R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه ششم – همزمانی

آنچه گذشت

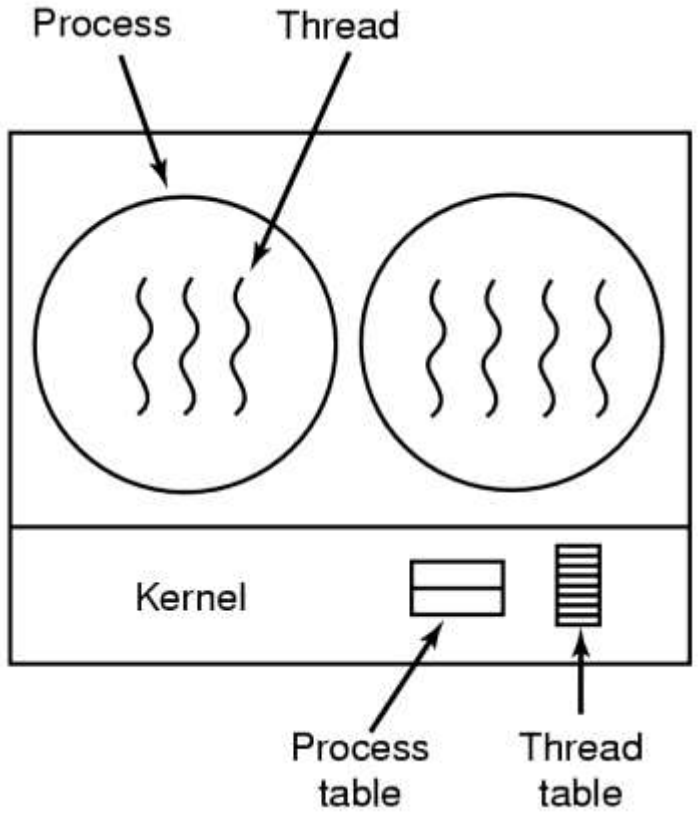
جلسه‌ی قبل، ادامه‌ی ریسمان، شروع همزمانی

Common Thread Strategies

- Manager/worker
  - *Manager thread handles I/O*
  - *Magaer assigns work to worker threads*
  - *Worker threads created dynamically*
  - *... or allocated from a thread-pool*
- Pipeline
  - *Each thread handles a different stage of an assembly line*
  - *Threads hand work off to each other in a producer-consumer relationship*

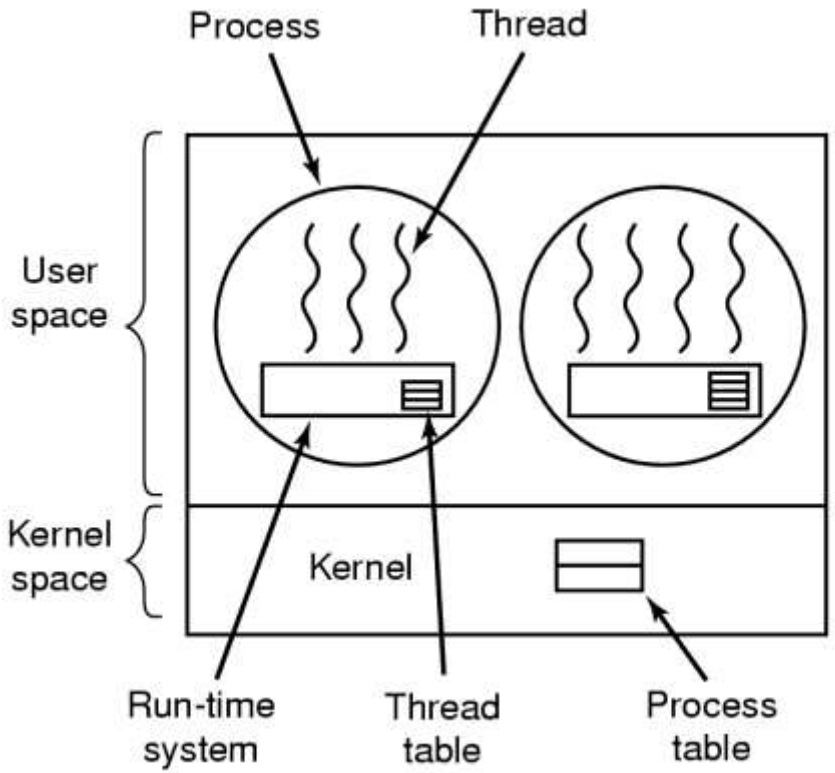
Kernel-Level Threads

Thread-switching
code is in the kernel



User-Level Threads

The thread-switching code is in user space



Concurrency Vs. Parallelism

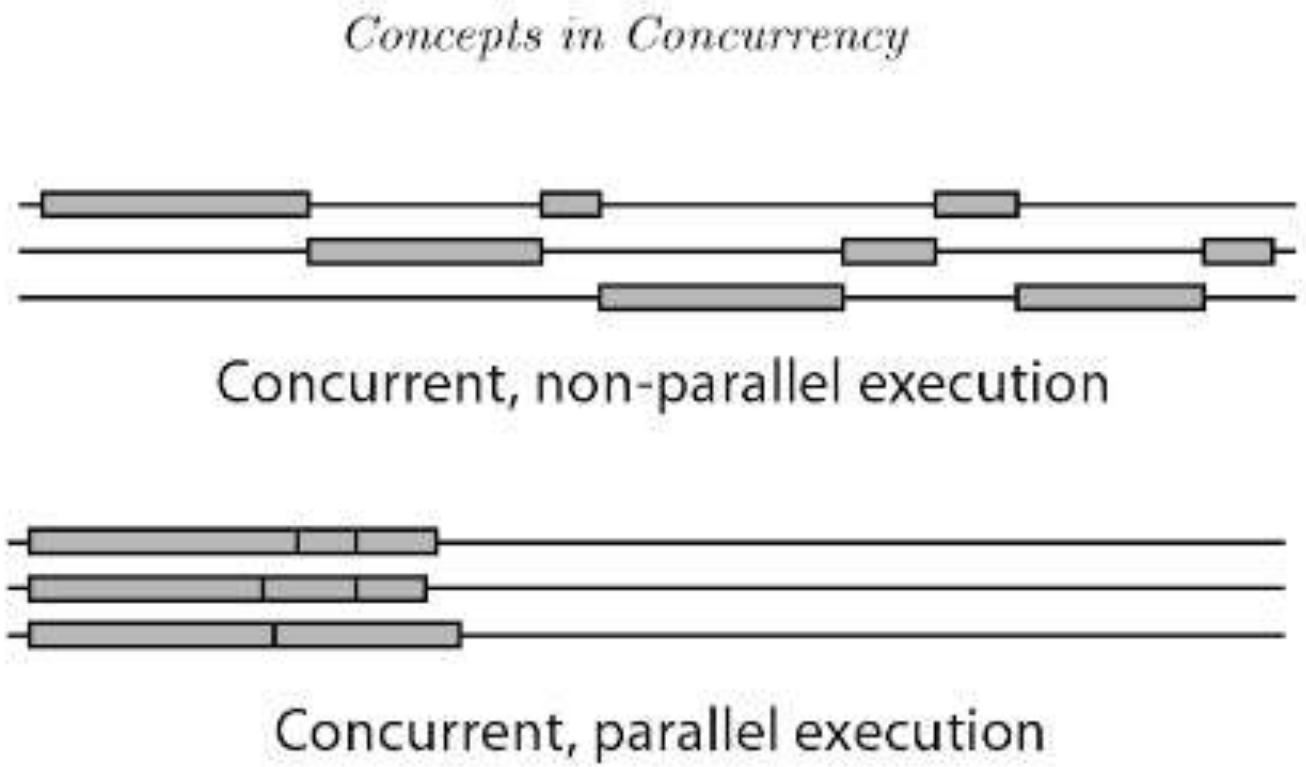


Image from: https://stackoverflow.com/a/53215726

Race Conditions

- A simple multithreaded program with a race:

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

Race Conditions

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

Critical Section

a part of a program where shared resources (like variables or memory) are accessed and modified

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

Critical Section

Mutual Exclusion Conditions

- **Mutual exclusion.** No two processes simultaneously in critical section
- **Progress.** No process running outside its critical section may block another process
- **Bounded waiting.** No process waits forever to enter its critical section

جلسه‌ی جدید

همزمانی

هدف این جلسه

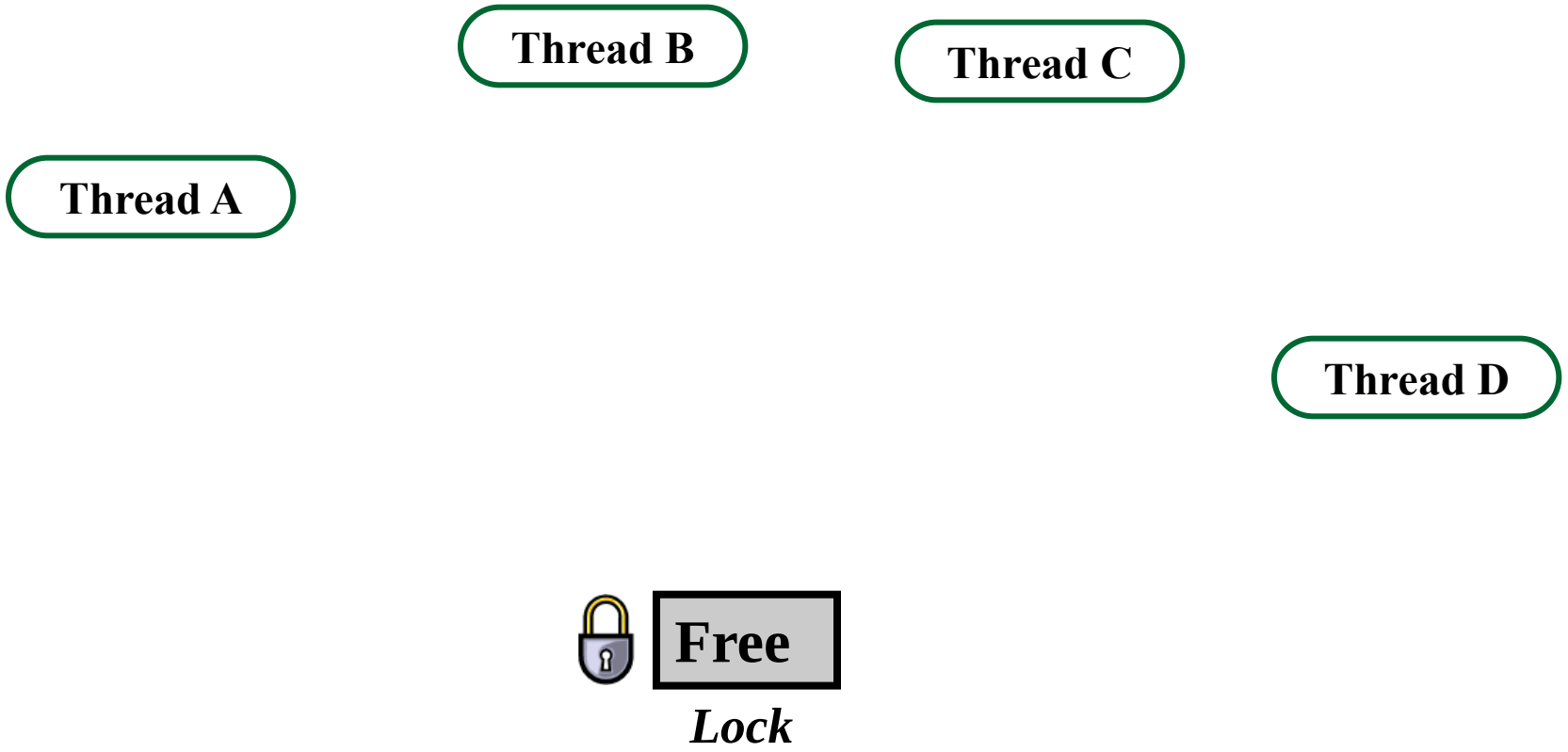
- بررسی راه حل‌های حل مسئله‌ی Mutual Exclusion
- راه حل‌های پیاده‌سازی Mutex
- معرفی Semaphore

MUTEX

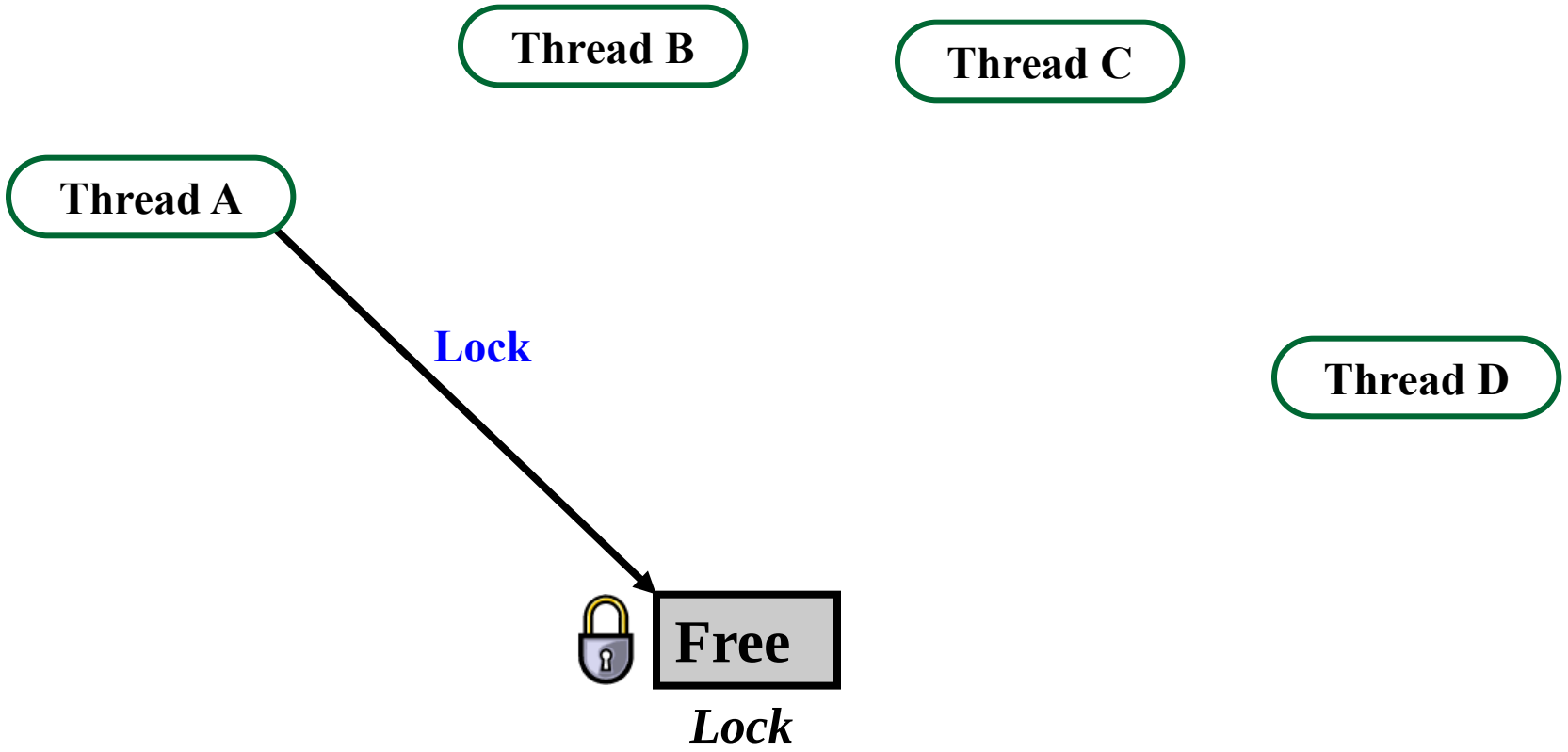
Enforcing Mutual Exclusion

- 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*

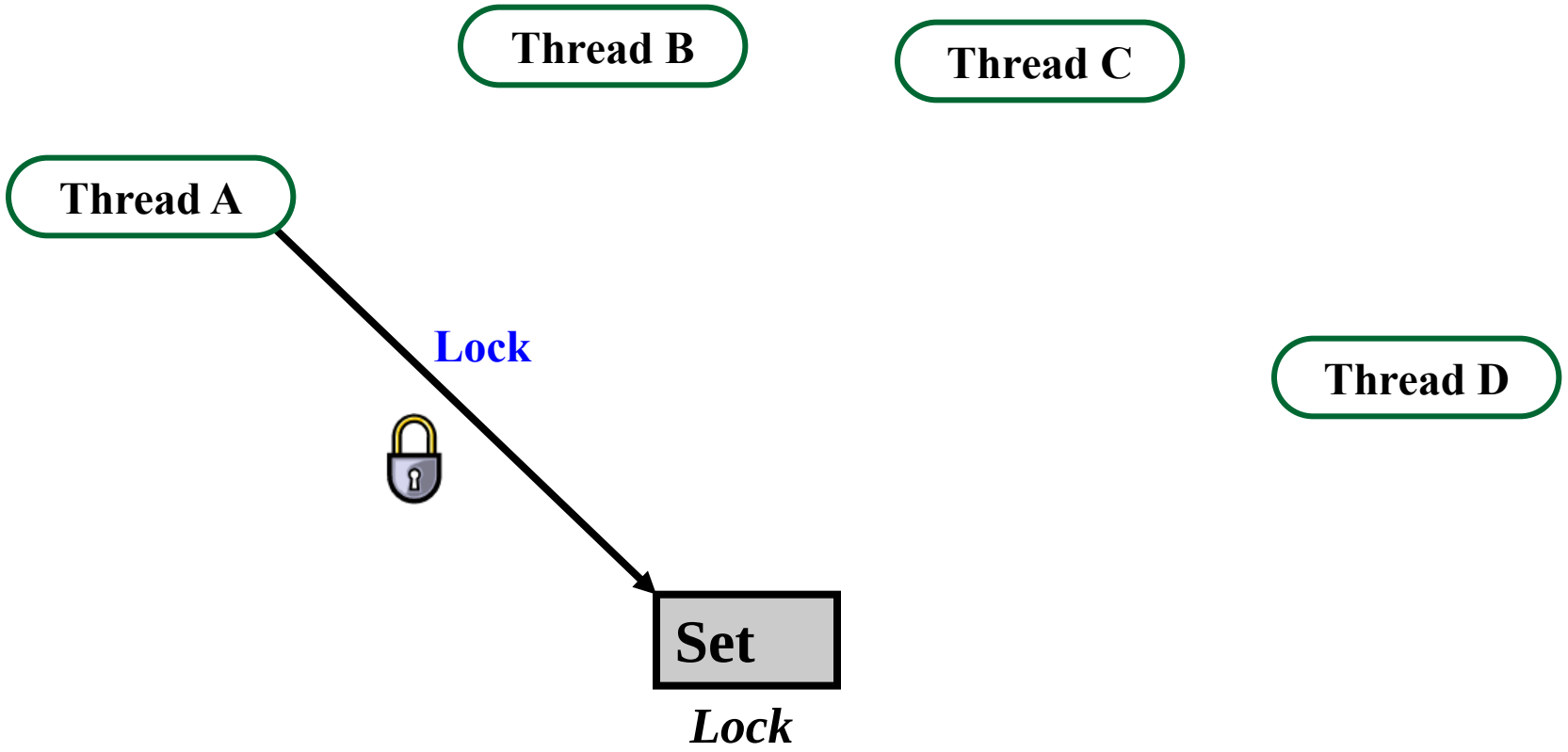
Acquiring and Releasing Locks



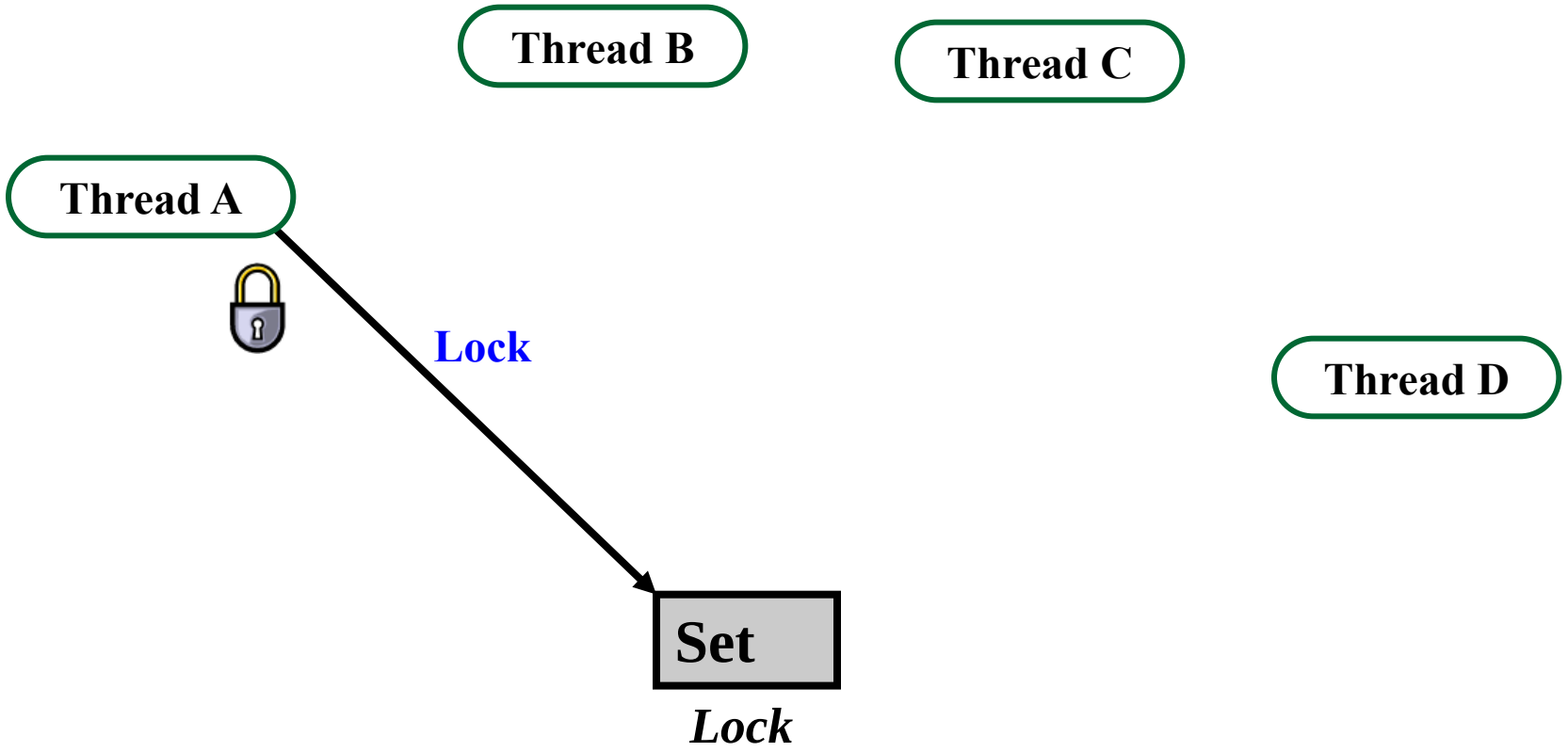
Acquiring and Releasing Locks



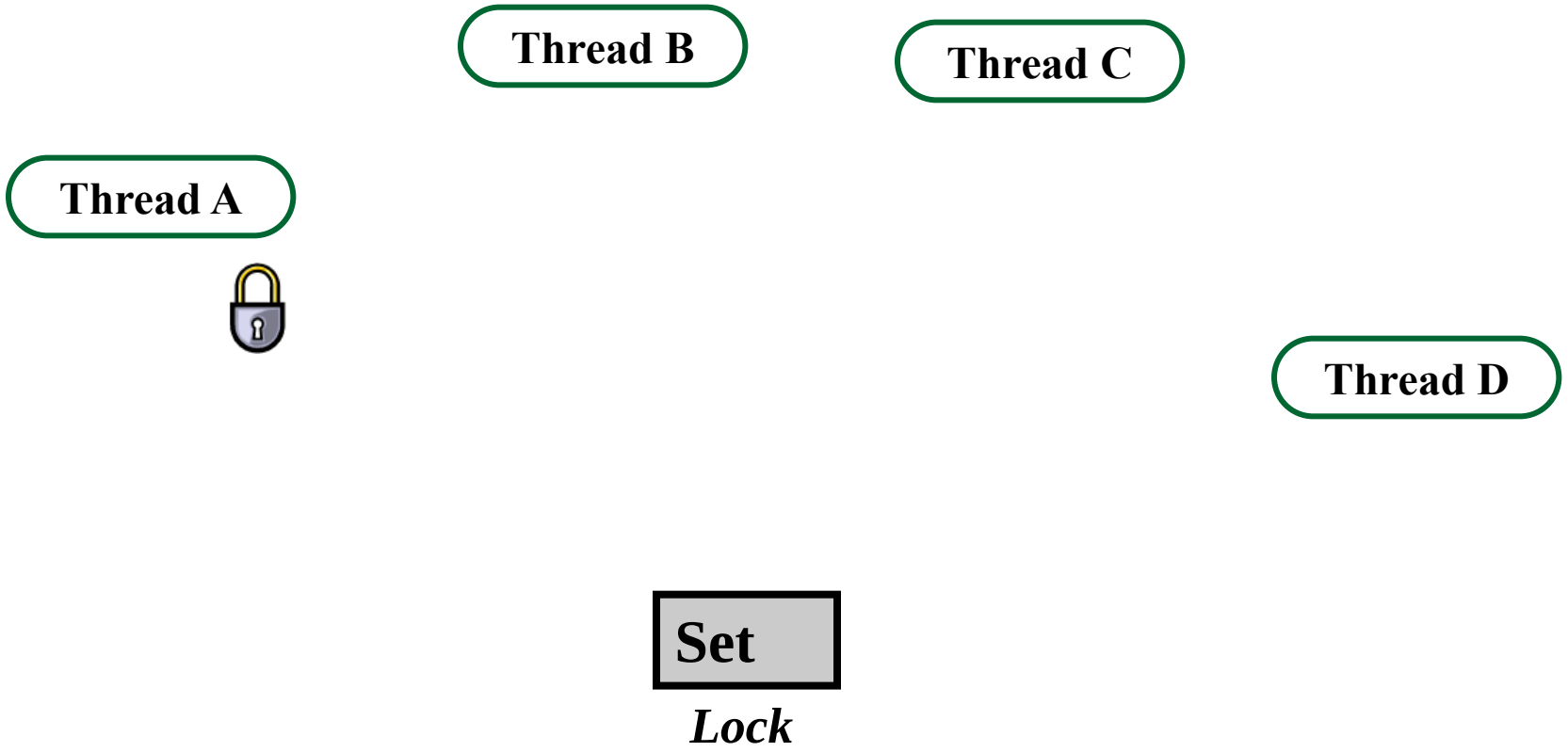
Acquiring and Releasing Locks



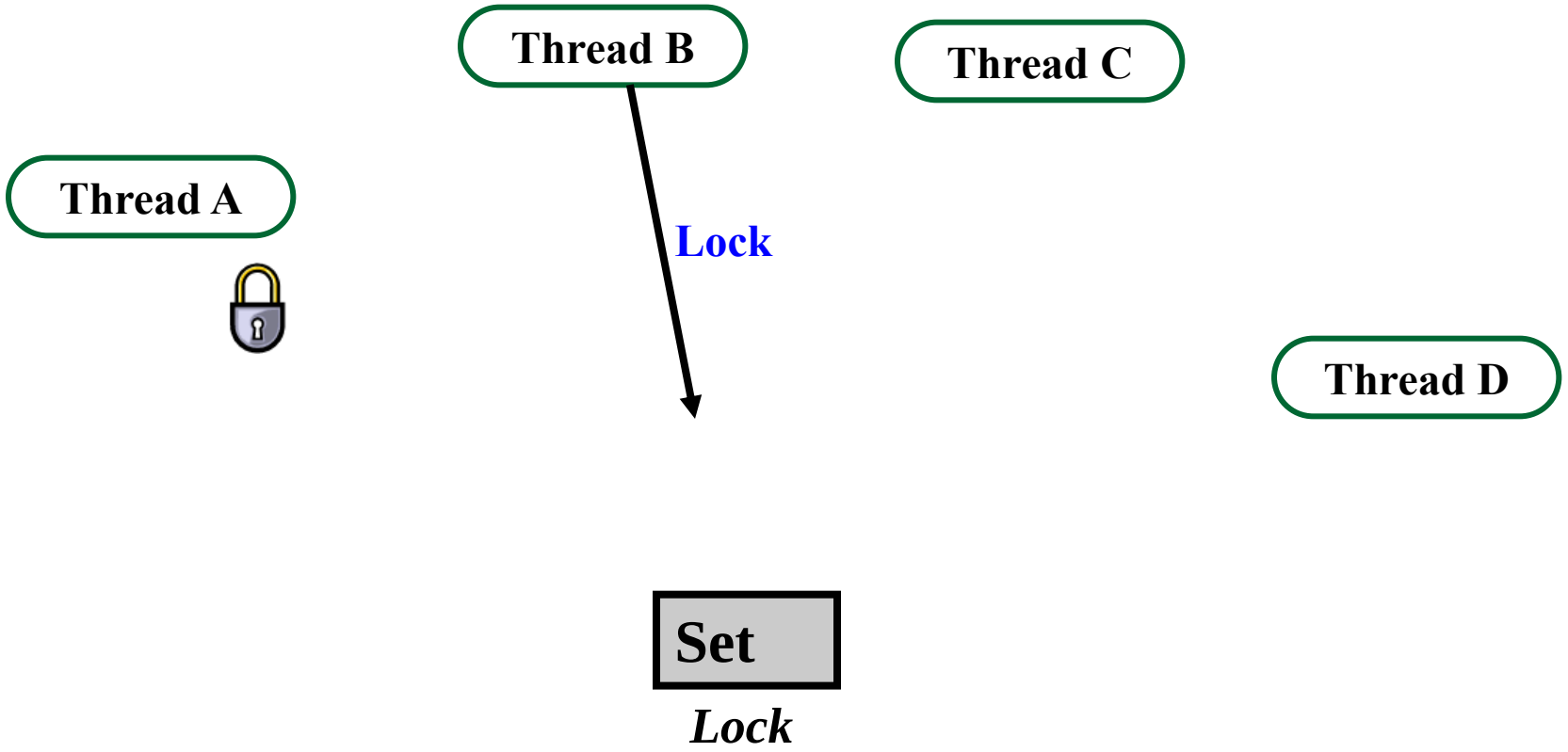
Acquiring and Releasing Locks



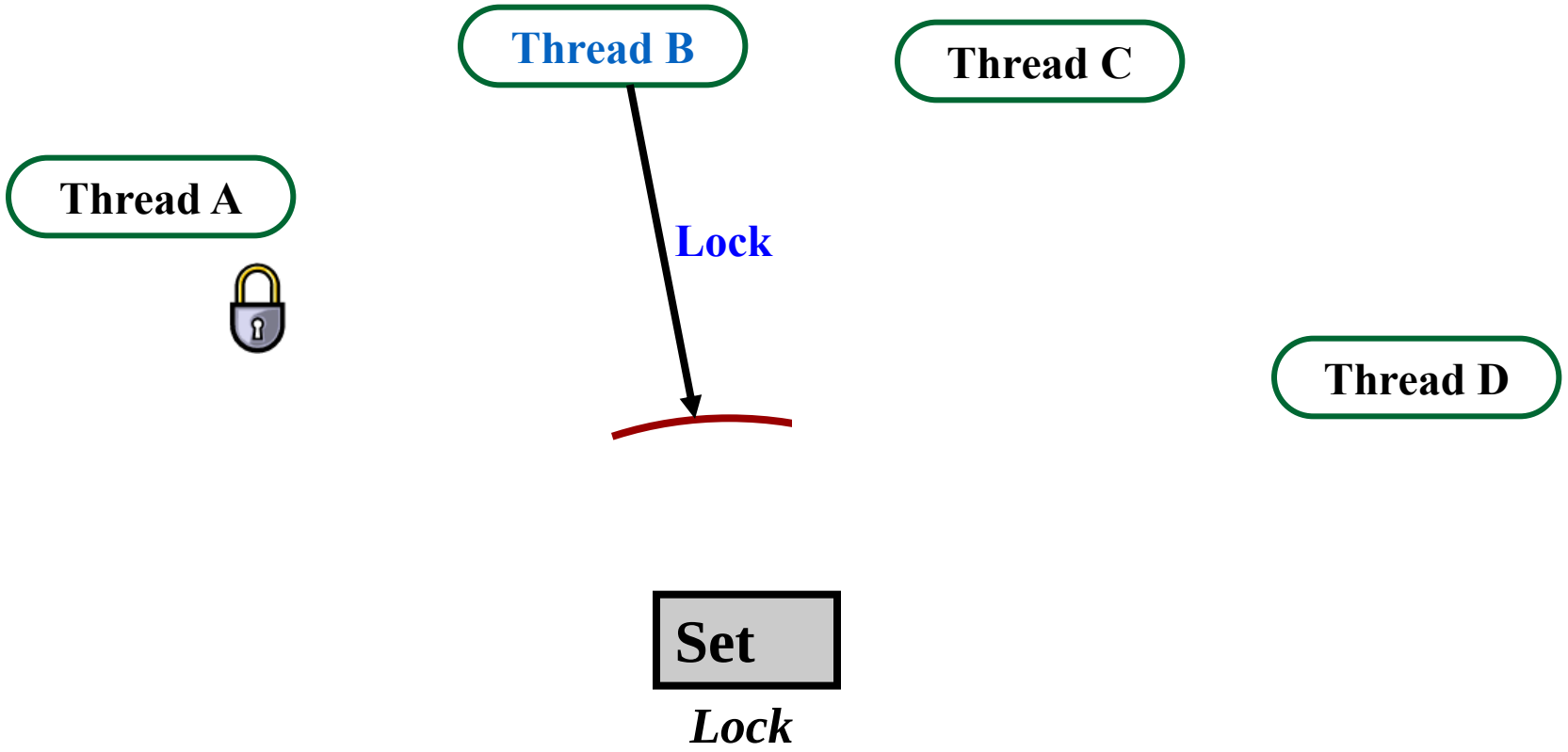
Acquiring and Releasing Locks



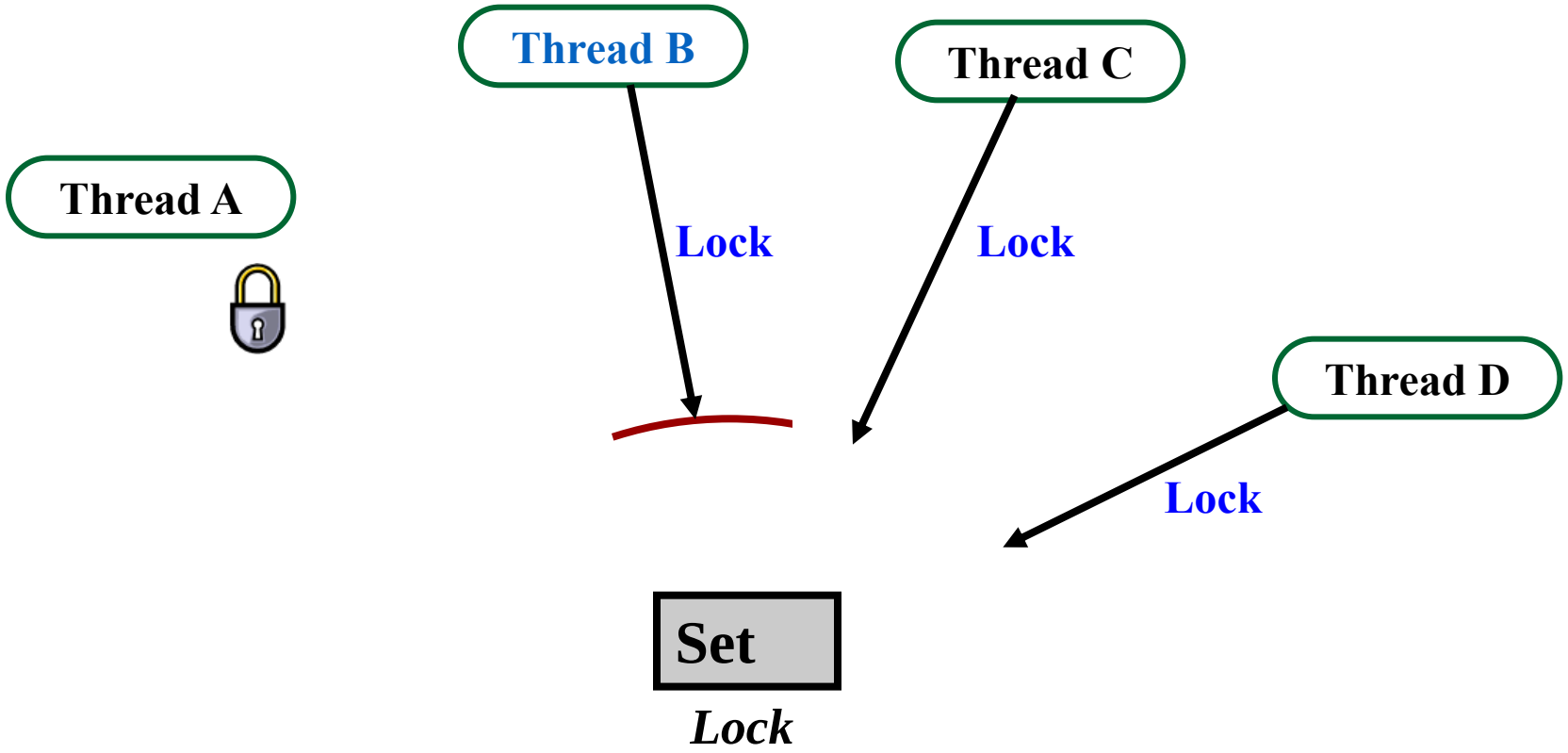
Acquiring and Releasing Locks



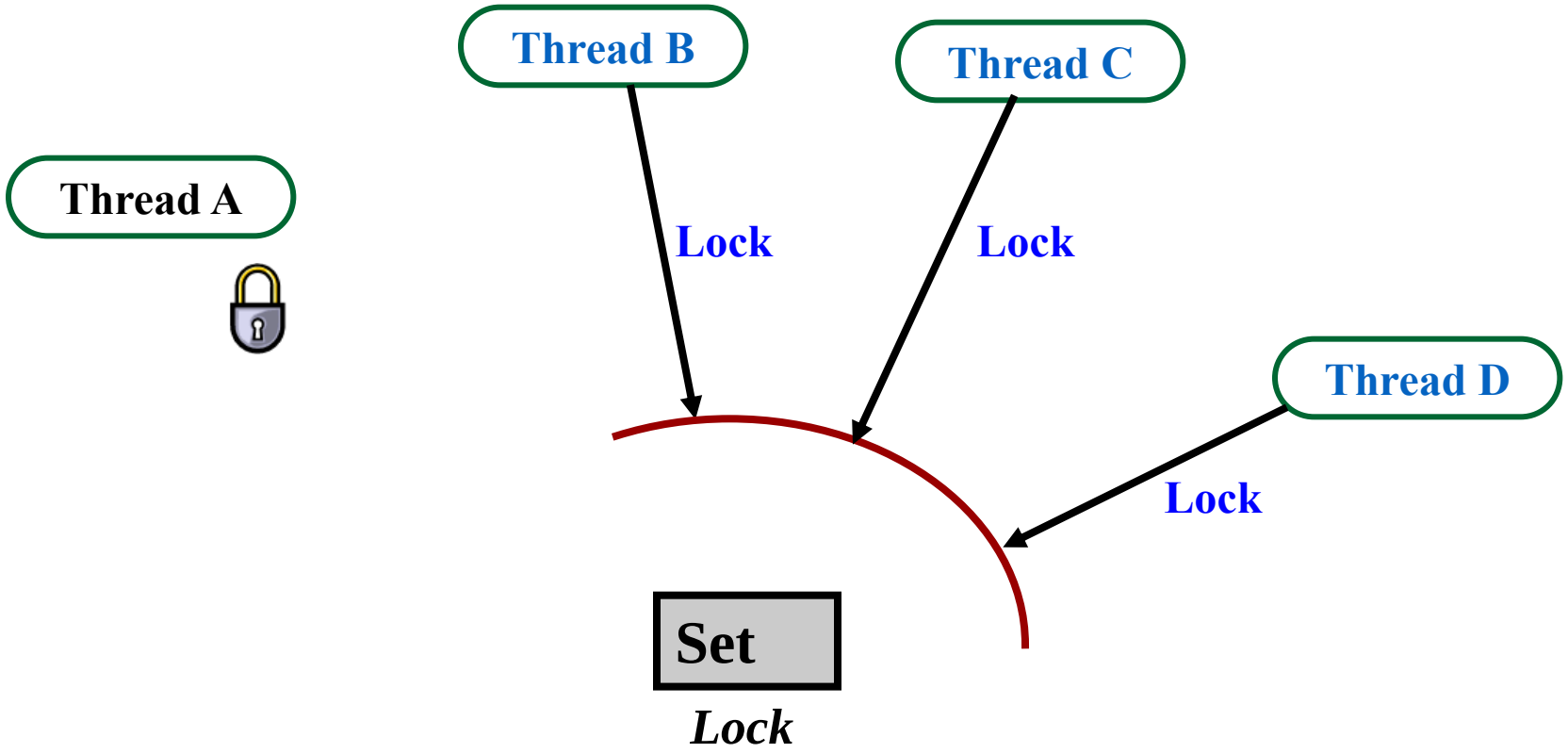
Acquiring and Releasing Locks



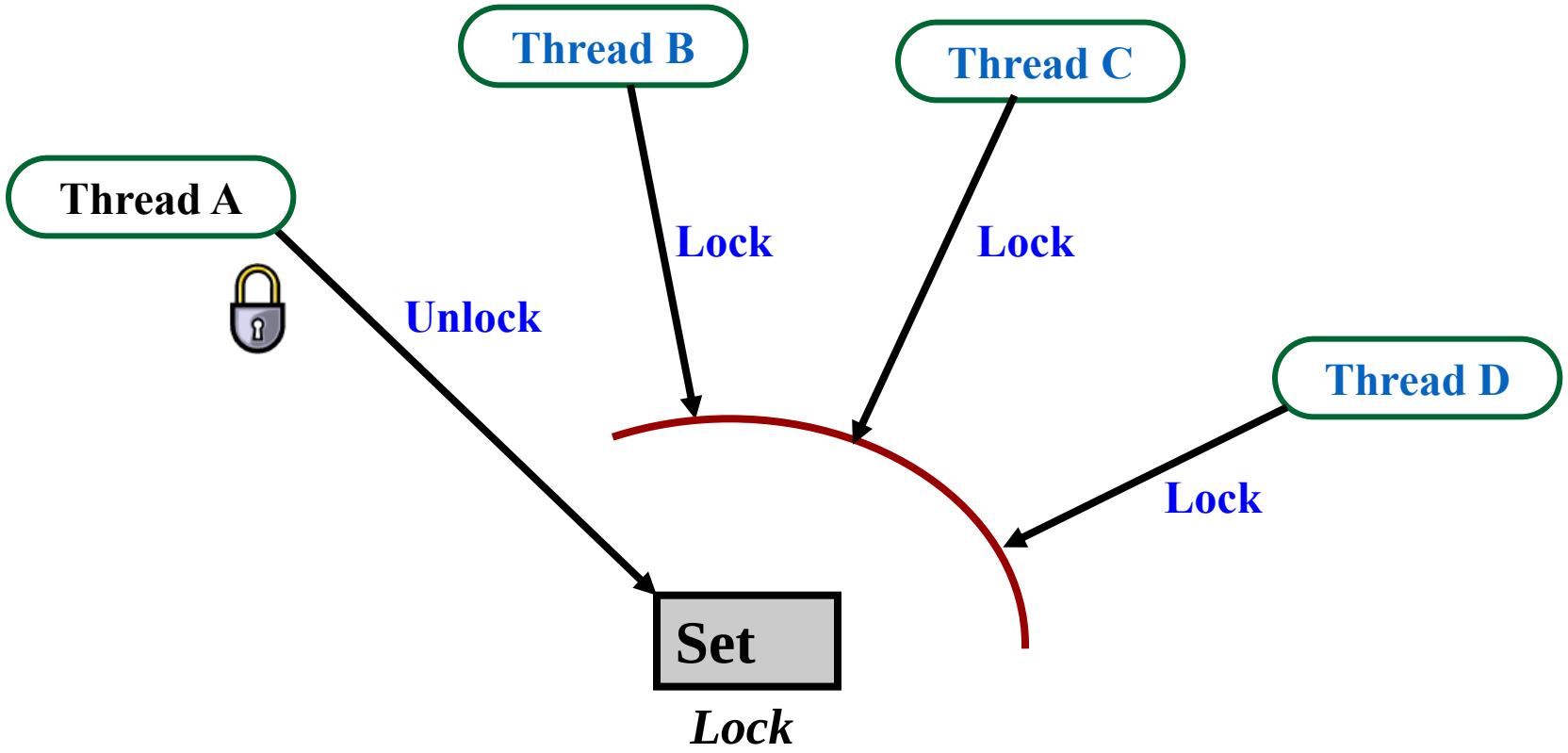
Acquiring and Releasing Locks



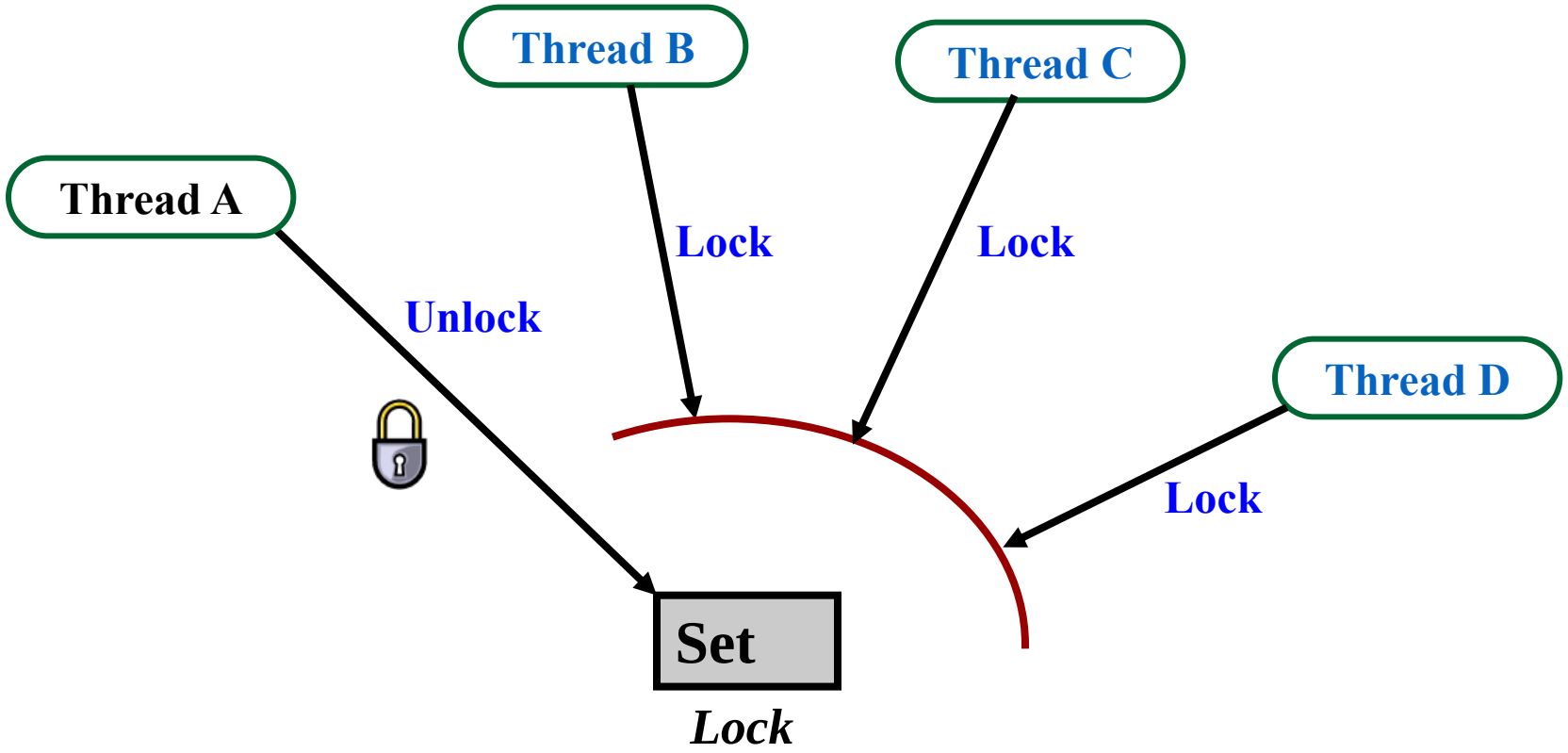
Acquiring and Releasing Locks



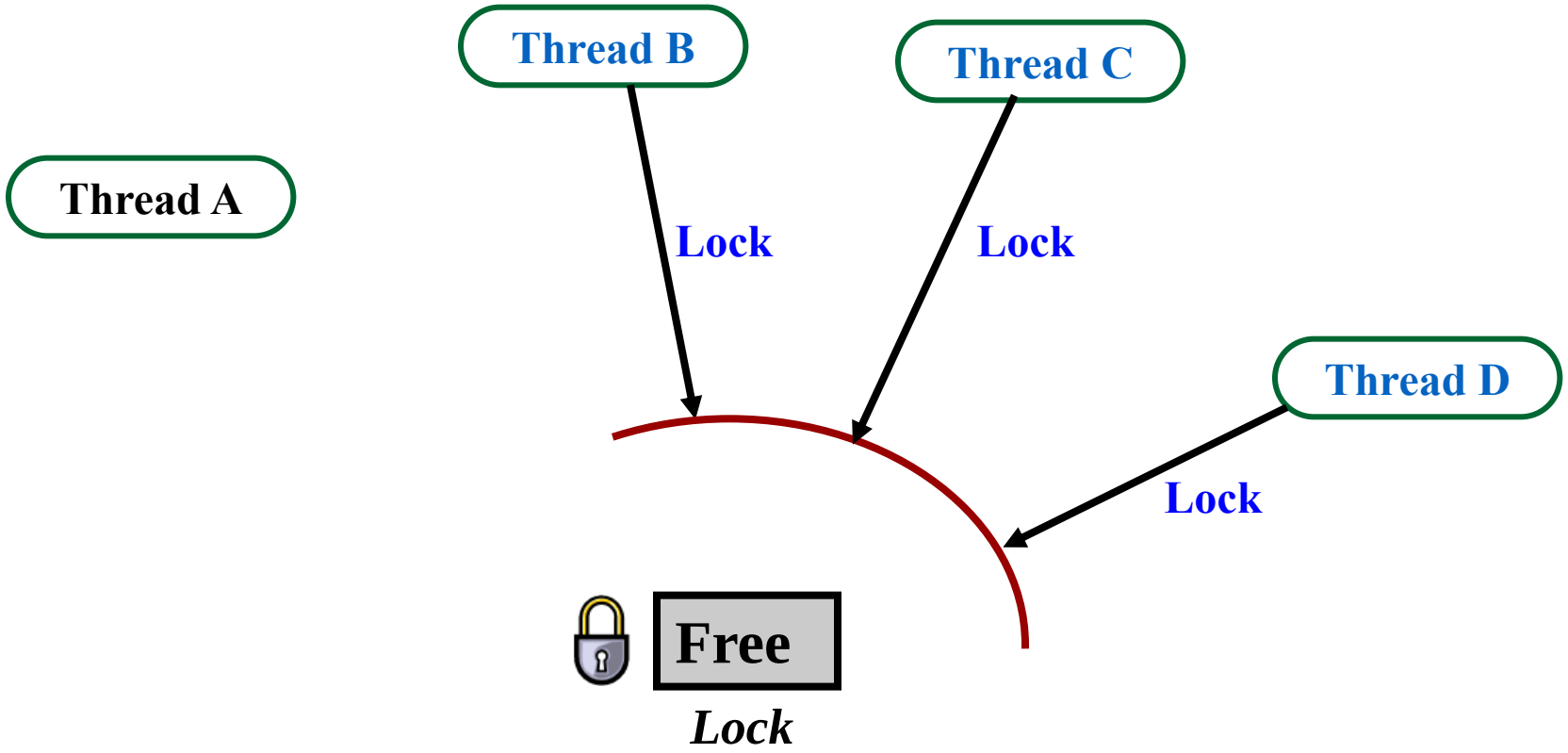
Acquiring and Releasing Locks



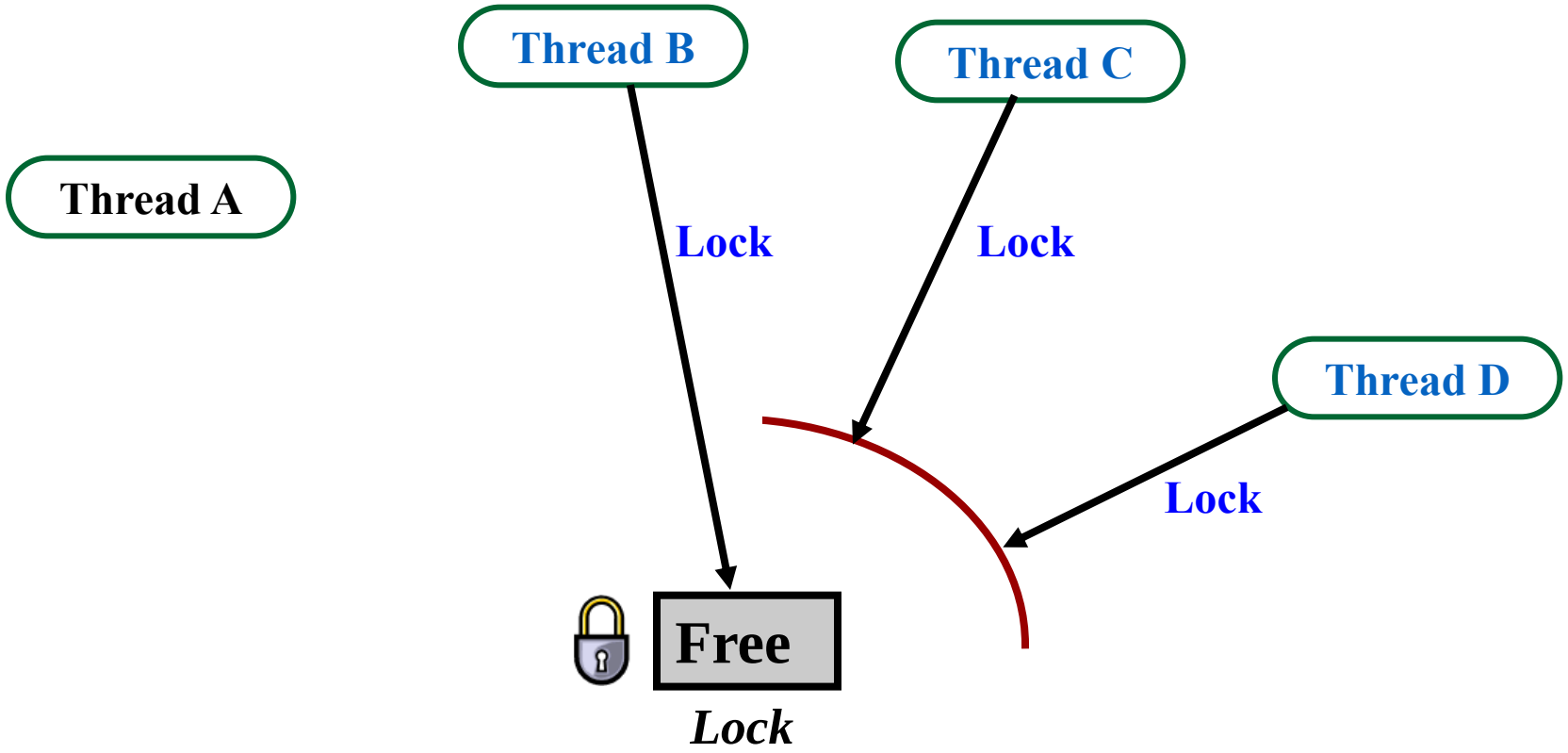
Acquiring and Releasing Locks



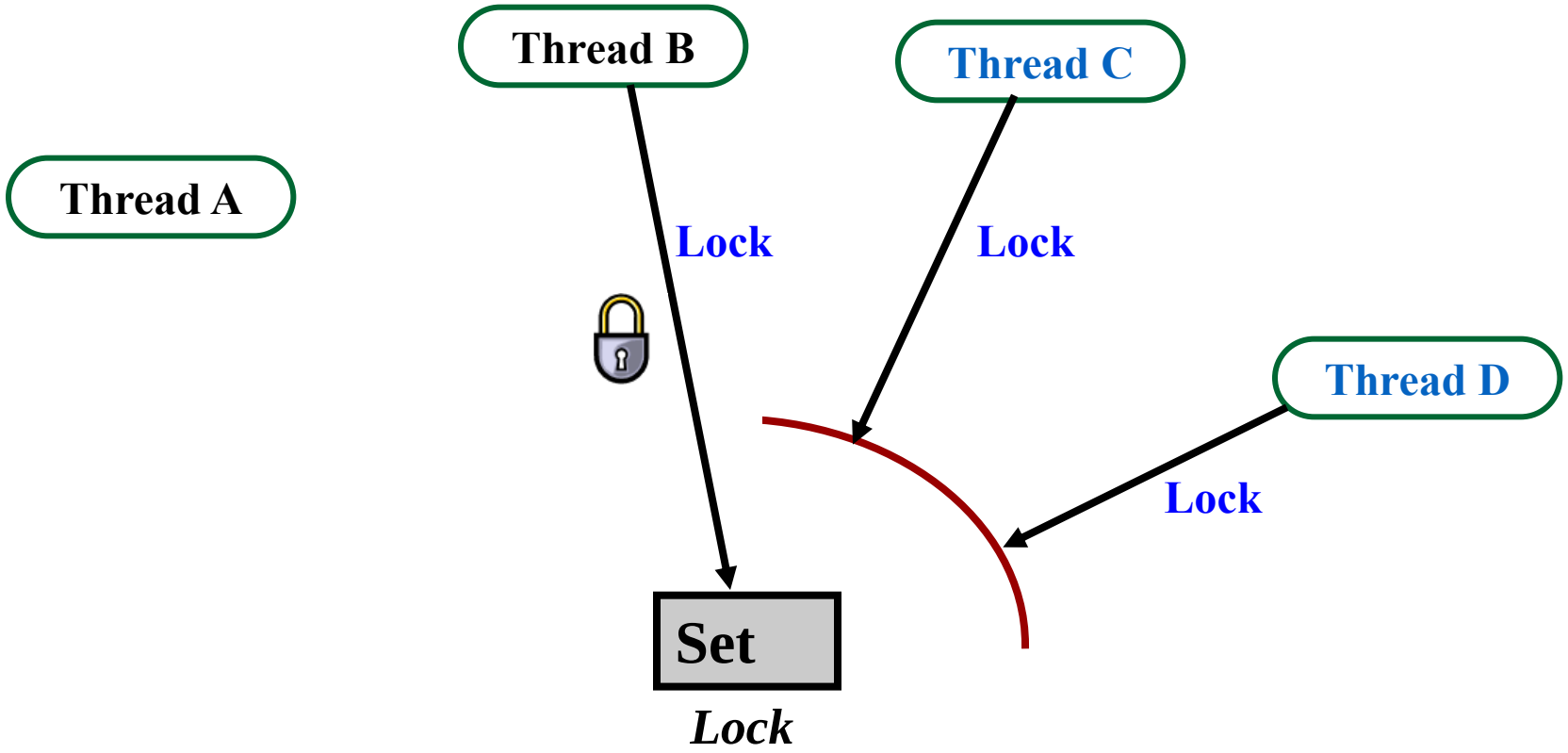
Acquiring and Releasing Locks



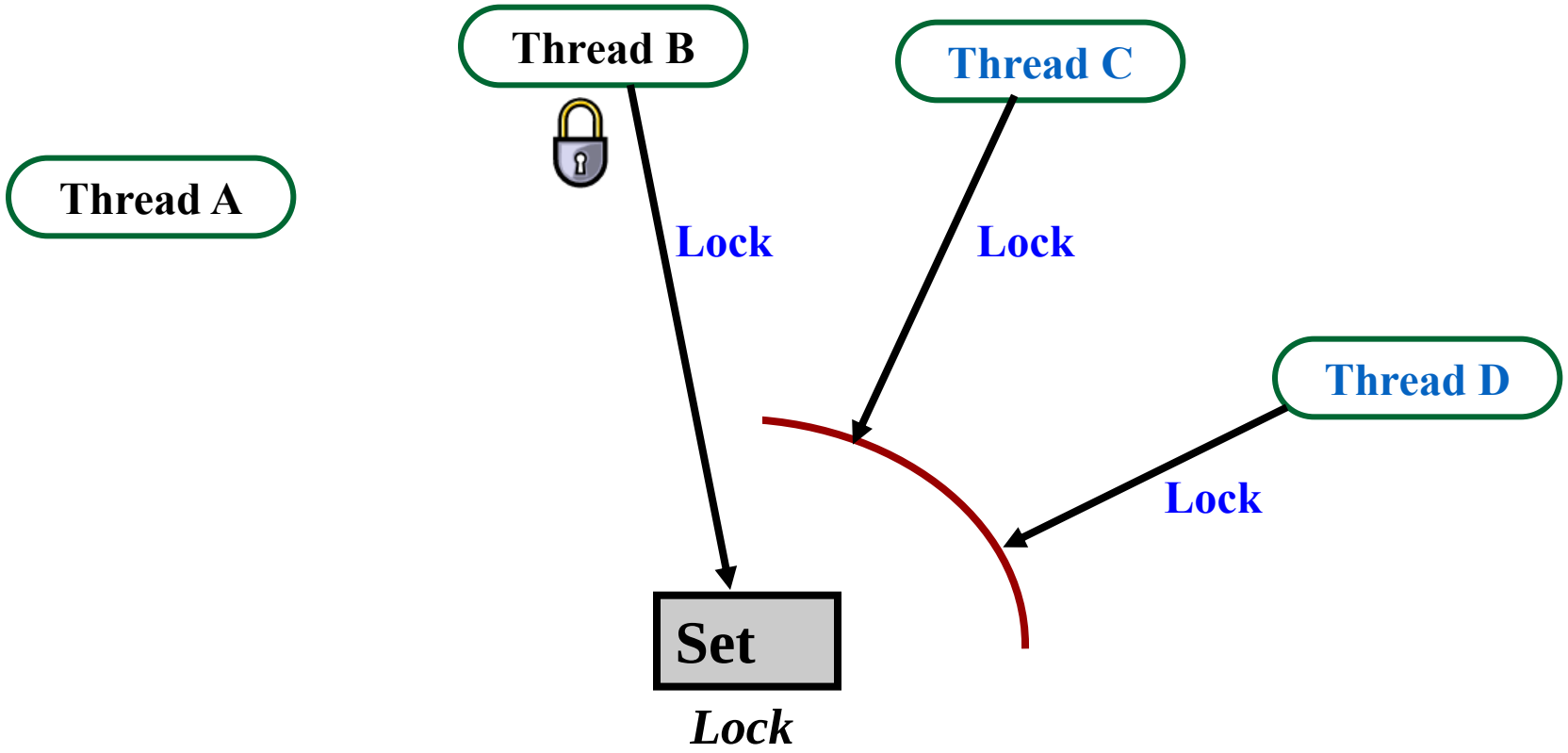
Acquiring and Releasing Locks



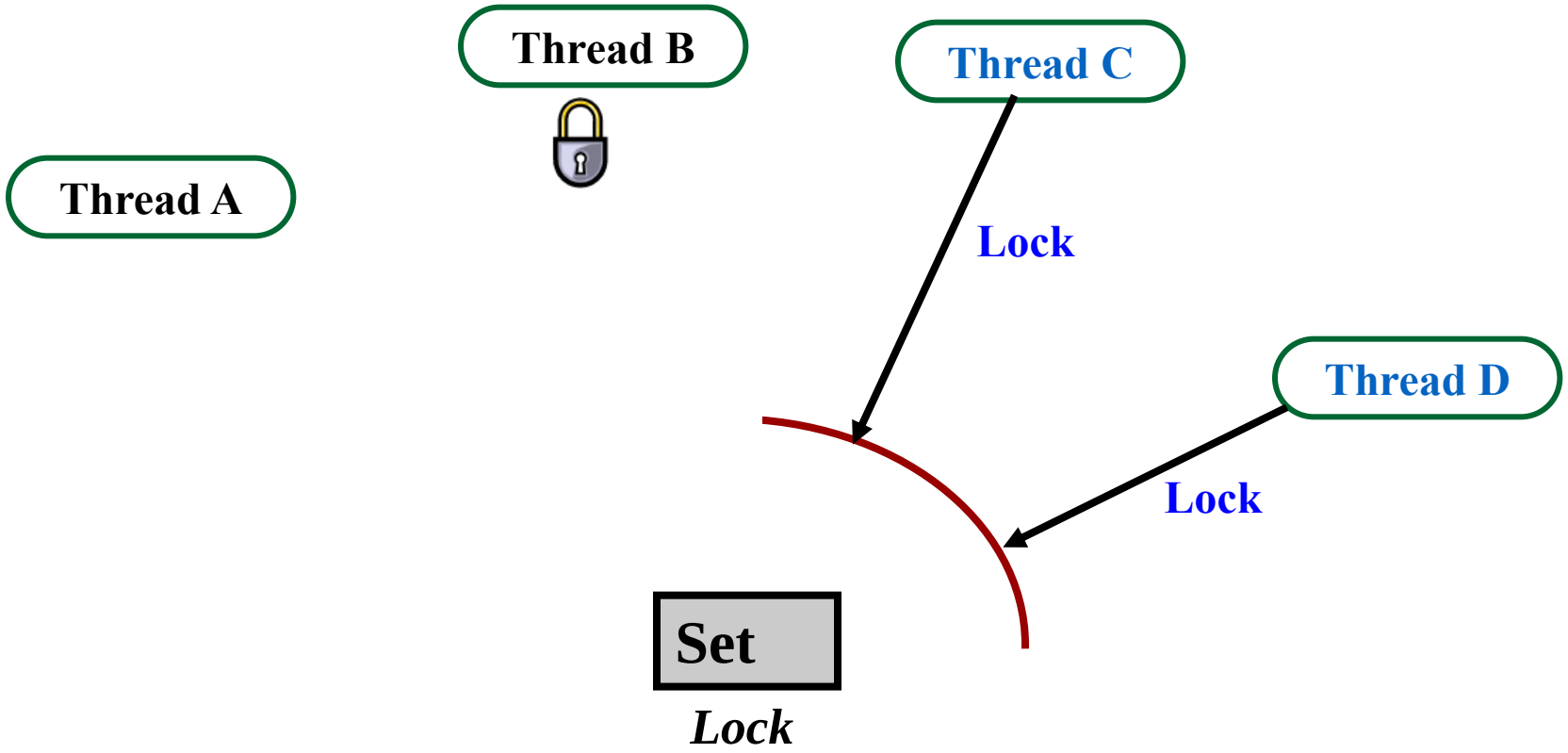
Acquiring and Releasing Locks



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 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?
- Lock and Unlock must be **atomic**

هدف:
پیاده‌سازی LOCK و UNLOCK برای MUTEX

ایده‌ی ۱

بدون هیچ کمکی از سیستم عامل یا سخت‌افزار – برای دو ریسمان

Software Solution 1

- Two thread solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two threads share one variable:
- int turn;
- The variable turn indicates whose turn it is to enter the critical section
- initially, the value of turn is set to i

Algorithm for thread *i*

```
while (true){

    while (turn = = j);

    /* critical section */

    turn = j;

    /* remainder section */

}
```

Lock:

Unlock:

Busy Wait

Algorithm for thread *j*

```
while (true){

    while (turn = = i);

    /* critical section */

    turn = i;

    /* remainder section */

}
```

Lock: `while (turn = = i);`

Unlock: `turn = i;`

Is Software Solution 1 Correct ?

- **Mutual exclusion**
  - 
- **Progress**
  - 
- **Bounded waiting**
  - 

ایده‌ی ۱ – شکست خورد ☹

بدون هیچ کمکی از سیستم عامل یا سخت‌افزار – برای دو ریسمان

ایده‌ی ۲

بدون هیچ کمکی از سیستم عامل یا سخت‌افزار – برای دو ریسمان

Software Solution 2

- Two thread solution
- Assume that the **load** and **store** machine-language instructions are **atomic**; that is, cannot be interrupted
- The two threads share two variables:
  - *int turn;*
  - *boolean flag[2]*
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a thread is ready to enter the critical section.
- flag[i] = true implies that thread Pi is ready!

Algorithm for Thread *i*

```
while (true){

    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

        /* critical section */

    flag[i] = false;

     /* remainder section */
}
```

Routine of two Threads

```
while (true){

    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

        /* critical section */

    flag[i] = false;

    /* remainder section */
}
```

```
while (true){

    flag[j] = true;
    turn = i;
    while (flag[i] && turn = = i);

        /* critical section */

    flag[j] = false;

    /* remainder section */
}
```

Routine of two Threads 1

```
while (true){

    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

        /* critical section */

    flag[i] = false;

    /* remainder section */
}
```

```
while (true){

    flag[j] = true;
    turn = i;
    while (flag[i] && turn = = i);

        /* critical section */

    flag[j] = false;

    /* remainder section */
}
```

Routine of two Threads 2

```
while (true){

    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

       /* critical section */

    flag[i] = false;

    /* remainder section */
}
```

```
while (true){

    flag[j] = true;
    turn = i;
    while (flag[i] && turn = = i);

       /* critical section */

    flag[j] = false;

    /* remainder section */
}
```

Routine of two Threads 3

```
while (true){

    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

       /* critical section */

    flag[i] = false;

    /* remainder section */
}
```

```
while (true){

    flag[j] = true;
    turn = i;
    while (flag[i] && turn = = i);

       /* critical section */

    flag[j] = false;

    /* remainder section */
}
```

Is Software Solution 2 Correct ?

- **Mutual exclusion**
  - ✅
- **Progress**
  - ✅
- **Bounded waiting**
  - ✅

Peterson’s Solution

- Software Solution 2 = Peterson’s Solution

کمک گرفتن از CPU

دو Instruction در CPU

- Test and Set
  - *TSET [R1],R2*
- Compare and Swap

The test_and_set Instruction

```
boolean test_and_set (boolean *target)
   {
          boolean rv = *target;
          *target = true;
          return rv:
   }
```

The compare_and_swap Instruction

```
int compare_and_swap(int *value, int expected, int new_value)
  {
   int temp = *value;
    if (*value == expected)
        *value = new_value;
     return temp;
 }
```

TEST AND LOCK INSTRUCTION

Test-and-Set-Lock (TSL) Instruction

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

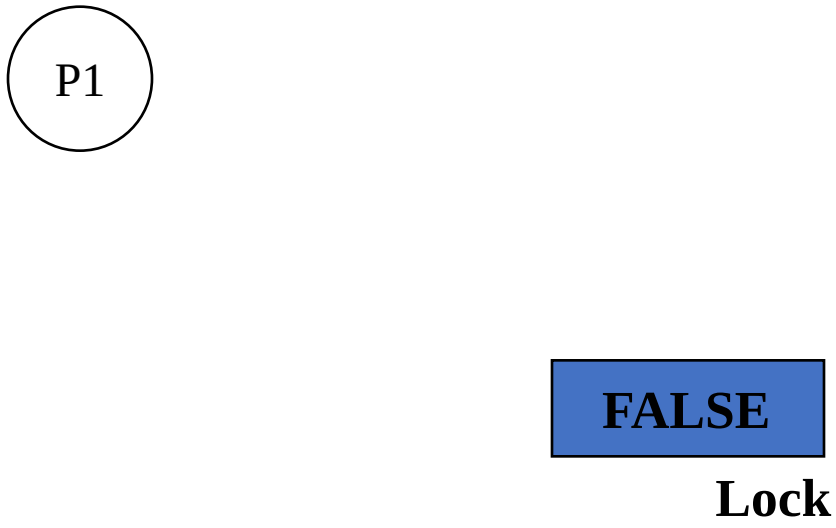
Test-and-Set-Lock (TSL) Instruction

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

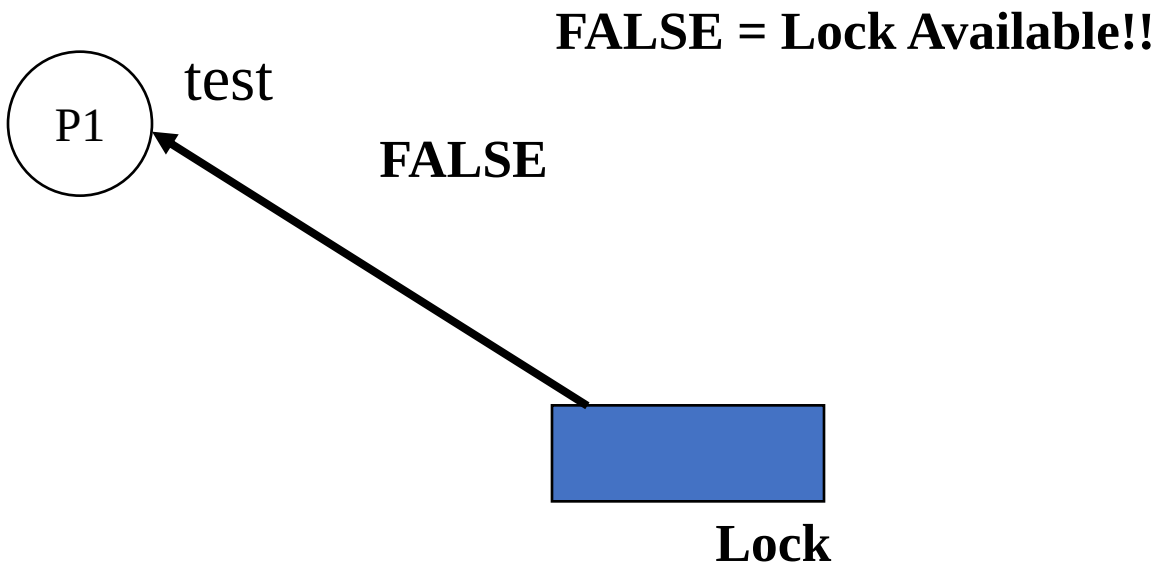
***If** the returned value was FALSE...*
- Then you got the lock!!!

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

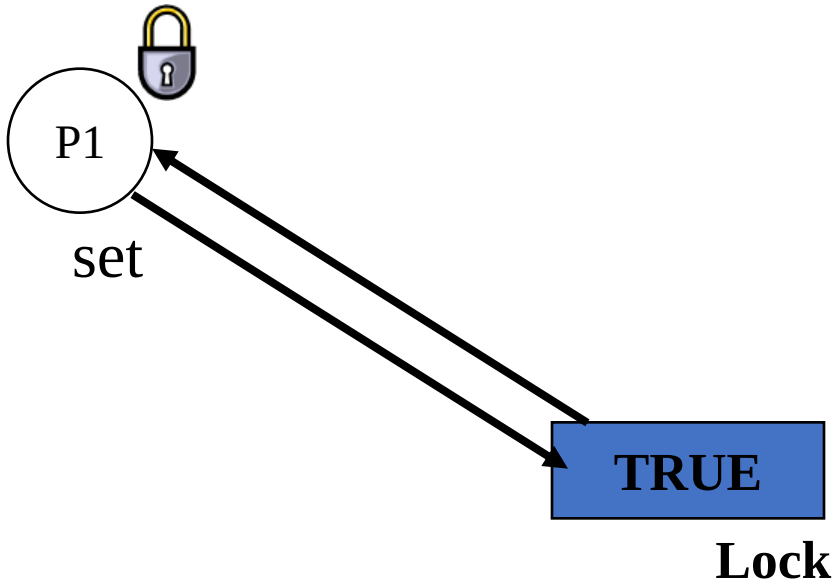
Test and Set Lock



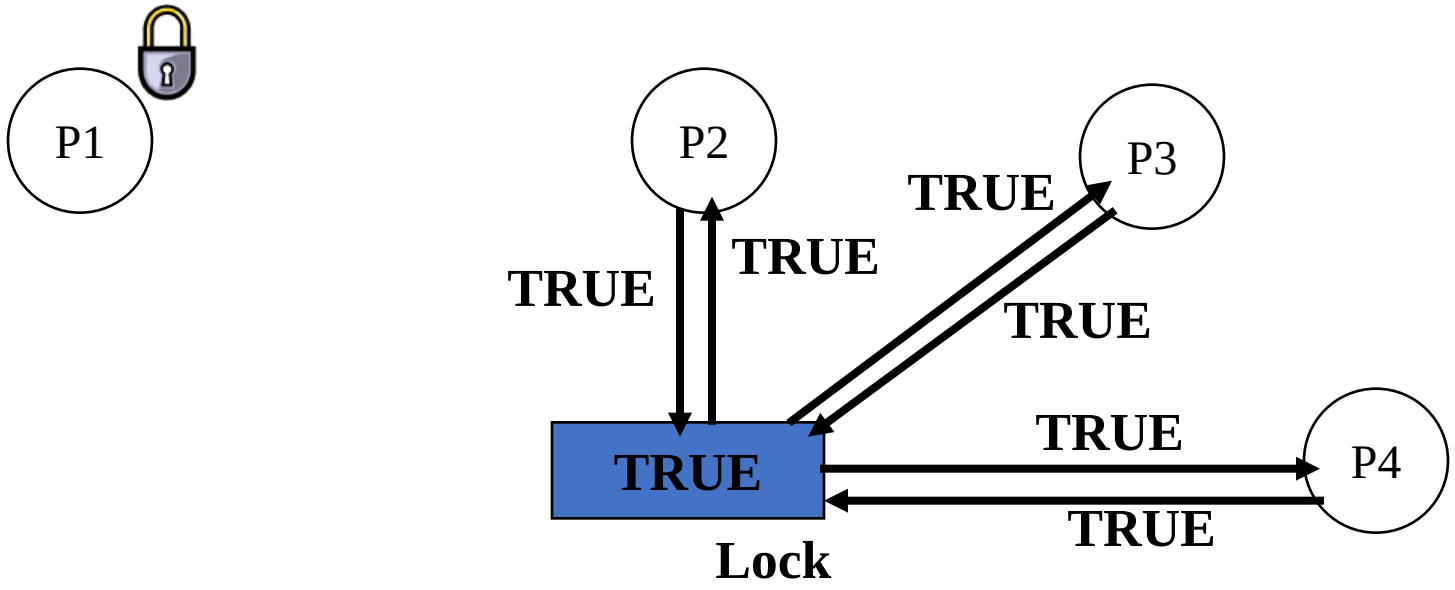
Test and Set Lock



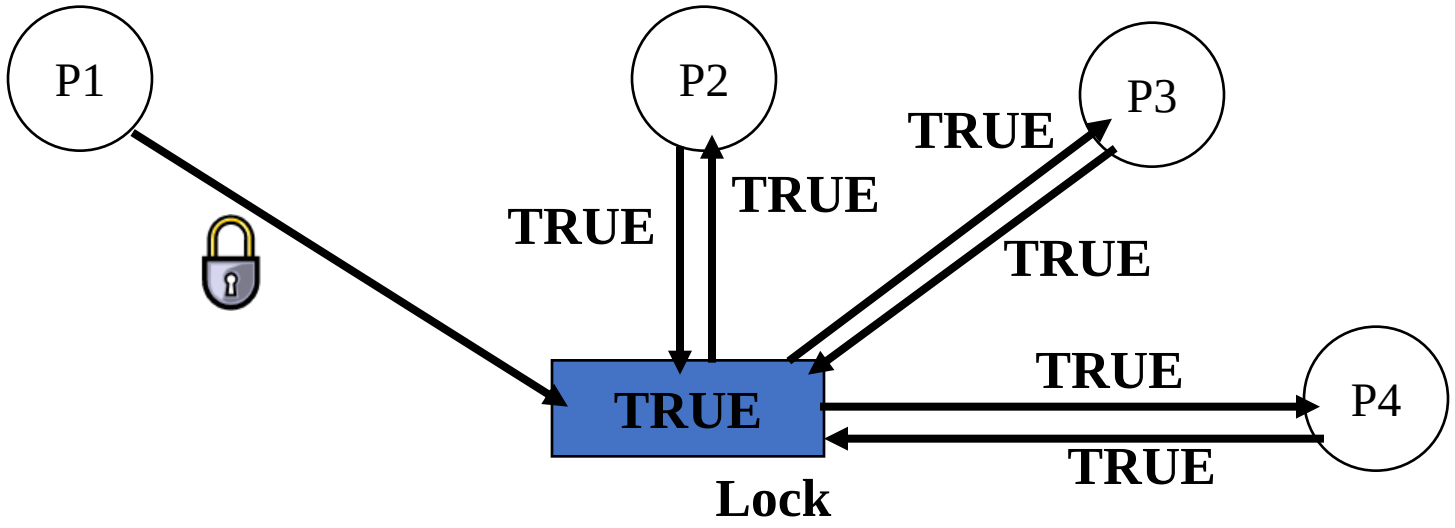
Test and Set Lock



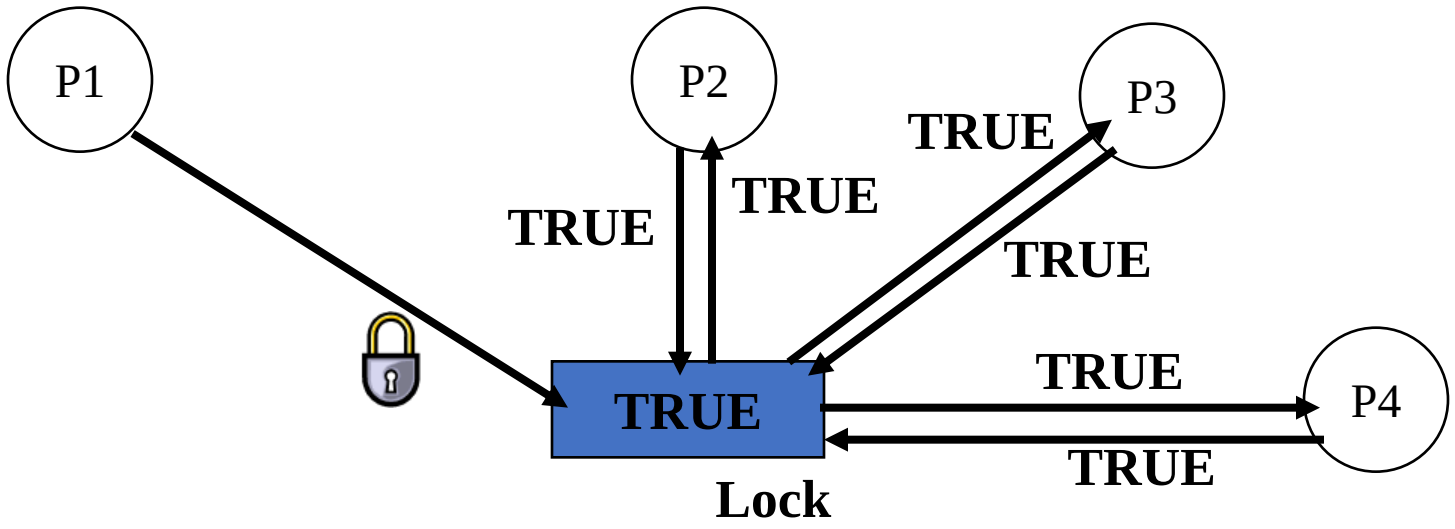
Test and Set Lock



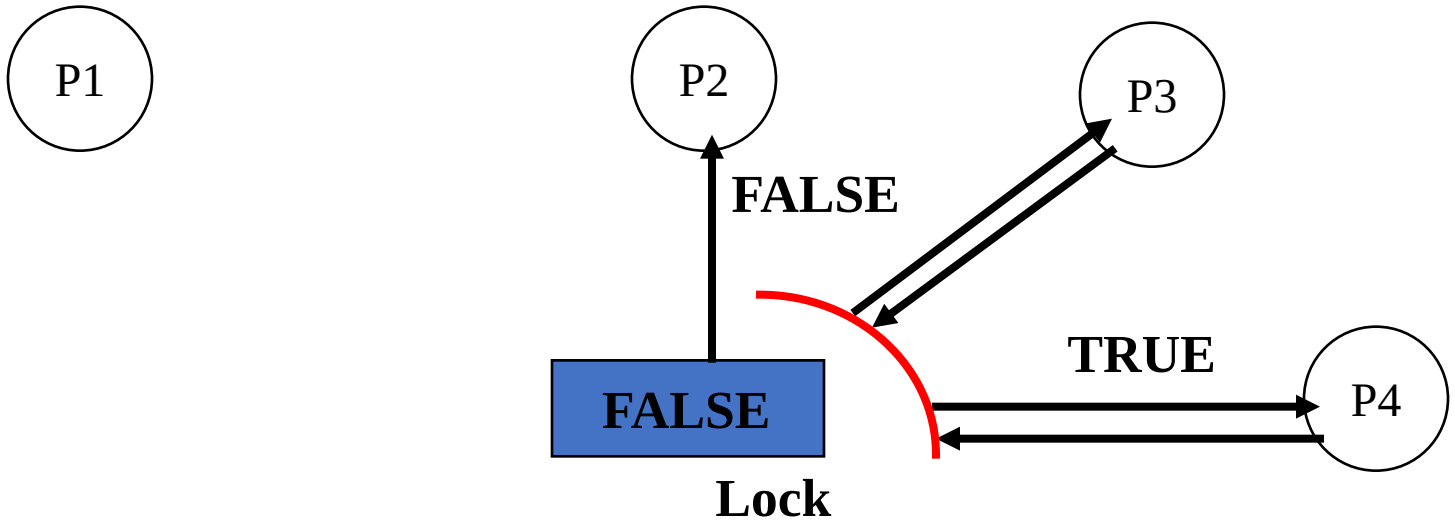
Test and Set Lock



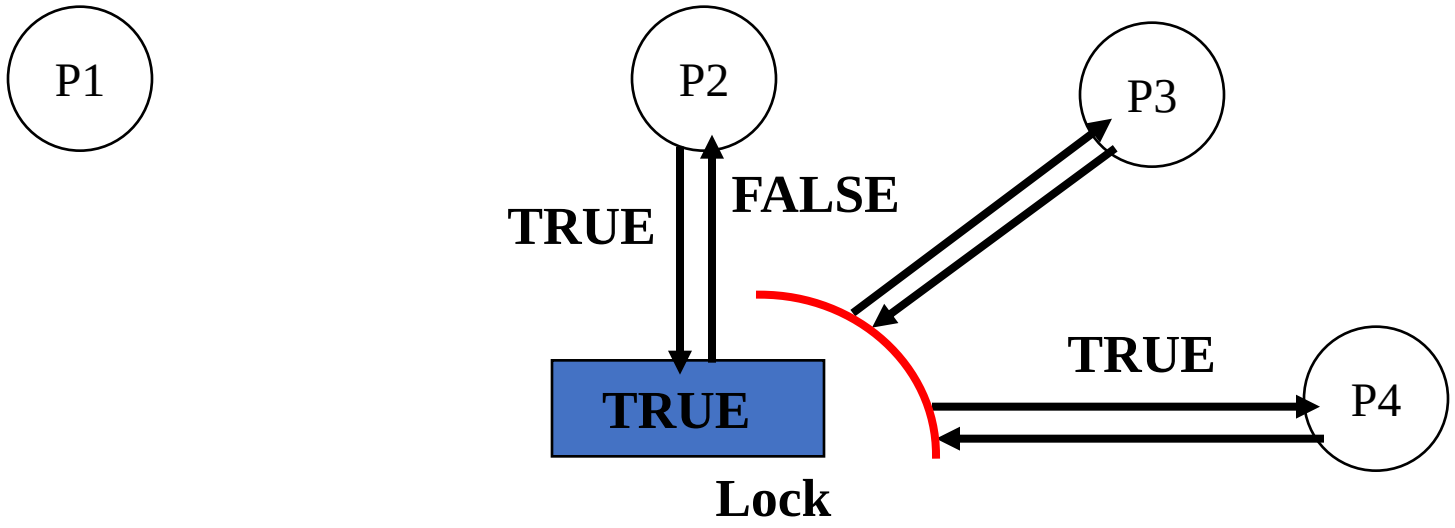
Test and Set Lock



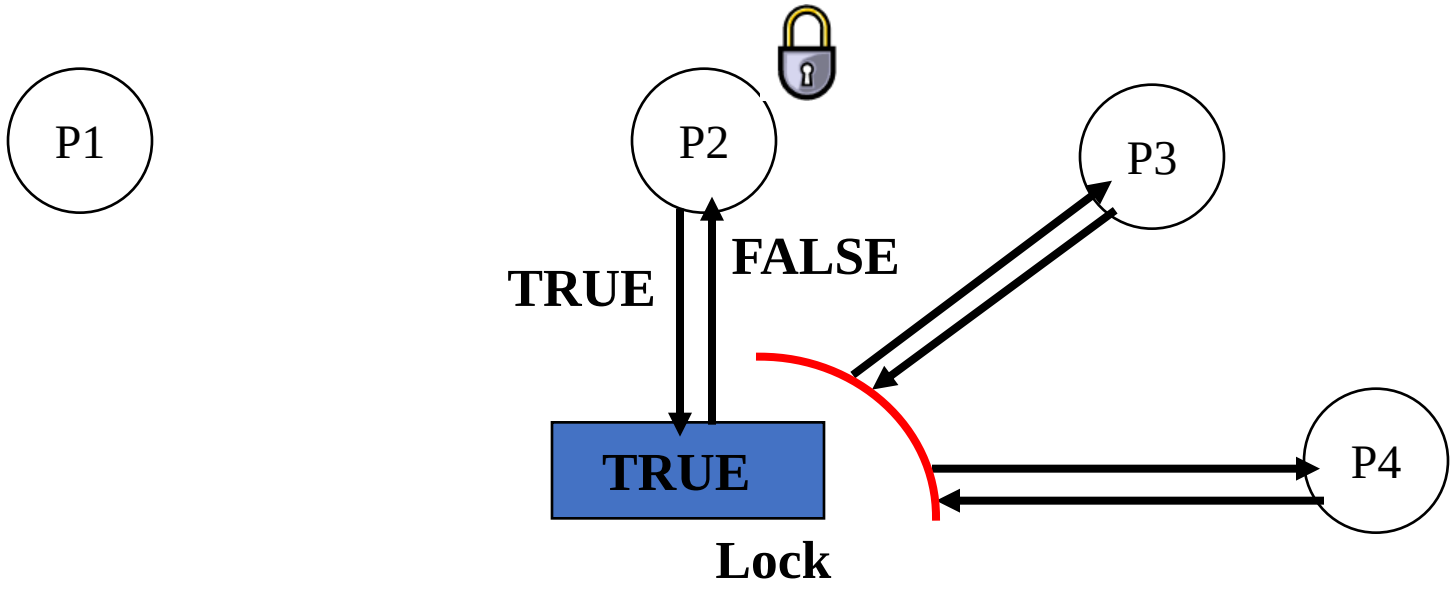
Test and Set Lock



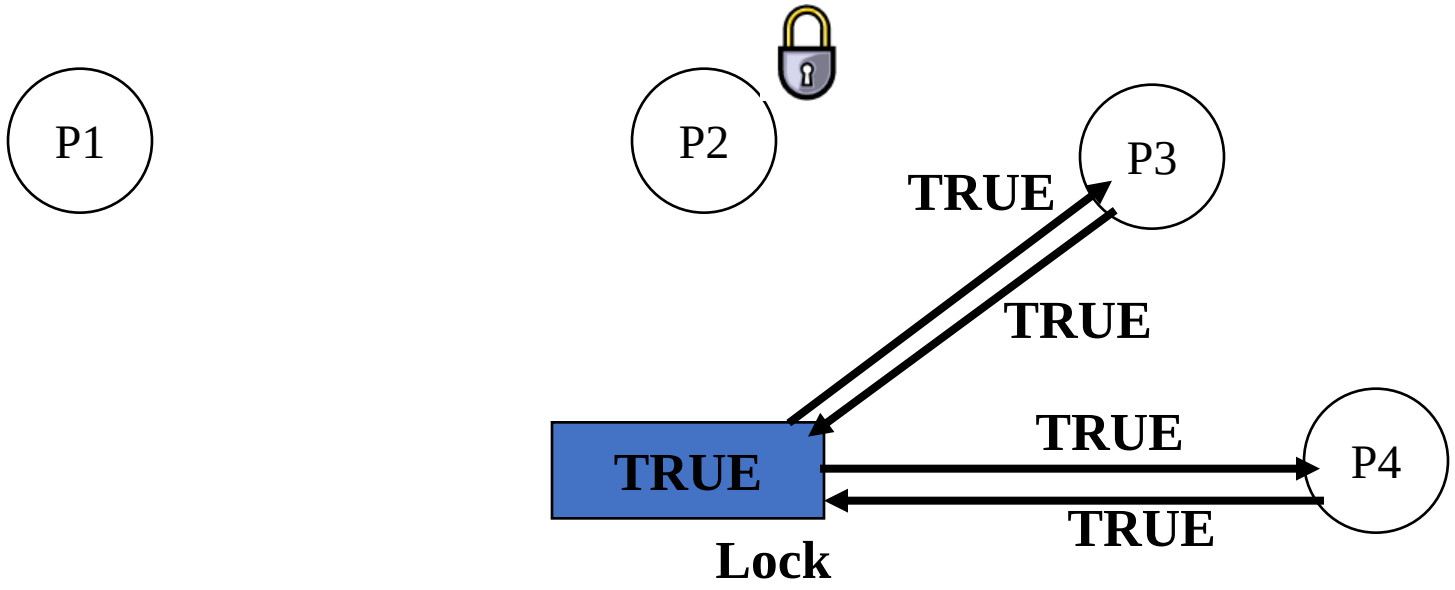
Test and Set Lock



Test and Set Lock



Test and Set Lock



Using TSL Directly

I

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

J

```
1 repeat
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))
3       no-op;
4   critical section
5   mylock = FALSE;
6   remainder section
7 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 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)*

How Can We Implement Them?

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

Synchronization in User Programs

- User threads call sleep and wakeup system calls
- Sleep and wakeup are system calls (in the kernel)
  - *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?*

کنترل هم‌روندی در هسته

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*

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 kernel code 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*
  - *Disabling interrupts on one CPU didn't prevent this!*
- Solution: use a mutex lock (based on TSL)
  - *Ensures mutual exclusion for all code that uses it*

Mutex is not enough

- Interrupt inside interrupt handler

Some Other 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 !*

Recap

- 1. What is a race condition?
- 2. How can we protect against race conditions?
- 3. Can locks be implemented simply by reading and writing to a binary variable in memory?
- 4. How can a kernel make synchronization-related system calls atomic on a uniprocessor?
  - *Why wouldn't this work on a multiprocessor?*
- 5. Why is it better to block rather than spin on a uniprocessor?
- 6. Why is it sometimes better to spin rather than block on a multiprocessor?

جلسه‌ی بعد

- معرفی Semaphore
- بررسی مسائل کلاسیک هم‌روندی