R

بسم الله الرحمن الرحیم

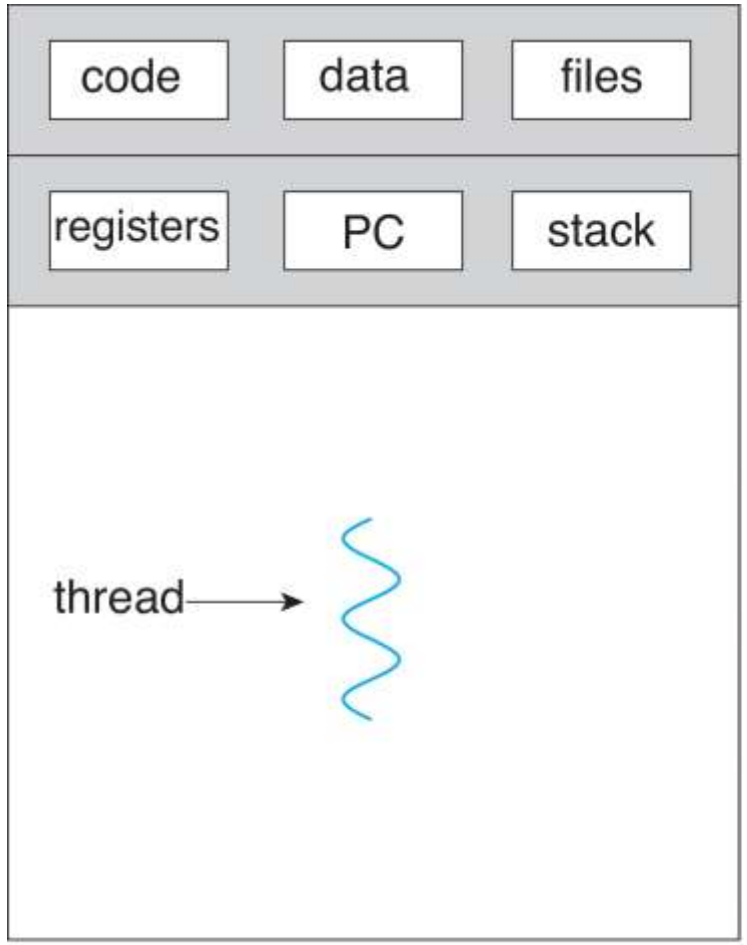
سیستم عامل

جلسه پنجم – ادامه‌ی ریسمان، همزمانی

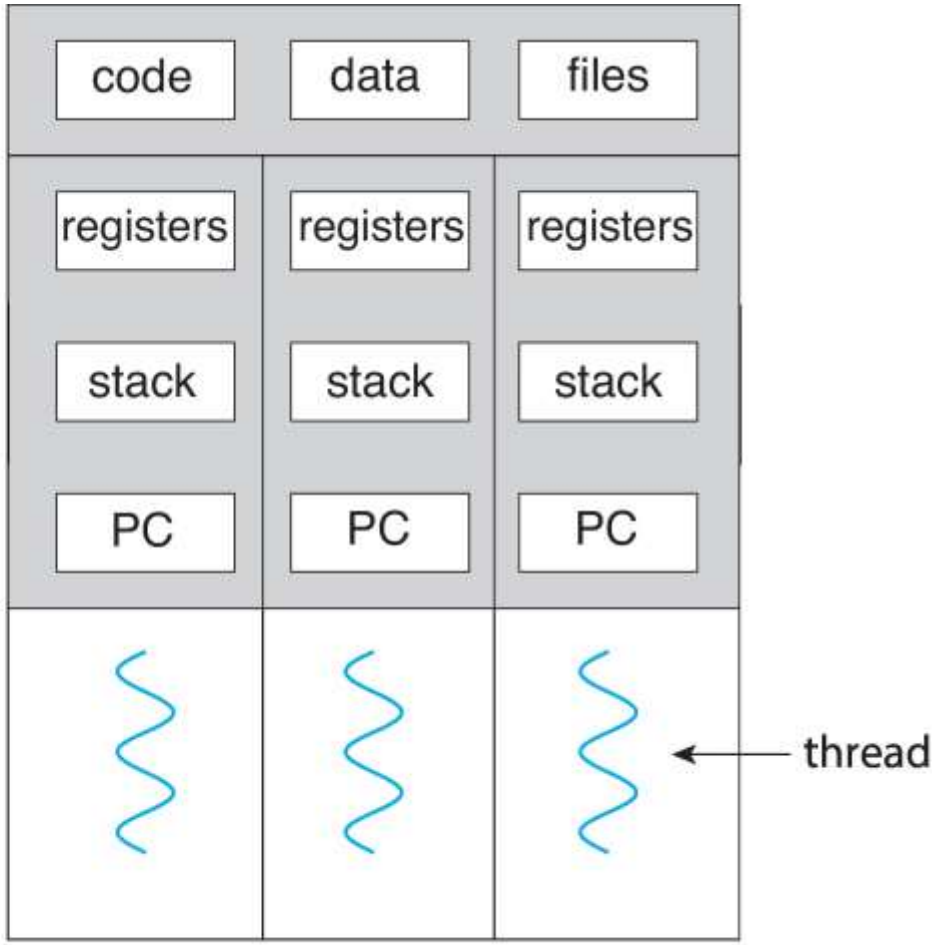
آنچه گذشت

جلسه‌ی قبل، ریسمان

ریسمان



single-threaded process



multithreaded process

Why Use Threads?

- **Utilize** multiple CPU's concurrently
- **Low cost** communication via shared memory
- Overlap computation and blocking on a single CPU
  - *Blocking due to I/O*
  - *Computation and communication*
- Handle asynchronous events

جلسه‌ی جدید

ادامه‌ی ریسمان

استراتژی‌های استفاده از ریسمان

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*

Manager / Worker

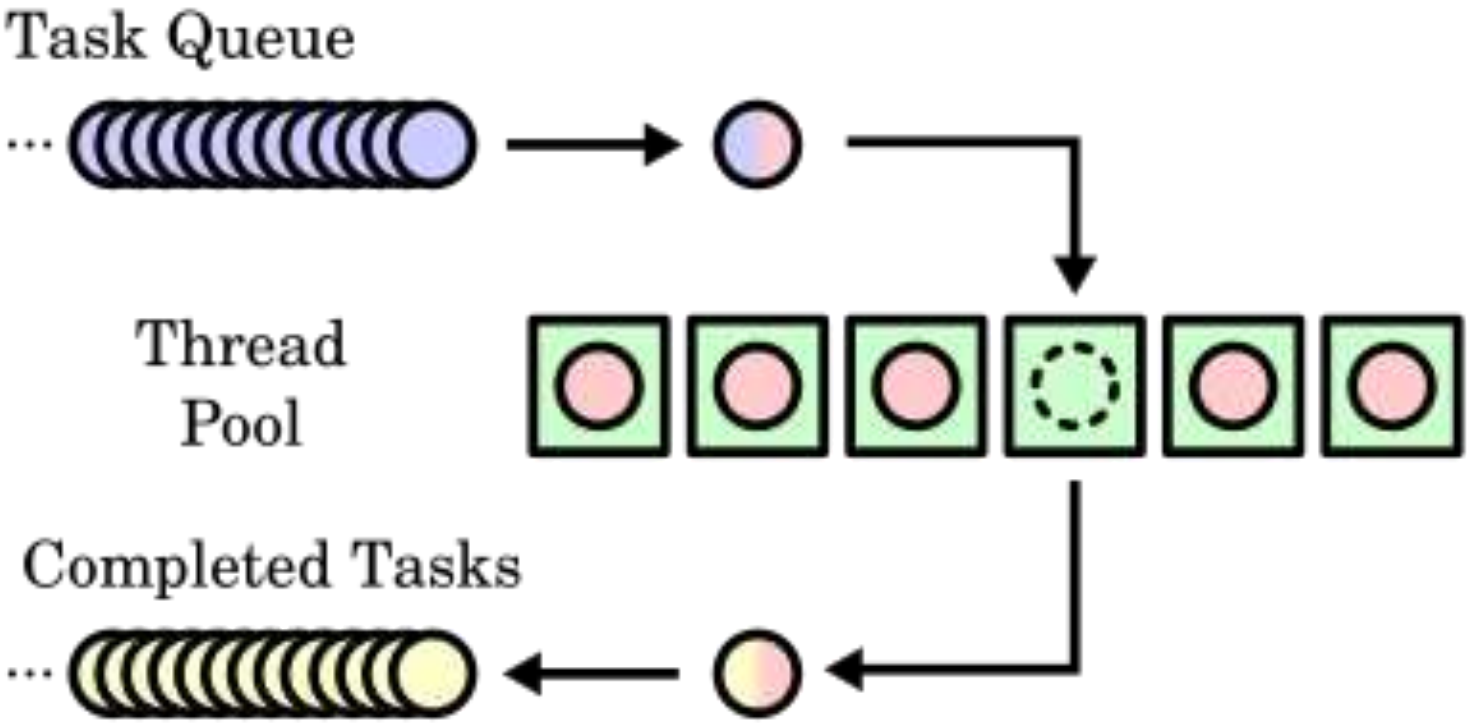


Image from https://en.wikipedia.org/wiki/Thread_pool

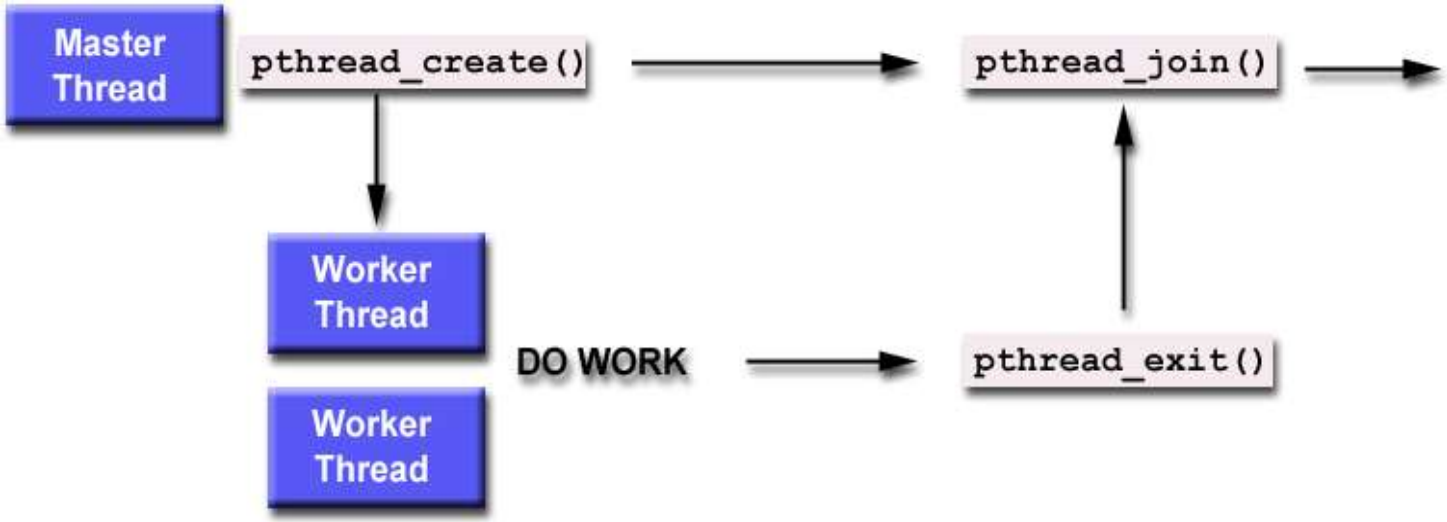
Java Thread Pools

- Three factory methods for creating thread pools in Executors class:
  - `static ExecutorService newSingleThreadExecutor()`
  - `static ExecutorService newFixedThreadPool(int size)`
  - `static ExecutorService newCachedThreadPool()`

Pthreads (continued)

- pthread_exit (status)
  - *Terminates the thread and returns "status" to any joining thread*
- pthread_join (threadid,status)
  - *Blocks the calling thread until thread specified by "threadid" terminates*
  - *Return status from pthread_exit is passed in "status"*
  - *One way of synchronizing between threads*
- pthread_yield ()
  - *Thread gives up the CPU and enters the run queue*

Using Create, Join and Exit



Pros & Cons of Threads

- Pros:
  - *Overlap I/O with computation!*
  - *Cheaper context switches*
  - *Better mapping to multiprocessors*
- Cons:
  - *Potential thread interactions*
  - *Complexity of debugging*
  - *Complexity of multi-threaded programming*
  - *Backwards compatibility with existing code*

پیاده‌سازی ریسمان

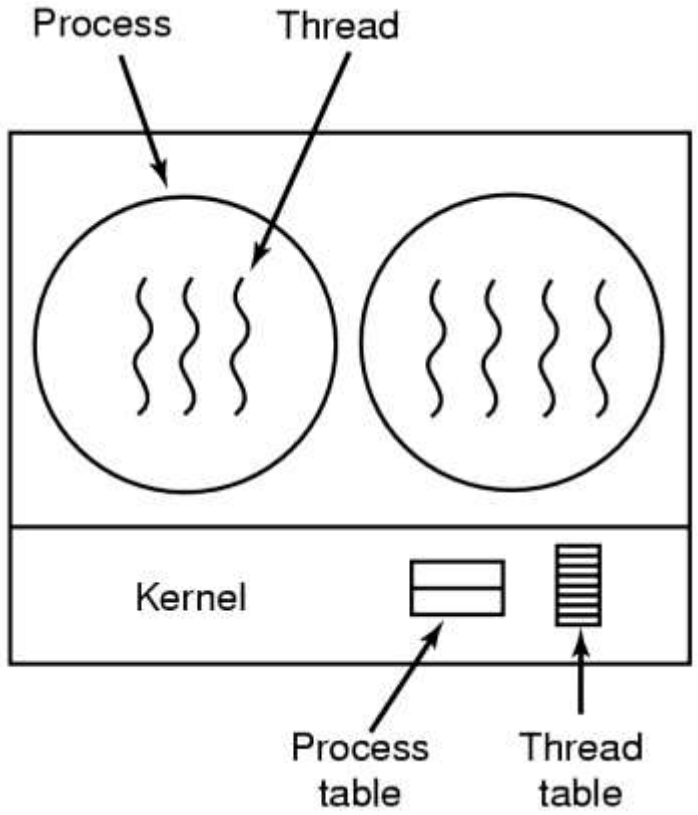
در سیستم عامل یا در کتاب‌خانه‌ی برنامه؟

Thread

- The idea of managing multiple abstract program counters above a single real one can be implemented using privileged or non-privileged code.
  - *Threads can be implemented in the OS or at user level*
- User level thread implementations
  - *Thread scheduler runs as user code (thread library)*
  - *Manages thread contexts in user space*
  - *The underlying OS sees only a traditional process above*

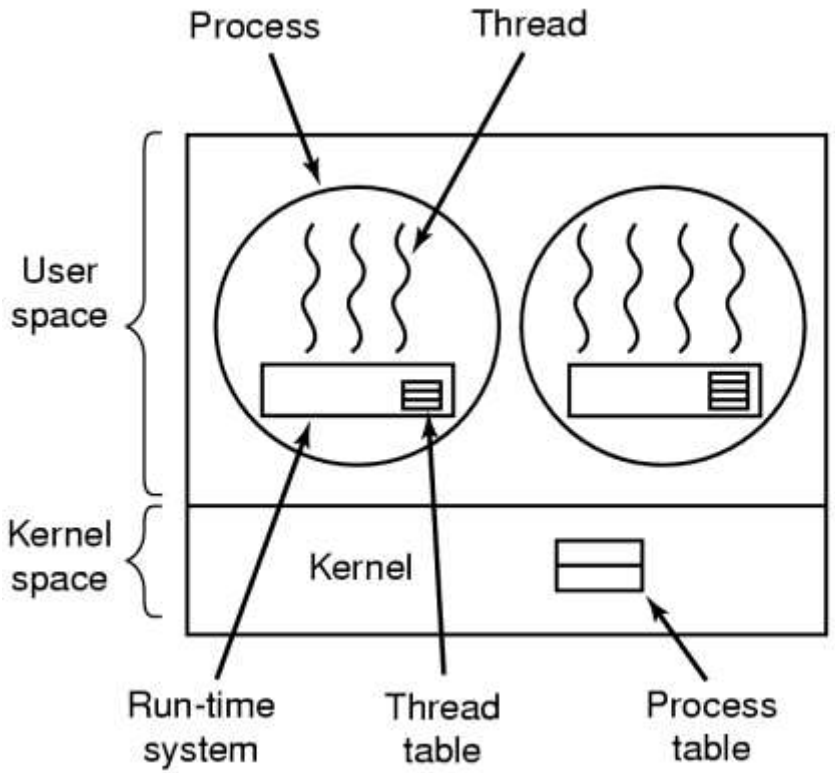
Kernel-Level Threads

Thread-switching
code is in the kernel



User-Level Threads

The thread-switching code is in user space



User-level threads (Green Threads)

- Advantages
  - *Cheap context switch costs among threads in the same process!*
  - *Calls are procedure calls not system calls!*
  - *User-programmable scheduling policy*
- Disadvantages
  - *How to deal with blocking system calls!*
  - *How to overlap I/O and computation!*

مسائل همزمانی

Concurrency

- *Two or more threads*
- *Parallel or pseudo parallel execution*
- *Can’t predict the relative execution speeds*
- *The threads access shared variables*

Concurrency Vs. Parallelism

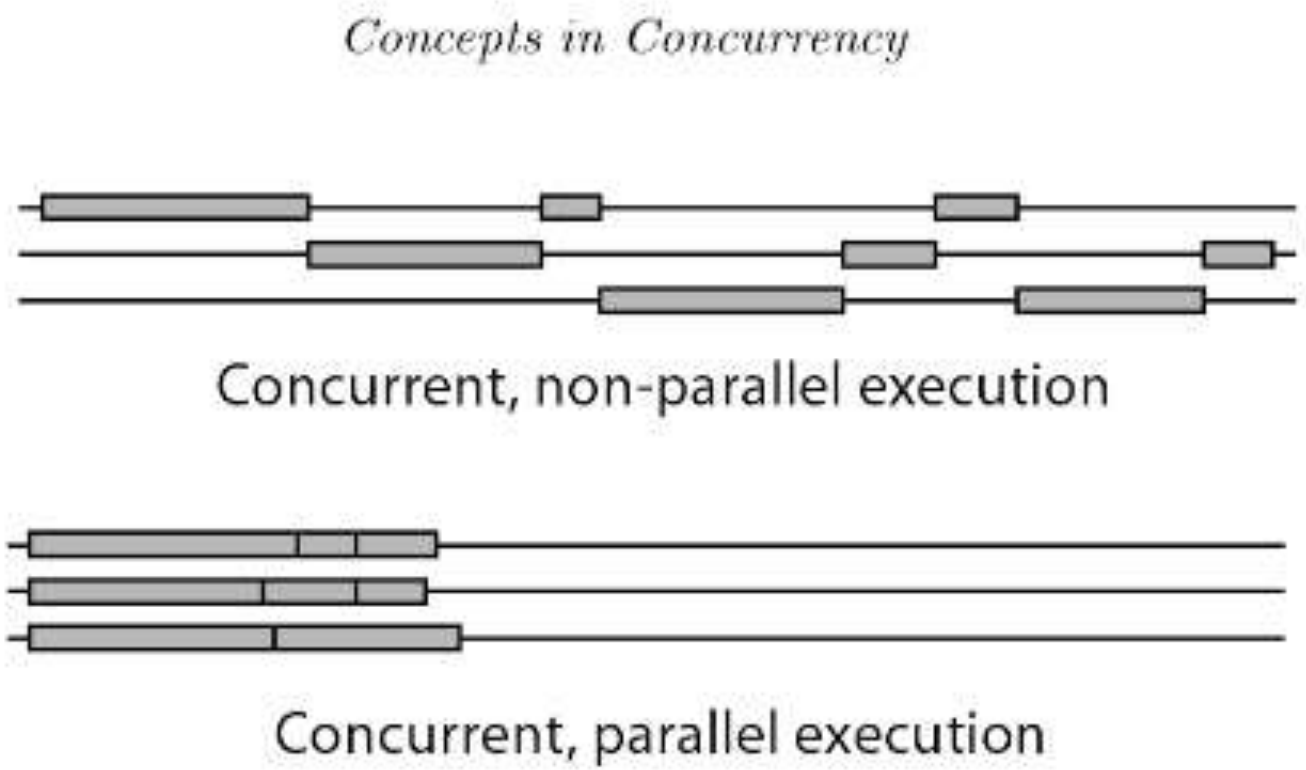


Image from: https://stackoverflow.com/a/53215726

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!*

وضعیت رقابتی

Race Condition

Race Conditions

- What is a race condition?
- Why do race conditions occur?

Race Conditions

- A simple multithreaded program with a race:

i++;

Race Conditions

- A simple multithreaded program with a race:

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

آیا مسئله‌ی وضعیت رقابتی مهم است؟

- مثال: بانک!
- فرض کنید یک متغیر برای میزان مانده حساب دارید.
- remainingMoney -= 100000
- اگر همزمان چندین برداشت از حساب کنید.

Race Conditions

- Why did this race condition occur?
  - *two or more threads have an inconsistent view of a shared memory region (ie., a variable)*
  - *values of memory locations are 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 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*

- Mutual Exclusion?
- Critical Section?

Critical Section

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

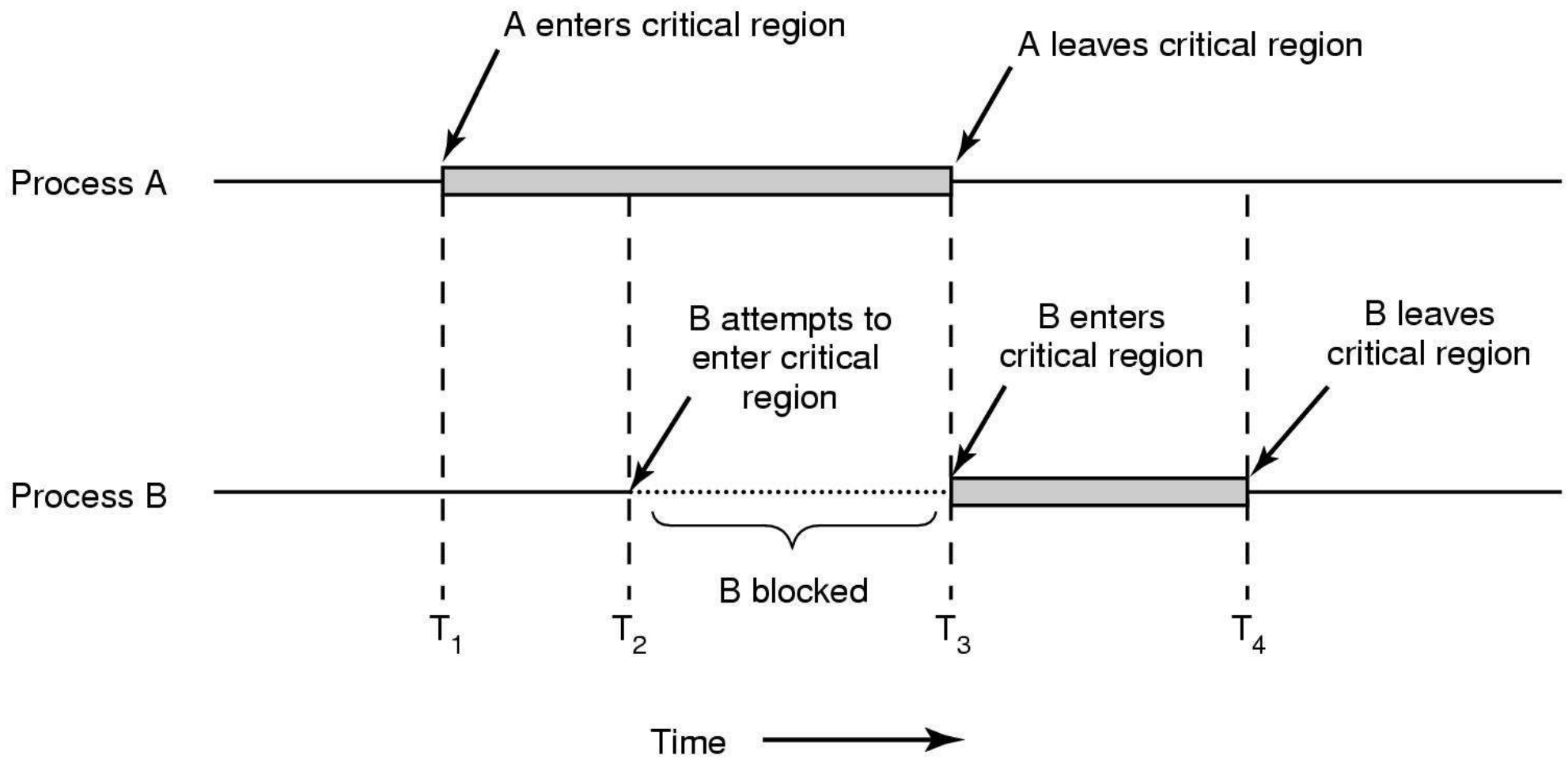
```
...
load i to register;
increment register;
store register to i;
...
```

Critical Section

Mutual Exclusion Conditions

- No two processes simultaneously in critical section
- No assumptions about speeds or numbers of CPUs
- No process running outside its critical section may block another process
- No process waits forever to enter its critical section

Mutual Exclusion Example

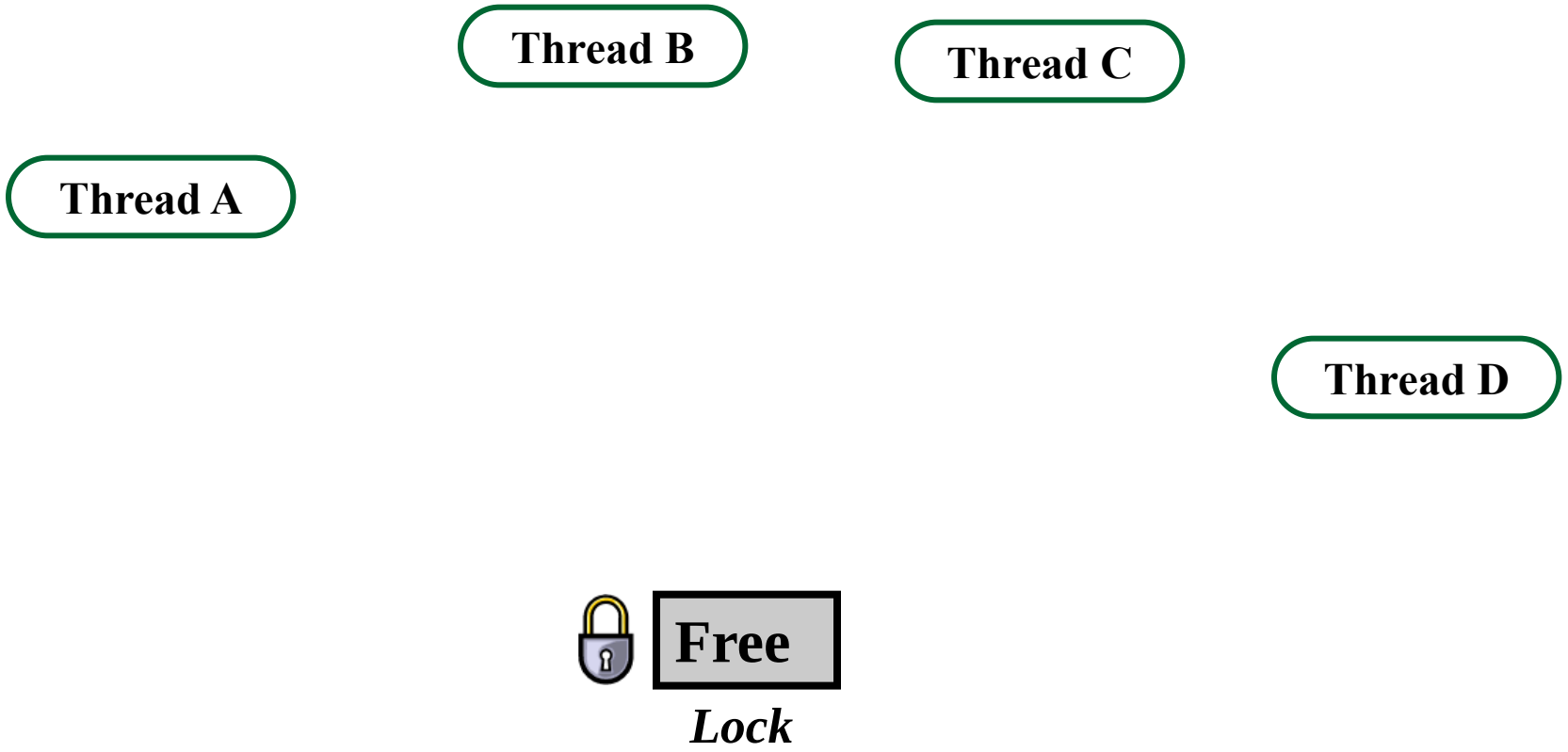


چطوری MUTUAL EXCLUSION؟

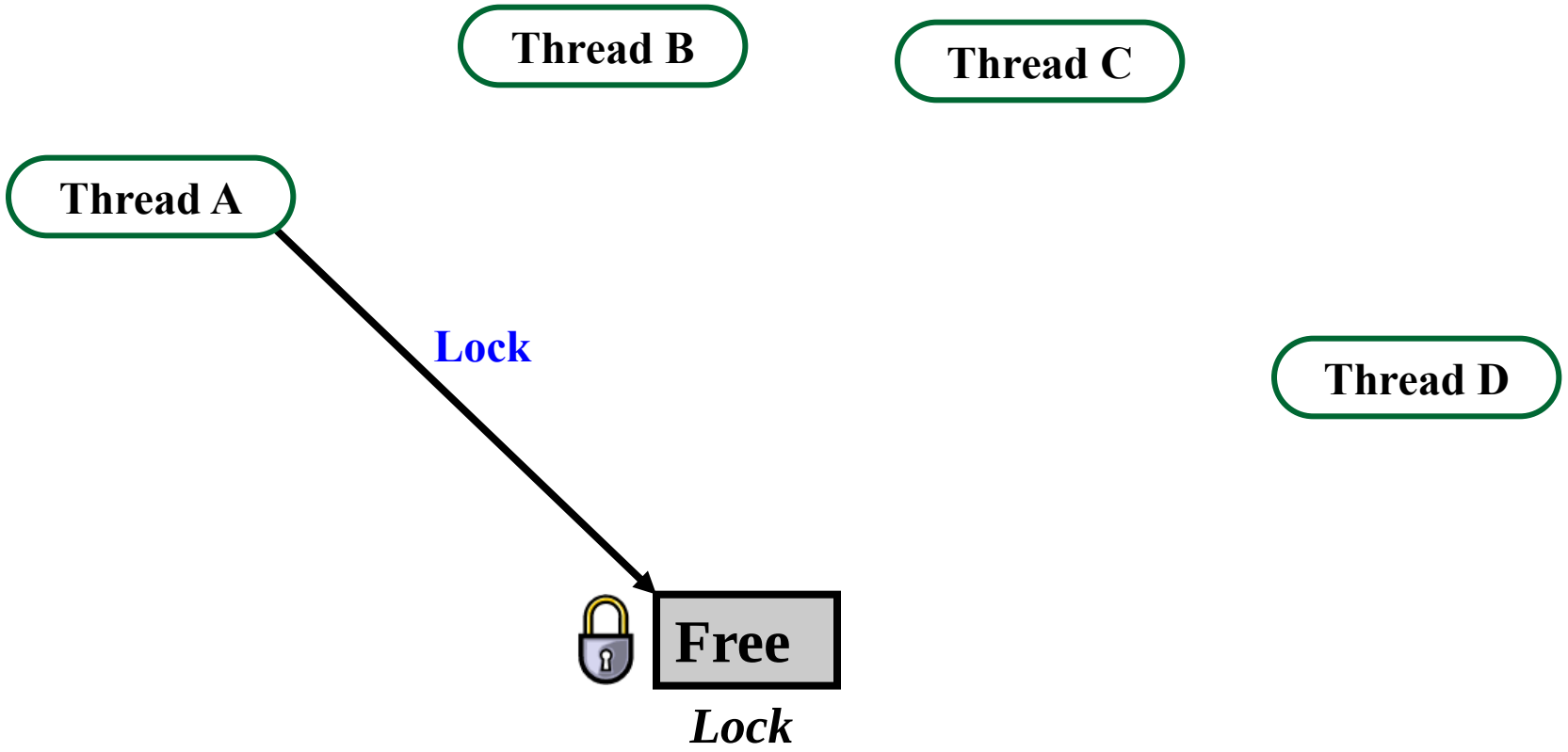
Enforcing Mutual Exclusion

- 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*

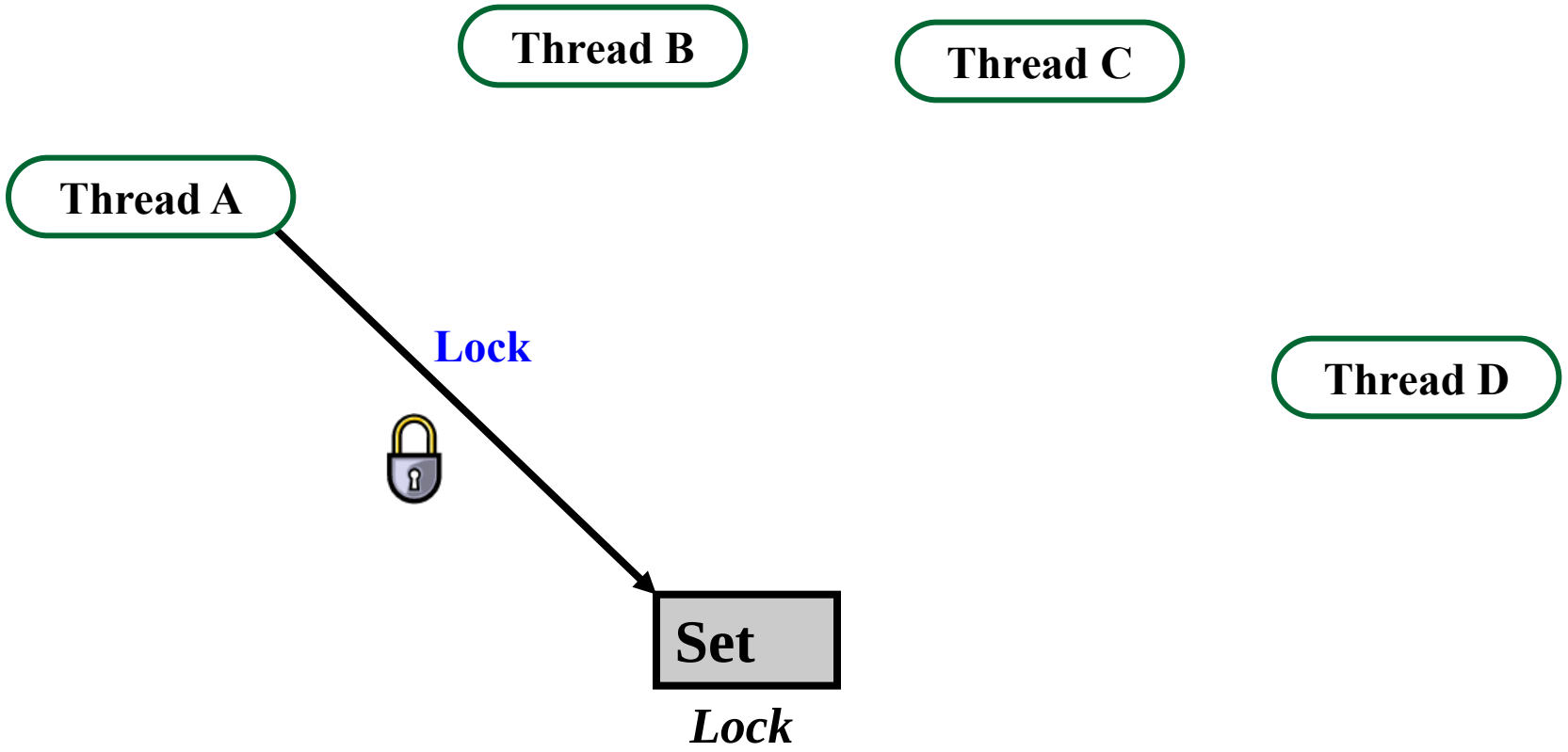
Acquiring and Releasing Locks



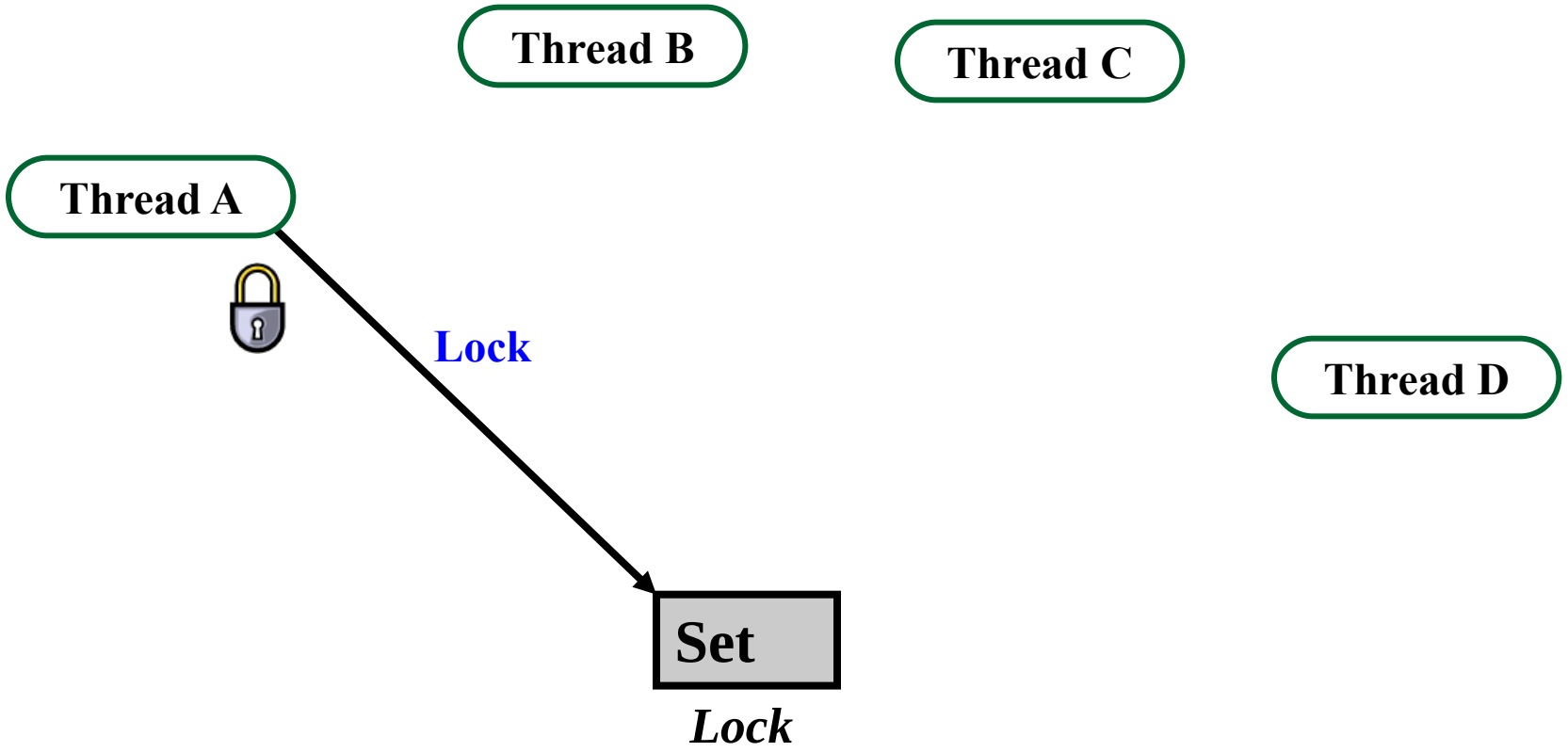
Acquiring and Releasing Locks



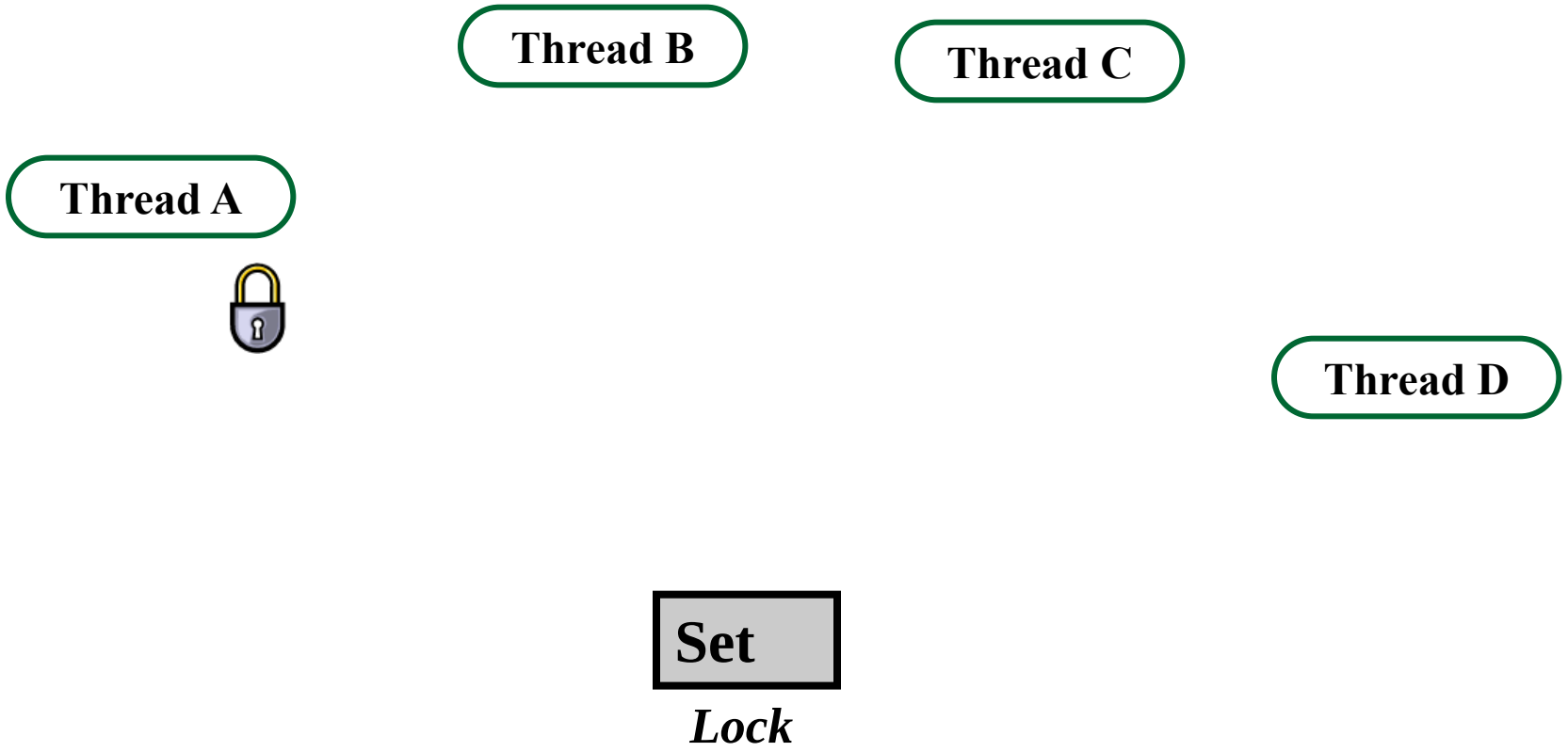
Acquiring and Releasing Locks



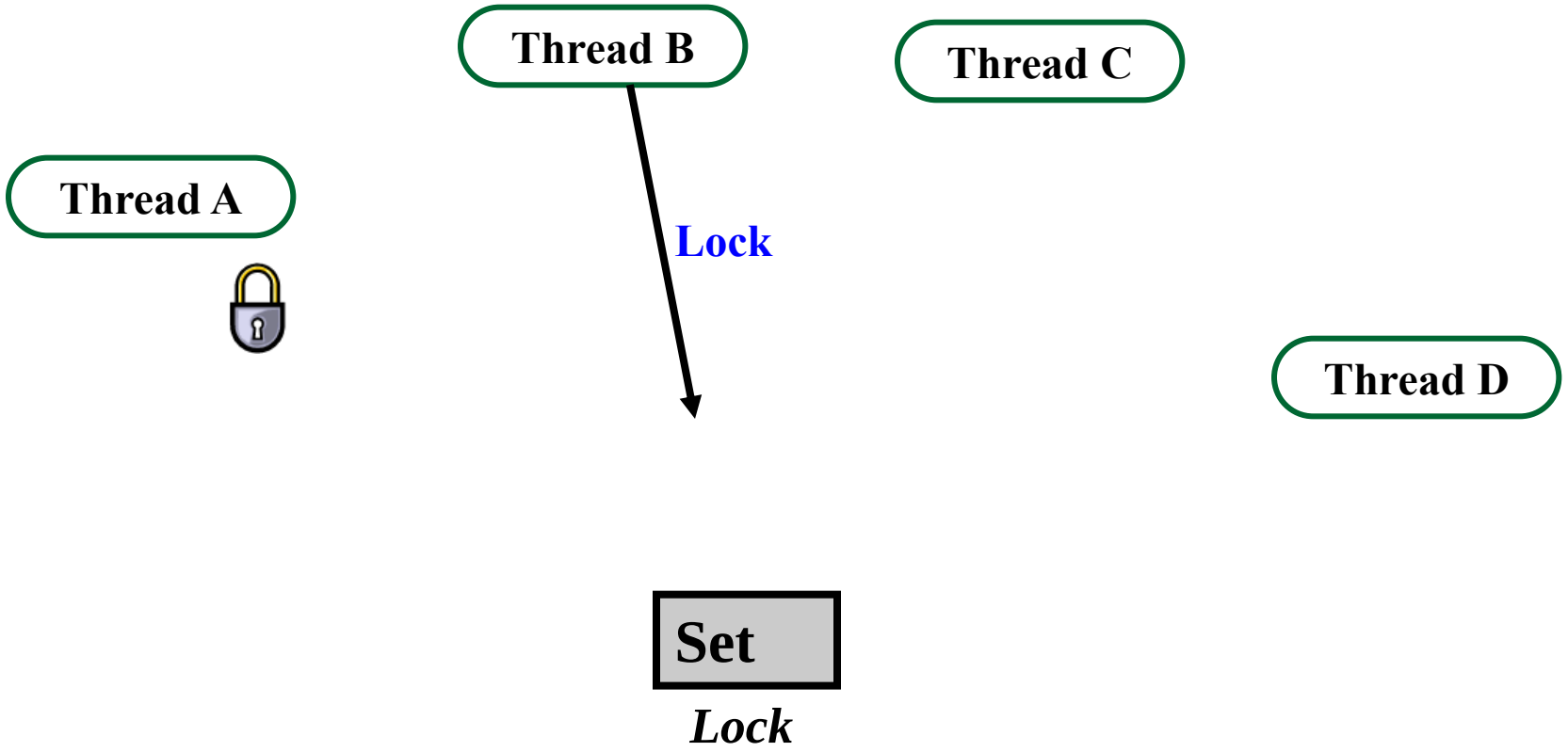
Acquiring and Releasing Locks



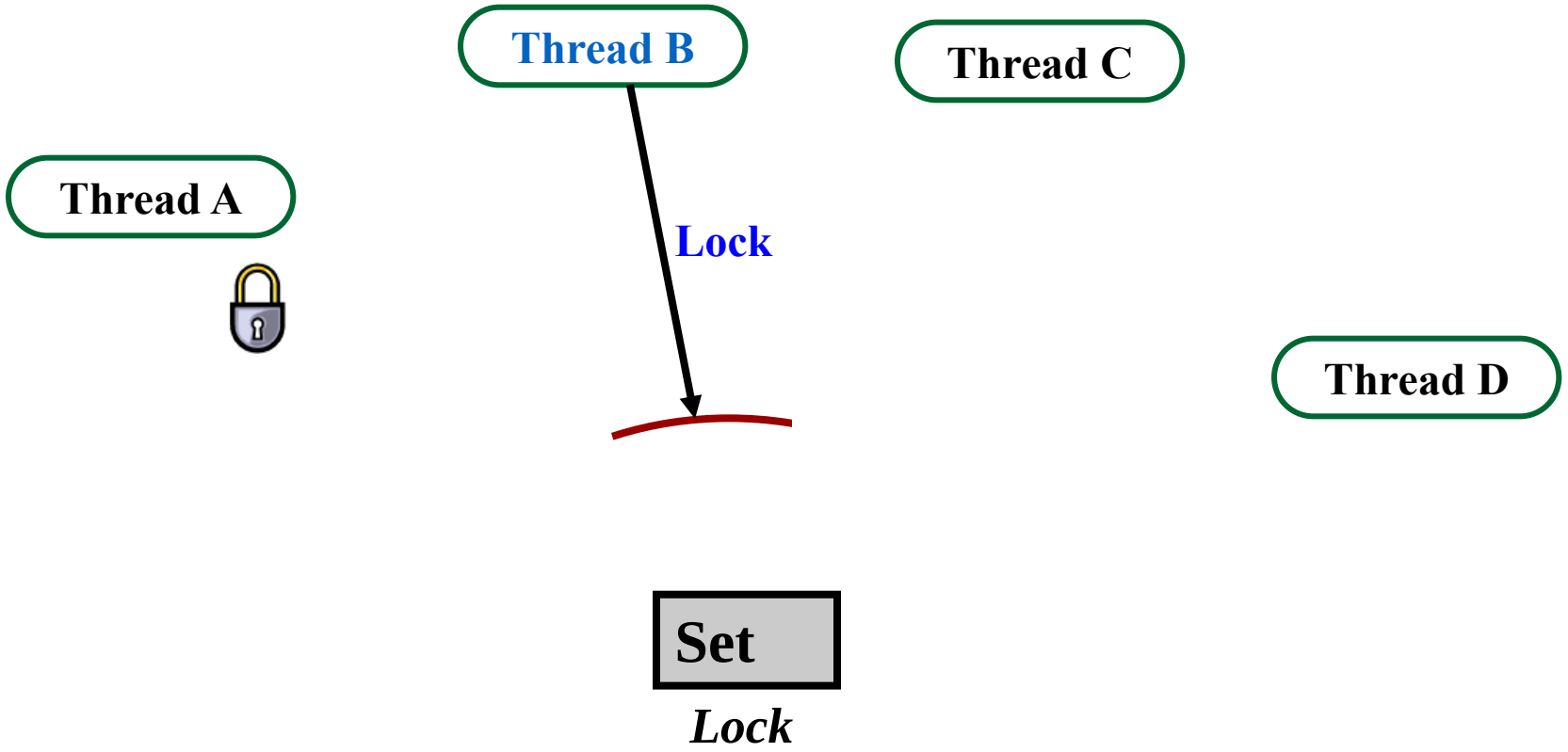
Acquiring and Releasing Locks



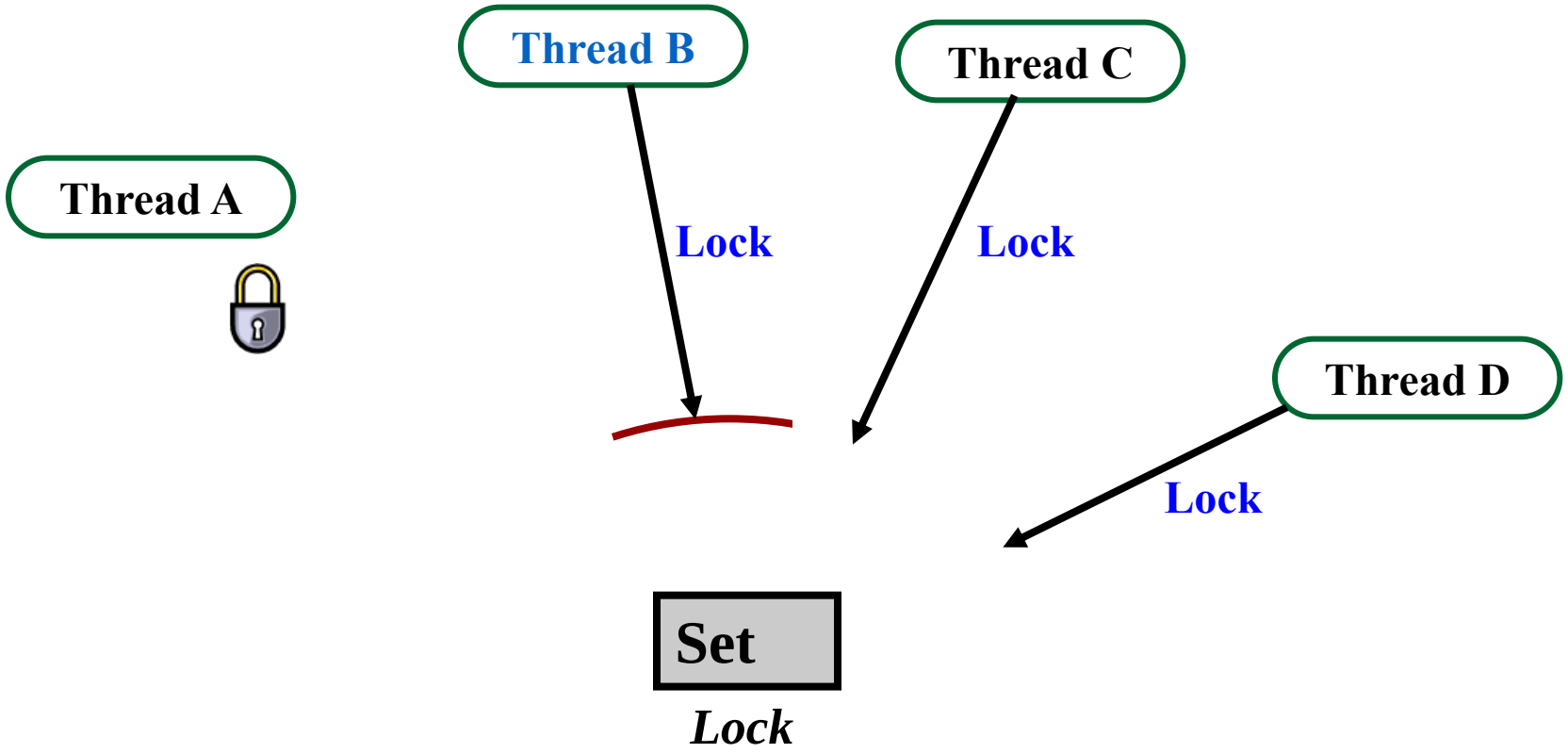
Acquiring and Releasing Locks



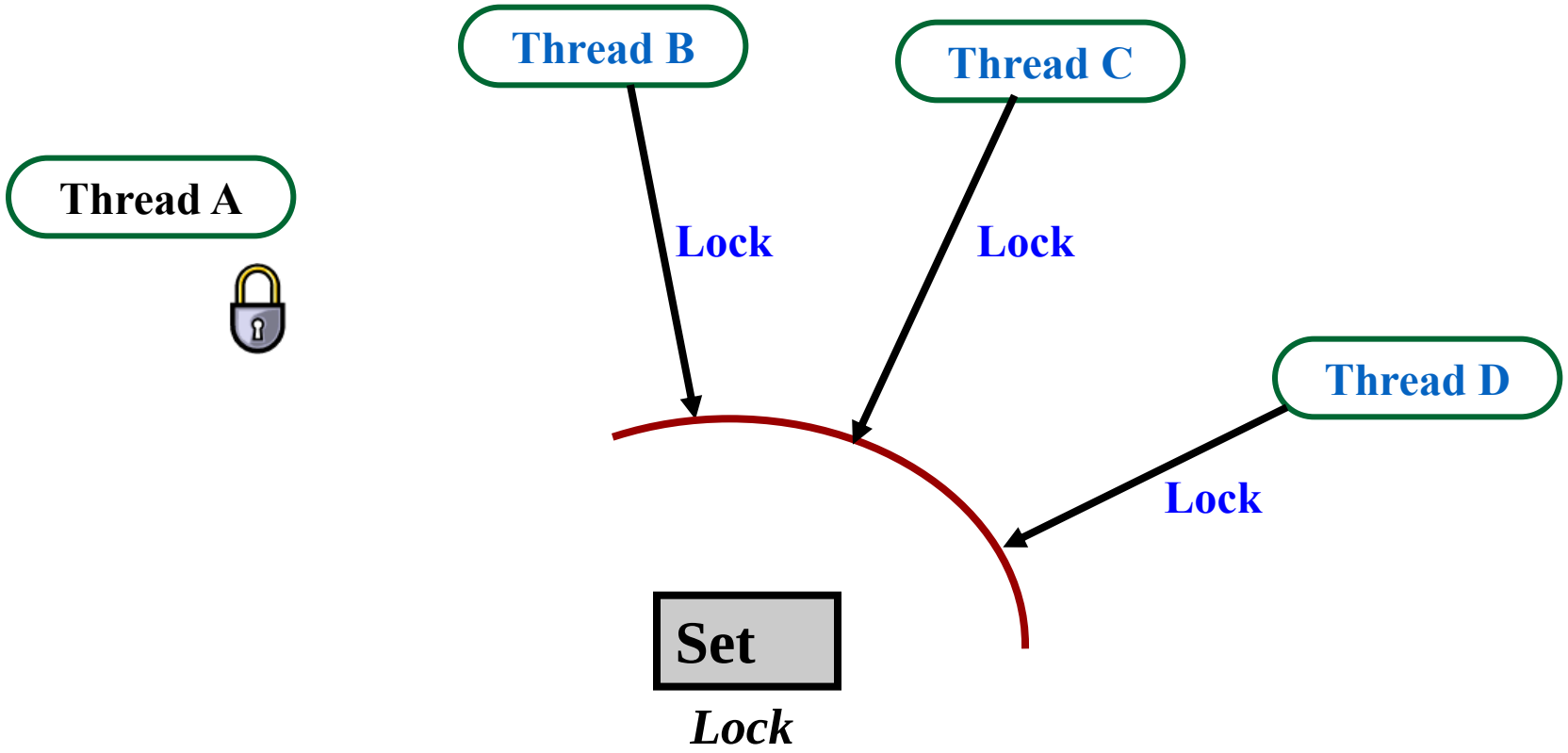
Acquiring and Releasing Locks



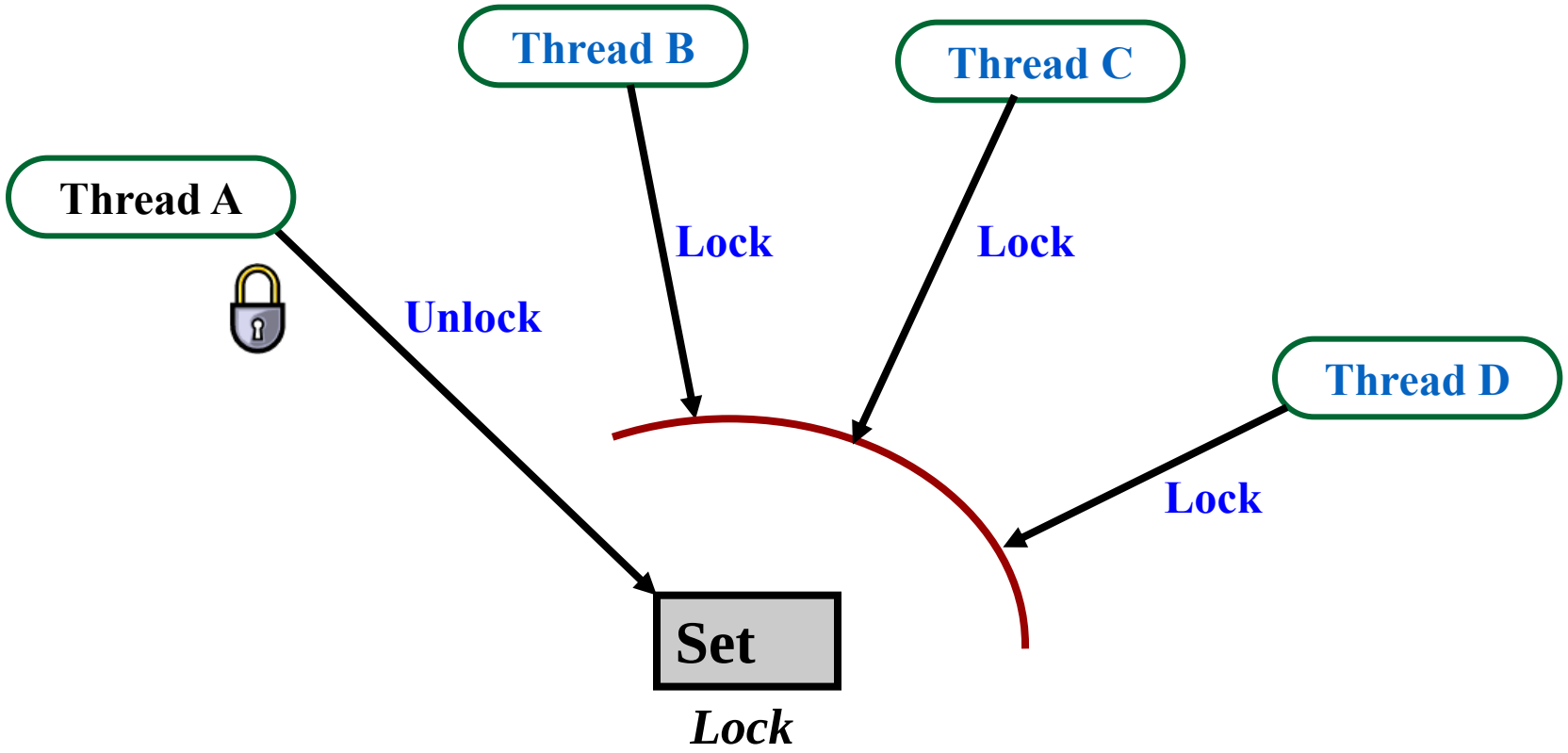
Acquiring and Releasing Locks



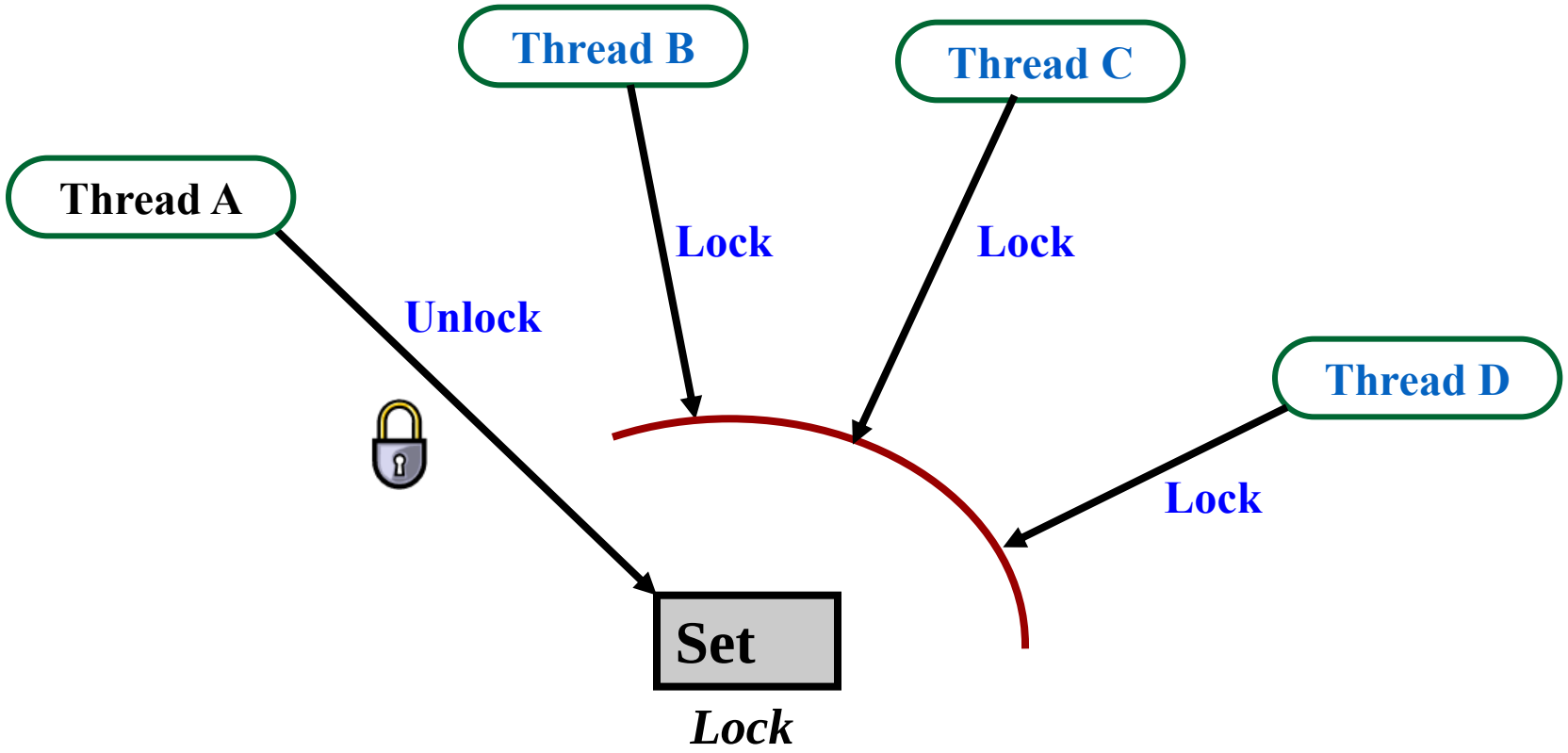
Acquiring and Releasing Locks



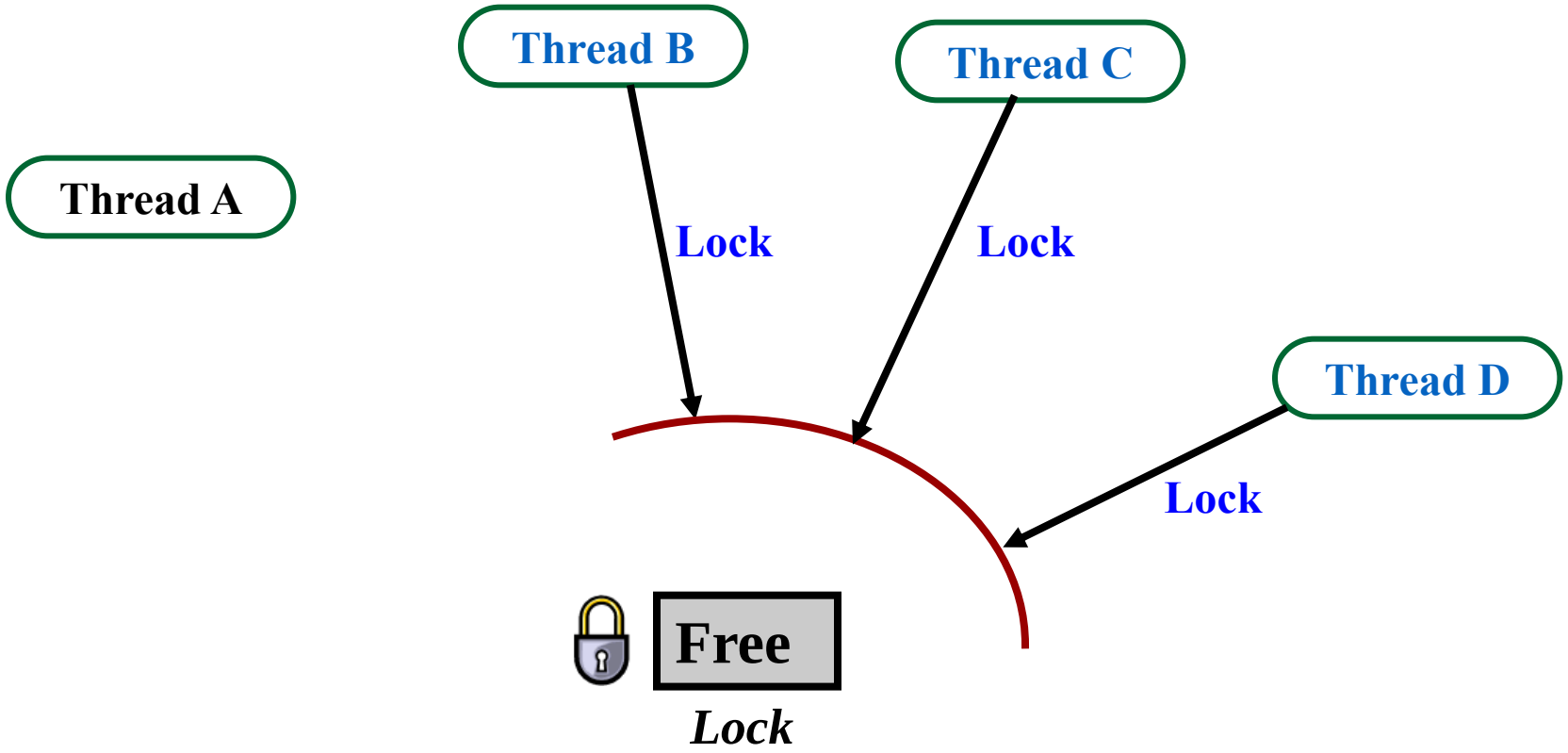
Acquiring and Releasing Locks



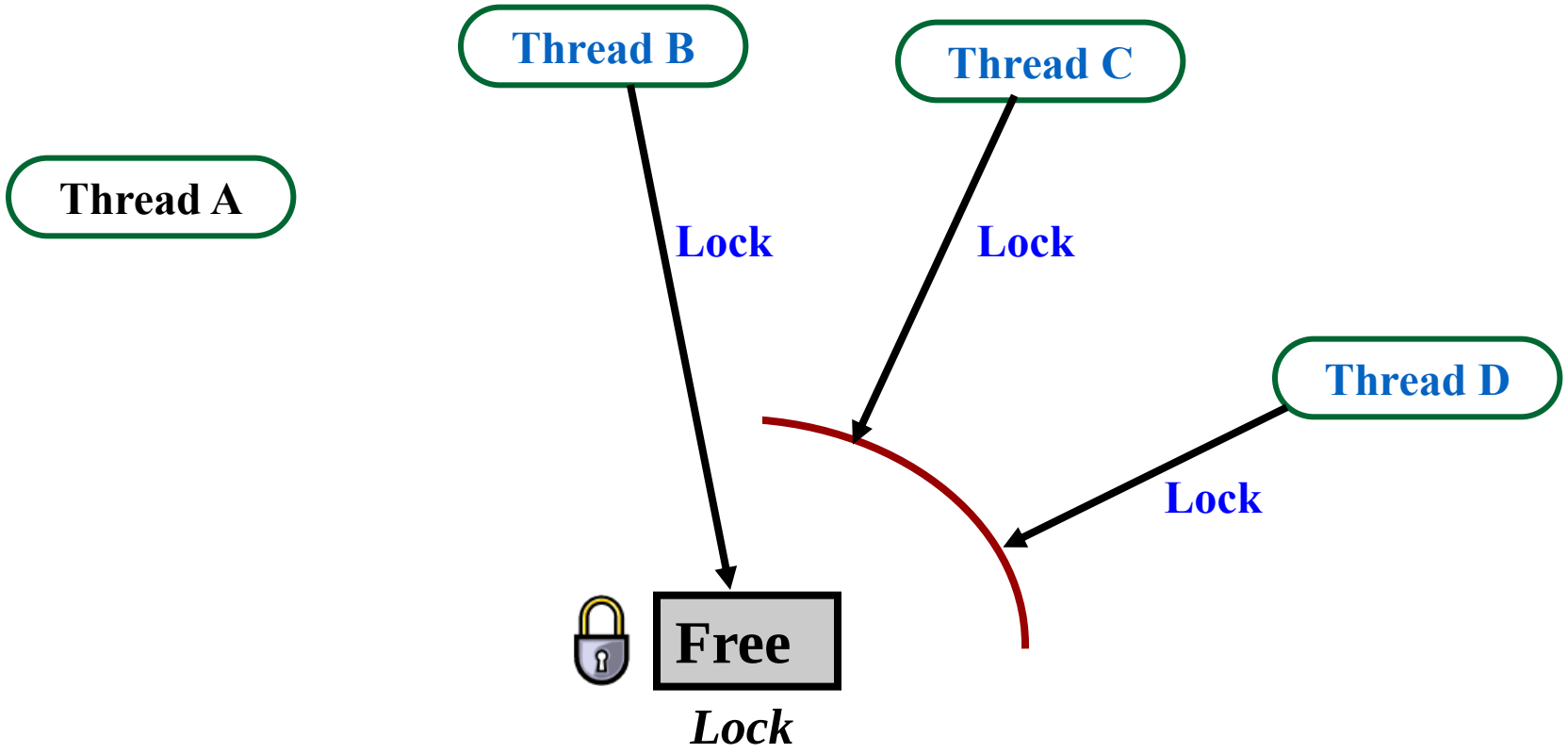
Acquiring and Releasing Locks



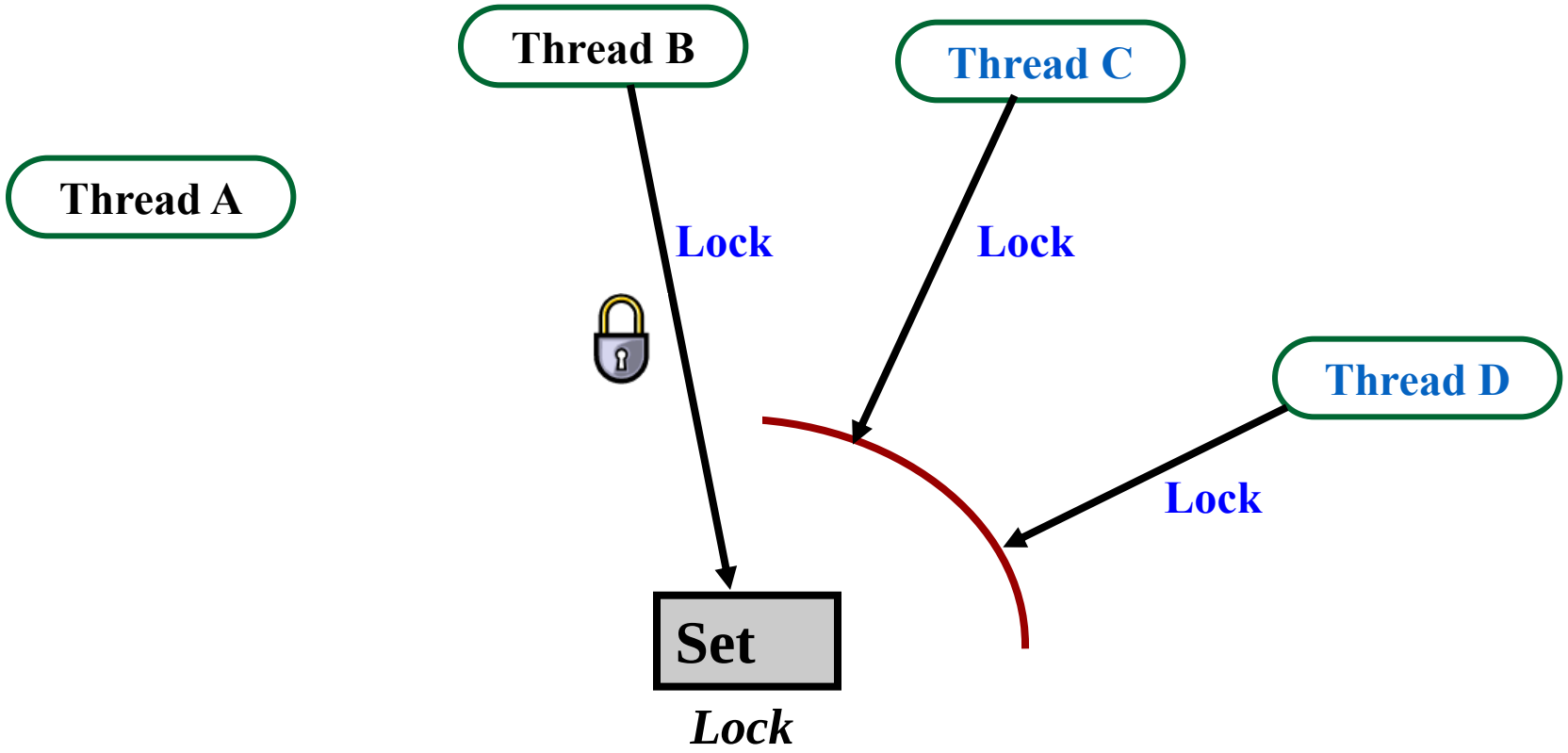
Acquiring and Releasing Locks



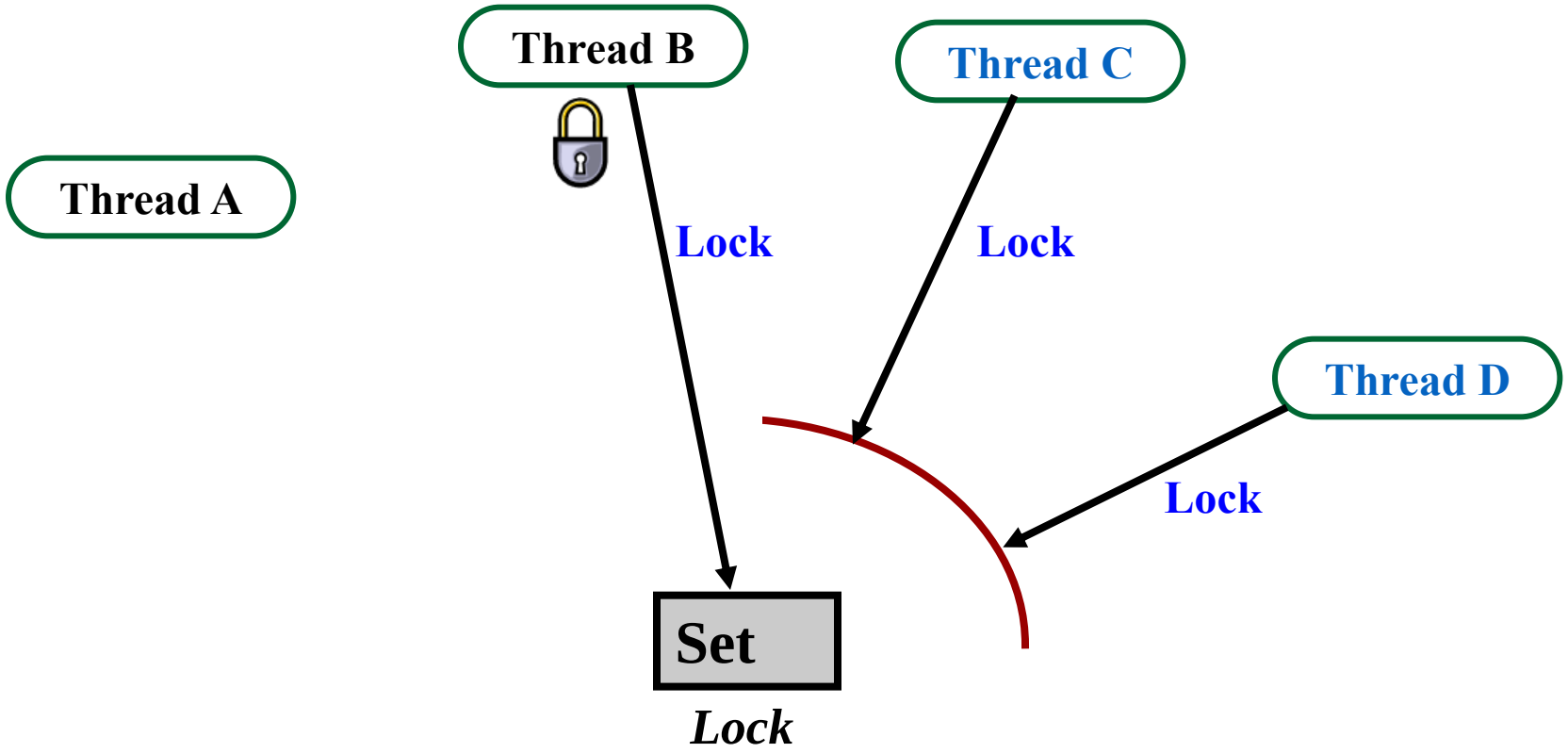
Acquiring and Releasing Locks



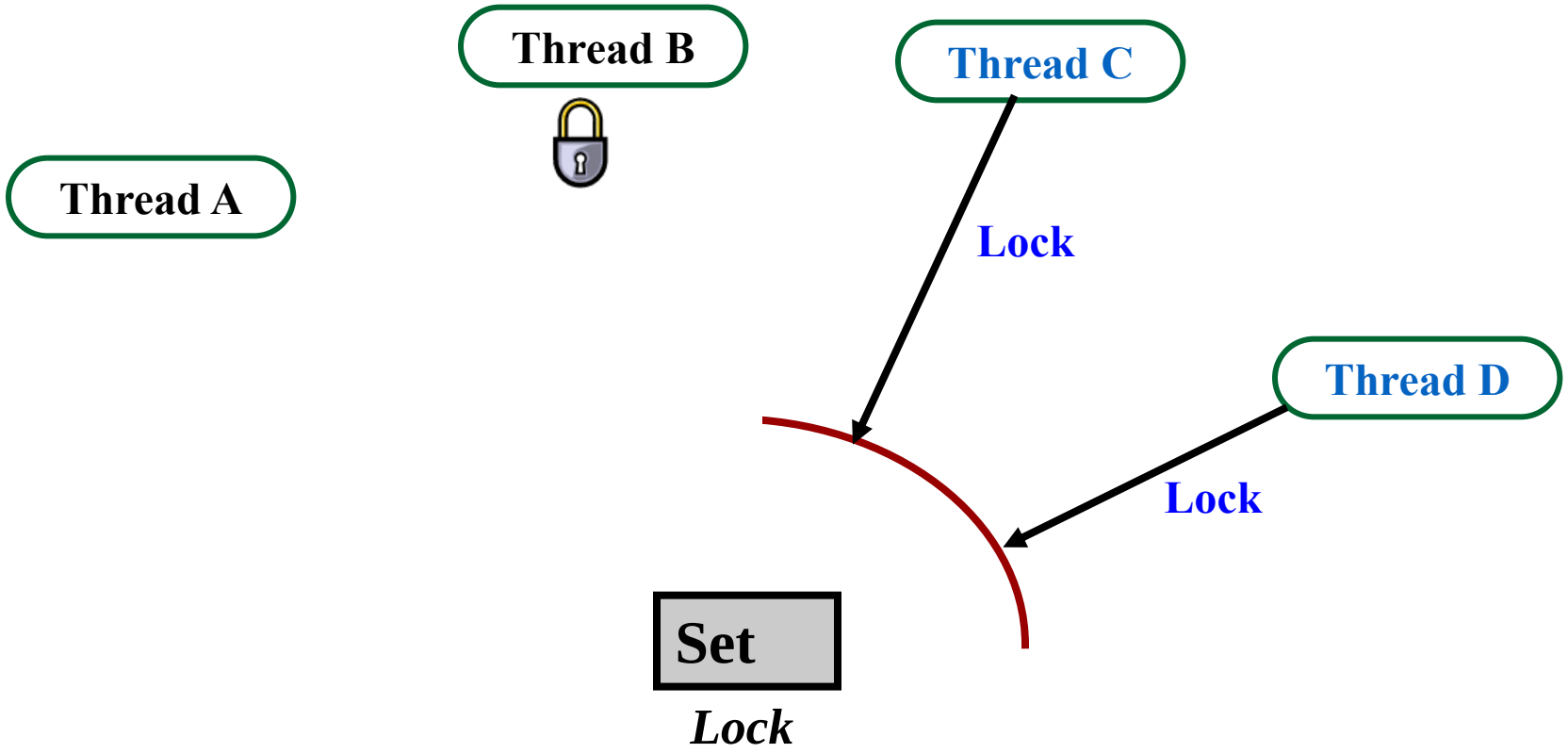
Acquiring and Releasing Locks



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
```

جلسه‌ی بعد

- چگونگی پیاده‌سازی Mutex
- Semaphoreها
- مسائل مشهور هم‌زمانی

آزمونک

زمان: ۲۰ دقیقه