

# Operating Systems

## Lecture 4

# Concurrent Programming & Synchronization Primitives

# Concurrency

Two or more threads run in parallel execution

The threads access shared variables

Can't predict the relative execution speeds (order)

# Concurrency

Example: One thread writes a variable that another thread reads

Problem – non-determinism:


*The relative order of one thread's reads and the other thread's writes may determine the outcome!*

# Race Condition Problem

Banking application

1. Check if account A has enough balance.
2. Add the money to account B.
3. Deduct the money from account A.

# Race Condition Problem

Thread 1	Thread 2	Balance of Account A + B
Check Account A Balance (\$500)		500 (A - 500, B - 0)
Add \$500 to Account B		1000 (A - 500, B - 500)
Deduct \$500 from Account A		500 (A - 0, B - 500)
	Check Account A Balance (\$0)	500 (A - 0, B - 500)
	Transfer Failed (Low Balance)	500 (A - 0, B - 500) 
		500 (A - 0, B - 500)
		500 (A - 0, B - 500)

# Race Condition Problem

Thread 1	Thread 2	Balance of Account A + B
Check Account A Balance (\$500)		500 (A - 500, B - 0)
	Check Account A Balance (\$500)	500 (A - 500, B - 0)
Add \$500 to Account B		1000 (A - 500, B - 500)
	Add \$500 to Account B	1500 (A - 500, B - 1000)
Deduct \$500 from Account A		1000 (A - 0, B - 1000)
	Deduct \$500 from Account A	1000 (A - 0, B - 1000)
		1000 (A - 0, B - 1000)

# Race Conditions

Race condition: whenever the result depends on the precise execution order of the threads!

What solutions can we apply?

- make threads coordinate with each other to ensure mutual exclusion in accessing critical sections of code



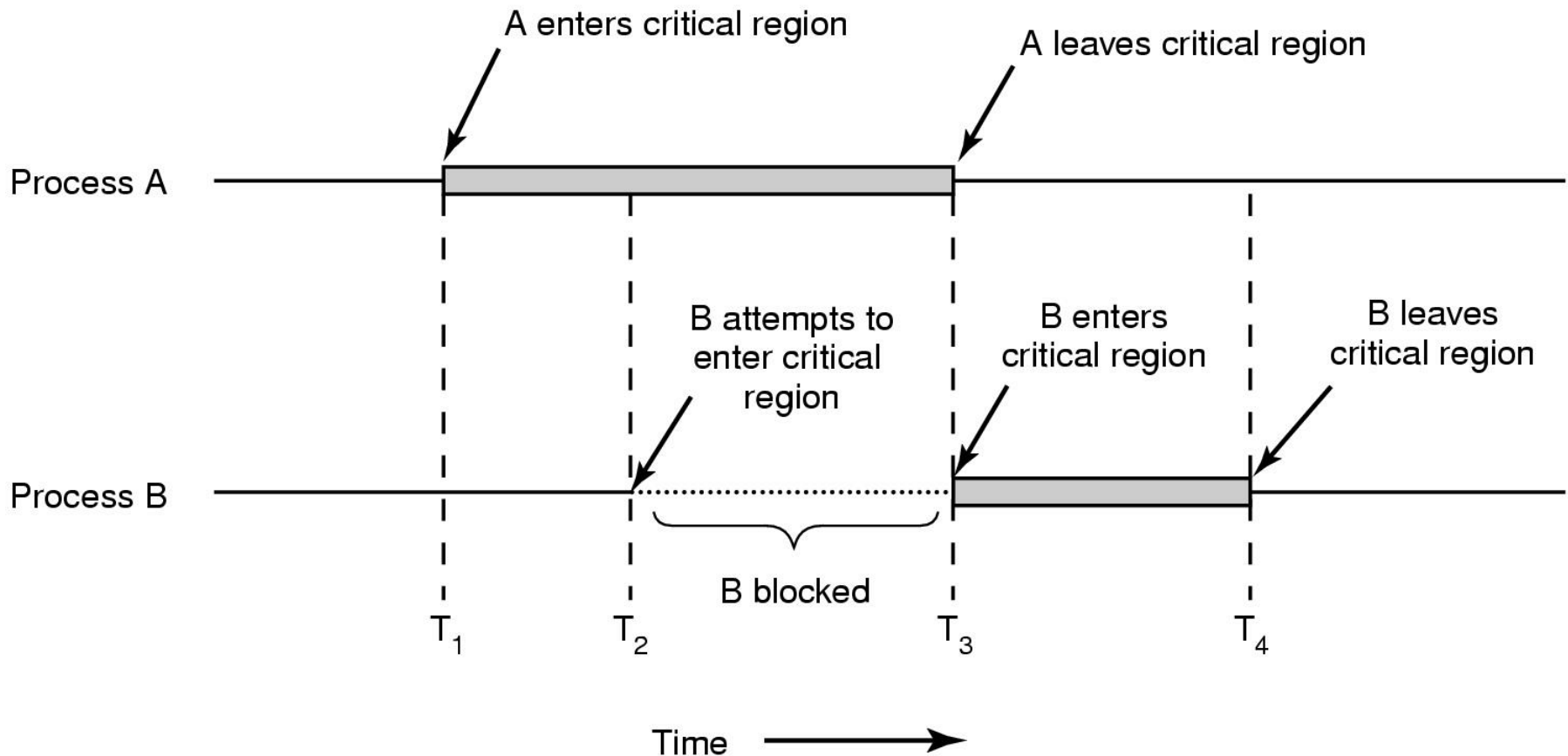
# Mutual Exclusion Conditions

No two processes (or threads) simultaneously in critical section

No process running outside its critical section may block another process

No process waits forever to enter its critical section

# Mutual Exclusion Example



# Enforcing Mutual Exclusion

What about using *locks*?

Locks can ensure exclusive access to shared data.

- Acquiring a lock prevents concurrent access
- Expresses intention to enter critical section

Assumption:

- Each shared data item has an associated lock
- All threads set the lock before accessing the shared data
- Every thread releases the lock after it is done

# Acquiring and Releasing Locks

Thread B

Thread C

Thread A

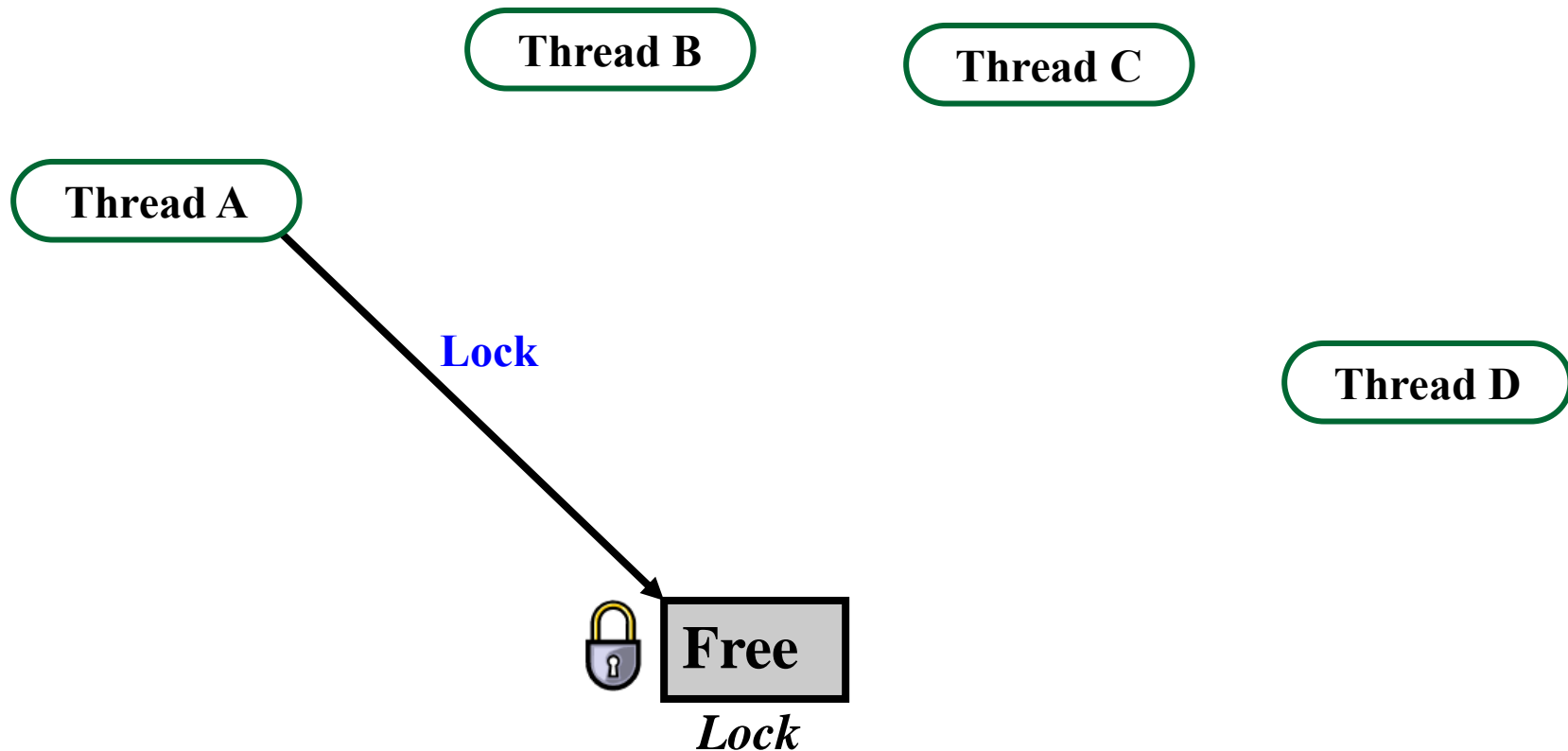
Thread D



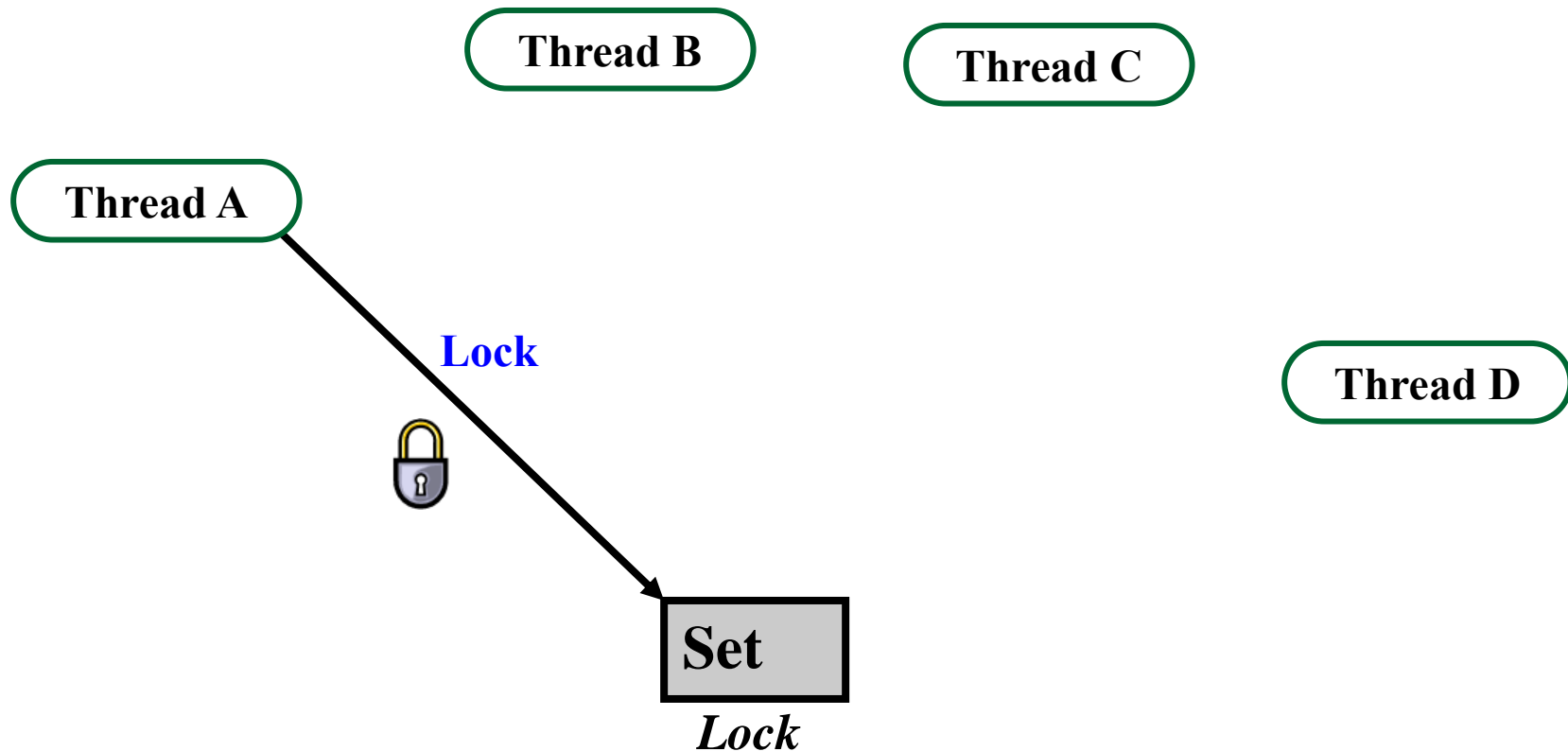
Free

*Lock*

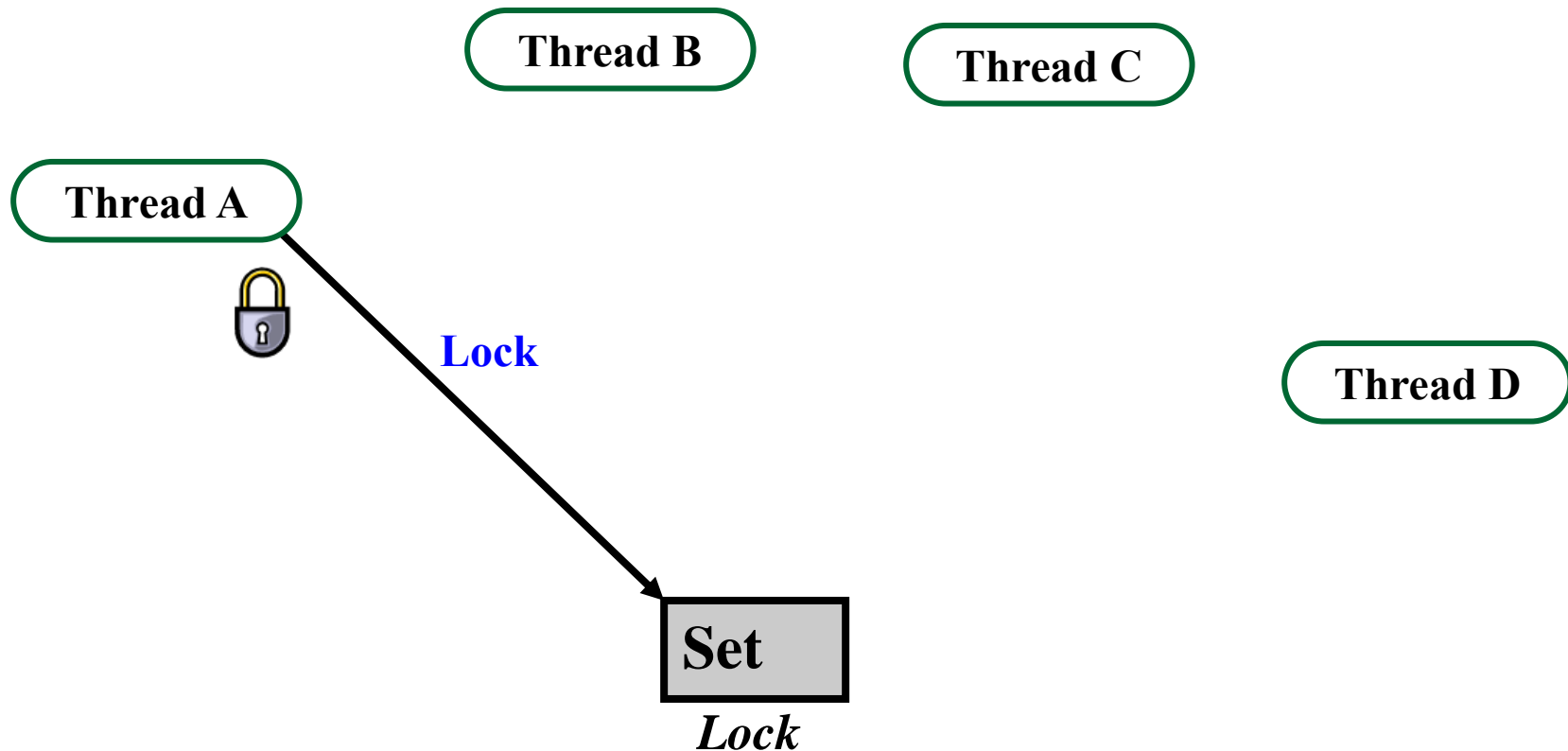
# Acquiring and Releasing Locks



# Acquiring and Releasing Locks



# Acquiring and Releasing Locks



# Acquiring and Releasing Locks

Thread B

Thread C

Thread A



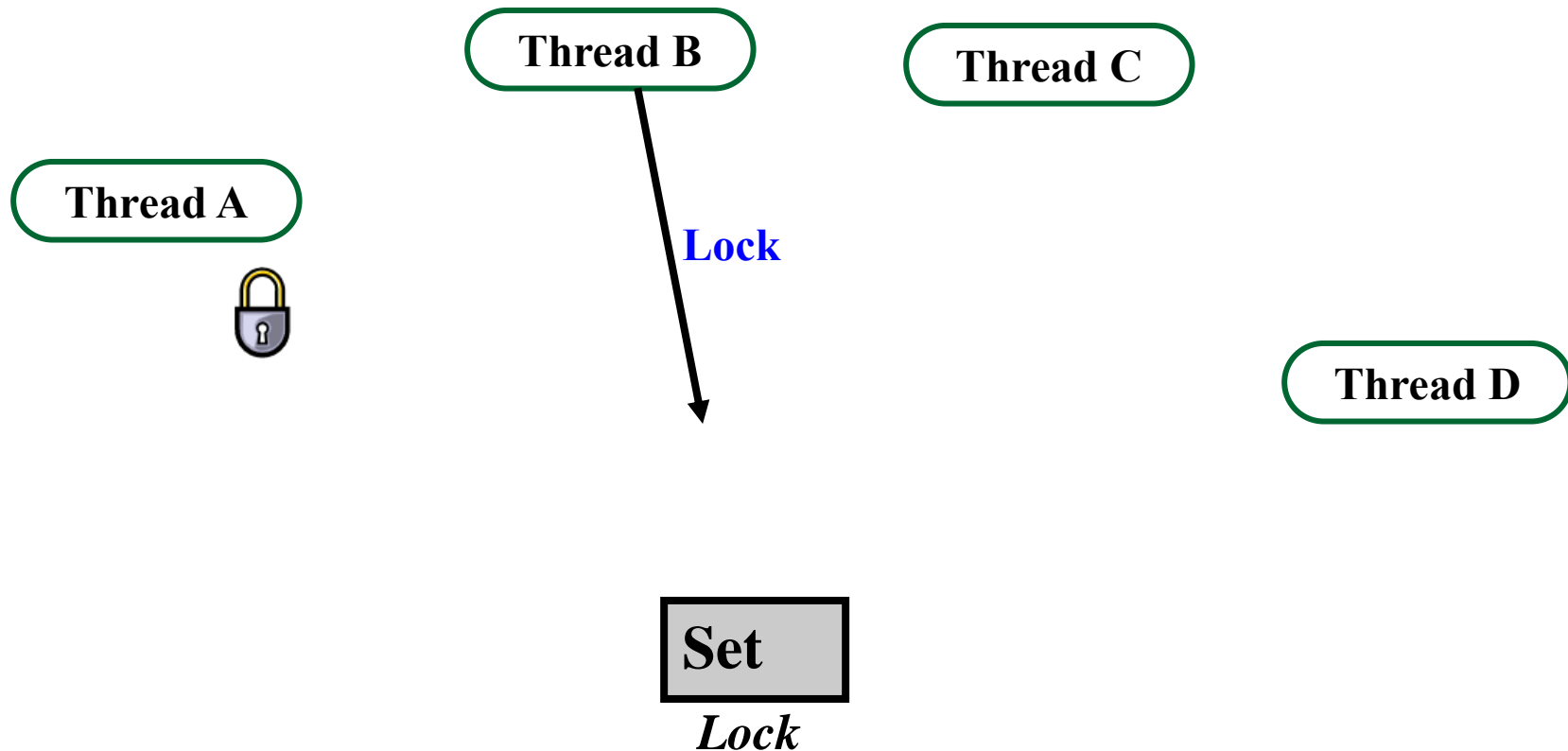
Thread D

Set

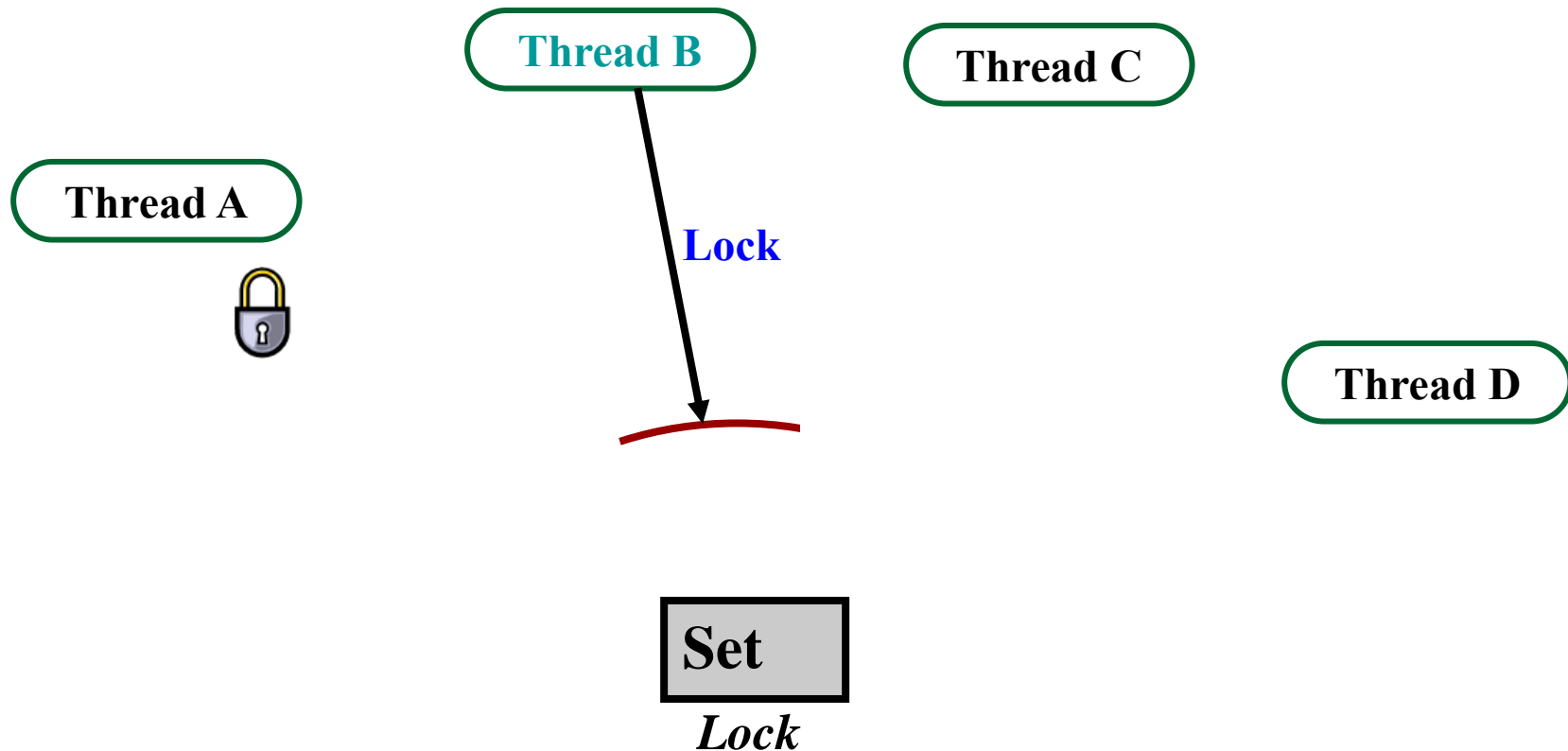
*Lock*



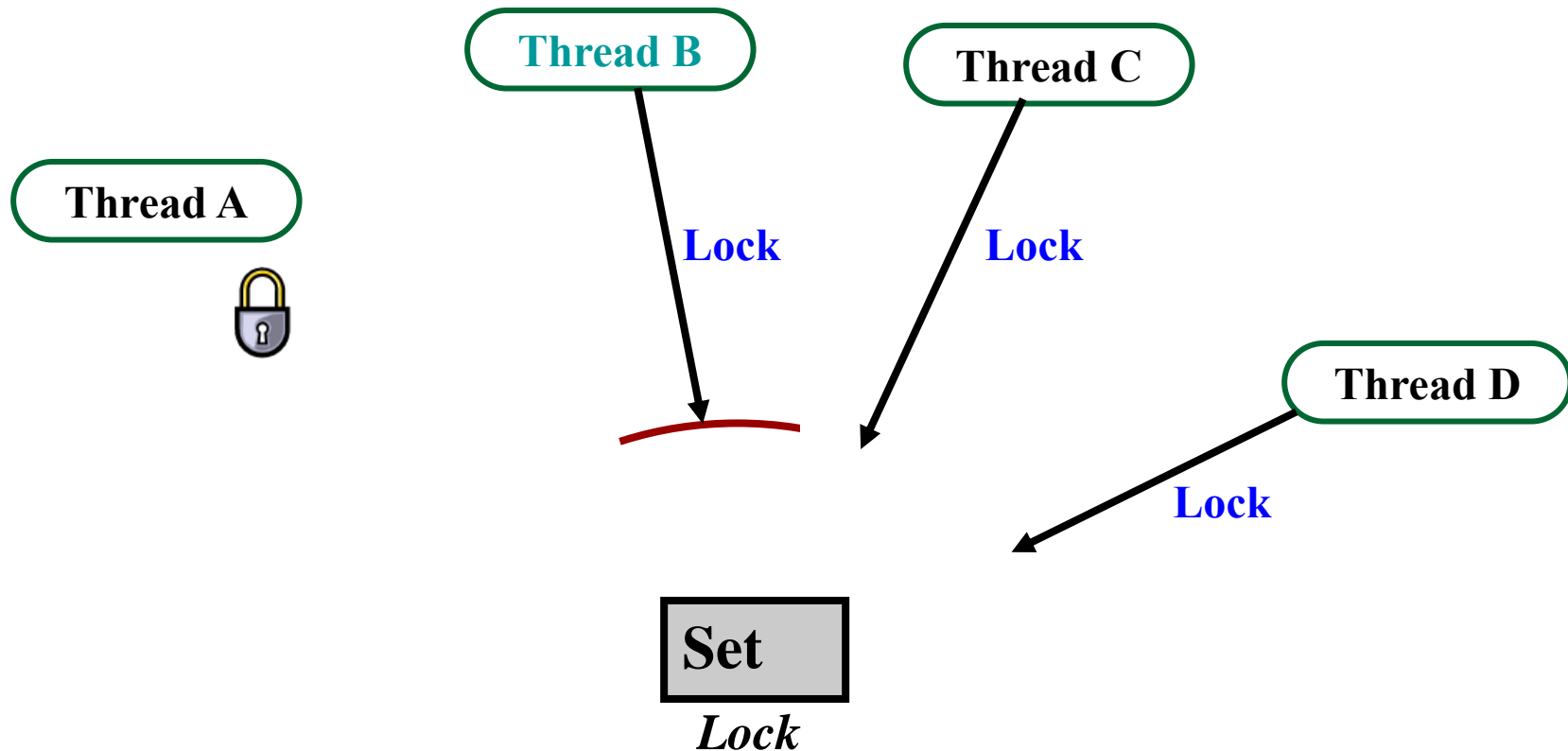
# Acquiring and Releasing Locks



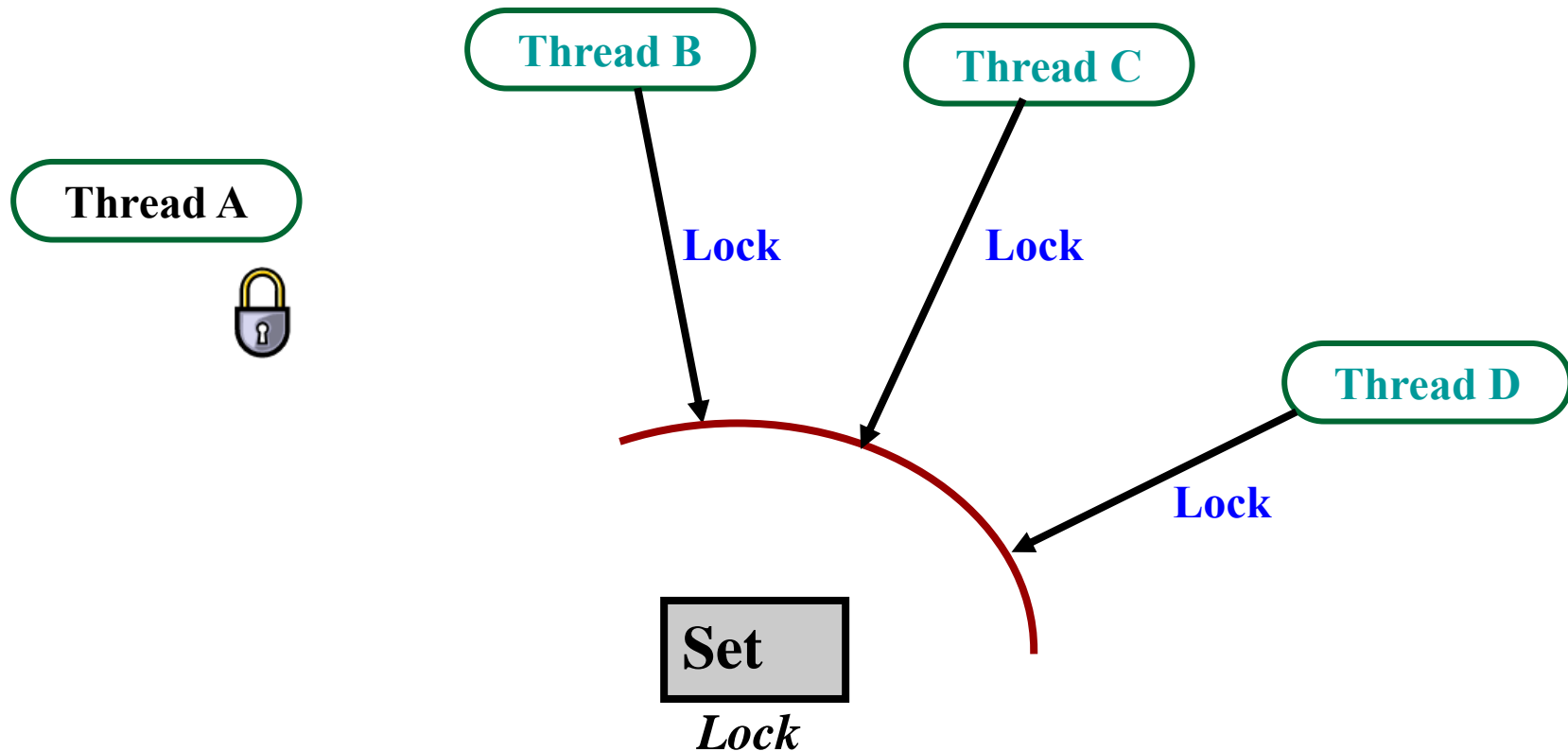
# Acquiring and Releasing Locks



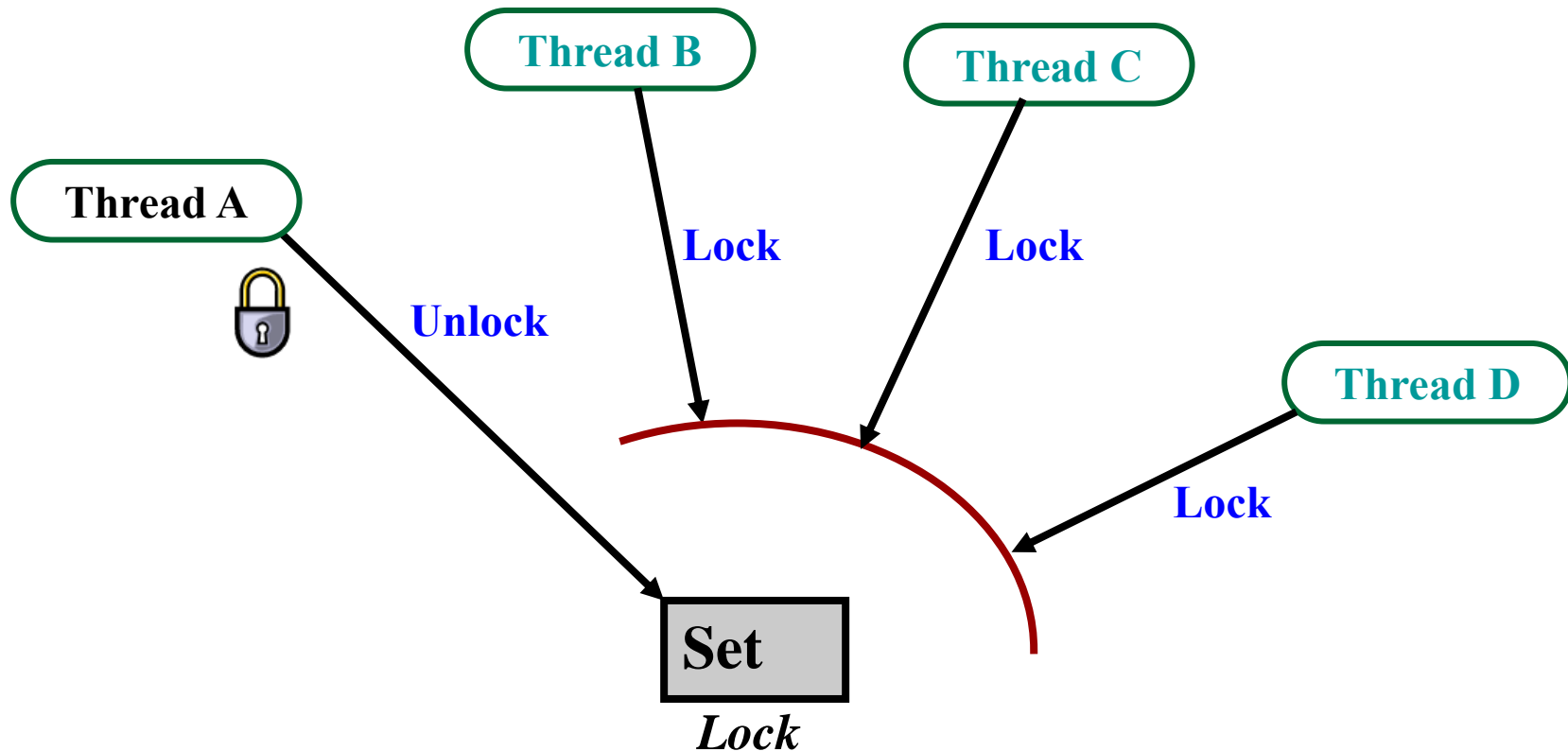
# Acquiring and Releasing Locks



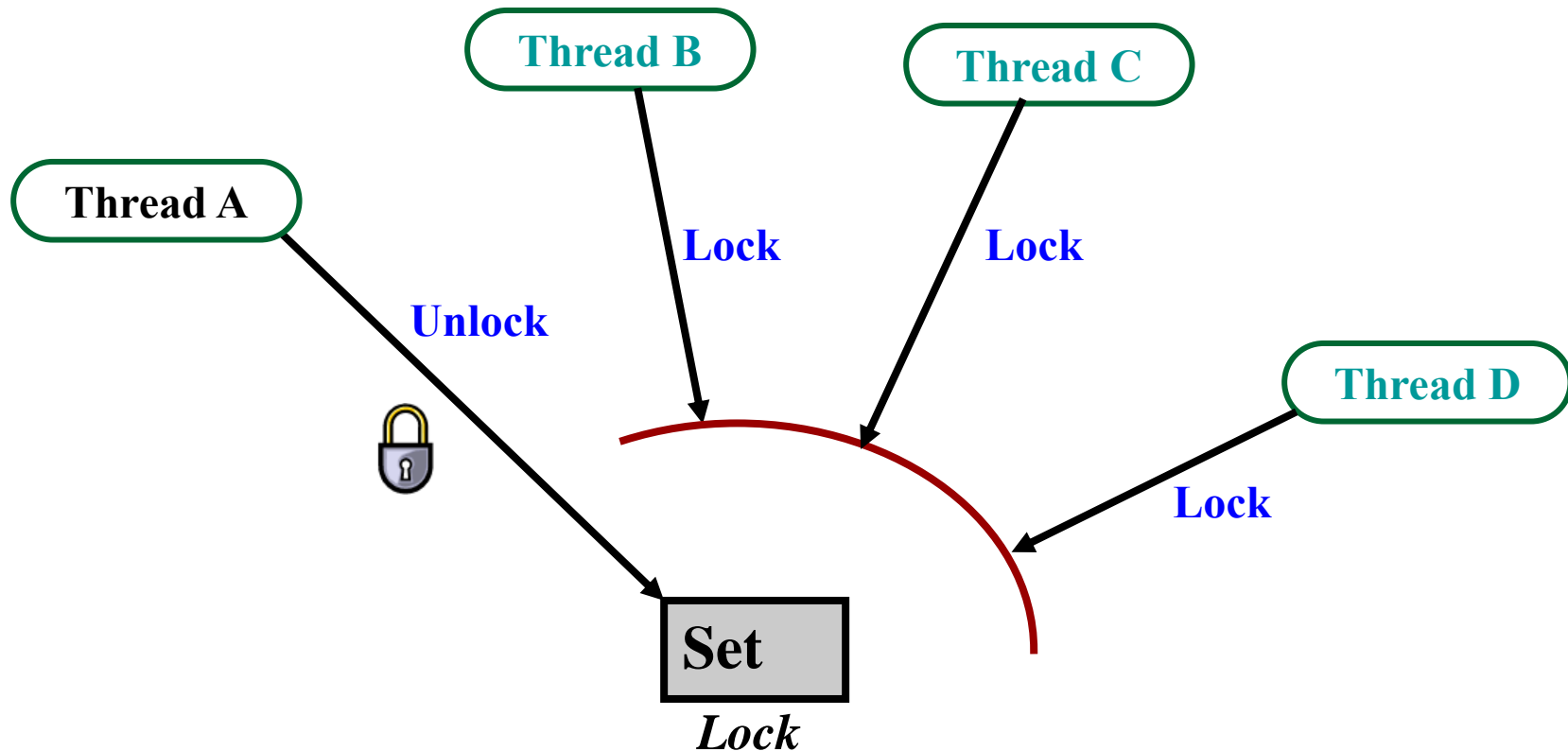
# Acquiring and Releasing Locks



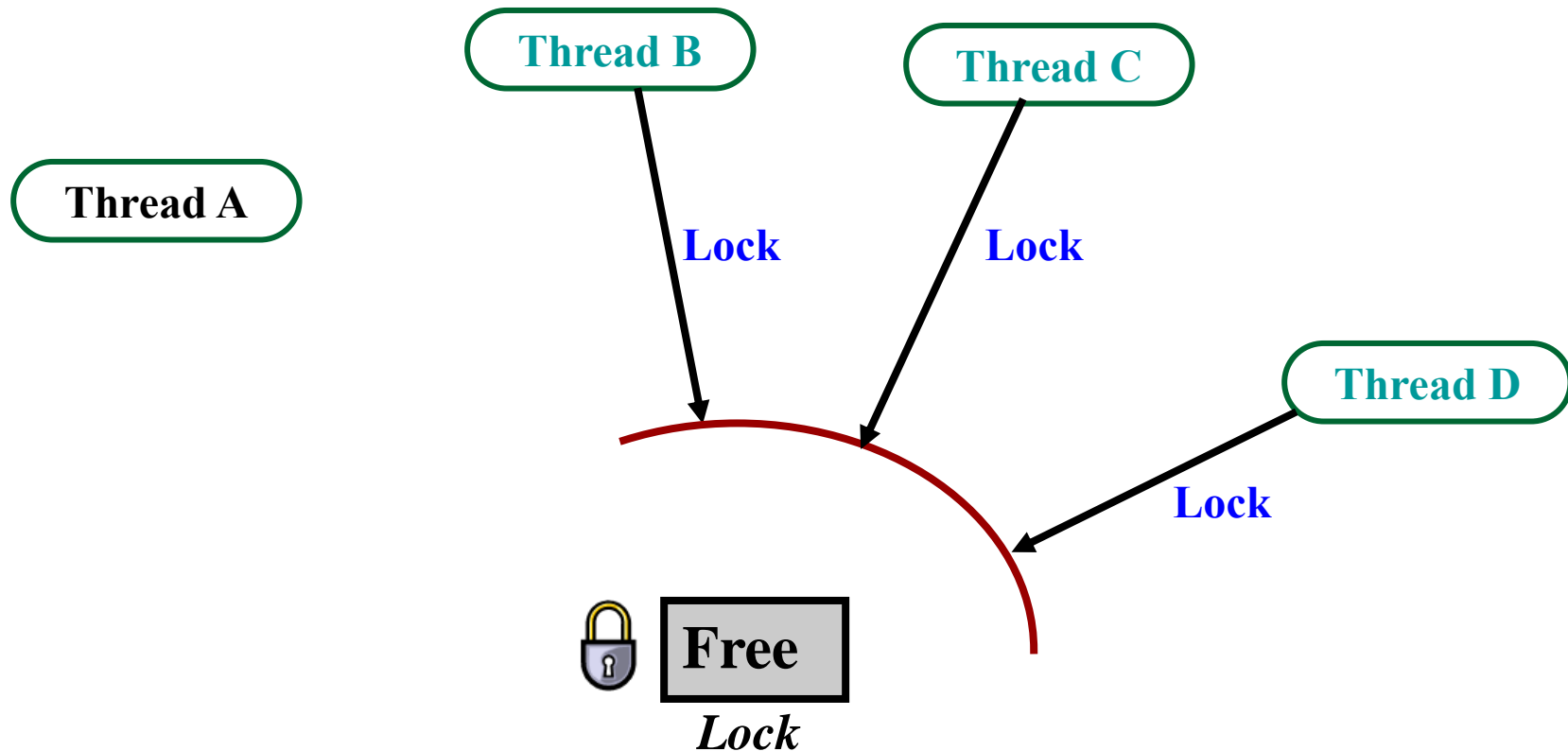
# Acquiring and Releasing Locks



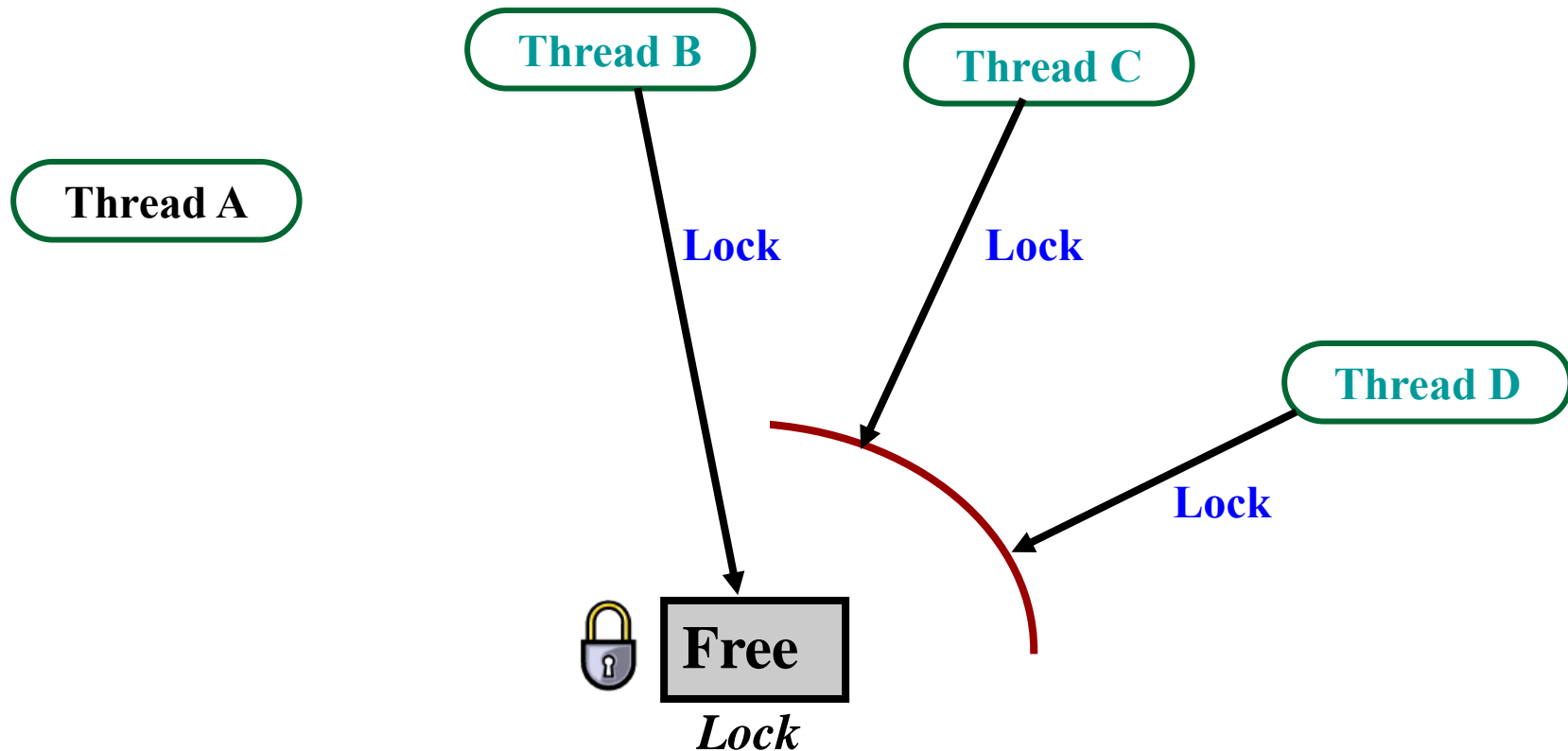
# Acquiring and Releasing Locks



# Acquiring and Releasing Locks

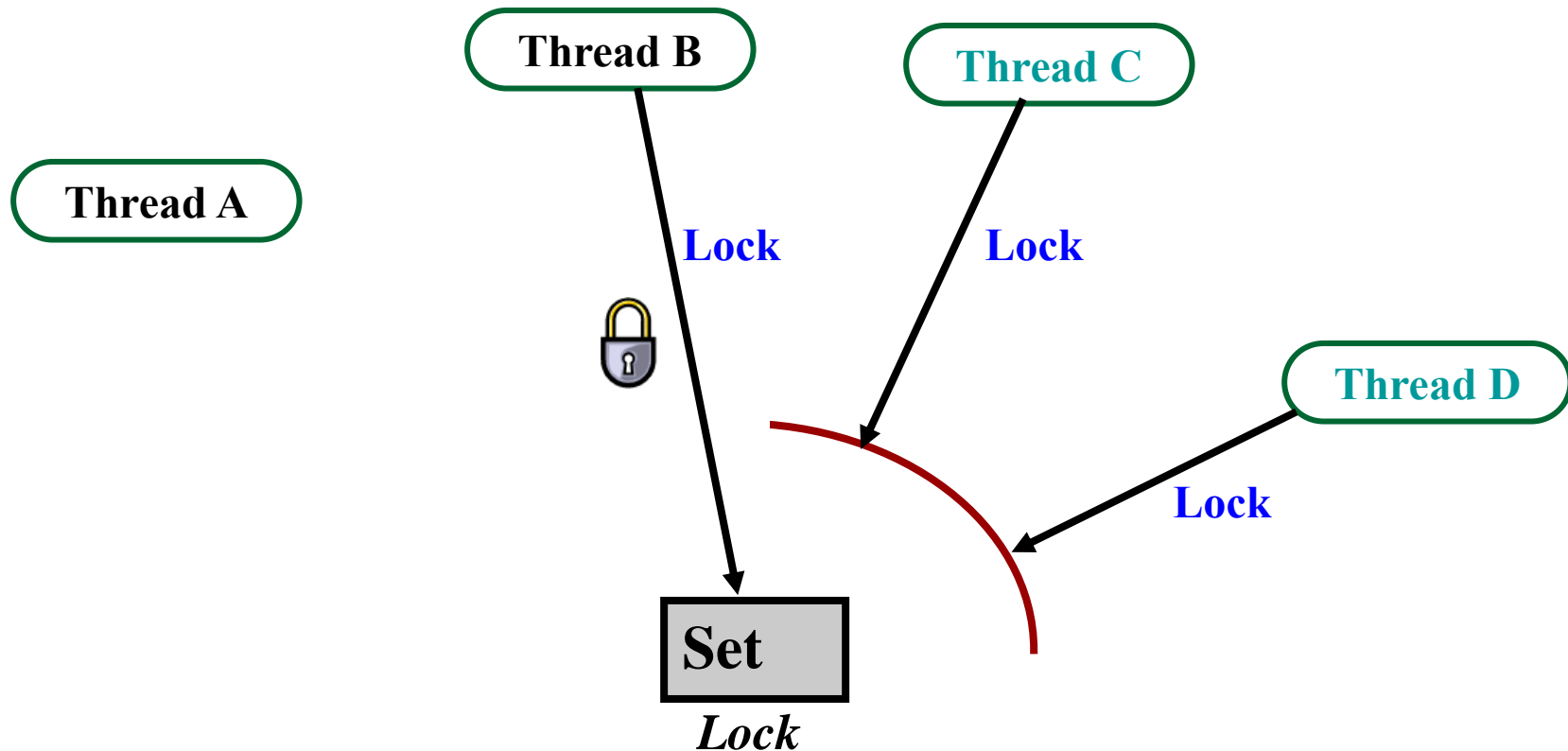


# Acquiring and Releasing Locks

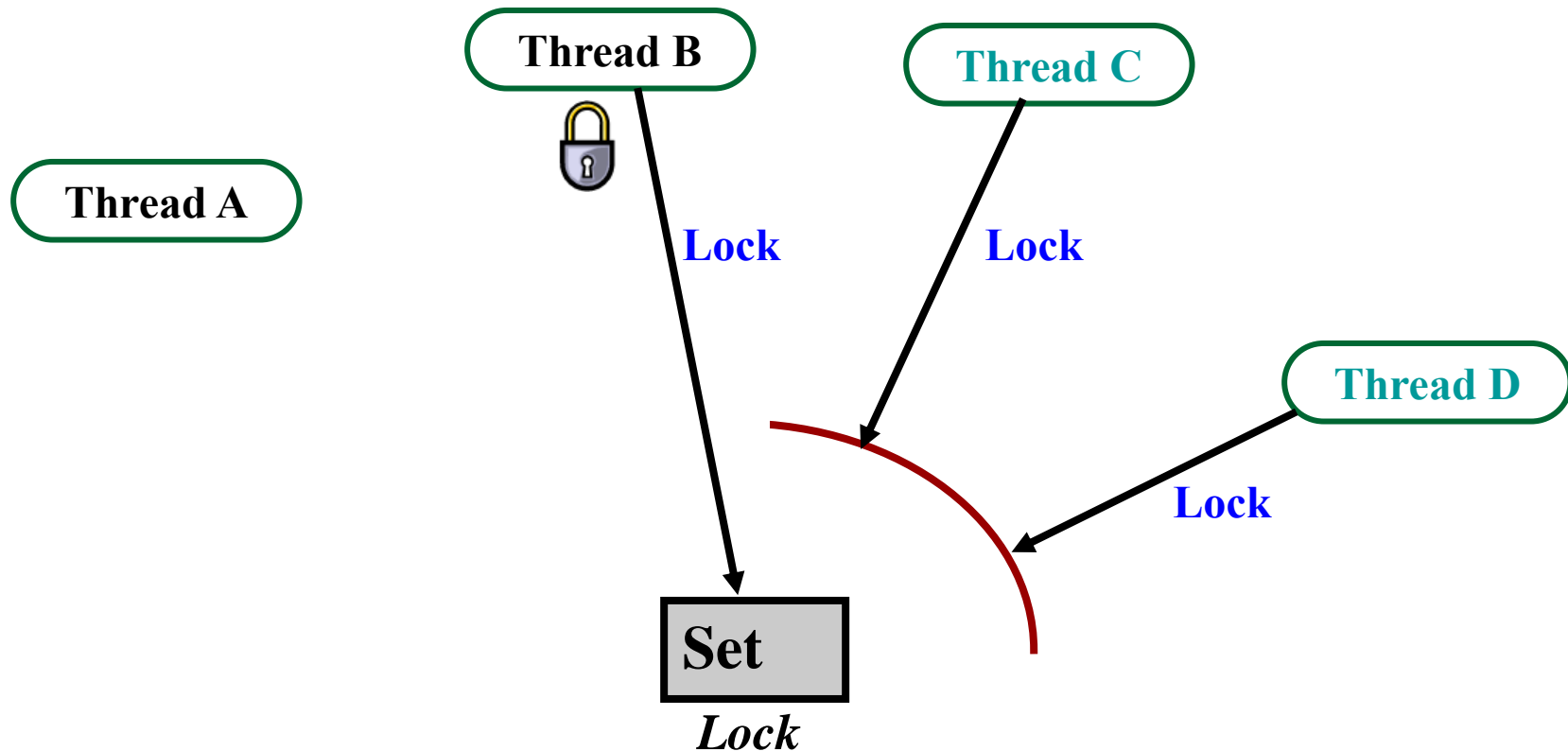




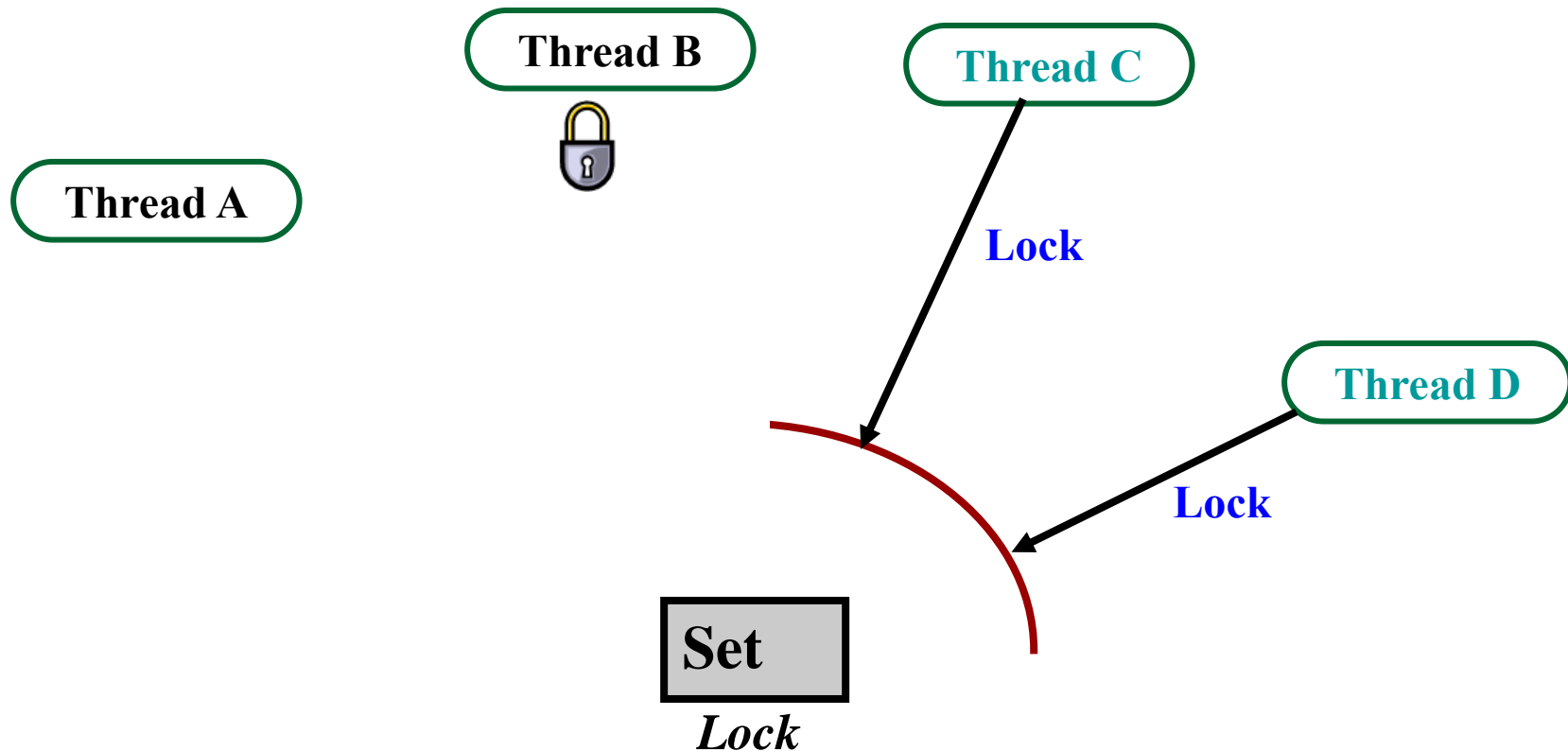
# Acquiring and Releasing Locks



# Acquiring and Releasing Locks



# Acquiring and Releasing Locks



# Mutual Exclusion (mutex) Locks

An abstract data type used for synchronization

The mutex is either:

Locked (“the lock is held”)

Unlocked (“the lock is free”)

# Mutex Lock Operations

## Lock (*mutex*)

Acquire the lock if it is free ... and continue  
Otherwise wait until it can be acquired

## Unlock (*mutex*)

Release the lock  
If there are waiting threads wake one up

# Using a Mutex Lock

Shared data:

**Mutex myLock ;**

```
1 repeat
2   Lock(myLock) ;
3   critical section
4   Unlock(myLock) ;
5   remainder section
6 until FALSE
```

```
1 repeat
2   Lock(myLock) ;
3   critical section
4   Unlock(myLock) ;
5   remainder section
6 until FALSE
```

# Implementing a Mutex Lock

If the lock was just a binary variable, how would we implement the lock and unlock procedures?

# Implementing a Mutex Lock

**Lock** and **Unlock** operations must be *atomic* !

Many computers have *some limited* hardware support for setting locks

- Atomic Test and Set Lock instruction
- Atomic compare and swap/exchange operation
- These can be used to implement mutex locks



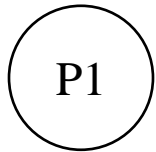
# Test-and-Set-Lock (TSL) Instruction

- ❑ A lock is a single word variable with two values  
0 = FALSE = not locked ; 1 = TRUE = locked
- ❑ Test-and-set-lock does the following *atomically*:
  - Get the (old) value
  - Set the lock to TRUE
  - Return the old value

**If** the returned value was FALSE...  
Then you got the lock!!!

**If** the returned value was TRUE...  
Then someone else has the lock  
(so try again later)

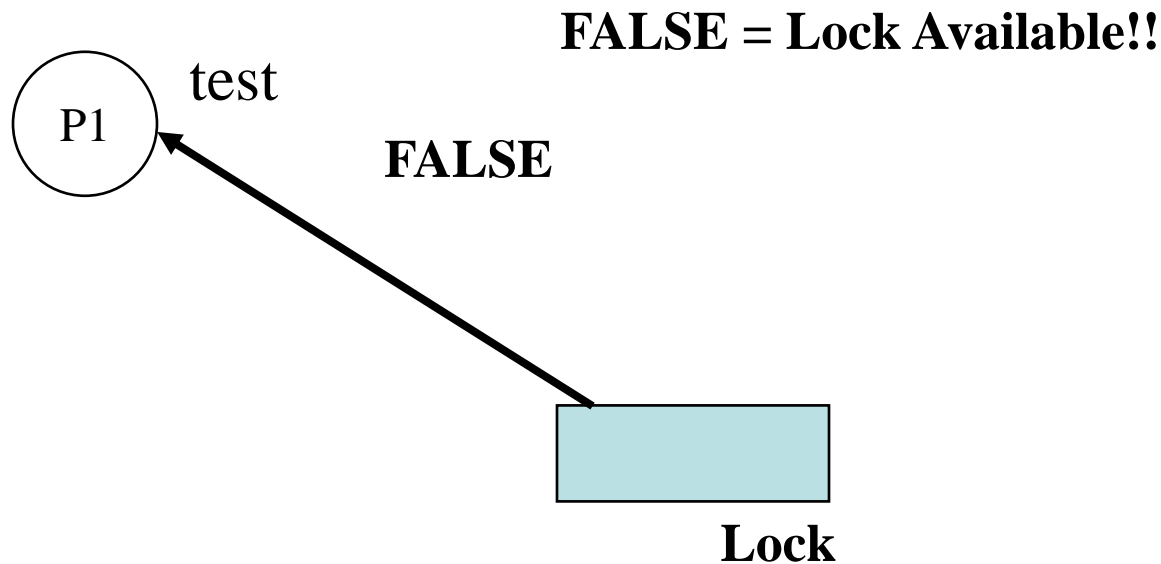
# Test and Set Lock



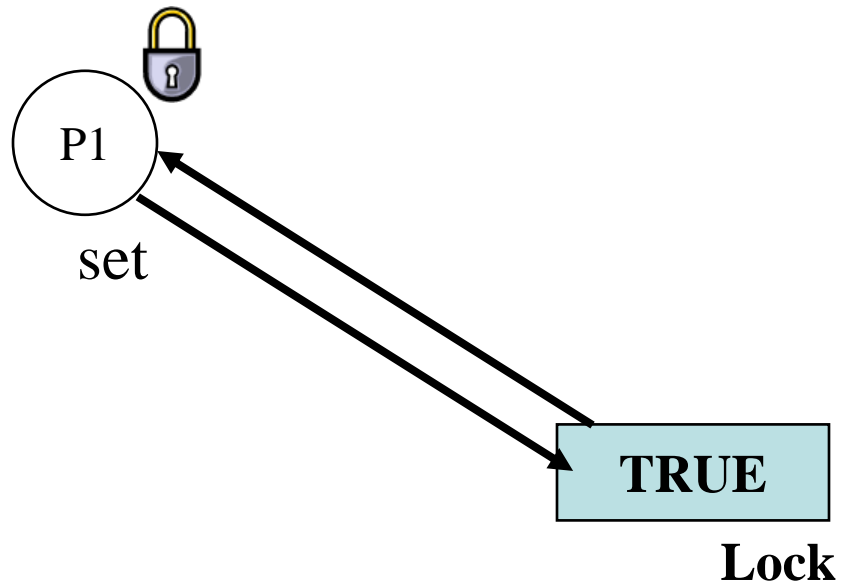
**FALSE**

**Lock**

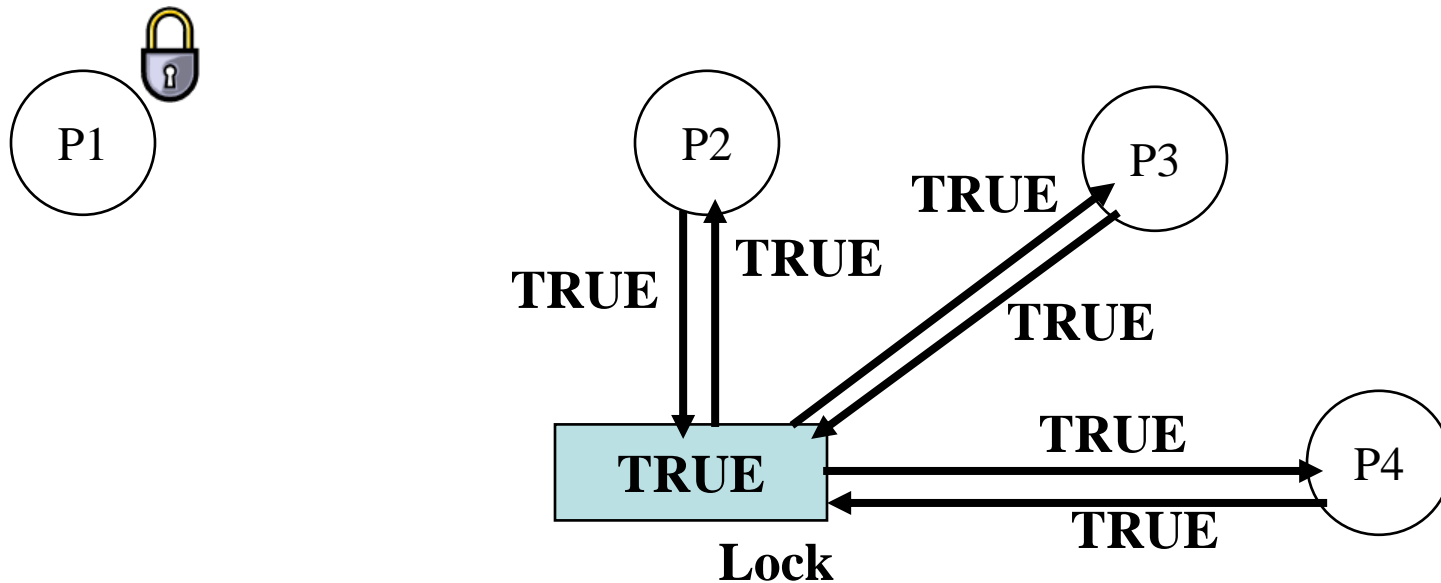
# Test and Set Lock



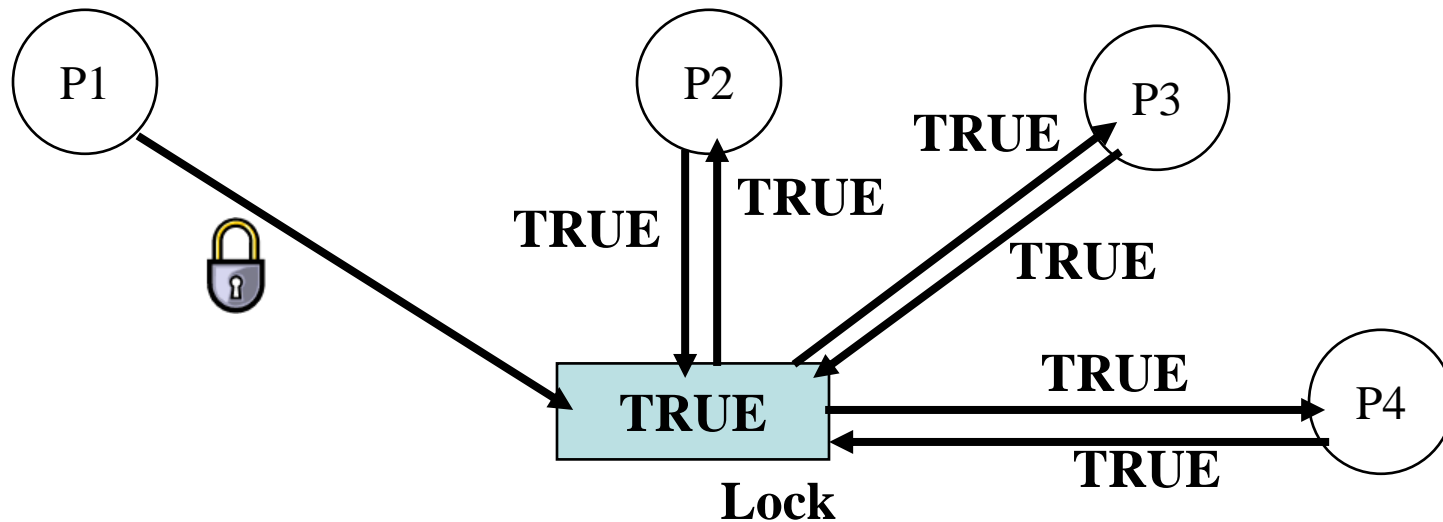
# Test and Set Lock



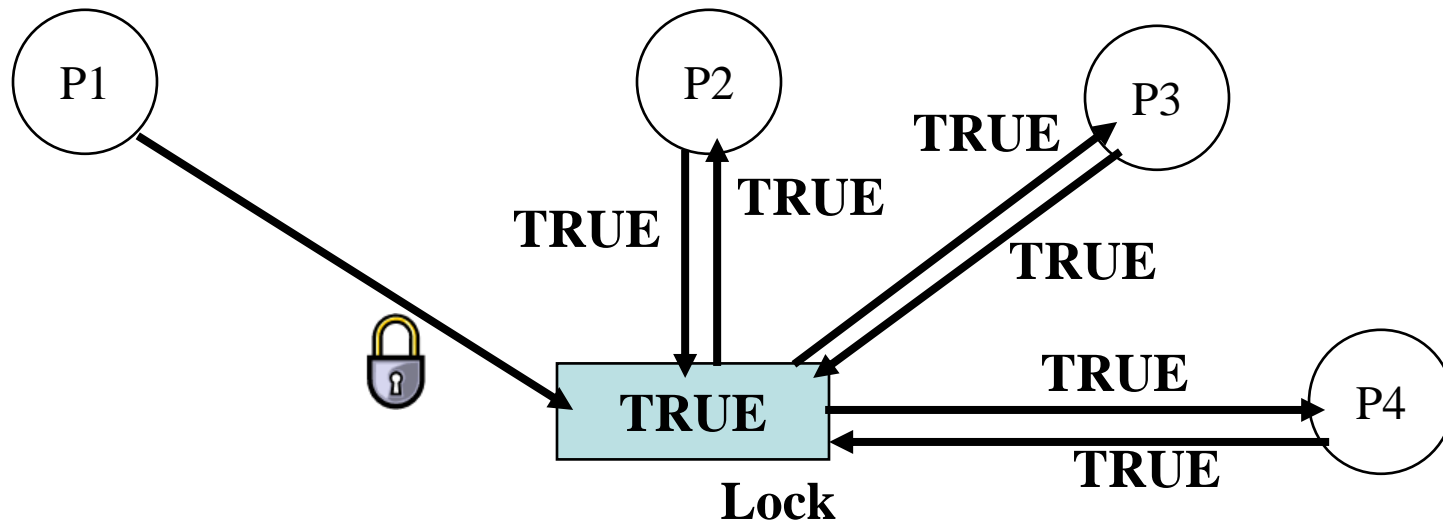
# Test and Set Lock



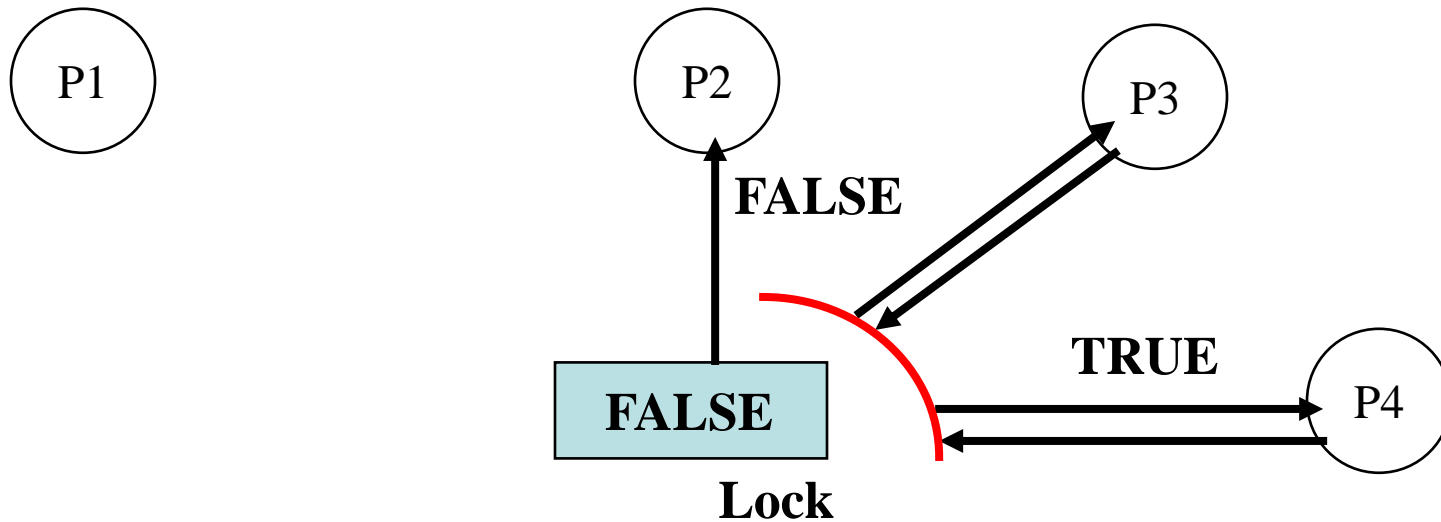
# Test and Set Lock



# Test and Set Lock

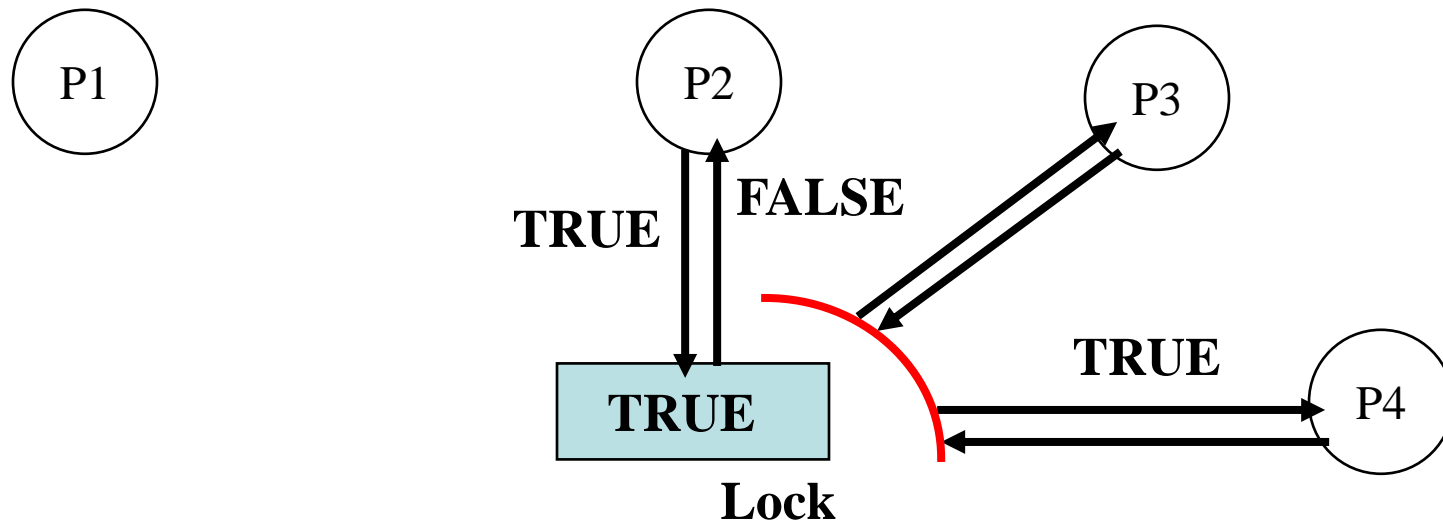


# Test and Set Lock

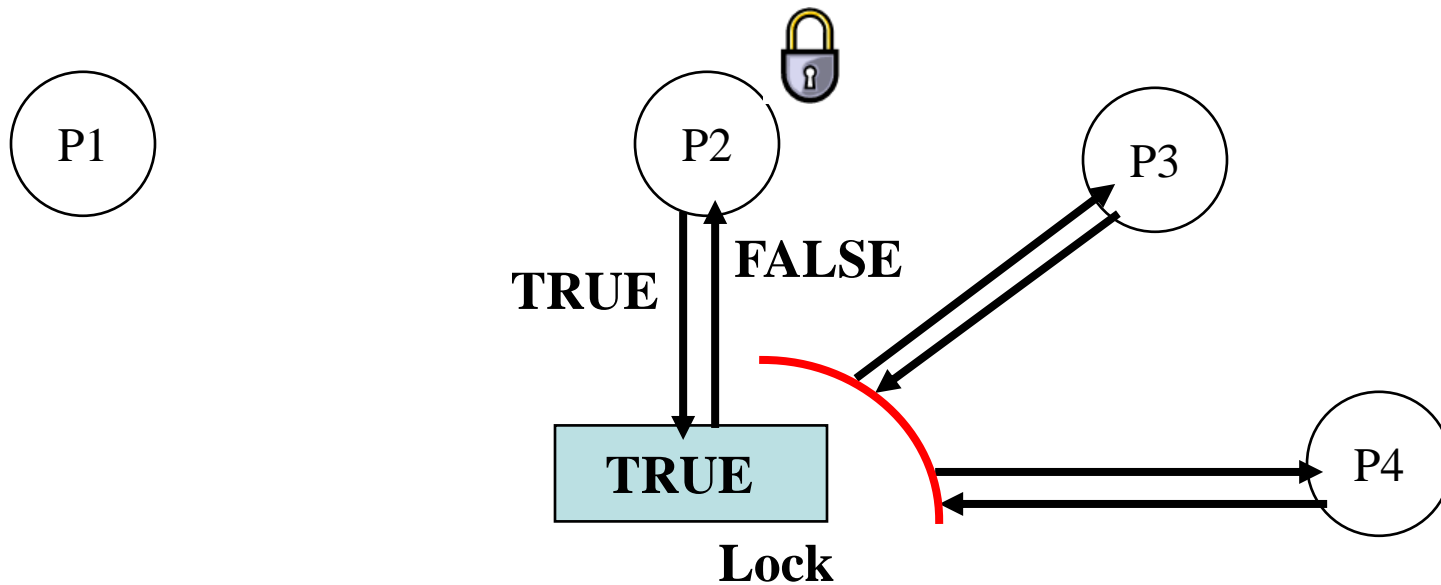




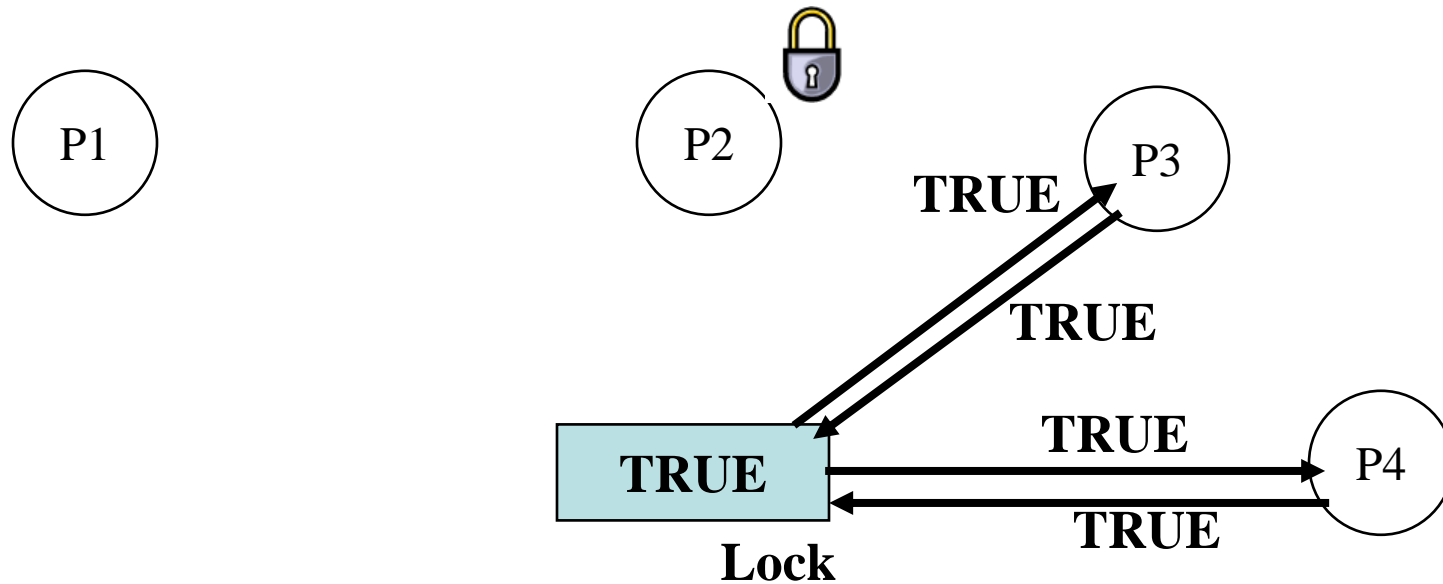
# Test and Set Lock



# Test and Set Lock



# Test and Set Lock



# Using TSL Directly

```
1 repeat                                     I
2   while(TSL(lock))
3     no-op;
4   critical section
5   Lock = FALSE;
6   remainder section
7 until FALSE
```

```
1 repeat                                     J
2   while(TSL(lock))
3     no-op;
4   critical section
5   Lock = FALSE;
6   remainder section
7 until FALSE
```

Guarantees that only one thread at a time will enter its critical section

# Implementing a Mutex With TSL

```
1 repeat
2   while(TSL(mylock)) } Lock (mylock)
3   no-op;
4   critical section
5   mylock = FALSE; } Unlock (mylock)
6   remainder section
7 until FALSE
```

Note that processes are **busy** while waiting

- this kind of mutex is called a **spin lock**

# Busy Waiting

Also called polling or spinning

- *The thread consumes CPU cycles to evaluate when the lock becomes free !*

Problem on a single CPU system...

- A busy-waiting thread can prevent the lock holder from running & completing its critical section & releasing the lock!
  - \* *time spent spinning is wasted on a single CPU system*
- Why not block instead of busy wait ?

# Blocking Primitives

## *Sleep*

- Put a thread to sleep
- Thread becomes BLOCKED

## *Wakeup*

- Move a BLOCKED thread back onto “Ready List”
- Thread becomes READY (or RUNNING)



The END