بسم الله الرحمن الرحیم

# «سیستم عامل»

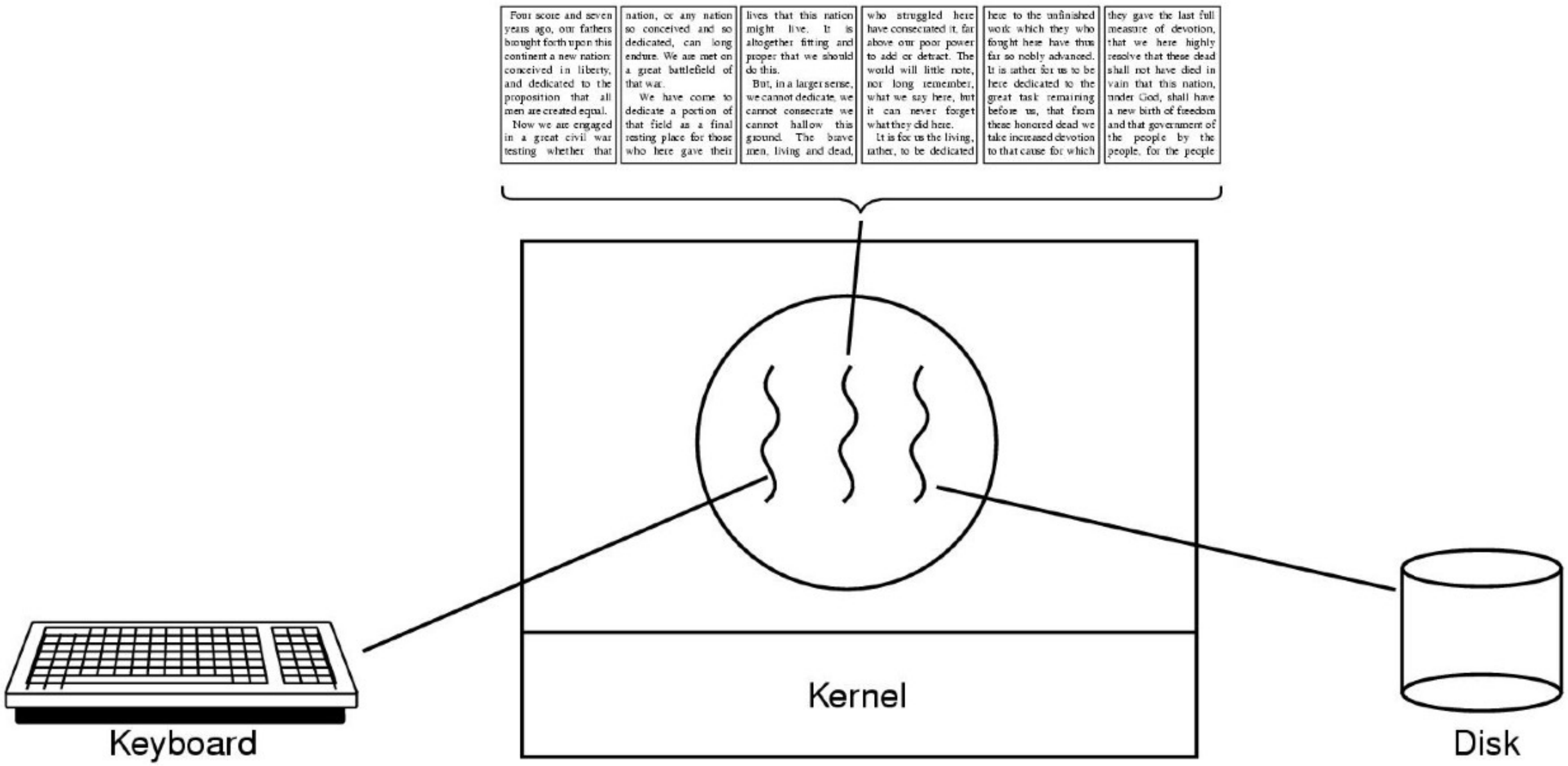


جلسه ۶: هم‌روندی (Concurrency)

# Interprocess communication

- Shared memory
- Message Passing
- Client/Server: Remote Procedure Call

# Thread

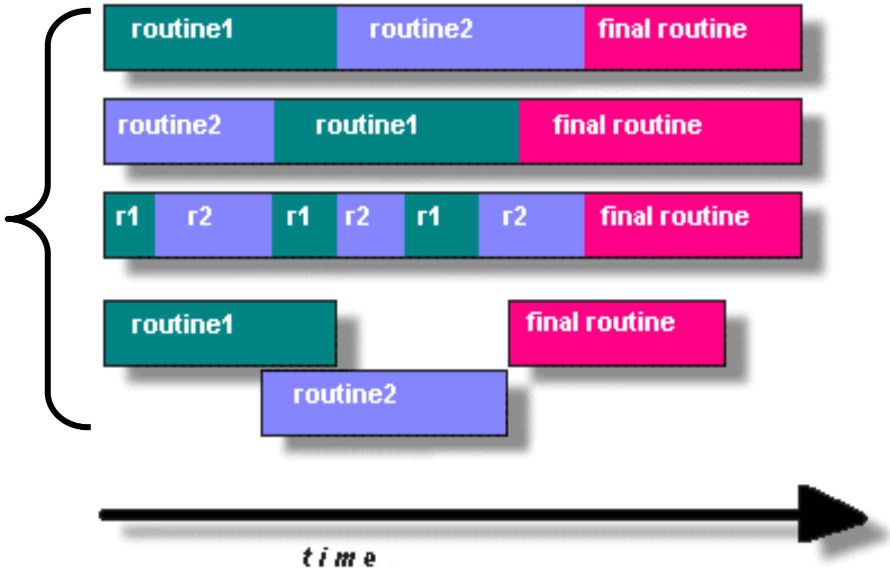



A word processor with three threads

# Concurrency همروندی

## Assumptions:

Alternative strategies for executing multiple rountines




## Example:

- One thread writes a variable
- The other thread reads from the same variable
- Problem – non-determinism:
  - The relative order of one thread's reads and the other thread's writes determines the end result!

# وضعیت رقابتی

# Race conditions

# Race conditions

- A simple multithreaded program with a race:

```
i++;
```

# Race conditions

- A simple multithreaded program with a race:

ریسمان ۱ (i++)

```
...
load i to register;
increment register;
store register to i;
...
```

ریسمان ۲ (i−−)

```
...
load i to register;
decrement register;
store register to i;
...
```

| | | | | |
|---|---|---|---|---|
| $T_0$: | *producer* | execute | $register_1 = \text{count}$ | $\{register_1 = 5\}$ |
| $T_1$: | *producer* | execute | $register_1 = register_1 + 1$ | $\{register_1 = 6\}$ |
| $T_2$: | *consumer* | execute | $register_2 = \text{count}$ | $\{register_2 = 5\}$ |
| $T_3$: | *consumer* | execute | $register_2 = register_2 - 1$ | $\{register_2 = 4\}$ |
| $T_4$: | *producer* | execute | $\text{count} = register_1$ | $\{count = 6\}$ |
| $T_5$: | *consumer* | execute | $\text{count} = register_2$ | $\{count = 4\}$ |

# Race conditions

- **Why did this race condition occur?**
  - two or more threads have an inconsistent view of a shared memory region (I.e., a variable)
  - values of memory locations replicated in registers during execution
  - context switches at arbitrary times during execution
  - threads can see “stale” memory values in registers

# Race Conditions

- **Race condition: whenever the output depends on the precise execution order of the processes!**
- **What solutions can we apply?**
  - prevent context switches by preventing interrupts
  - make threads coordinate with each other to ensure mutual exclusion in accessing critical sections of code

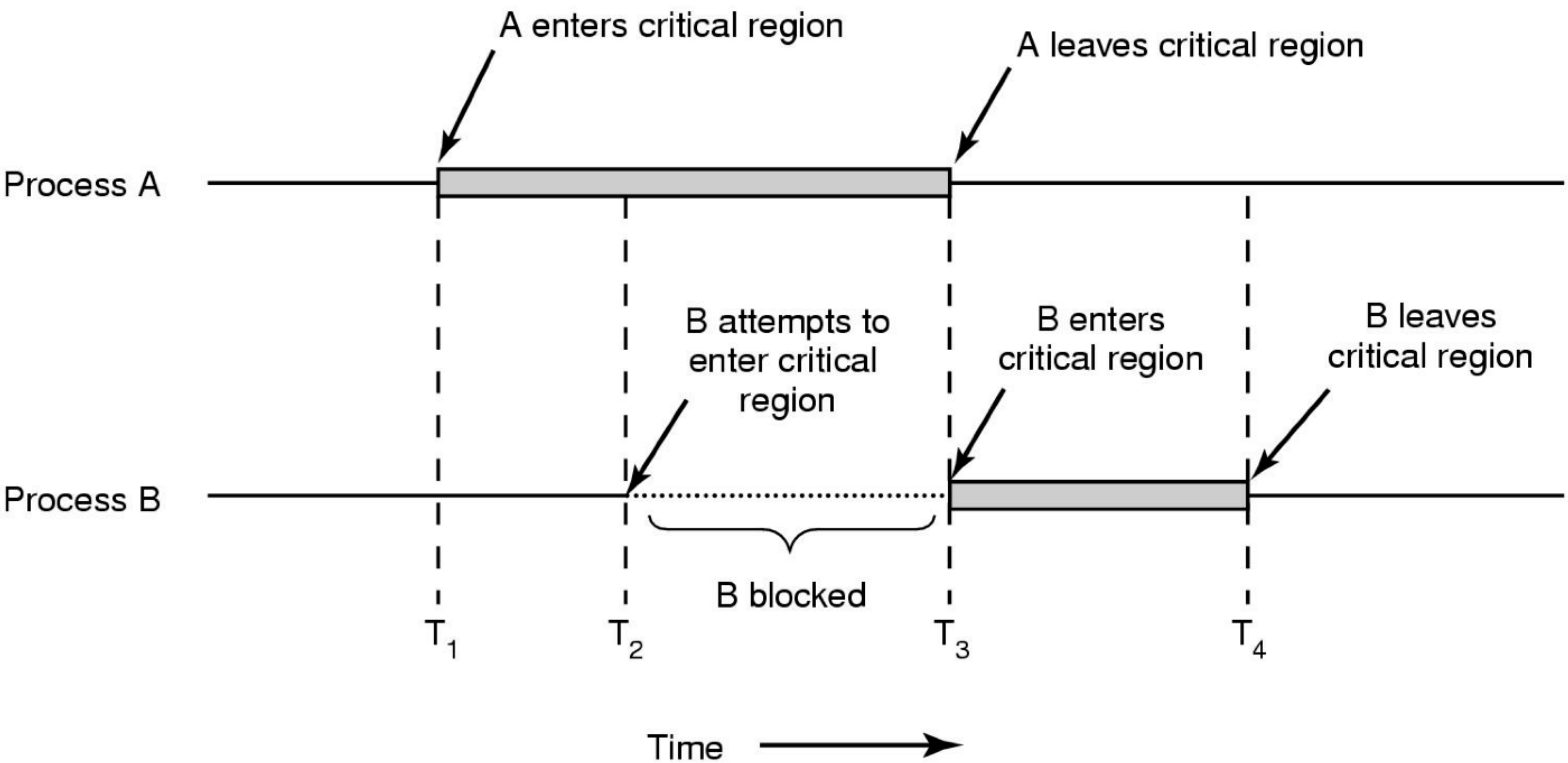
# وضعیت رقابتی
# Race conditions

## mutex —

# Mutual exclusion conditions

- Mutual exclusion. No two processes simultaneously in critical section
    - No assumptions made about speeds or numbers of CPUs
- Progress. No process running outside its critical section may block another process
- Bounded waiting. No process must wait forever to enter its critical section

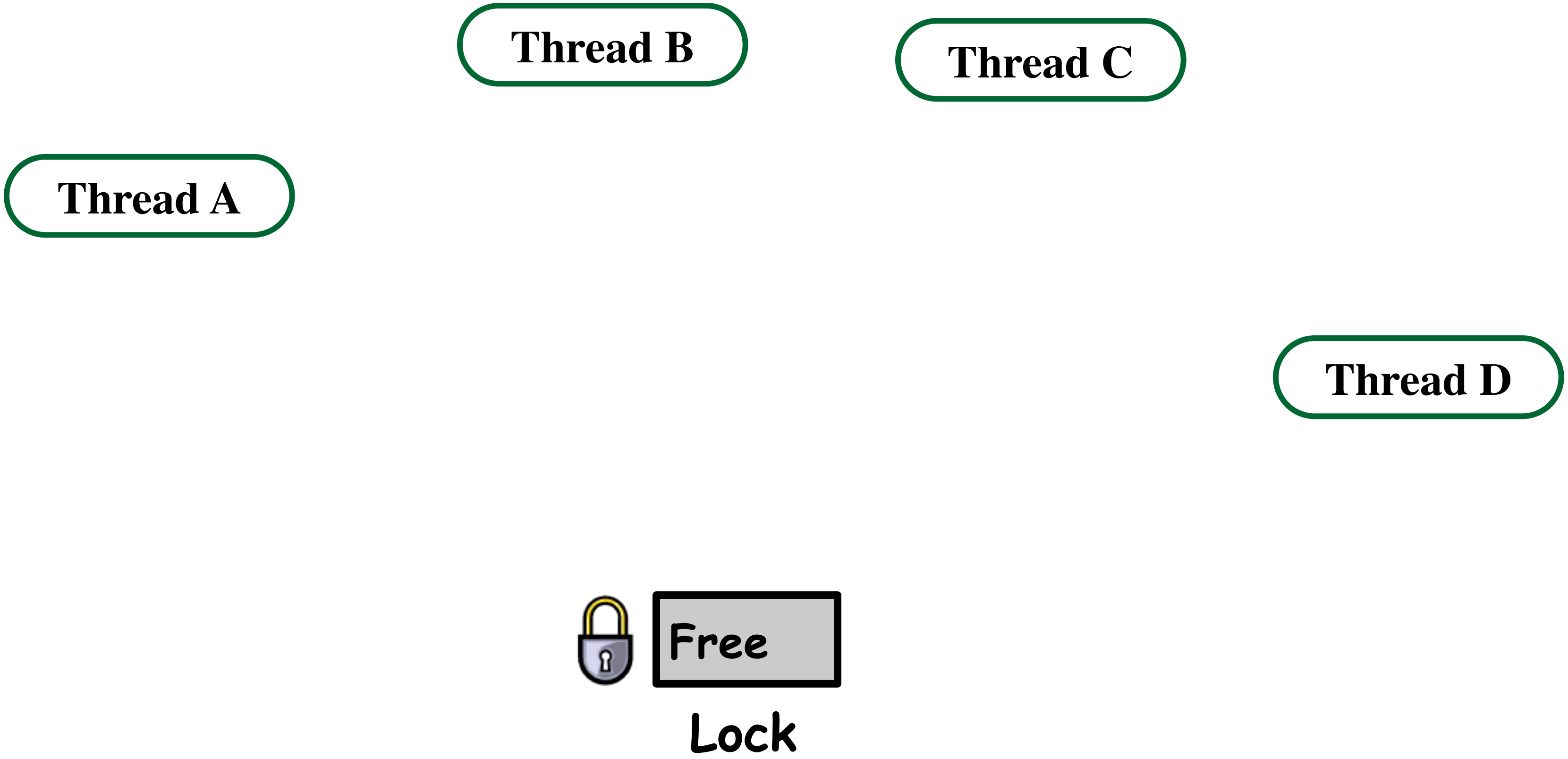
# Using mutual exclusion for critical sections



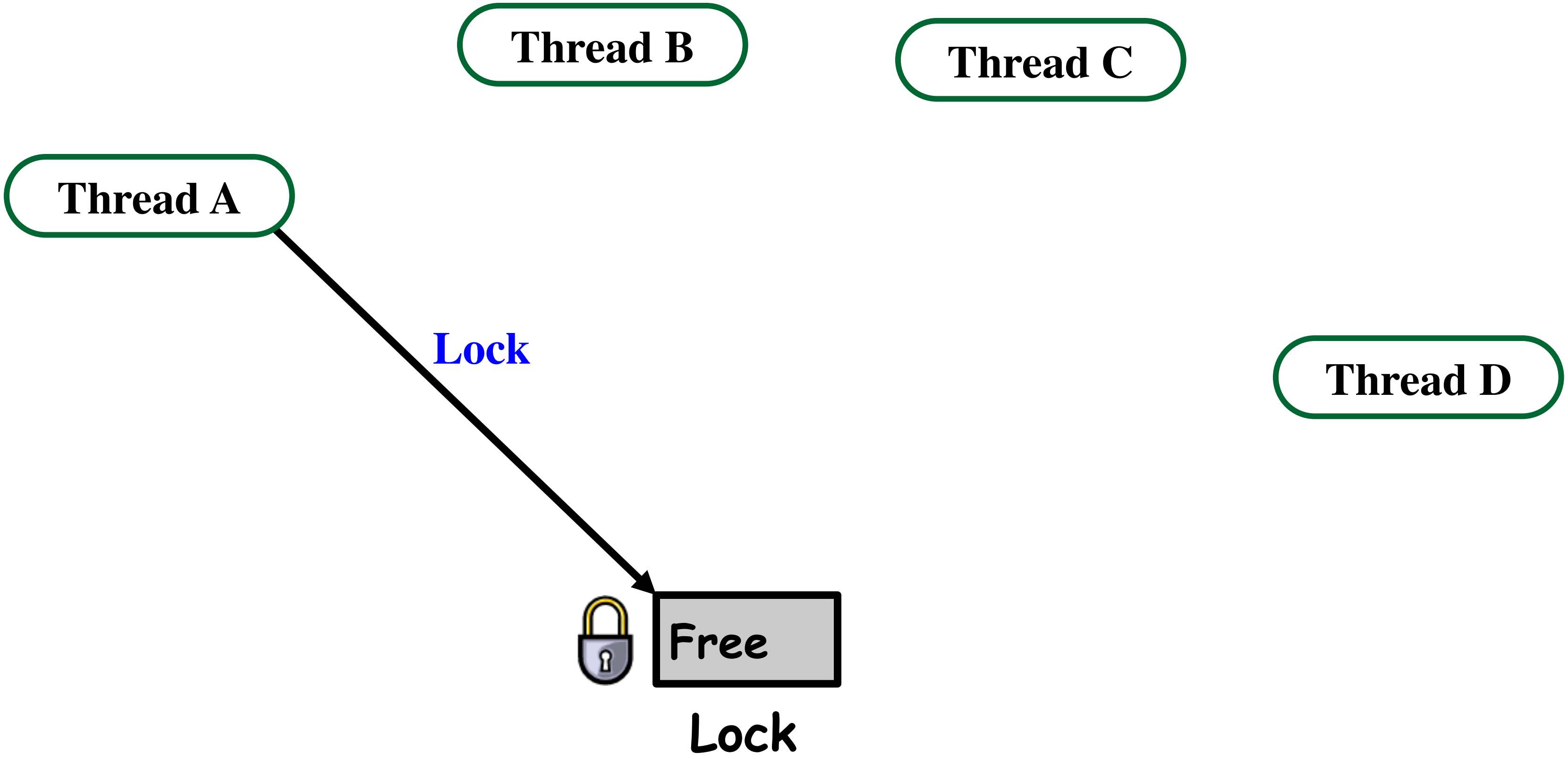
# How can we enforce mutual exclusion?

- **What about using locks ?**
- **Locks solve the problem of 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

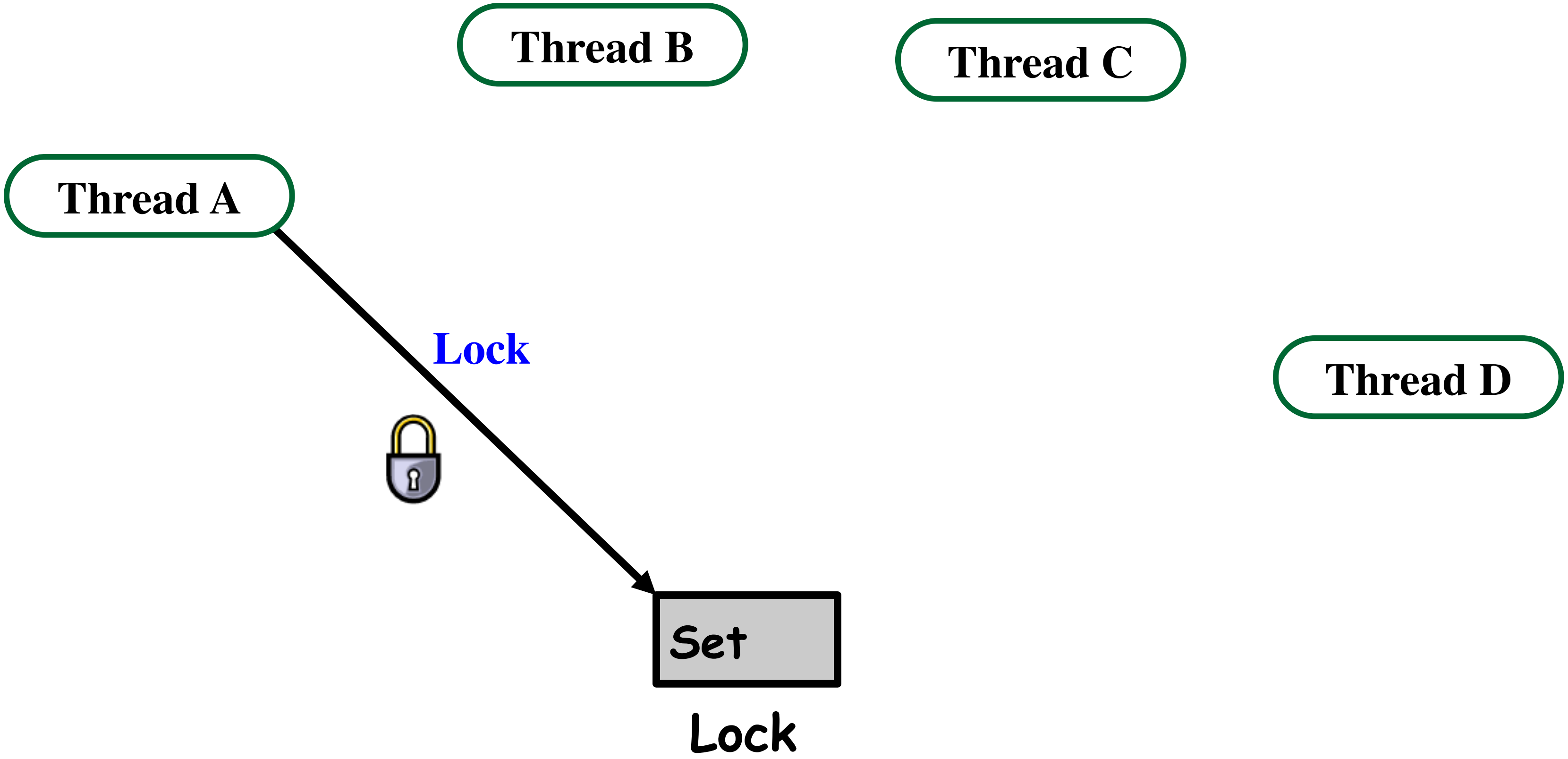
# Acquiring and releasing locks



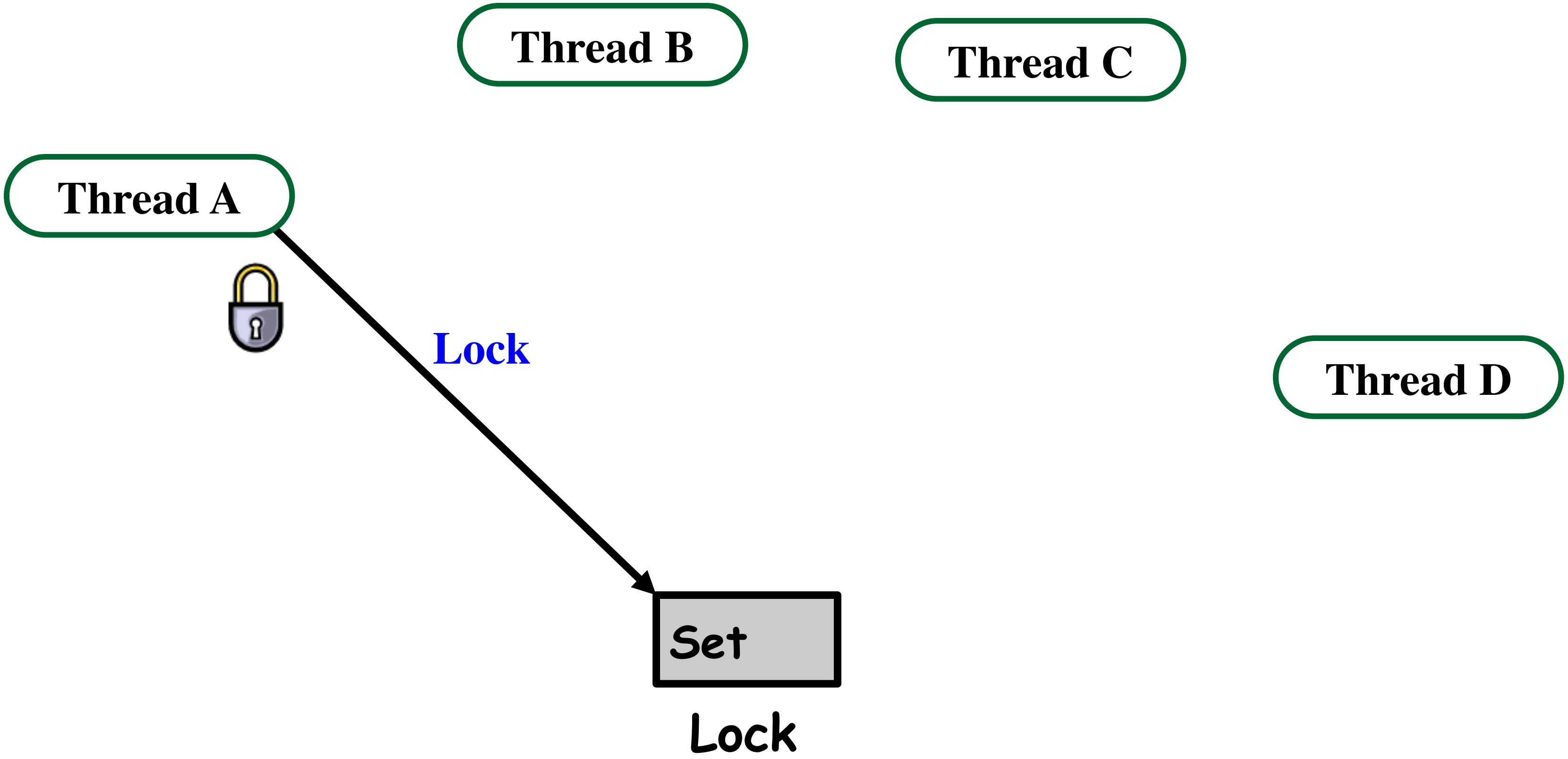
# Acquiring and releasing locks



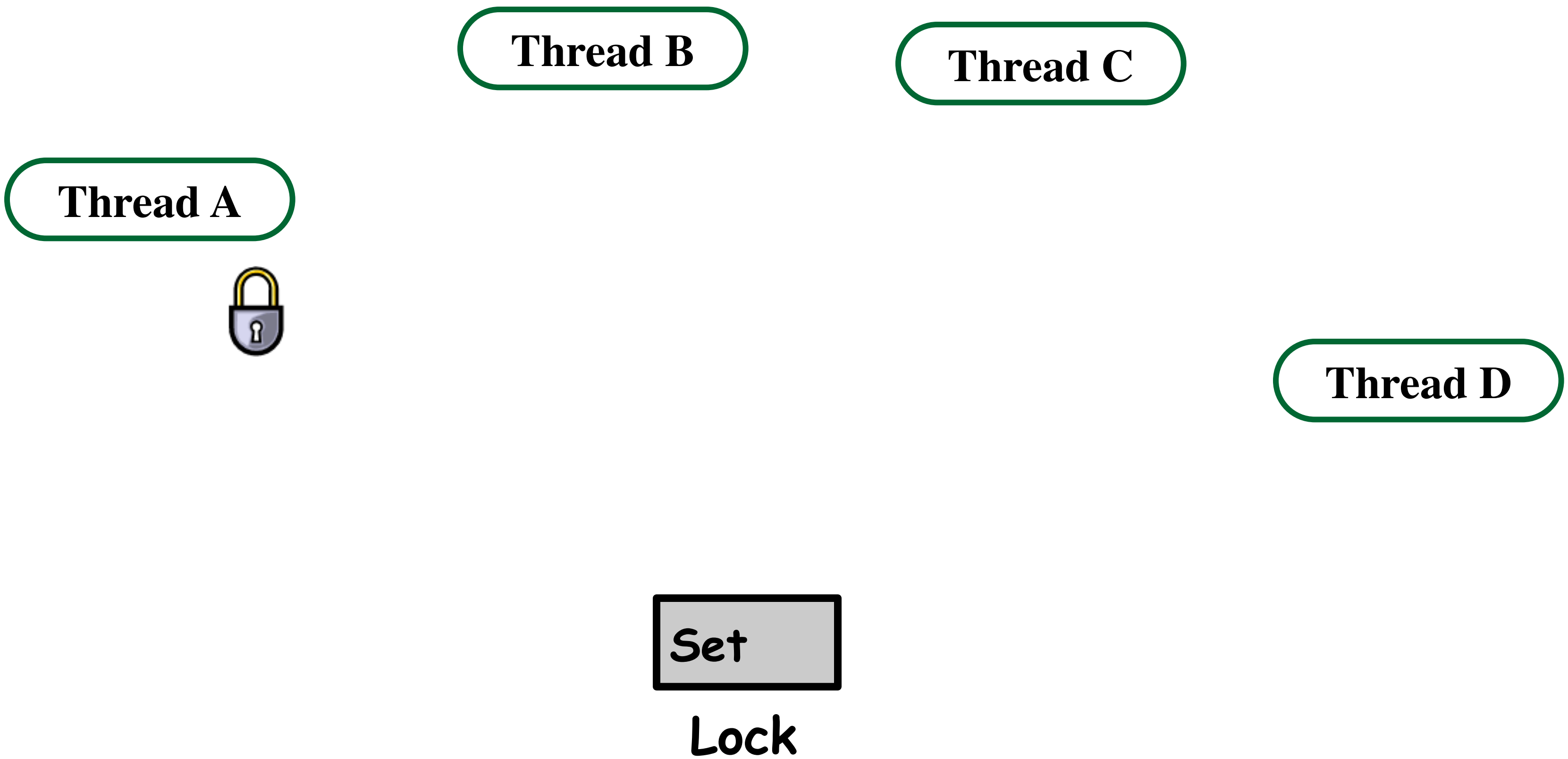
# Acquiring and releasing locks



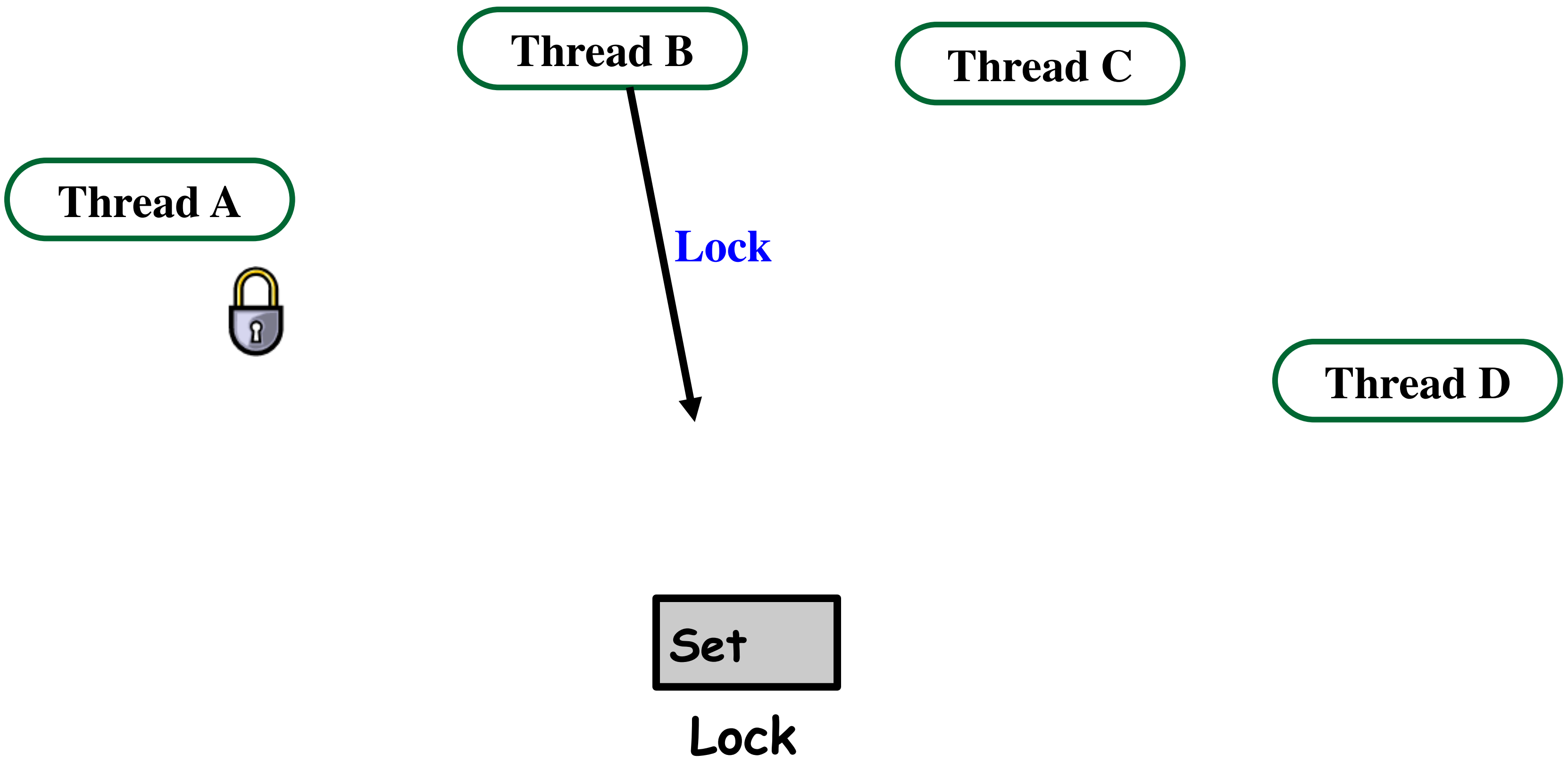
# Acquiring and releasing locks



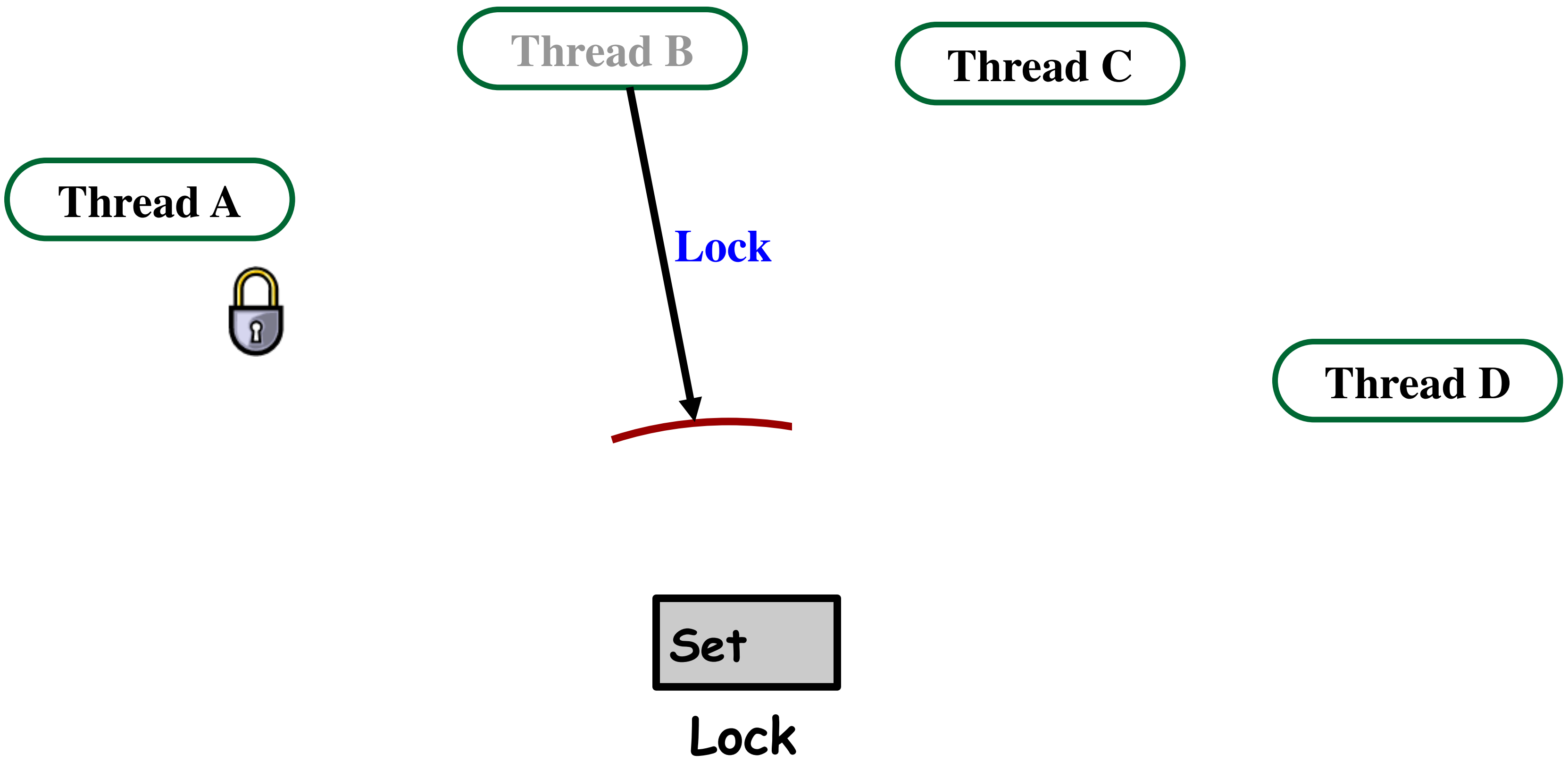
# Acquiring and releasing locks



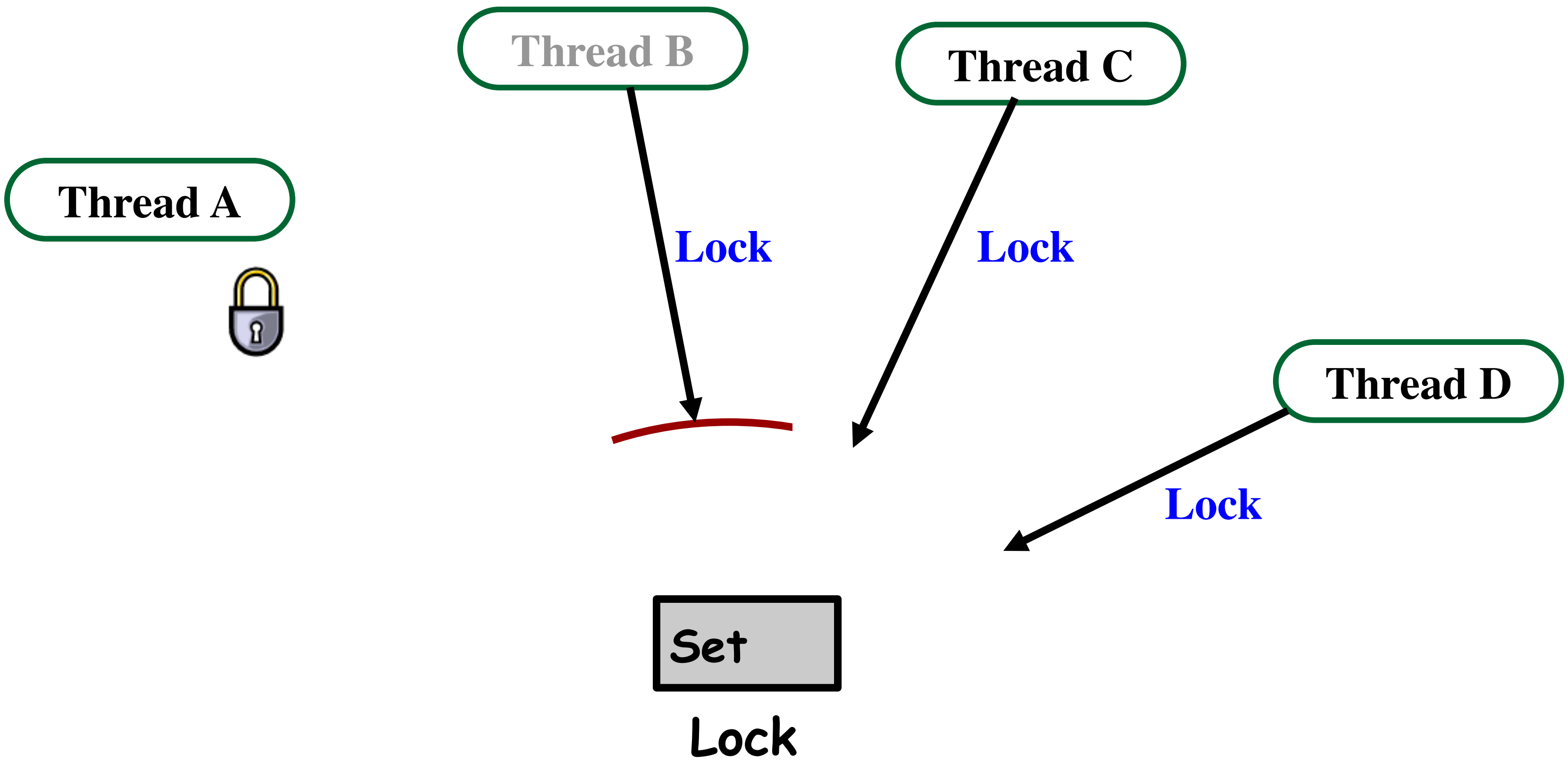
# Acquiring and releasing locks



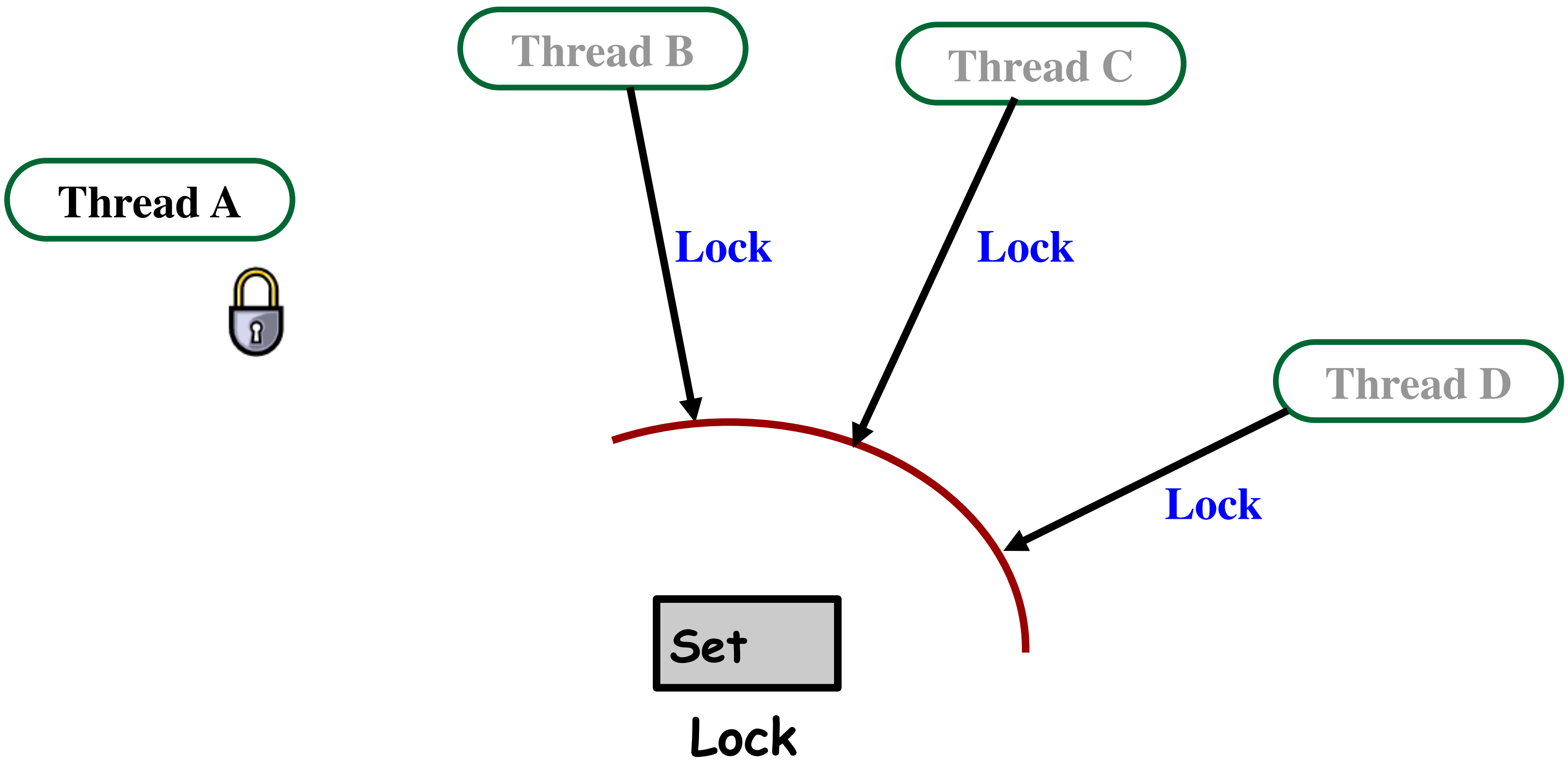
# Acquiring and releasing locks



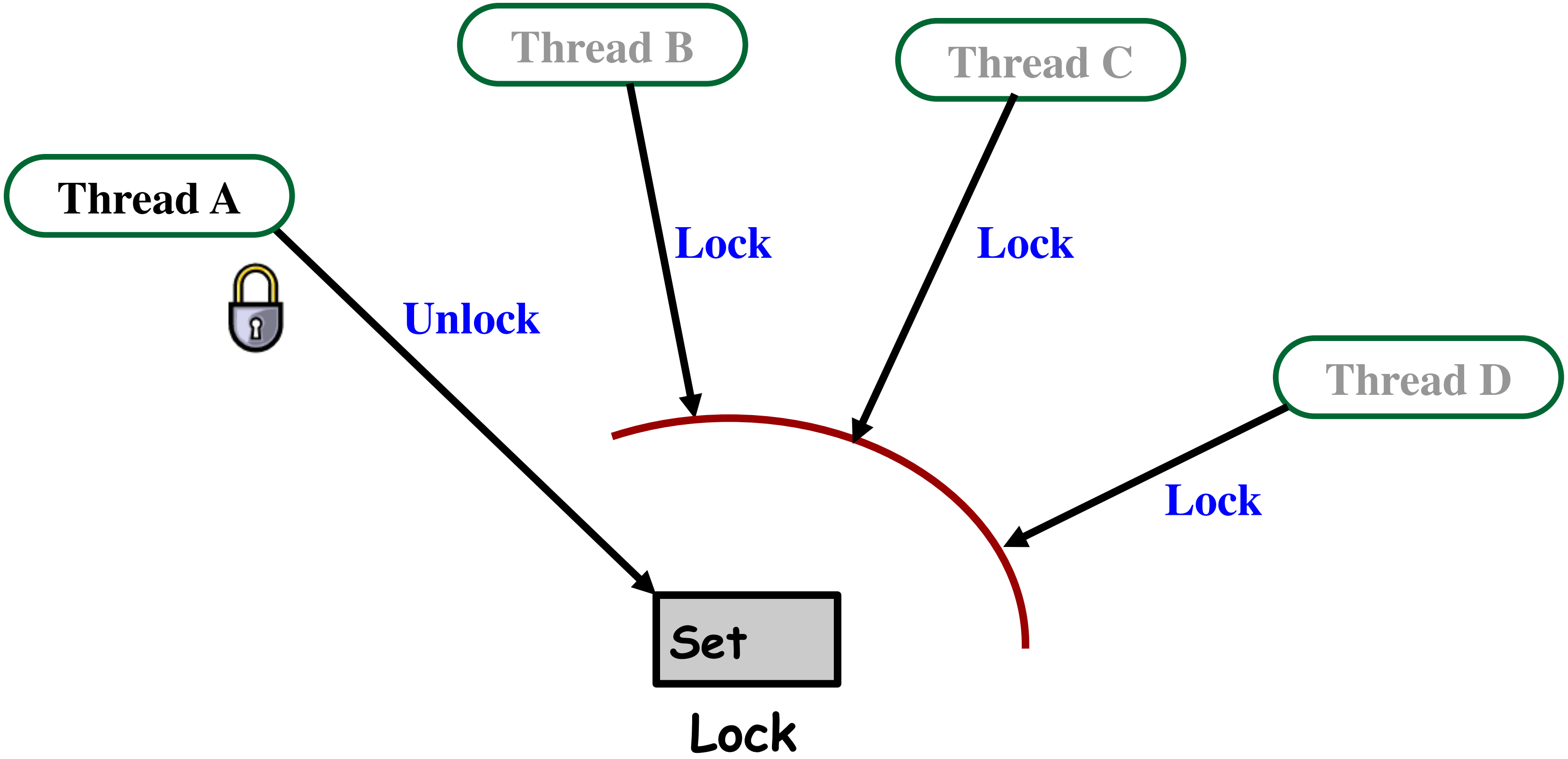
# Acquiring and releasing locks



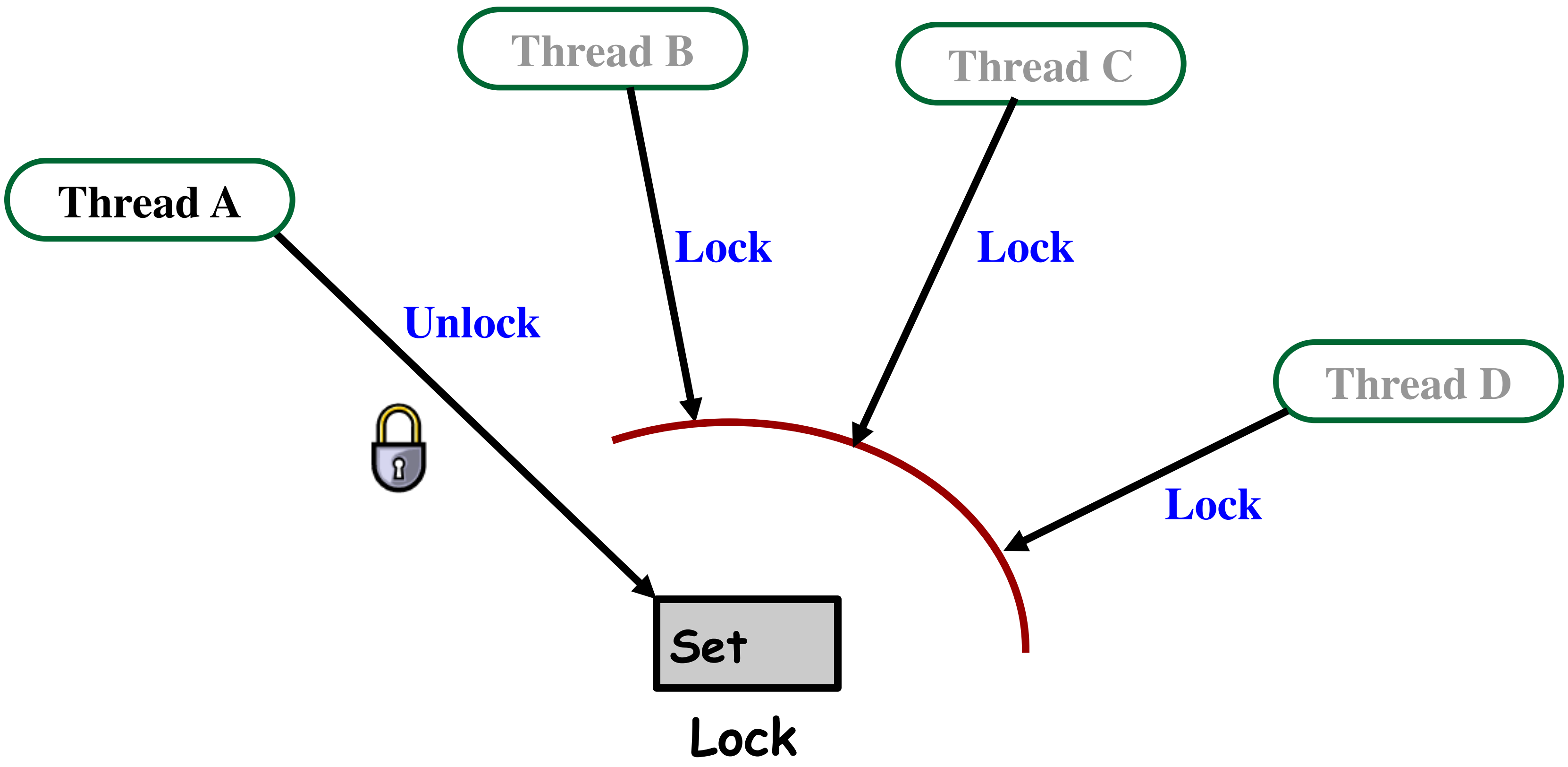
# Acquiring and releasing locks



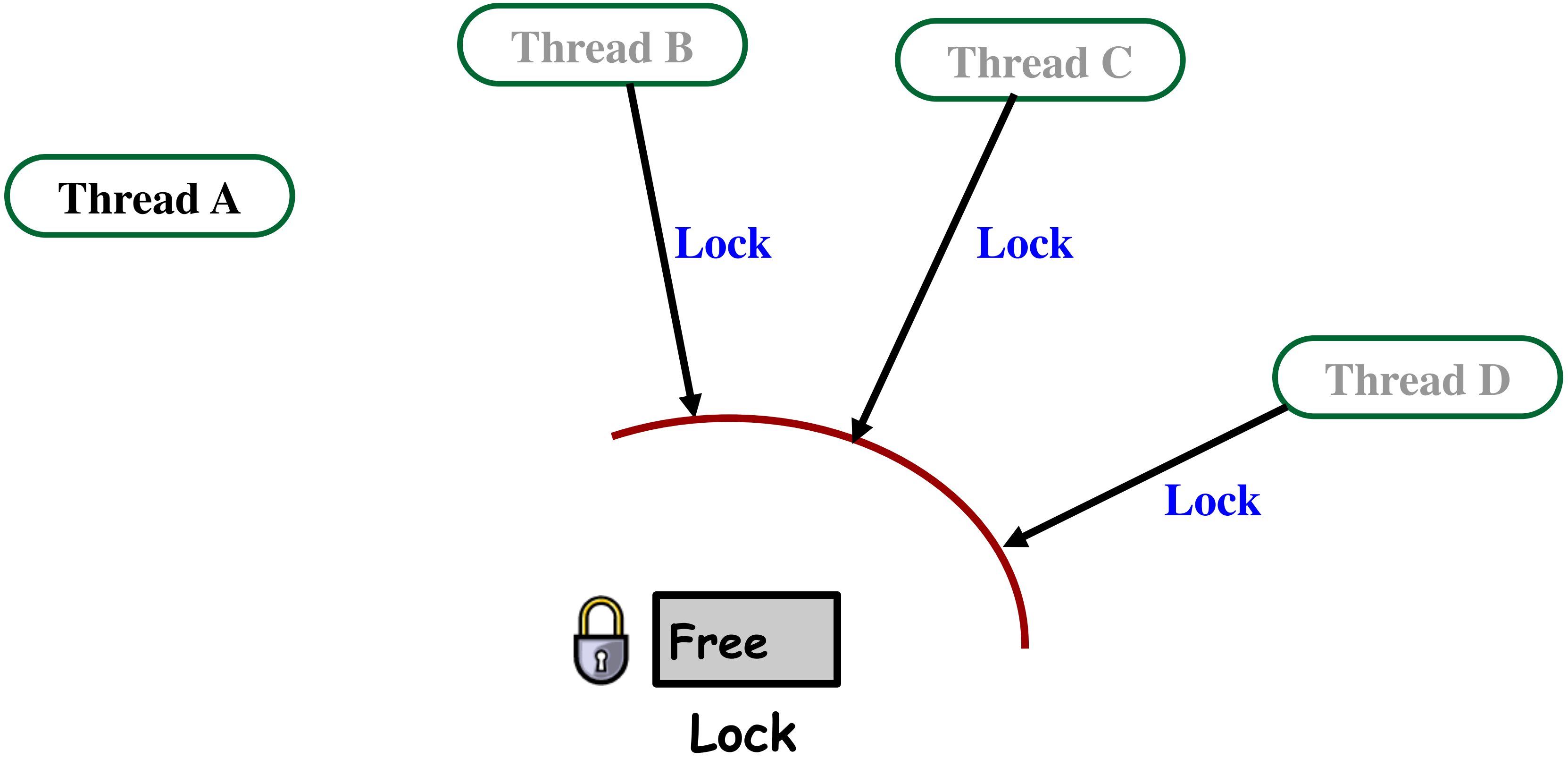
# Acquiring and releasing locks



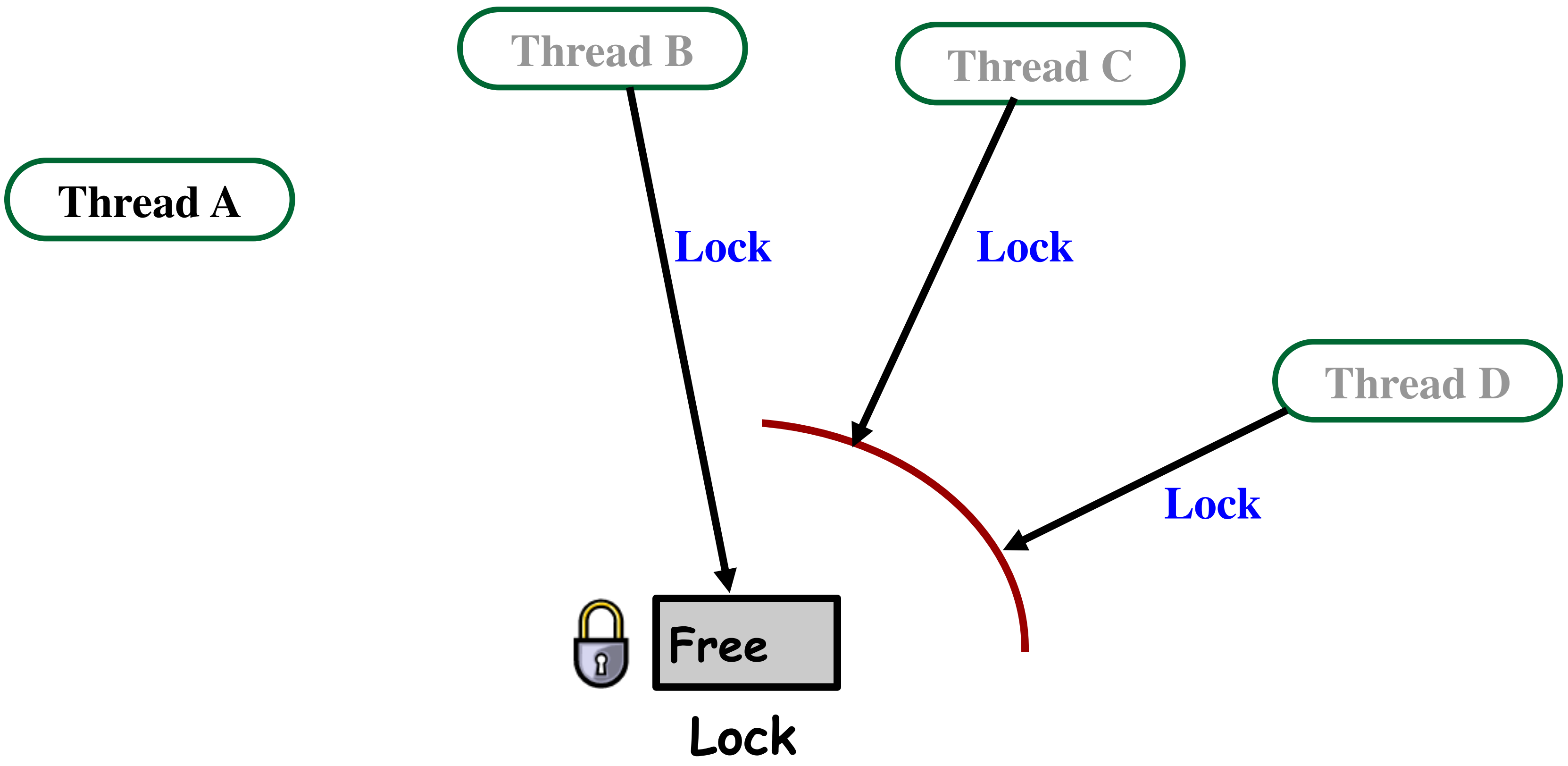
# Acquiring and releasing locks



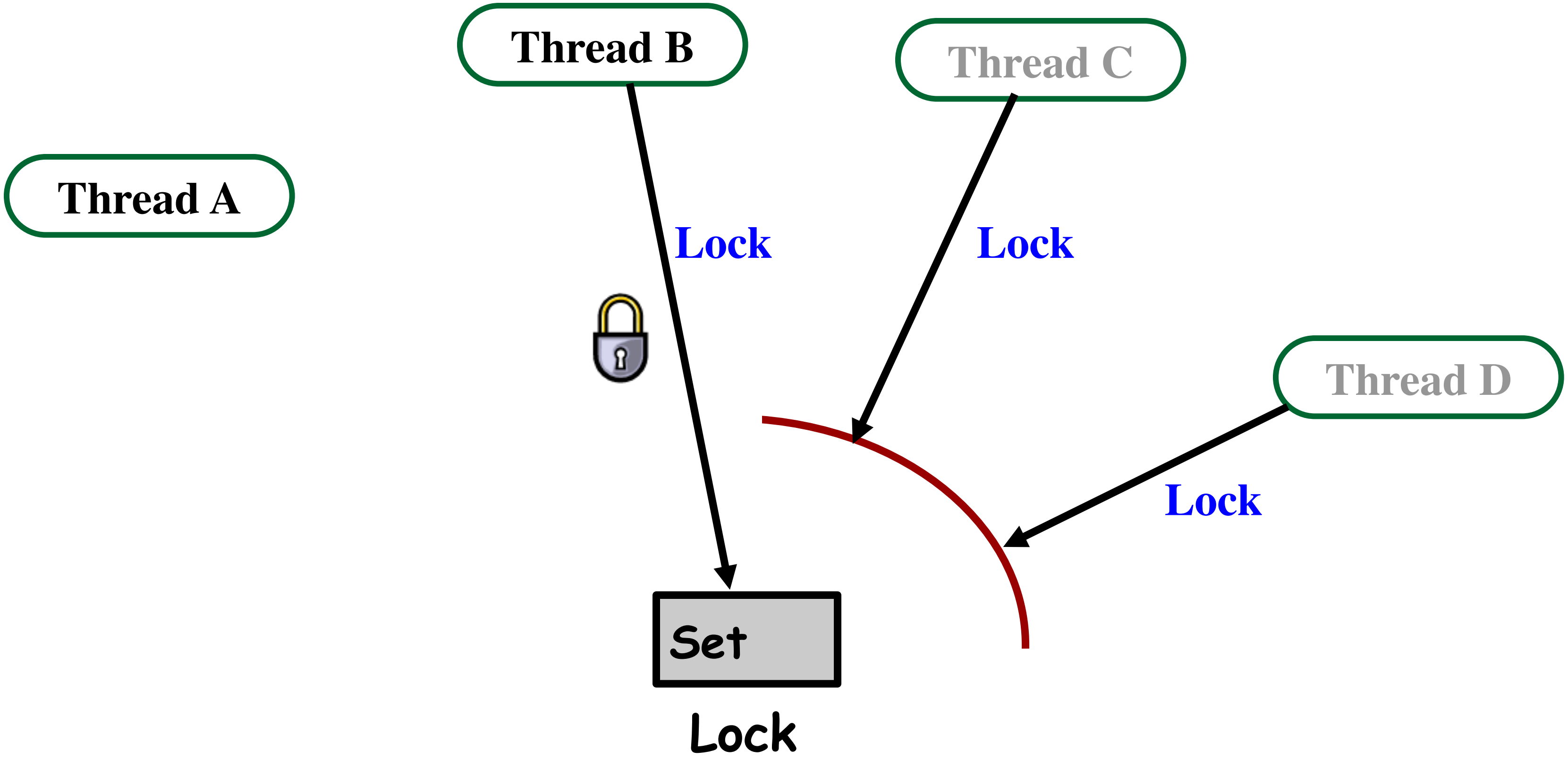
# Acquiring and releasing locks



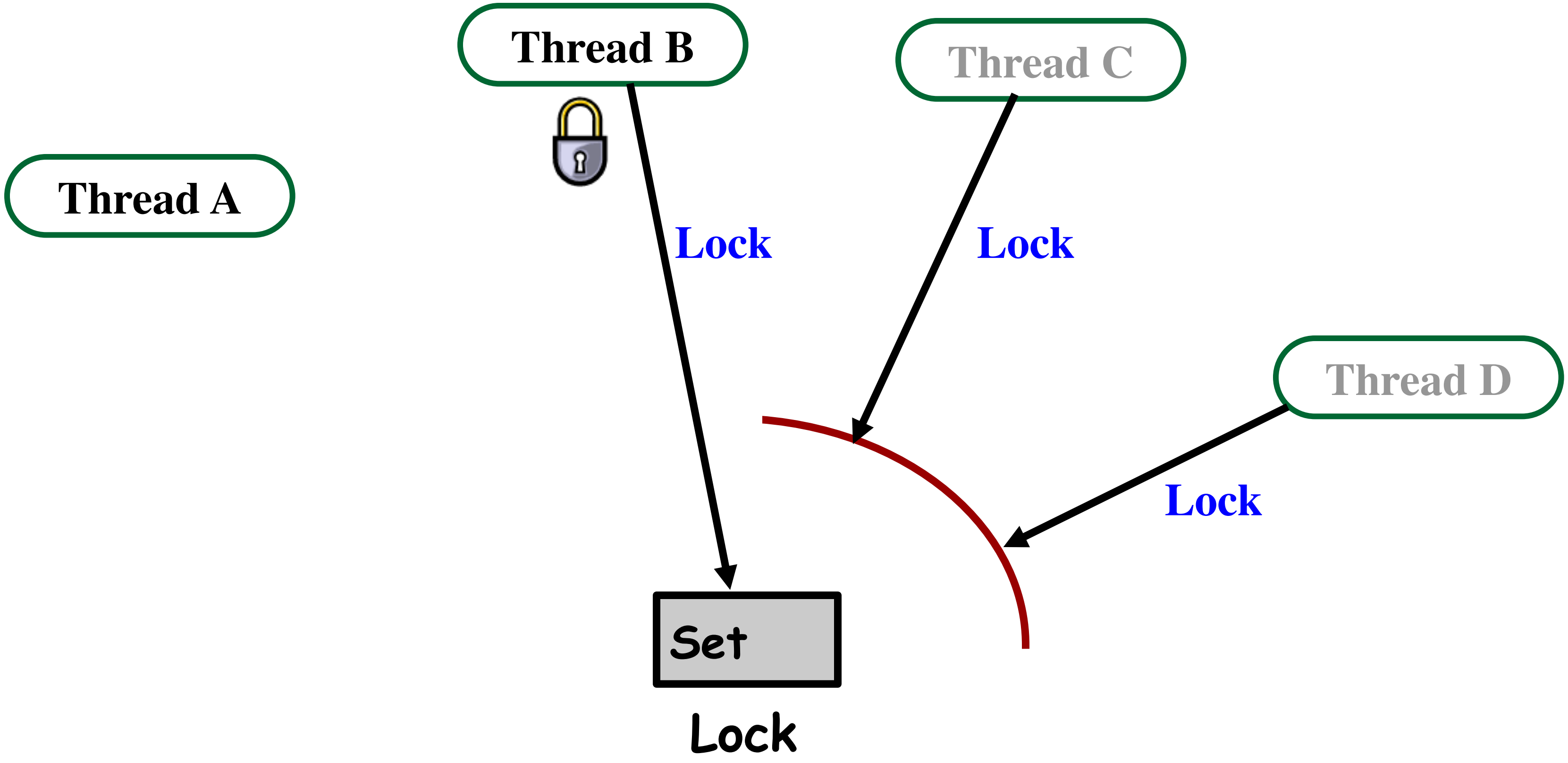
# Acquiring and releasing locks



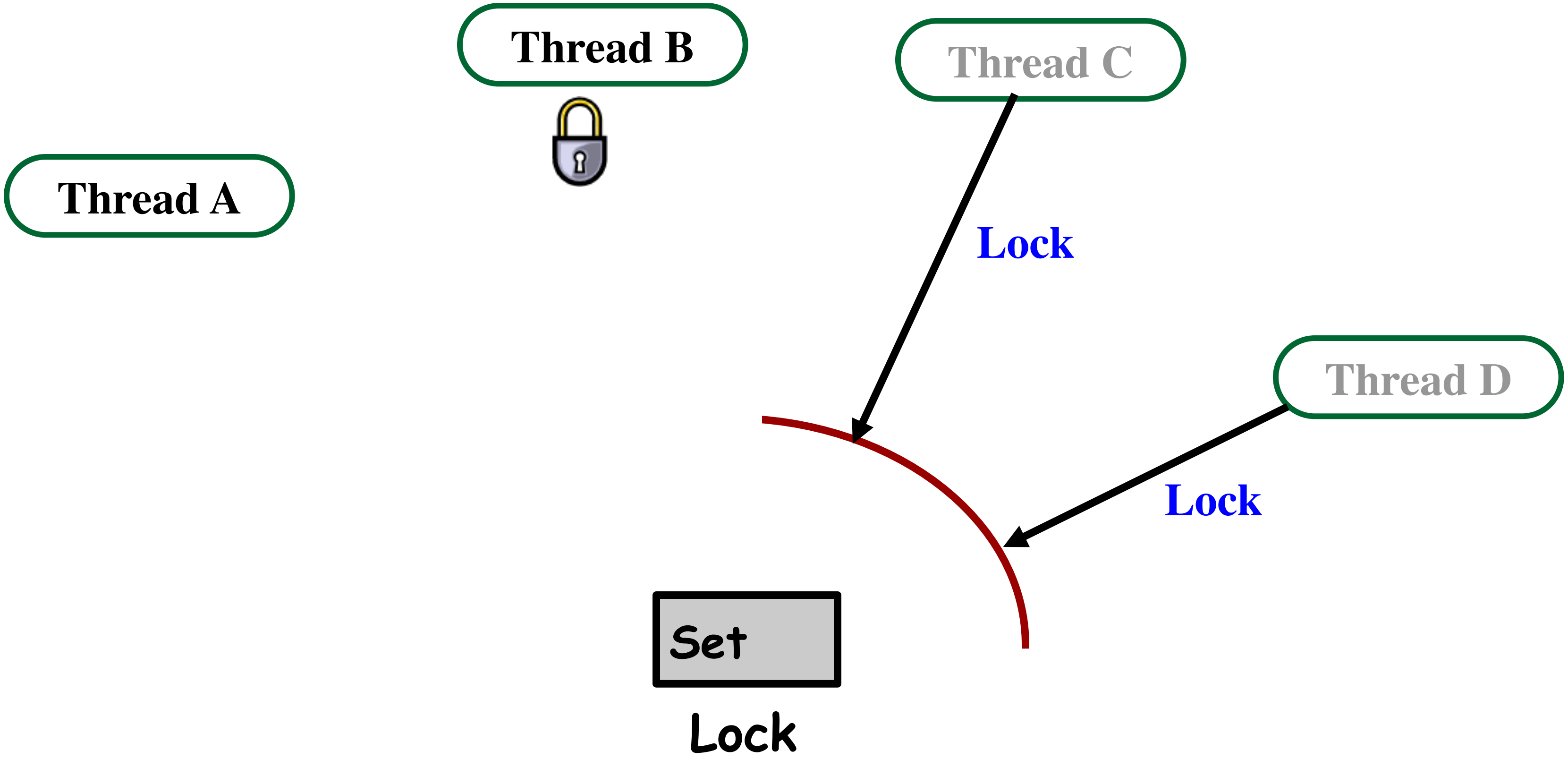
# Acquiring and releasing locks



# Acquiring and releasing locks



# Acquiring and releasing locks



# 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 up one of them

# How to use a mutex?

<u>Shared data:</u>

```
Mutex myLock;
```

```
repeat
   Lock(myLock);
   critical section
   Unlock(myLock);
   remainder section
until FALSE
```

```
repeat
   Lock(myLock);
   critical section
   Unlock(myLock);
   remainder section
until FALSE
```

# But how can we implement a mutex?

- What if the lock is a binary variable
- How would we implement the lock and unlock procedures?

# But how can we implement a mutex?

- **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 operation
- **These can be used to implement mutex locks**

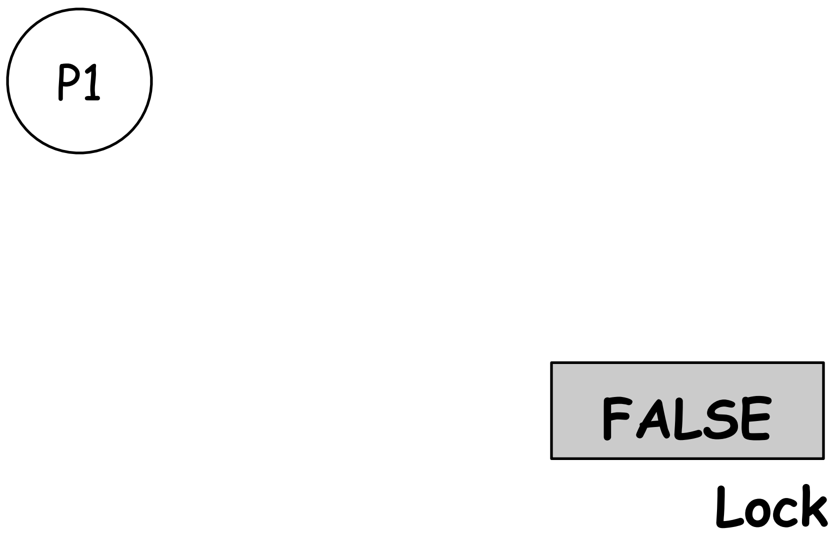
# Test-and-set-lock instruction (TSL, tset)

- **A lock is a single word variable with two values**
  - 0 = FALSE = not locked
  - 1 = TRUE = locked

- **Test-and-set does the following <u>atomically</u>:**
  - Get the (old) value
  - Set the lock to TRUE
  - Return the old value

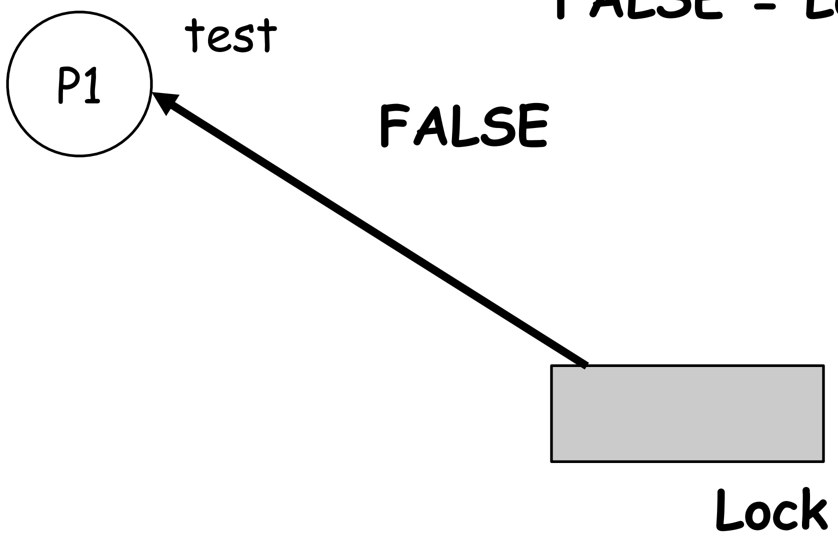
**If** the returned value was FALSE...
**Then** you got the lock!!!
**If** the returned value was TRUE...
**Then** someone else has the lock
(so try again later)

# Test and set lock

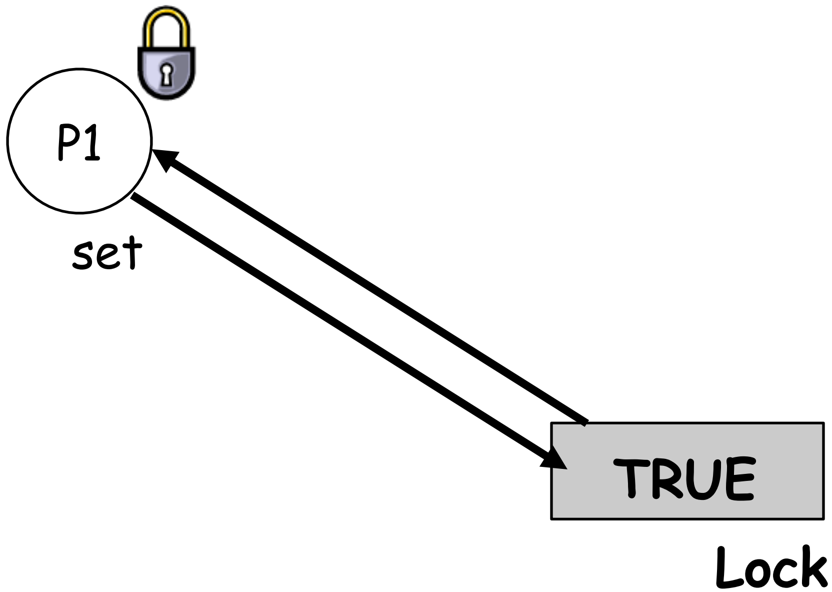


# Test and set lock

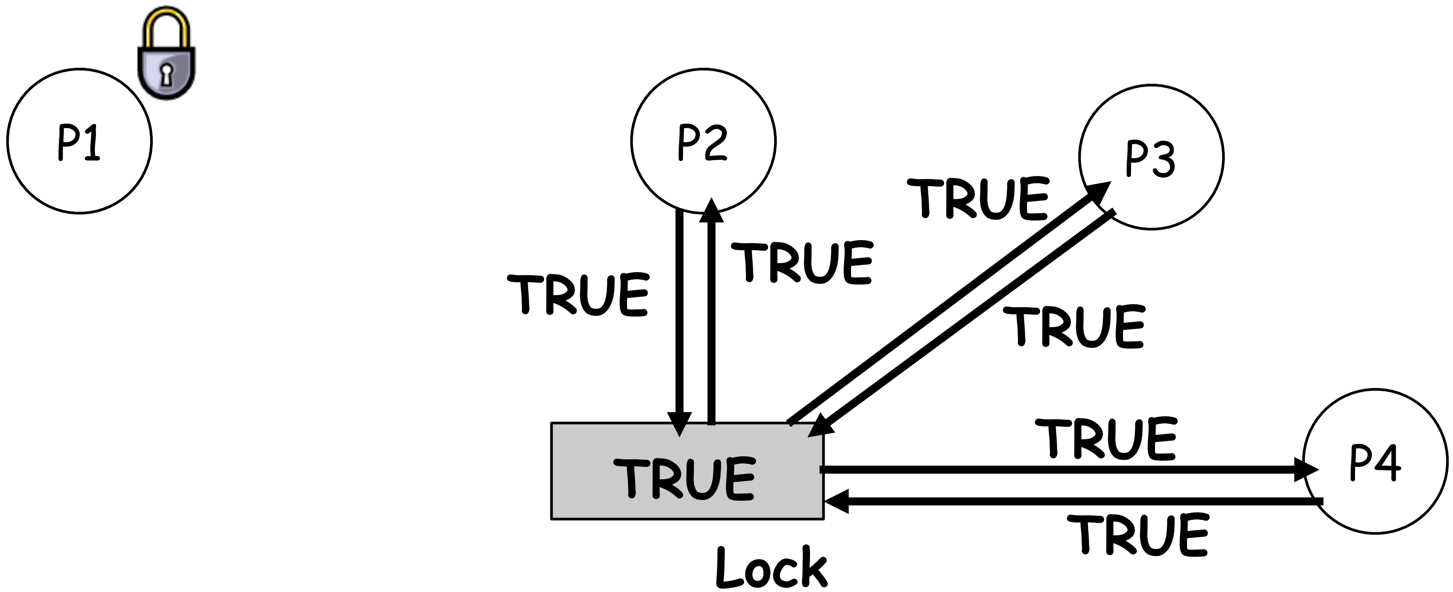
FALSE = Lock Available!!



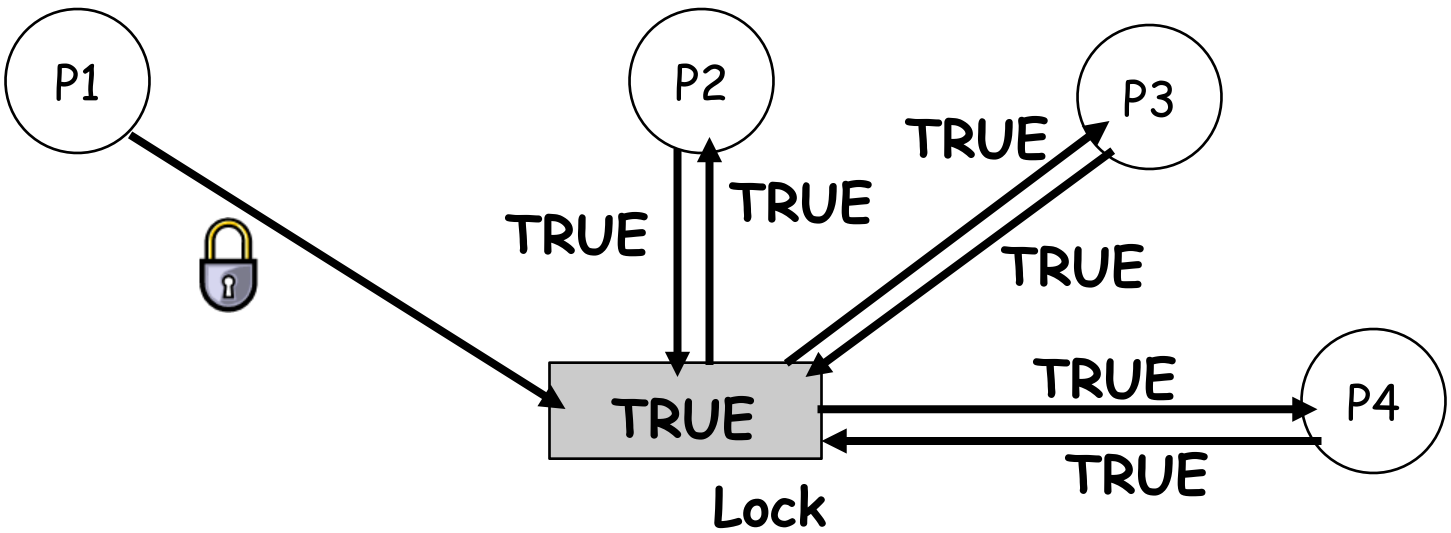
# Test and set lock



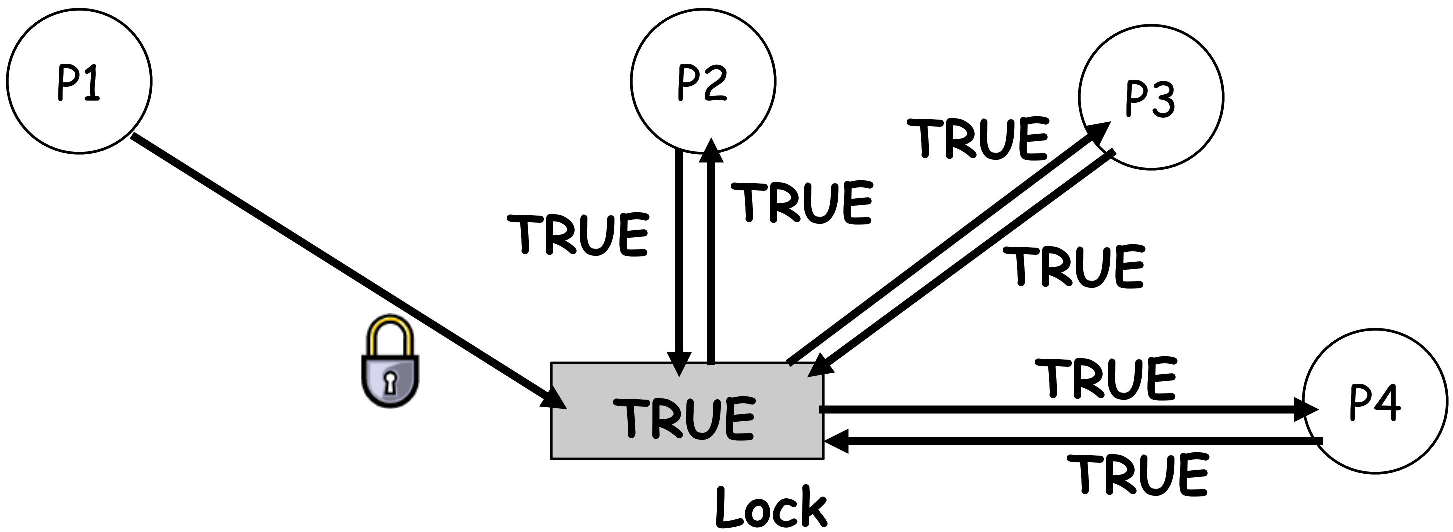
# Test and set lock



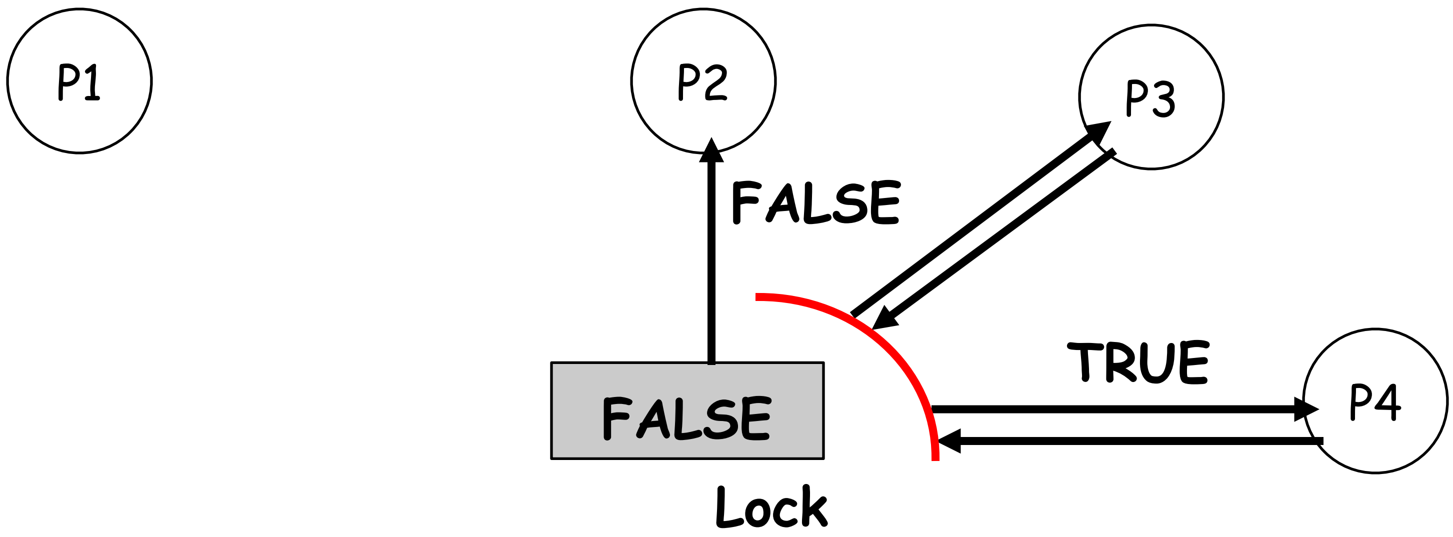
# Test and set lock



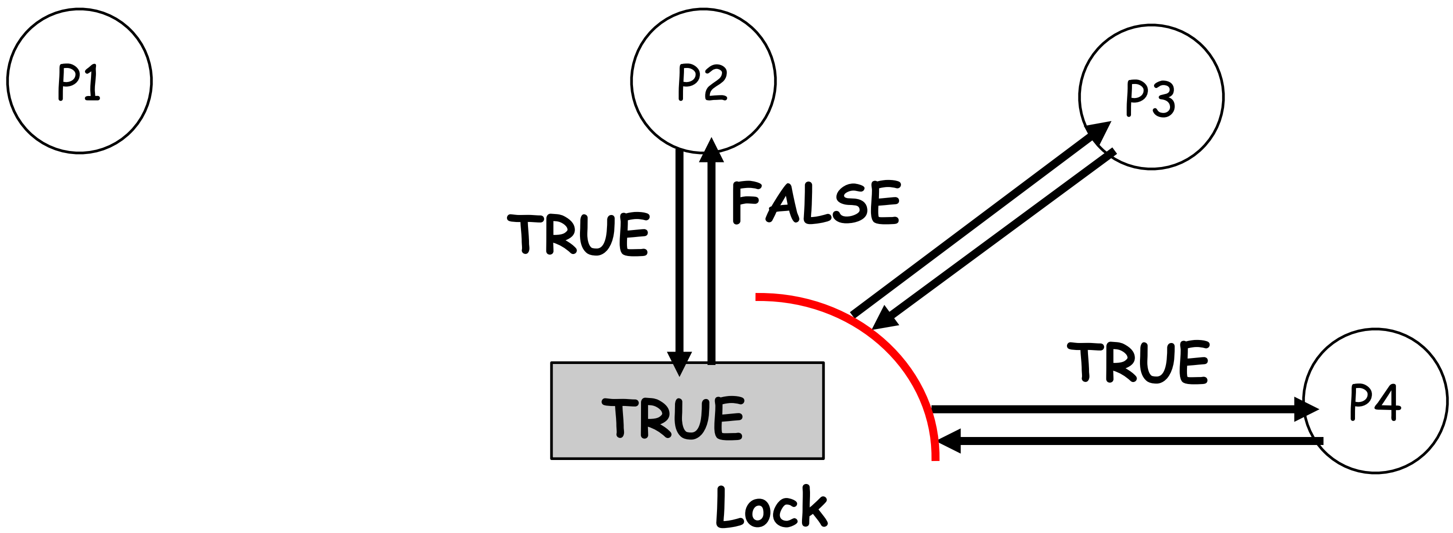
# Test and set lock



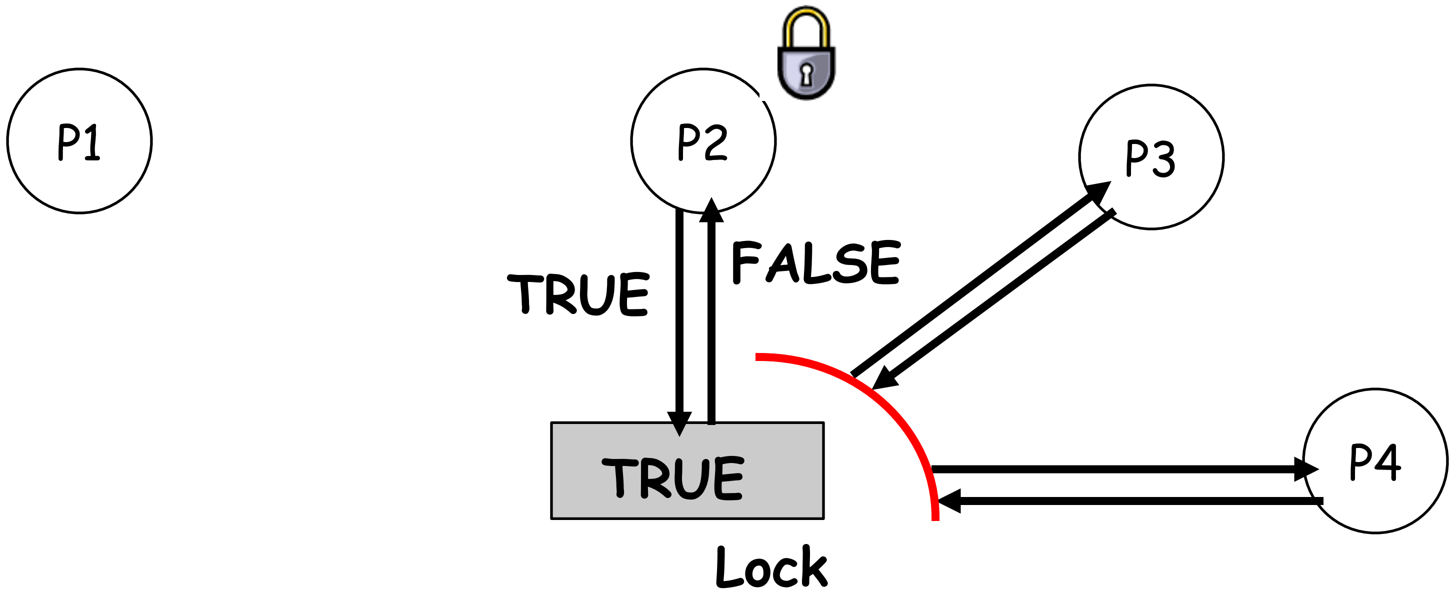
# Test and set lock



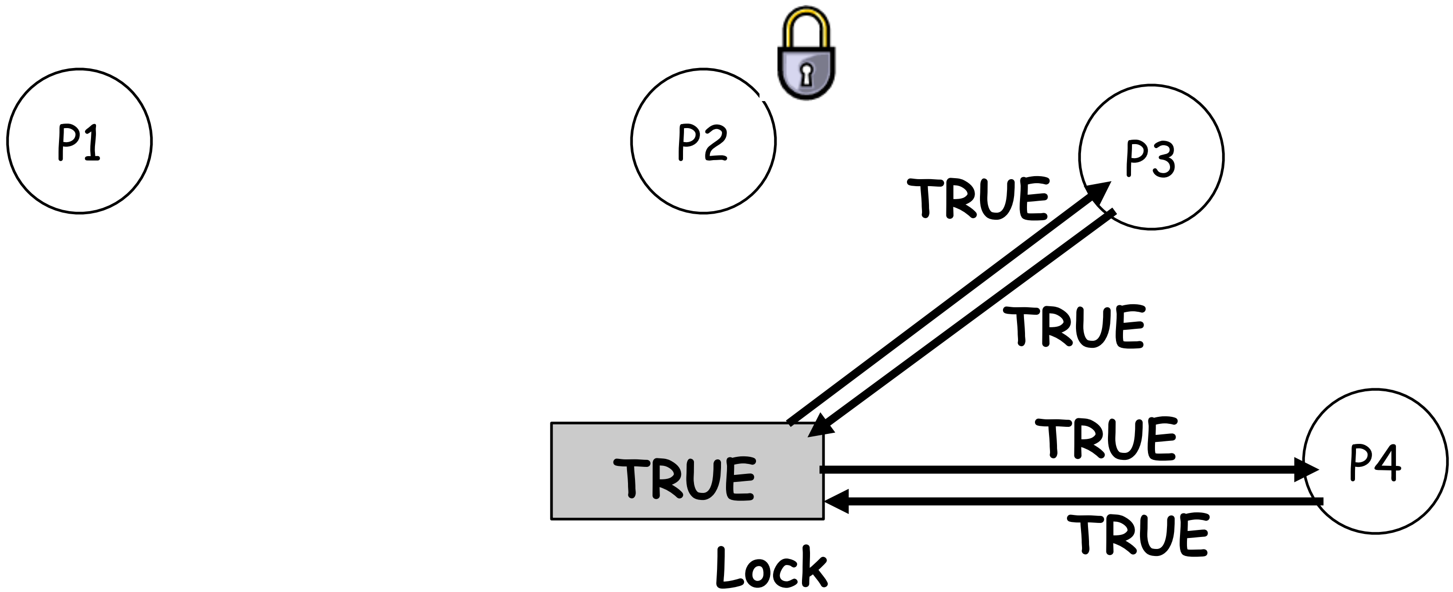
# Test and set lock



# Test and set lock



# Test and set lock



# Using TSL directly for critical sections

I

```
repeat
  while(TSL(lock))
     no-op;
  critical section
  Lock = FALSE;
  remainder section
until FALSE
```

J

```
repeat
  while(TSL(lock))
     no-op;
  critical section
  Lock = FALSE;
  remainder section
until FALSE
```

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

# Implementing a mutex with TSL

```
repeat
  while(TSL(mylock))
     no-op;
  critical section
  mylock = FALSE;
  remainder section
until FALSE
```

Lock (mylock)

Unlock (mylock)

- **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 synchronization primitives

- **Sleep**
  - Put a thread to sleep
  - Thread becomes BLOCKED
- **Wakeup**
  - Move a BLOCKED thread back onto "Ready List"
  - Thread becomes READY (or RUNNING)
- **Yield**
  - Put calling thread on ready list and schedule next thread
  - Does not BLOCK the calling thread!
    - Just gives up the current time-slice

# کنترل همروندی در هسته

# But how can these be implemented?

- **In User Programs:**
  - System calls to the kernel
- **In Kernel:**
  - Calls to the thread scheduler routines

# Concurrency control in user programs

- **User threads call sleep and wakeup system calls**
- **Scheduler routines in the kernel implement sleep and wakeup**
  - they manipulate the "ready list"
  - but the ready list is shared data
  - the code that manipulates it is a critical section
    - What if a timer interrupt occurs during a sleep or wakeup call?
- **Problem:**
  - How can scheduler routines be programmed to execute correctly in the face of concurrency?

# Concurrency in the kernel

**Solution 1: Disable interrupts during critical sections**

- Ensures that interrupt handling code will not run
- … but what if there are multiple CPUs?

**Solution 2: Use mutex locks based on TSL for critical sections**

- Ensures mutual exclusion for all code that follows that convention
- ... but what if your hardware doesn't have TSL?

# Disabling interrupts

- **Disabling interrupts in the OS vs disabling interrupts in user processes**
  - why not allow user processes to disable interrupts?
  - is it ok to disable interrupts in the OS?
  - what precautions should you take?

# Disabling interrupts in the kernel

Scenario 1:

A thread is running; wants to access shared data

Disable interrupts

Access shared data ("critical section")

Enable interrupts

# Disabling interrupts in the kernel

Problem:

Interrupts are already disabled and a thread wants to access the critical section

...using the above sequence...

- Ie. One critical section gets nested inside another

# Disabling interrupts in the kernel

**Problem: Interrupts are already disabled.**

- Thread wants to access critical section using the previous sequence...

**Save previous interrupt status (enabled/disabled)**

**Disable interrupts**

**Access shared data ("critical section")**

**Restore interrupt status to what it was before**

# Disabling interrupts is not enough on MPs…

- **Disabling interrupts during critical sections**
  - Ensures that interrupt handling code will not run
  - But what if there are multiple CPUs?
  - A thread on a different CPU might make a system call which invokes code that manipulates the ready queue

- **Using a mutex lock (based on TSL) for critical sections**
  - Ensures mutual exclusion for all code that follows that convention

# Mutex is not enough

- Interrupt inside interrupt handler
  - 

# Some tricky issues …

- **The interrupt handling code that saves interrupted state is a critical section**
  - It could be executed concurrently if multiple almost simultaneous interrupts happen
  - Interrupts must be disabled during this (short) time period to ensure critical state is not lost
- **What if this interrupt handling code attempts to lock a mutex that is held?**
  - What happens if we sleep with interrupts disabled?
  - What happens if we busy wait (spin) with interrupts disabled?

# Implementing mutex locks without TSL

- **If your CPU did not have TSL, how would you implement blocking mutex lock and unlock calls using interrupt disabling?**
  - … this is your next Blitz project !