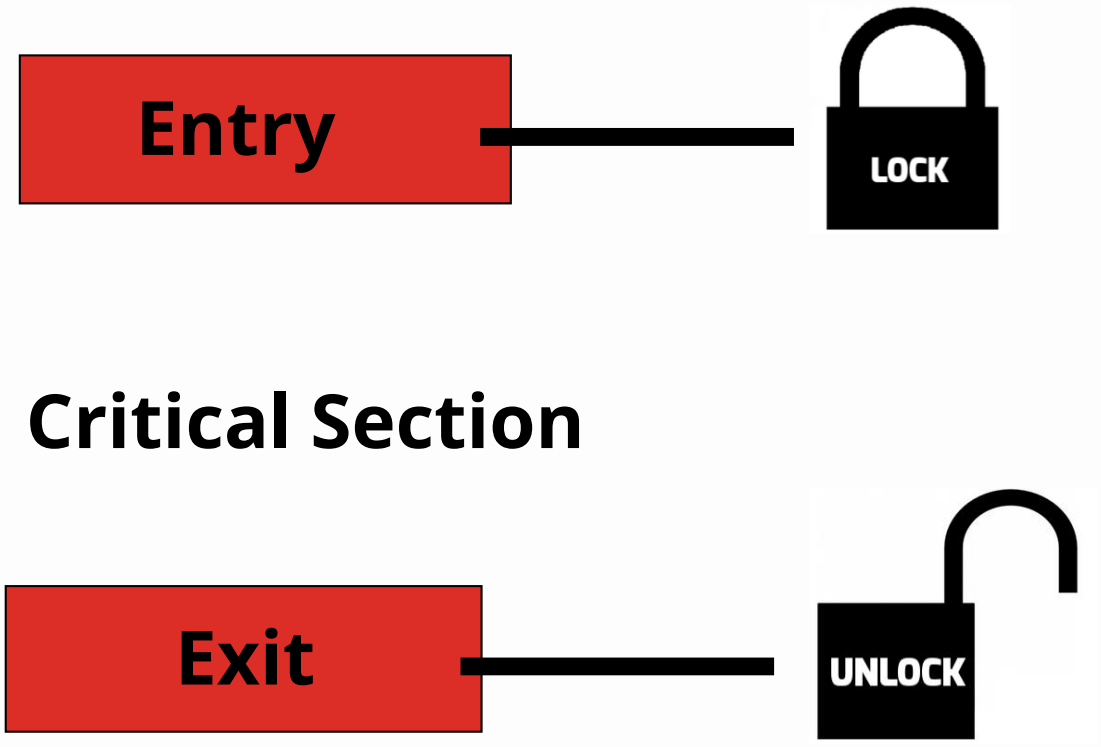
- An indeterminate program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when.

- To avoid these problems, threads should use some kind of mutual exclusion primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

Outline

- Why there is concurrency problem?
- Key concurrency terms
- Lock implementation
  - Using Lock object
  - Using Synchronized keyword

How to guarantee ME (Mutual Exclusion)?



A Lock is similar to a phone booth

- Phone booth can accommodate only one person
- If the booth is empty the first person goes inside
- 2nd person has to wait until the first person leaves the booth

Bangkok ,1986

Acquiring and Releasing Locks

Thread B

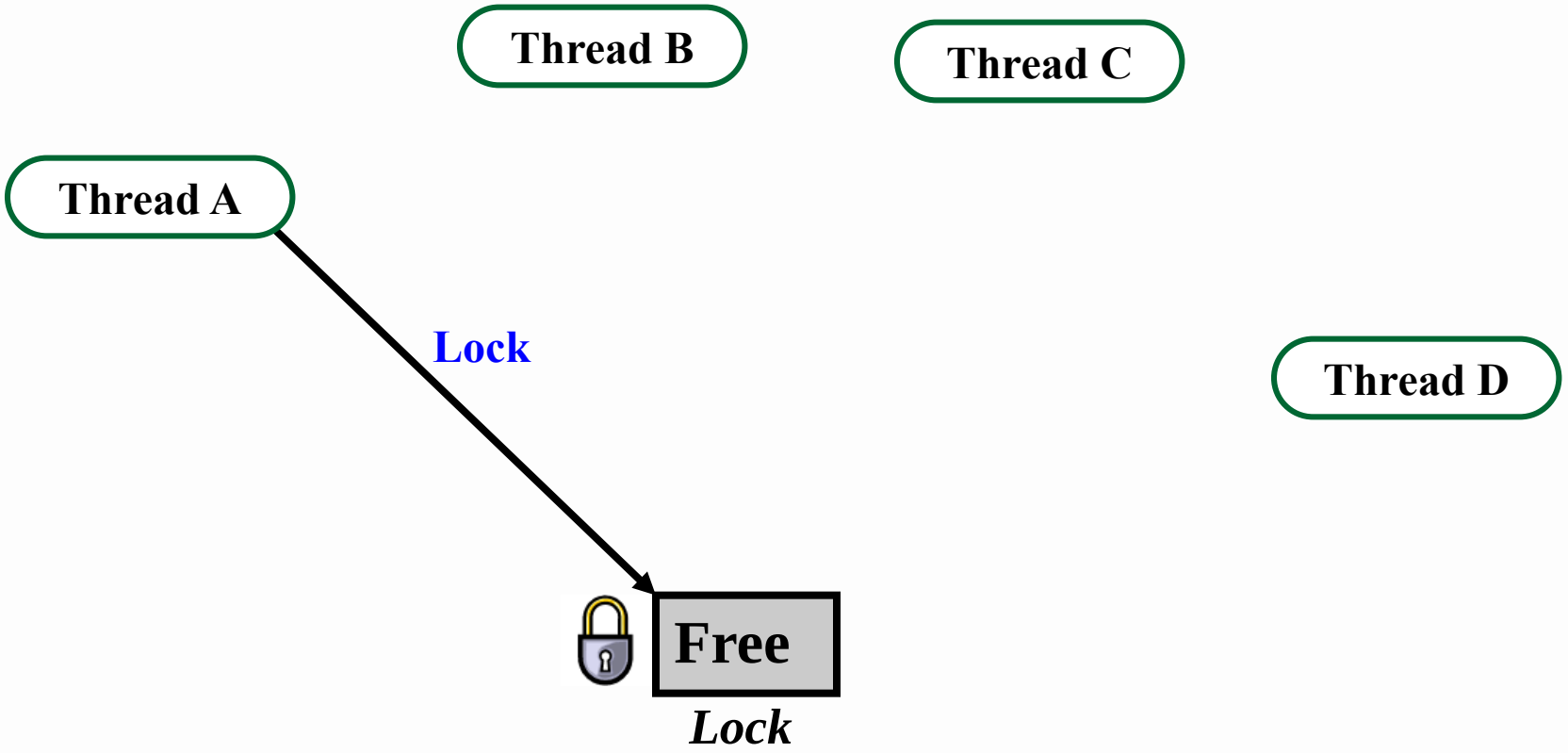
Thread C

Thread A

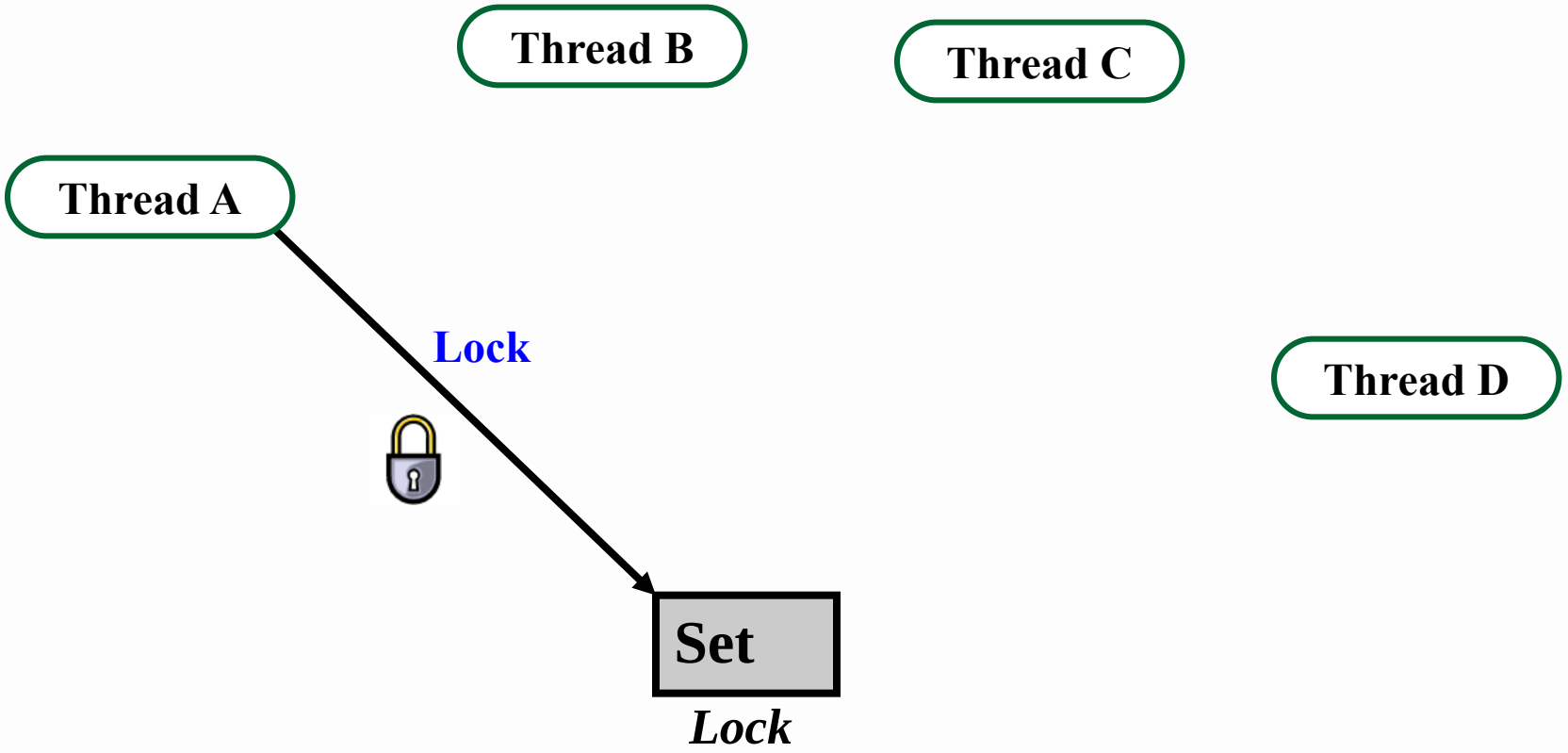
Thread D

Lock

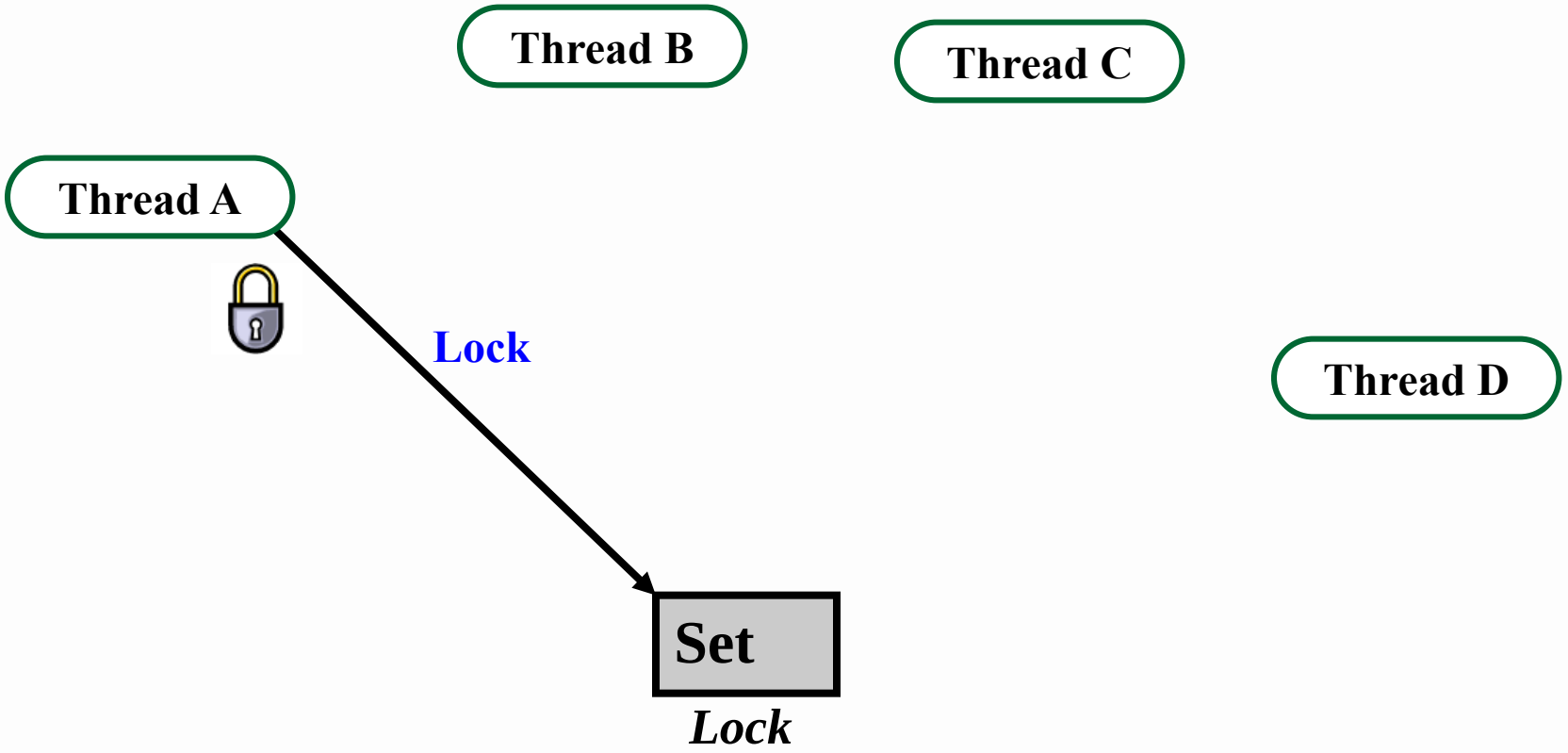
Acquiring and Releasing Locks



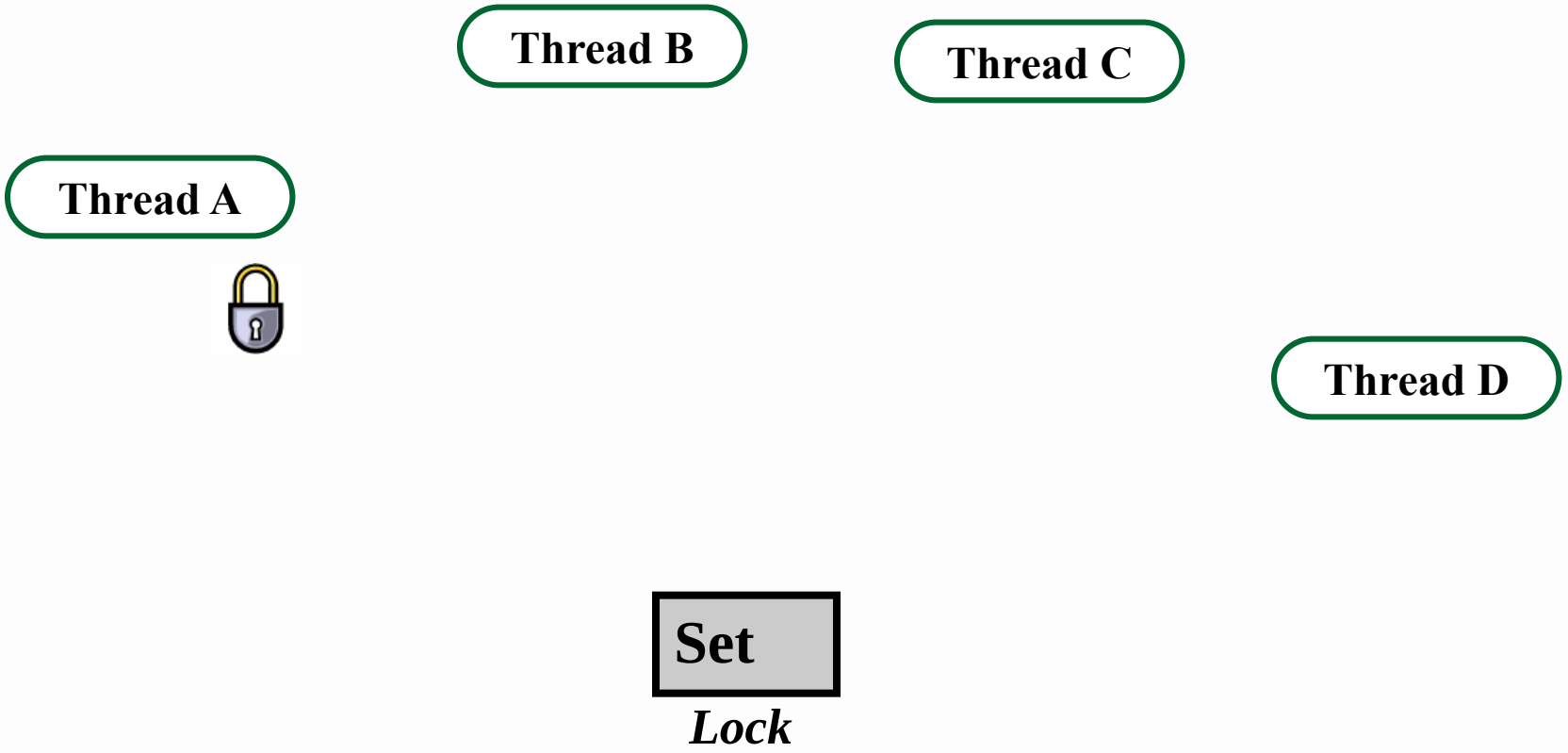
Acquiring and Releasing Locks



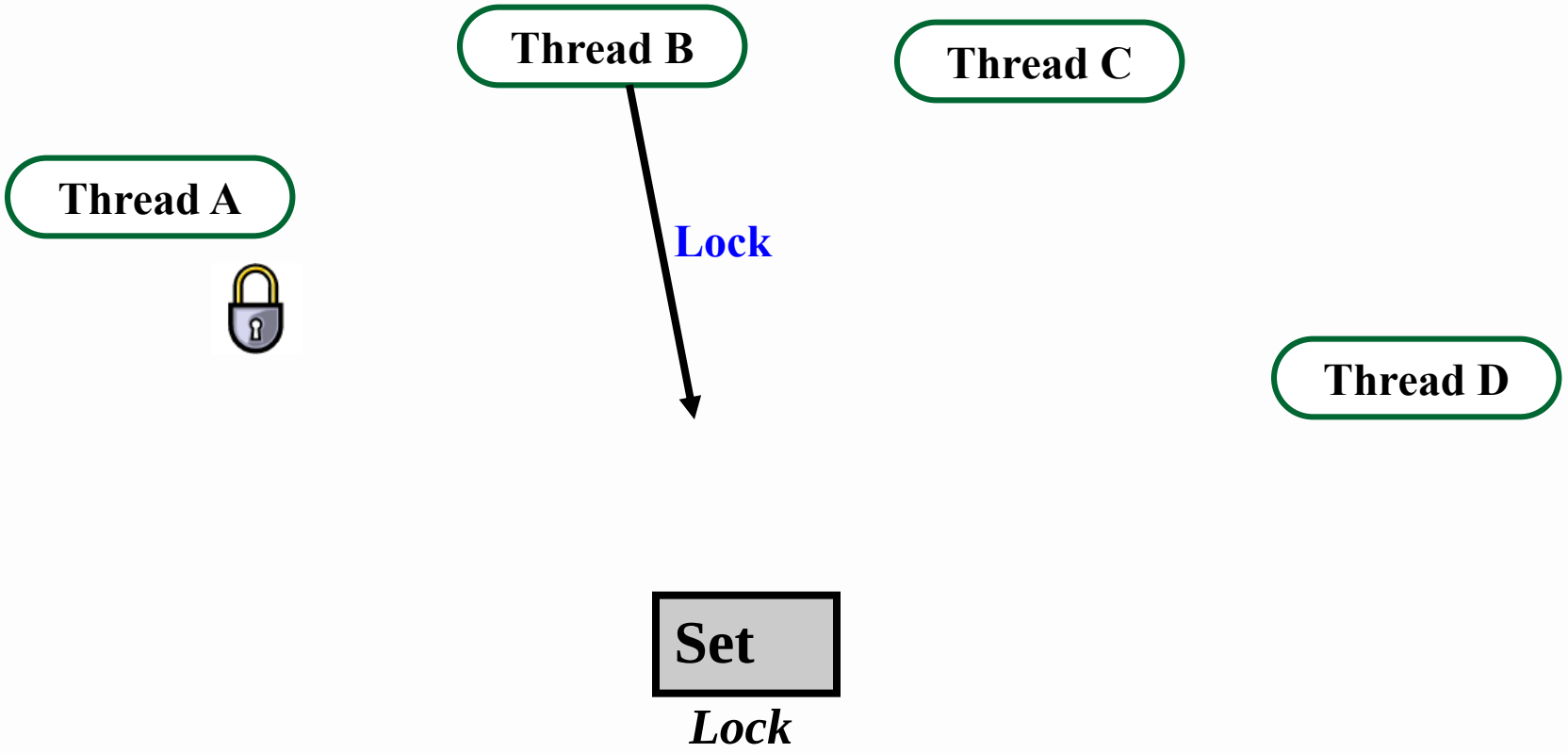
Acquiring and Releasing Locks



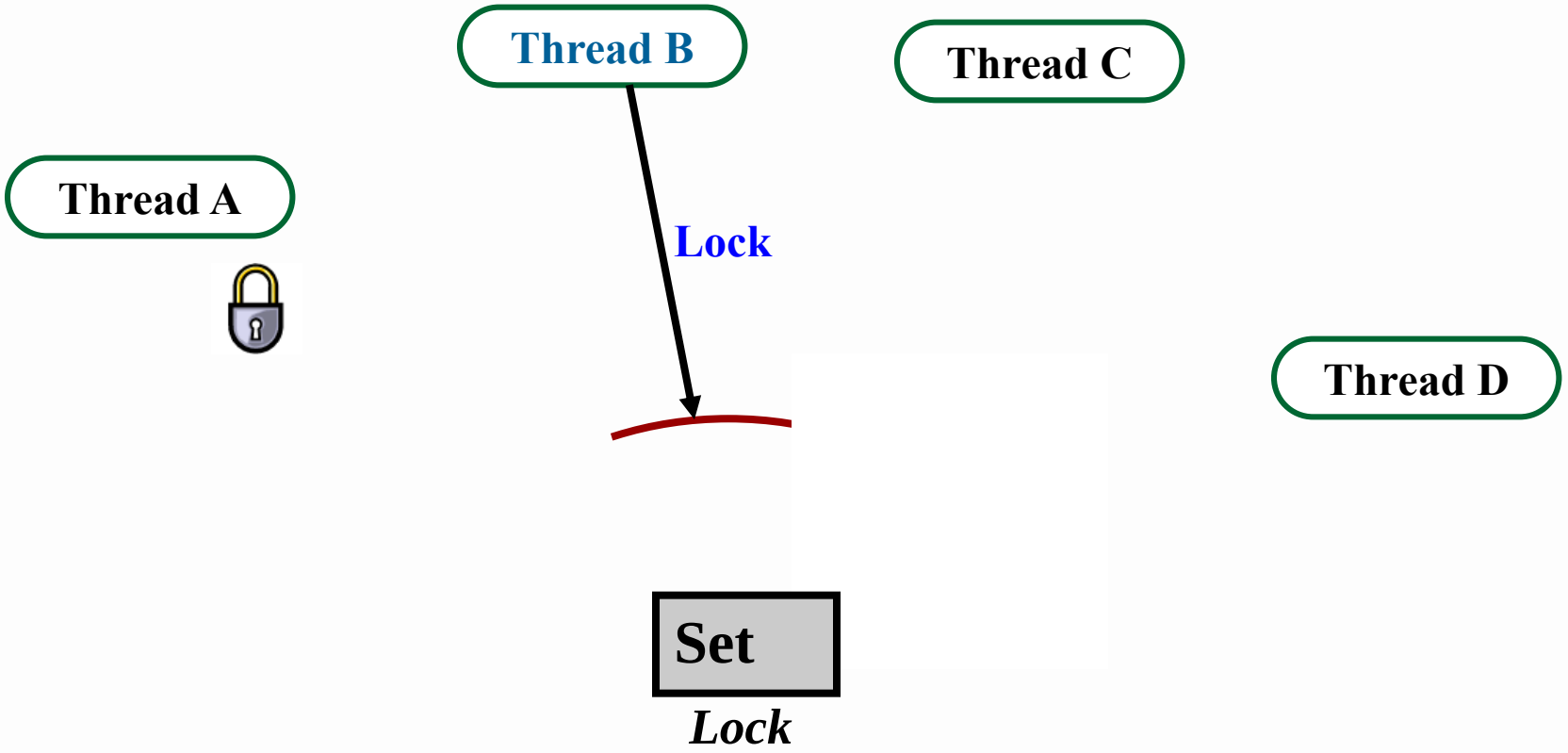
Acquiring and Releasing Locks



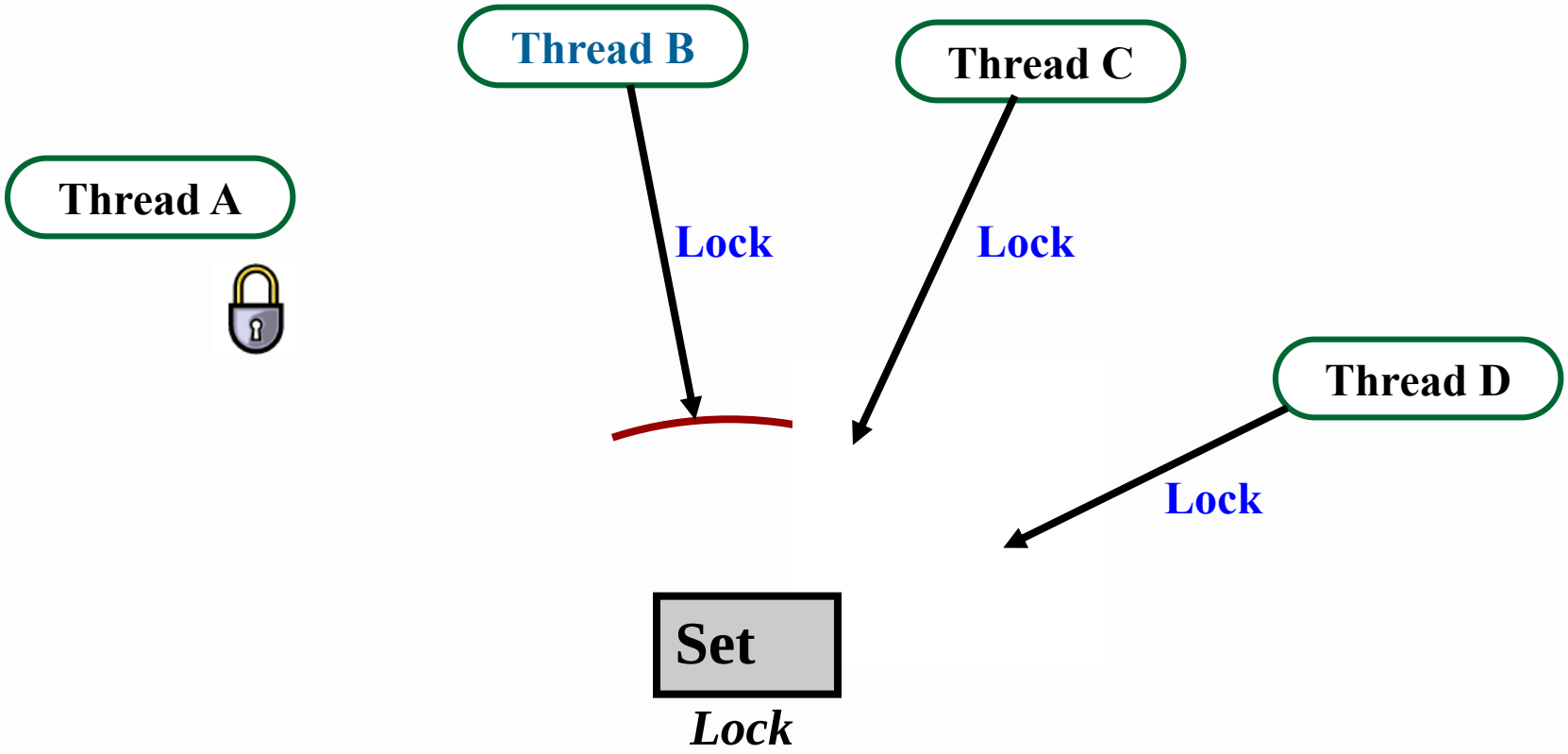
Acquiring and Releasing Locks



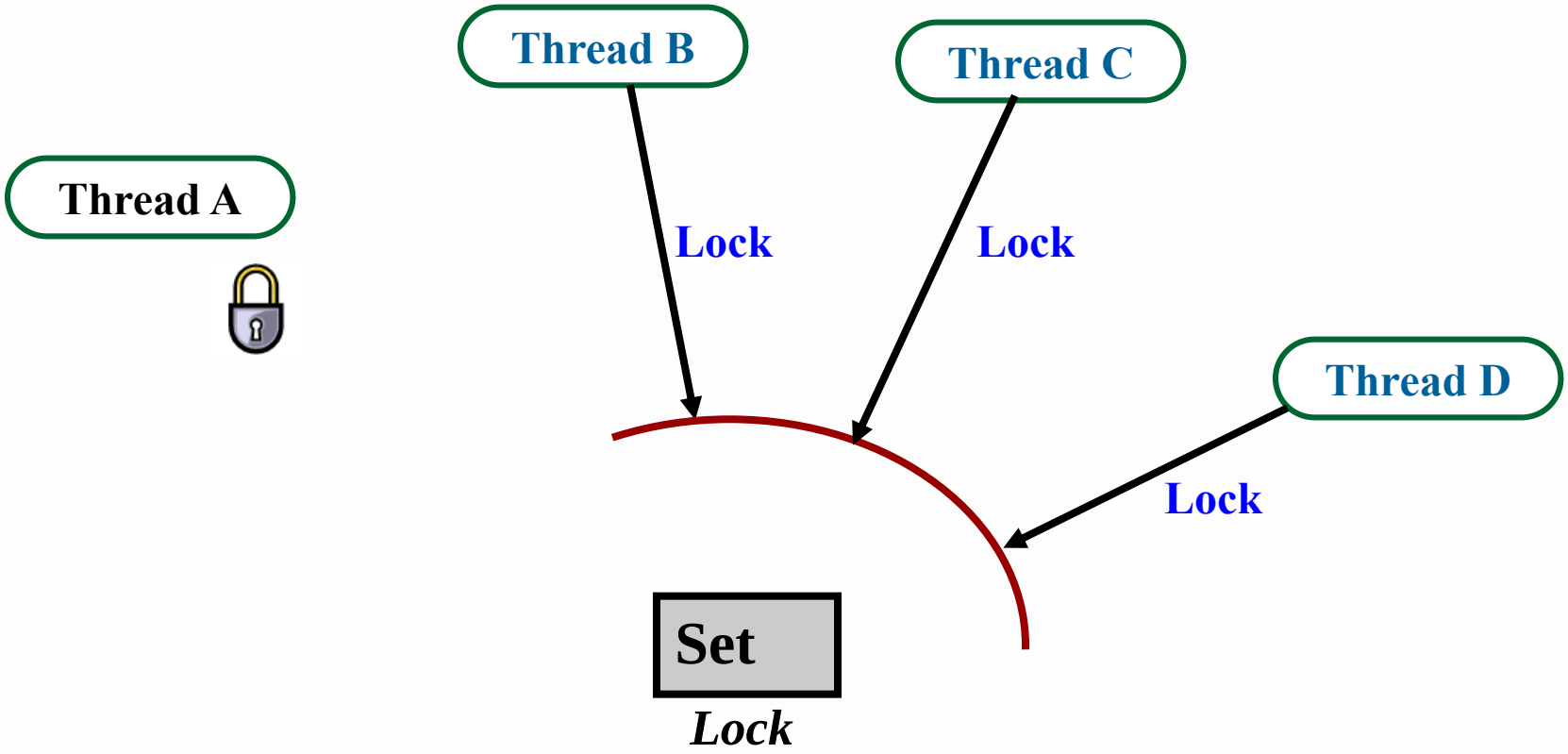
Acquiring and Releasing Locks



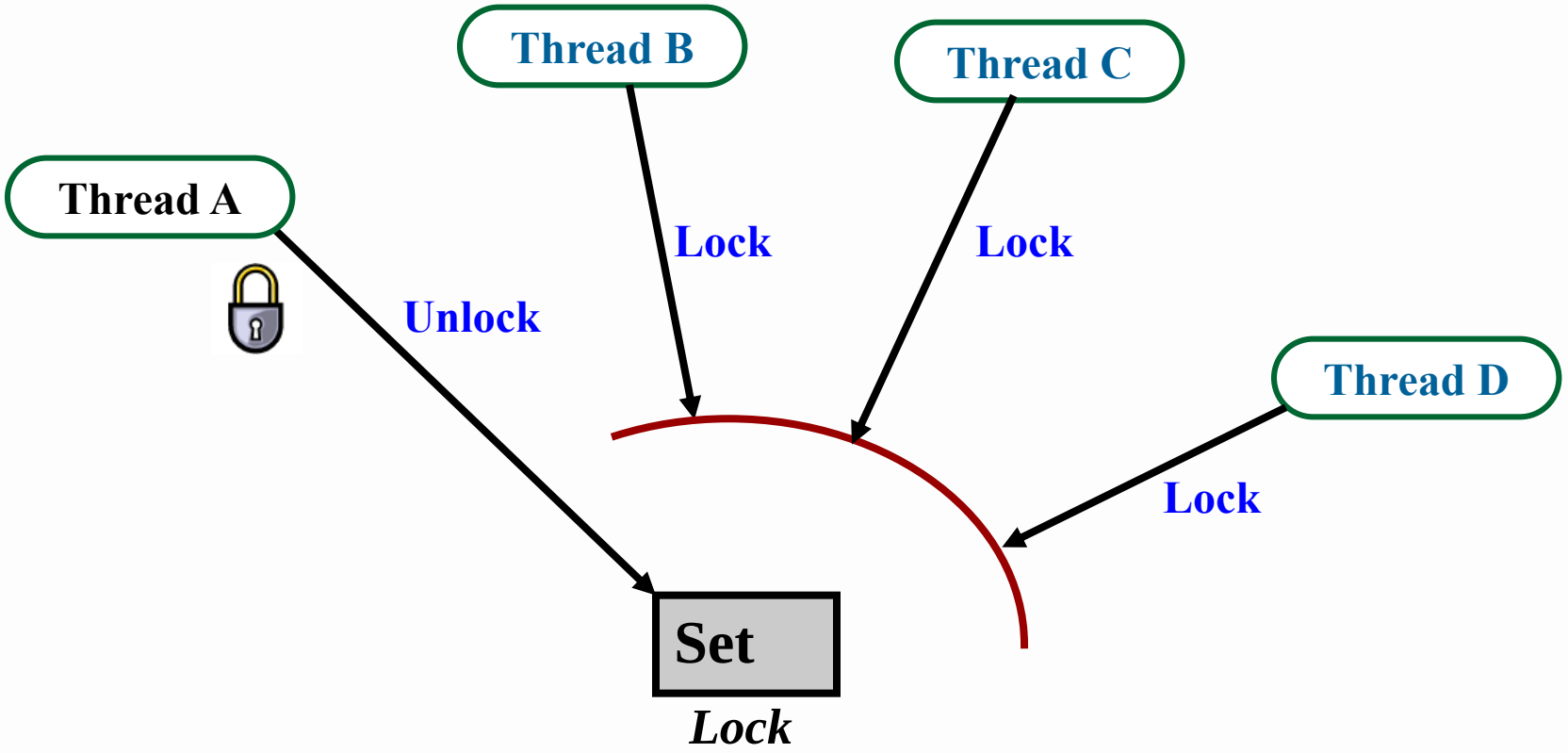
Acquiring and Releasing Locks



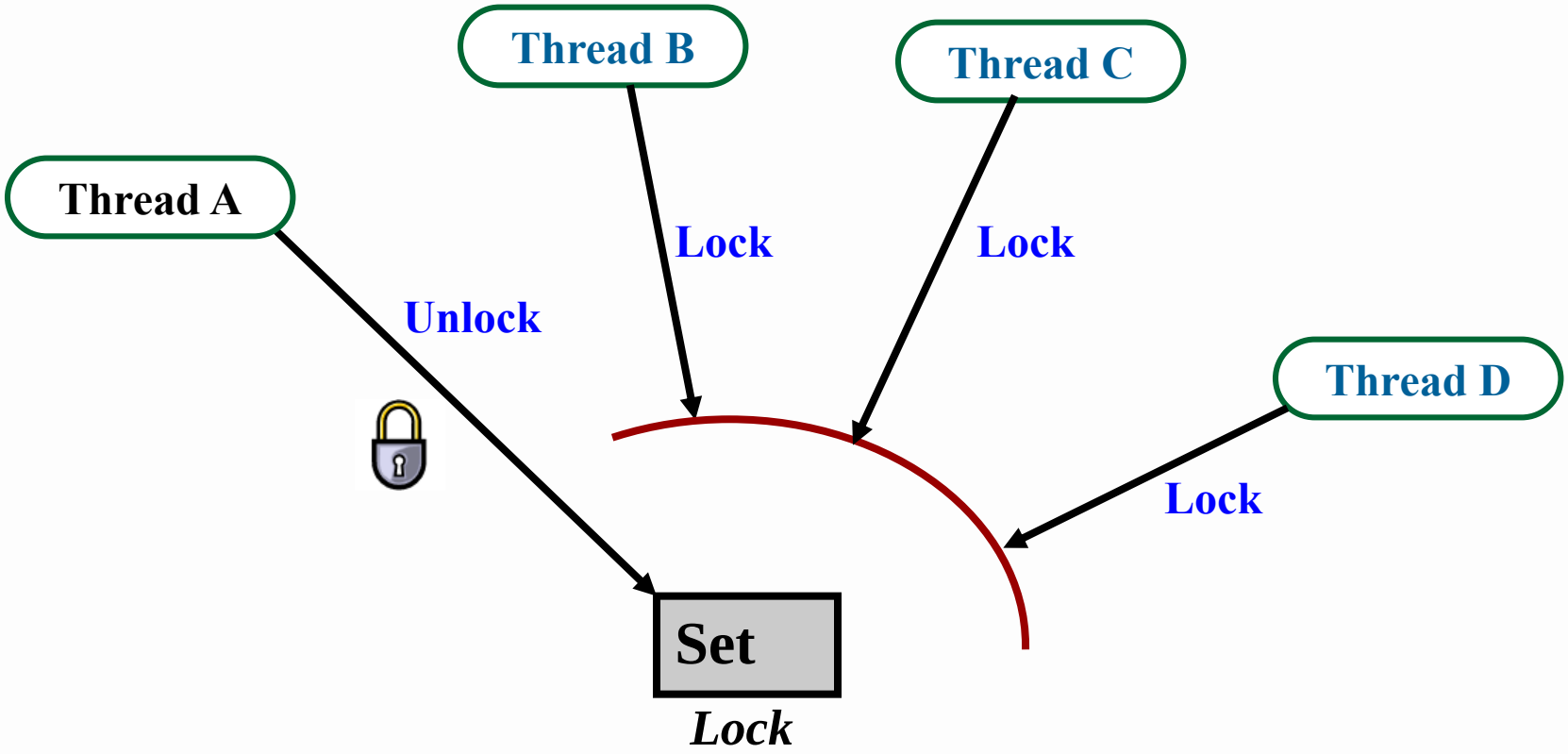
Acquiring and Releasing Locks



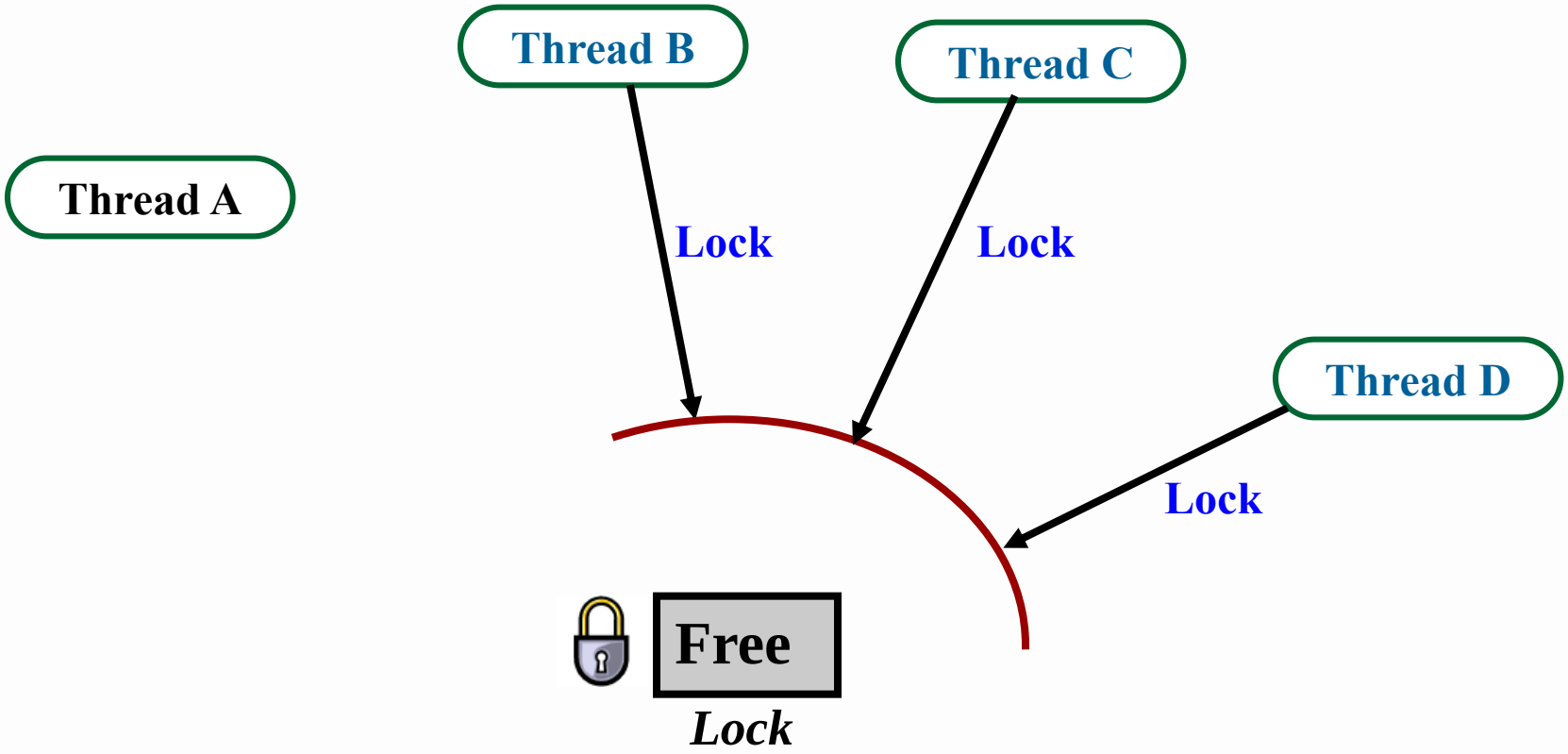
Acquiring and Releasing Locks



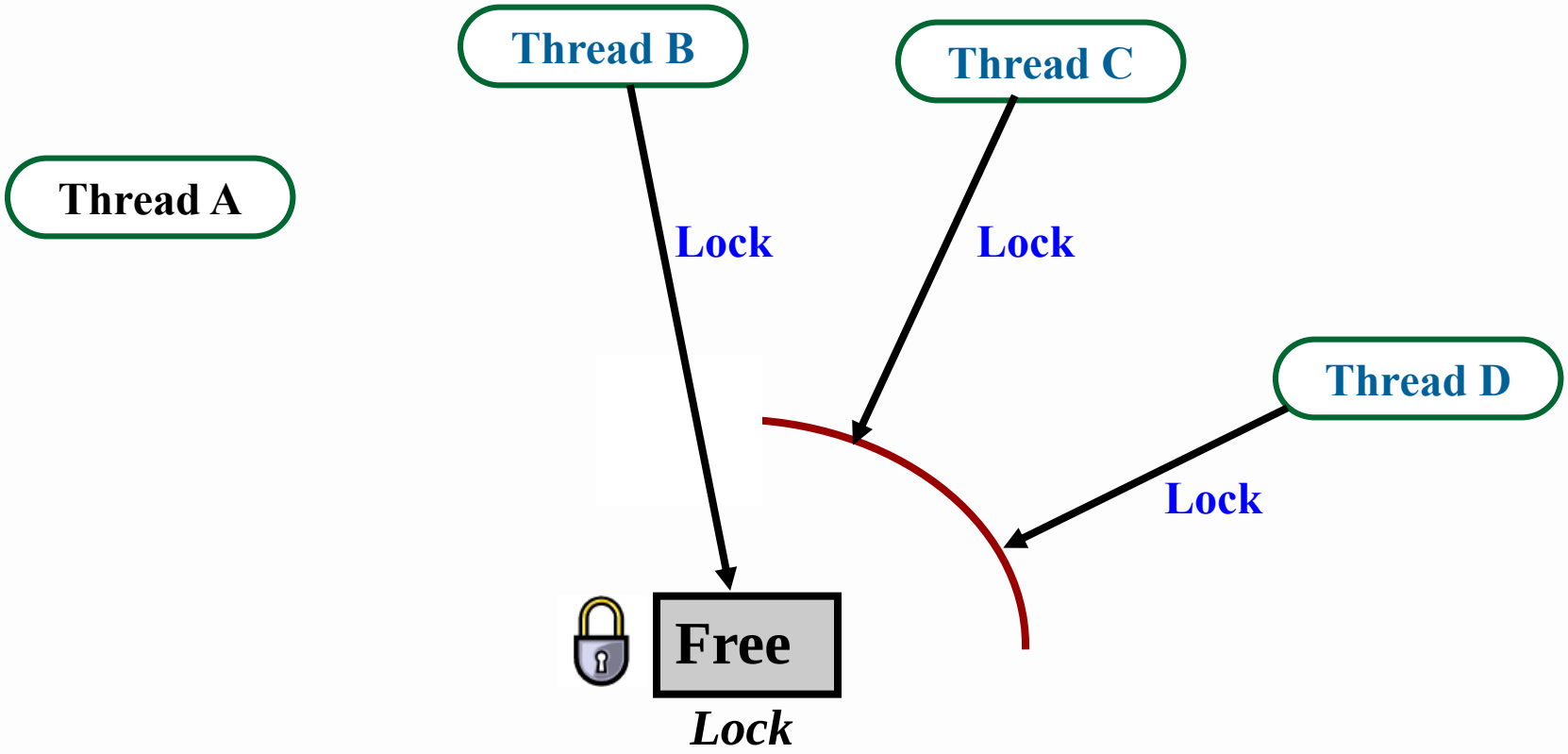
Acquiring and Releasing Locks



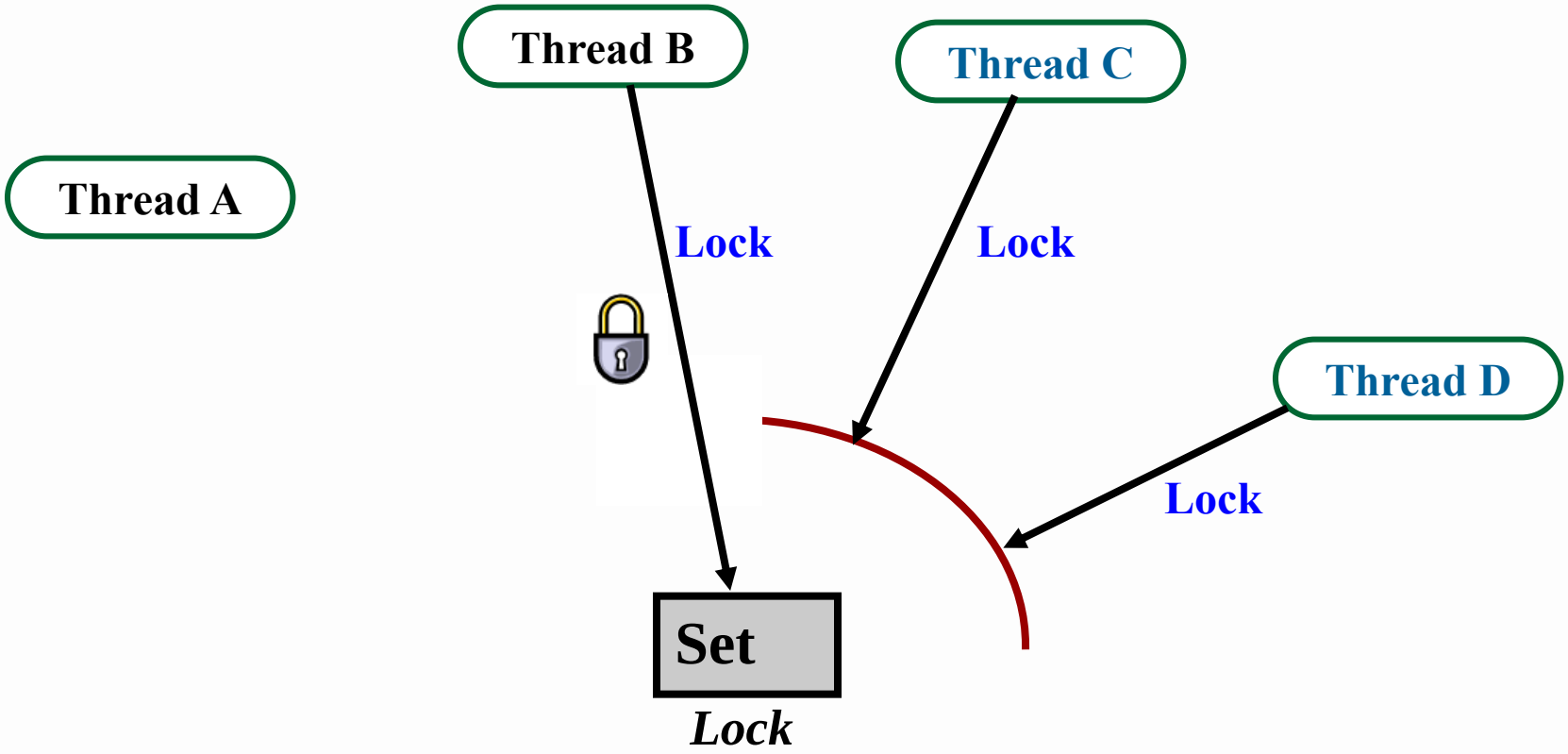
Acquiring and Releasing Locks



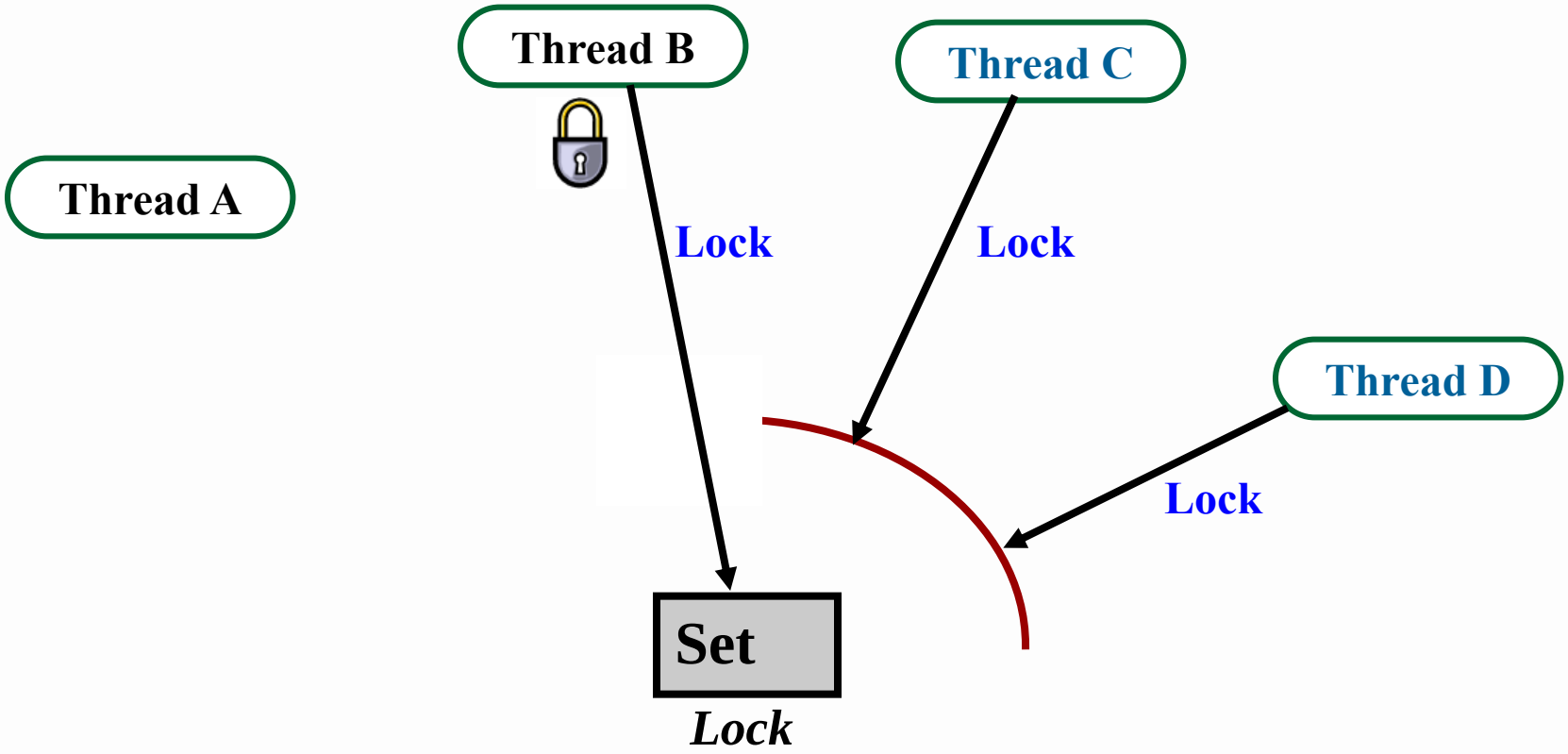
Acquiring and Releasing Locks



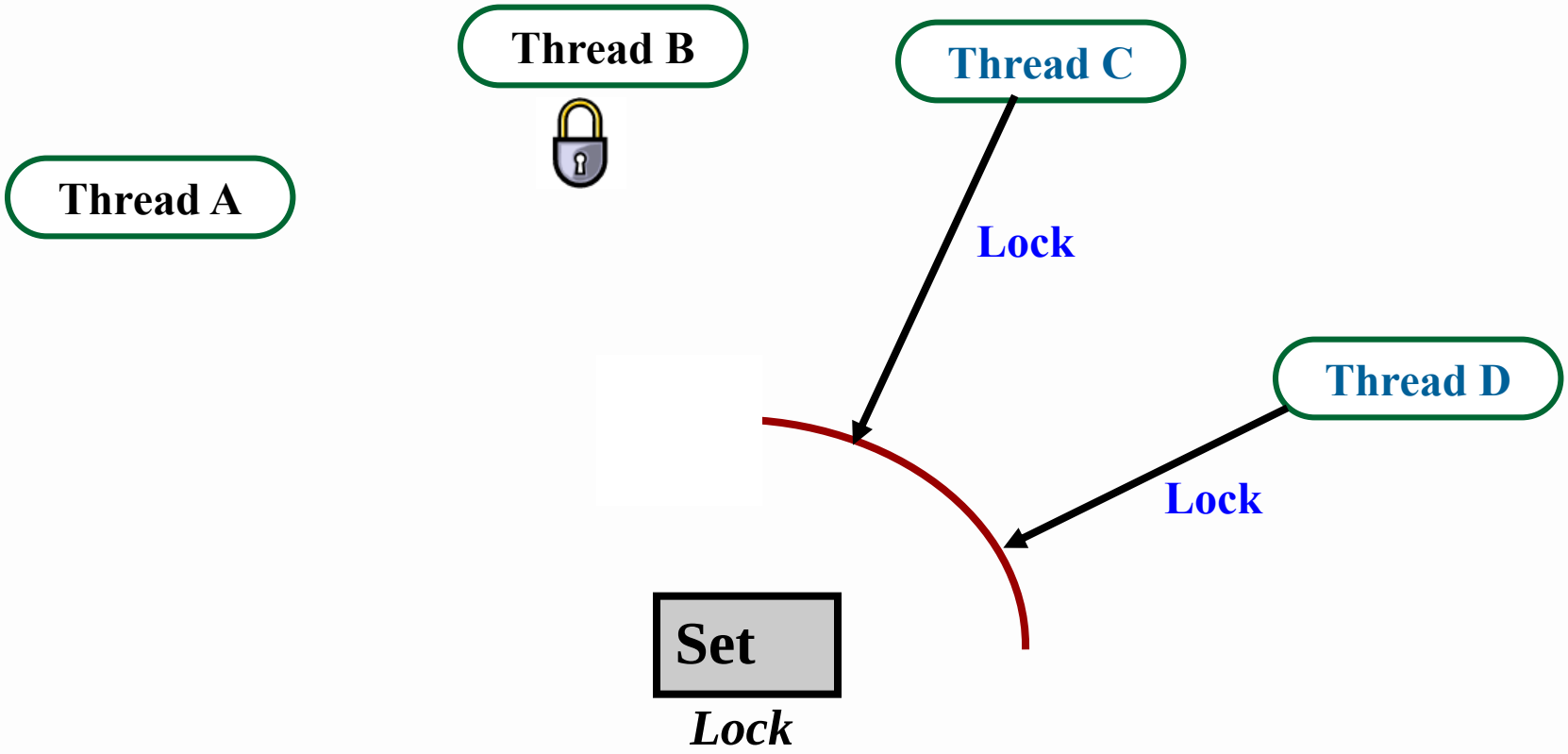
Acquiring and Releasing Locks



Acquiring and Releasing Locks



Acquiring and Releasing Locks



C implementation of Lock

```
#include  <pthread.h>

1 pthread_mutex_t lock =
PTHREAD_MUTEX_INITIALIZER;

2 Pthread_mutex_lock(&lock);

3 balance = balance + 1;
// critical section

4 Pthread_mutex_unlock(&lock);
```

Java implementation of Lock

- Using Lock object

- Using “Synchronized” keyword
  - synchronized methods
  - synchronized statements

Locks (Interface)

```
public interface Lock {

  public void lock();

  public void unlock();


}
```

acquire lock

release lock

Locks (Interface)

- Lock: interface

- Classes implement interface Lock
  - ReentrantLock
  - ReentrantReadWriteLock.ReadLock
  - ReentrantReadWriteLock.WriteLock

Using ReentrantLock Class

```
public class MyClass {
  private int shared_variable;
  private ReentrantLock lock;
  //ReentrantLock class implements Lock interface
  public void myFunction() {
   lock.lock();
   try {

       shared_variable++;

   } finally {
       lock.unlock();
   }
  }
}
```

Using Locks

```
public class MyClass {
  private int shared_variable;
  private ReentrantLock lock;
  public void myFunction() {
    lock.lock();
    try {

        shared_variable++;

    } finally {
      lock.unlock();
    }
  }
}
```

acquire Lock

Using ReentrantLock Class

```
public class MyClass {
  private int shared_variable;
  private ReentrantLock lock;
  public void myFunction() {
   lock.lock();
   try {

       shared_variable++;

   } finally {
     lock.unlock();
   }
  }
}
```

Release lock
(no matter what)

Using ReentrantLock Class

```
public class MyClass {
  private int shared_variable;
  private ReentrantLock lock;
  public void myFunction() {
   lock.lock();
   try {
      shared_variable++;
   } finally {
     lock.unlock();
   }
  }
}
```

critical section

Java implementation of Lock

- Using Lock object

- Using “Synchronized” keyword
  - synchronized methods
  - synchronized statements

Using “Synchronized” keyword

- Each Java object has an associated intrinsic lock
  - The lock is initially un-owned
  - As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

- **synchronized** keyword forces the current thread to obtain an object’s intrinsic lock

ME in Java using **synchronized**

```
public synchronized void MyFunction(){

   // critical section

}
```


```
synchronized(shared_object){

    // critical section

}
```

Example: Synchronized methods

```
public class SynchronizedCounter {

    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Two effects of Synchronized methods

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Synchronized methods

- Question:
  - Can constructors be synchronized?

- Answer:
  - No
  - using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates the object should have access to it while the object is being constructed

Example: Synchronized statements

```
public void addName(String name) {

    // other work goes here

    synchronized(this) {

       lastName = name;
       nameCount++;

    }

    // other work goes here

}
```

Example: Synchronized statements

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Using an object that provides intrinsic lock

These two are equivalent

```
public synchronized void MyFunction(){


   // critical section


}
```

```
public void MyFunction(){
   synchronized(this){

   // critical section

   }
}
```

Using synchronized

- The synchronized keyword automatically implements:
  - lock
  - try
  - finally
  - unlock

Which one is better? Lock interface or synchronization blocks?

- **Answer: Lock interface (Reentrant locks)!**

- Lock interface has more benefits…

Which one is better? Lock interface or synchronization blocks?

- With the Lock interface, you have the option of calling ***tryLock()***

```
if(lock.tryLock() ) {
   //Lock is acquired
} else{
   //Lock isn't acquired; act accordingly
}
```

- You also have the option to specify a timeout (from nanoseconds to days!)

```
if(lock.tryLock(50, TimeUnit.SECONDS) ) {
   //Lock is acquired
} else{
   //Lock isn't acquired; act accordingly
}
```

Which one is better? Lock interface or synchronization blocks?

- **Answer: Lock interface (Reentrant locks)!**
- A thread blocked on an intrinsic lock cannot be interrupted. All or nothing. (inflexible)
- If a thread becomes blocked and the semantics of your program will never allow it to become unblocked (it's stuck until you kill the JVM)
- With the Lock interface, you use it with the ***lockInterruptibly()*** method

ReentrantLock interface

```
public interface Lock {

 public void lock();

 public void lockInterruptibly();

 public void unlock();

 public Boolean tryLock();

 public Boolean tryLock(long time, TimeUnit
unit);

}
```

Acknowledgement

- Chapter 26
  - Operating Systems: Three Easy Pieces
- Java documentation
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- 4.ppt
  - Intro to Operating System at Portland State University
  - by Jonathan Walpole

SWINBURNE UNIVERSITY OF TECHNOLOGY

Questions?