CS 333
Introduction to Operating Systems

Class 4 – Concurrent Programming and Synchronization Primitives

Jonathan Walpole
Computer Science
Portland State University

What does a typical thread API look like?

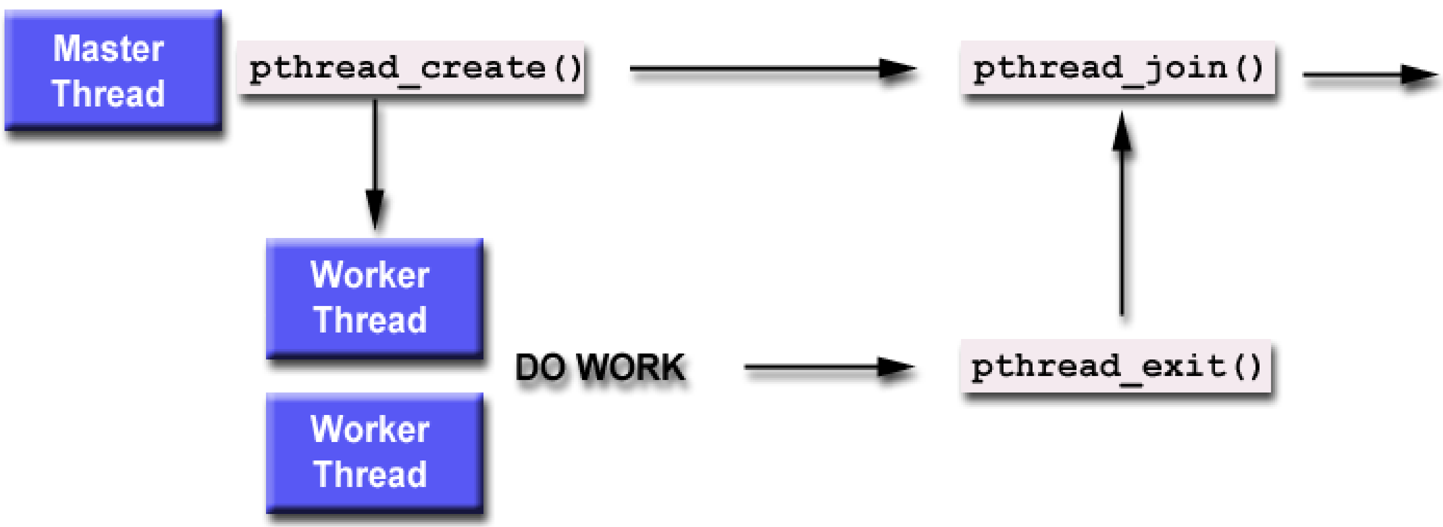
- POSIX standard threads (Pthreads)
- First thread exists in main(), typically creates the others

- **pthread_create (thread,attr,start_routine,arg)**
  - Returns new thread ID in "thread"
  - Executes routine specified by "start_routine" with argument specified by "arg"
  - Exits on return from routine or when told explicitly

Thread API (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 primitives



An example Pthreads program

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
  printf("\n%d: Hello World!\n", threadid);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc, t;
  for(t=0; t<NUM_THREADS; t++)
  {
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    if (rc)
    {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Program Output

```
Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
3: Hello World!
Creating thread 4
4: Hello World!
```

For more examples see: http://www.llnl.gov/computing/tutorials/pthreads

Pros & cons of threads

- **Pros**
  - Overlap I/O with computation!
  - Cheaper context switches
  - Better mapping to shared memory multiprocessors

- **Cons**
  - Potential thread interactions due to concurrency
  - Complexity of debugging
  - Complexity of multi-threaded programming
  - Backwards compatibility with existing code

Concurrency

Assumptions:

- Two or more threads
- Each executes in (pseudo) parallel
- We can't predict exact running speeds
- The threads can interact via access to shared variables

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!*

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;
...
```

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

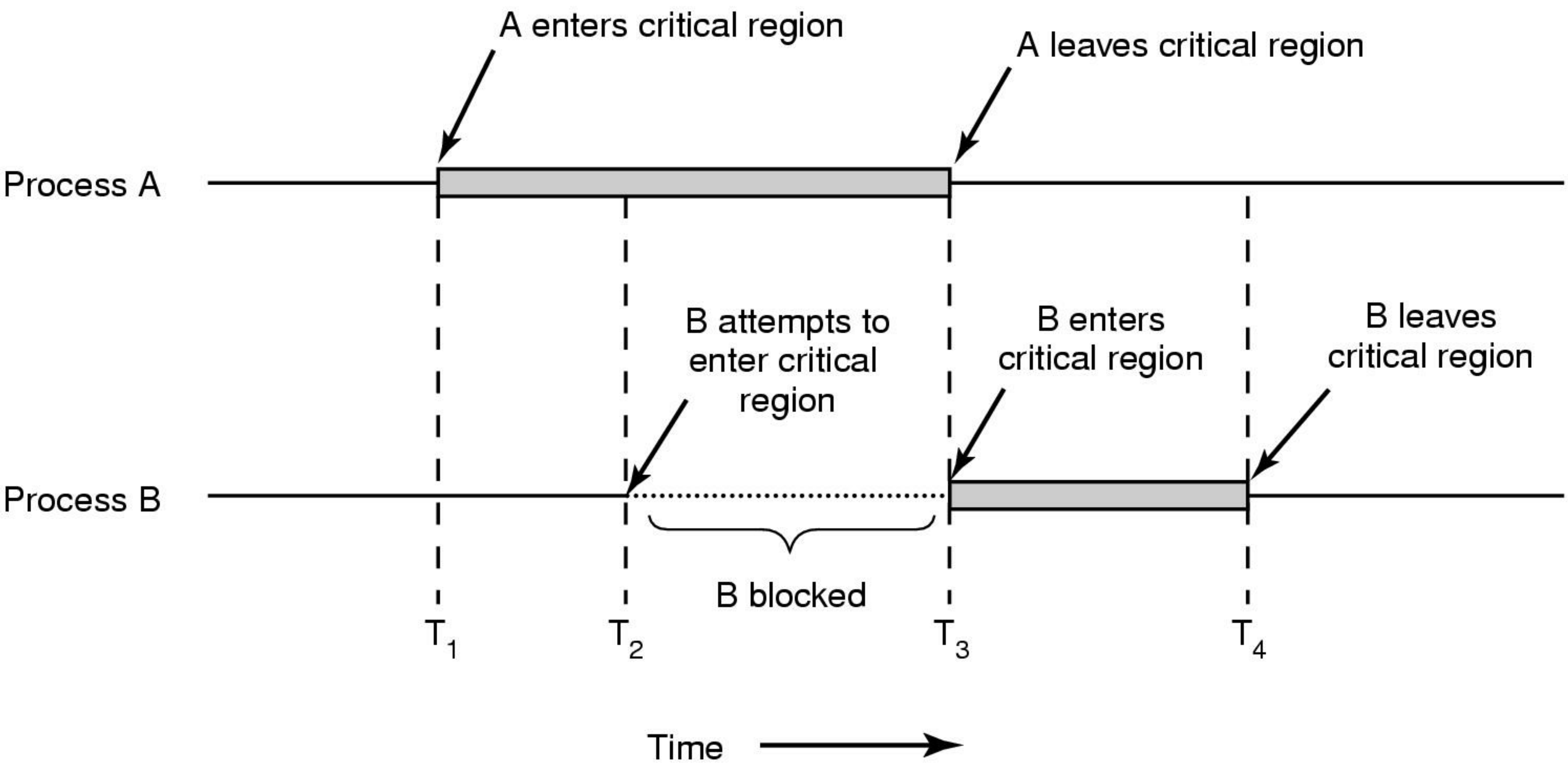
Race Conditions

- **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

Mutual exclusion conditions

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

Using mutual exclusion for critical sections

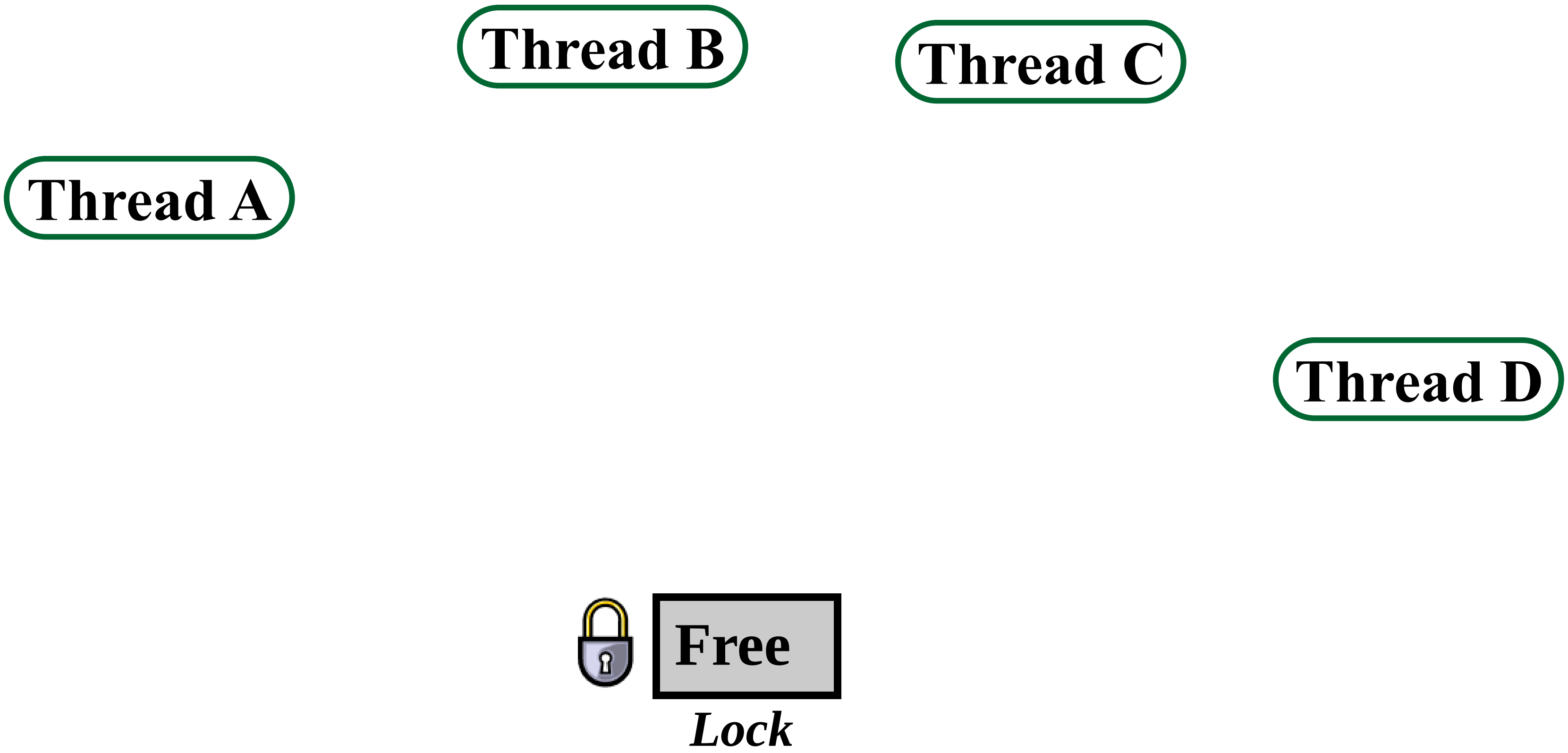


How can we enforce mutual exclusion?

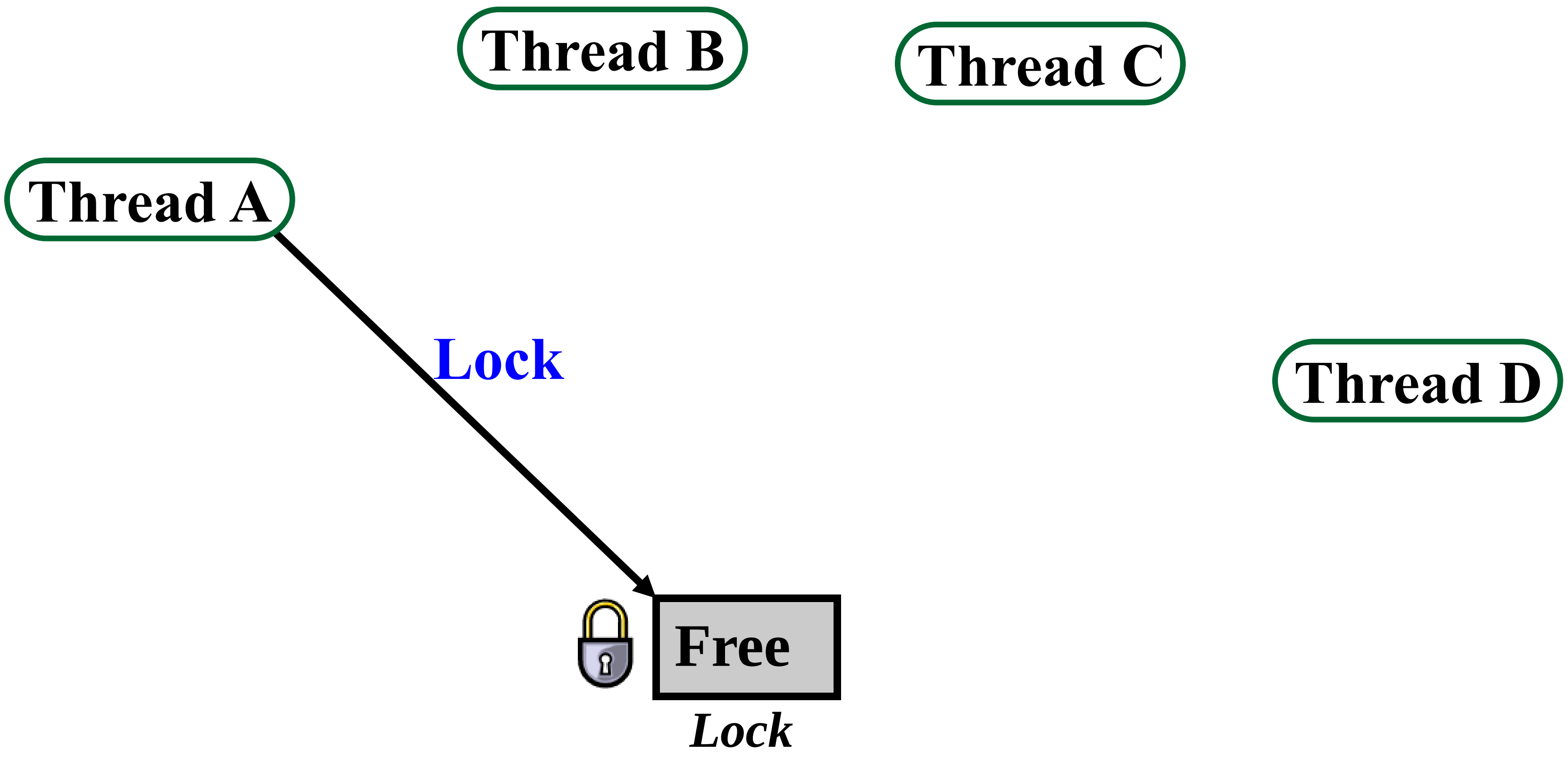
- **What about using *locks* ?**
- **Locks solve the problem of exclusive access to shared data.**
  - Acquiring a lock prevents concurrent access
  - Expresses intention to enter critical section

- ***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

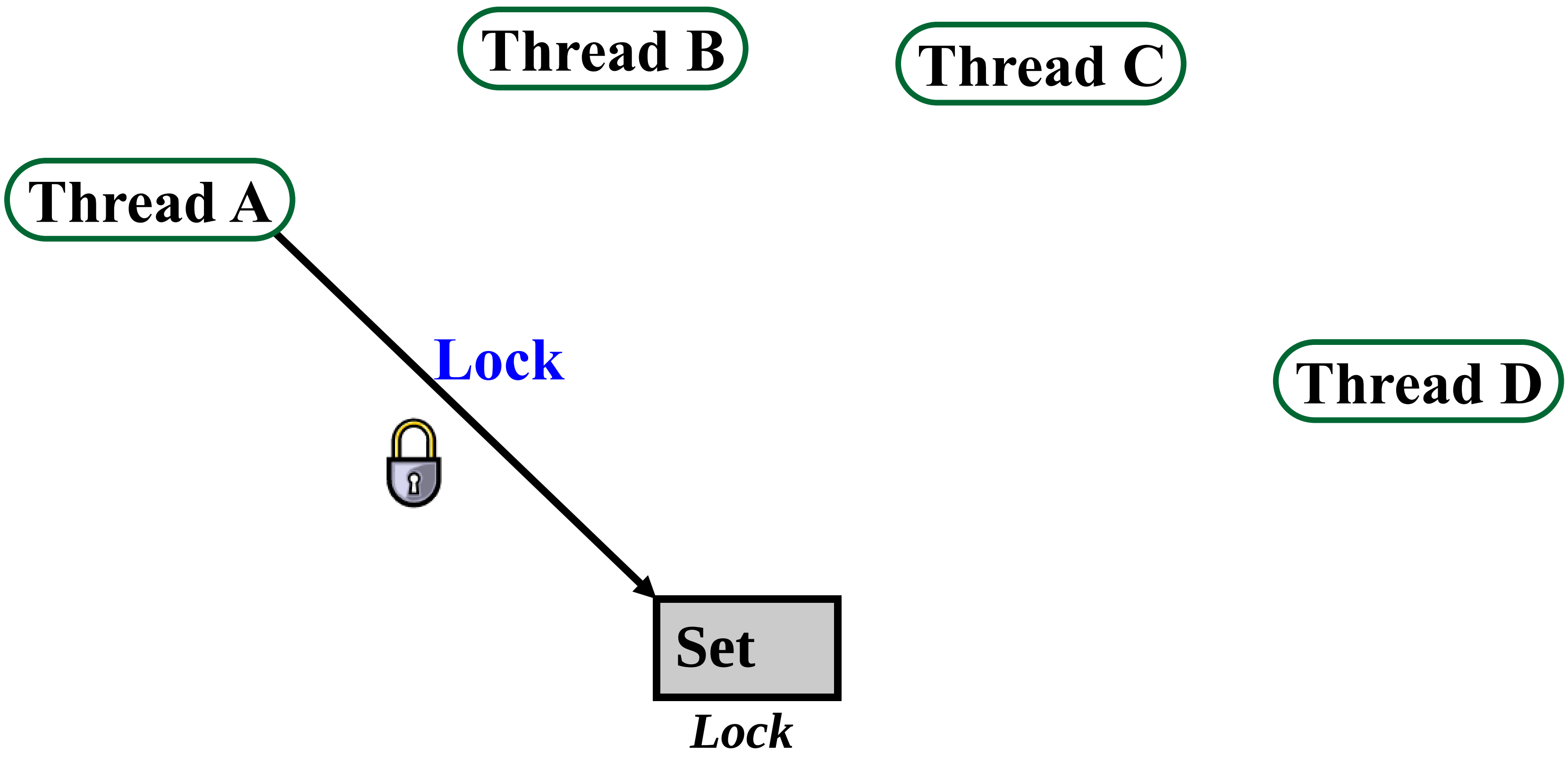
Acquiring and releasing locks



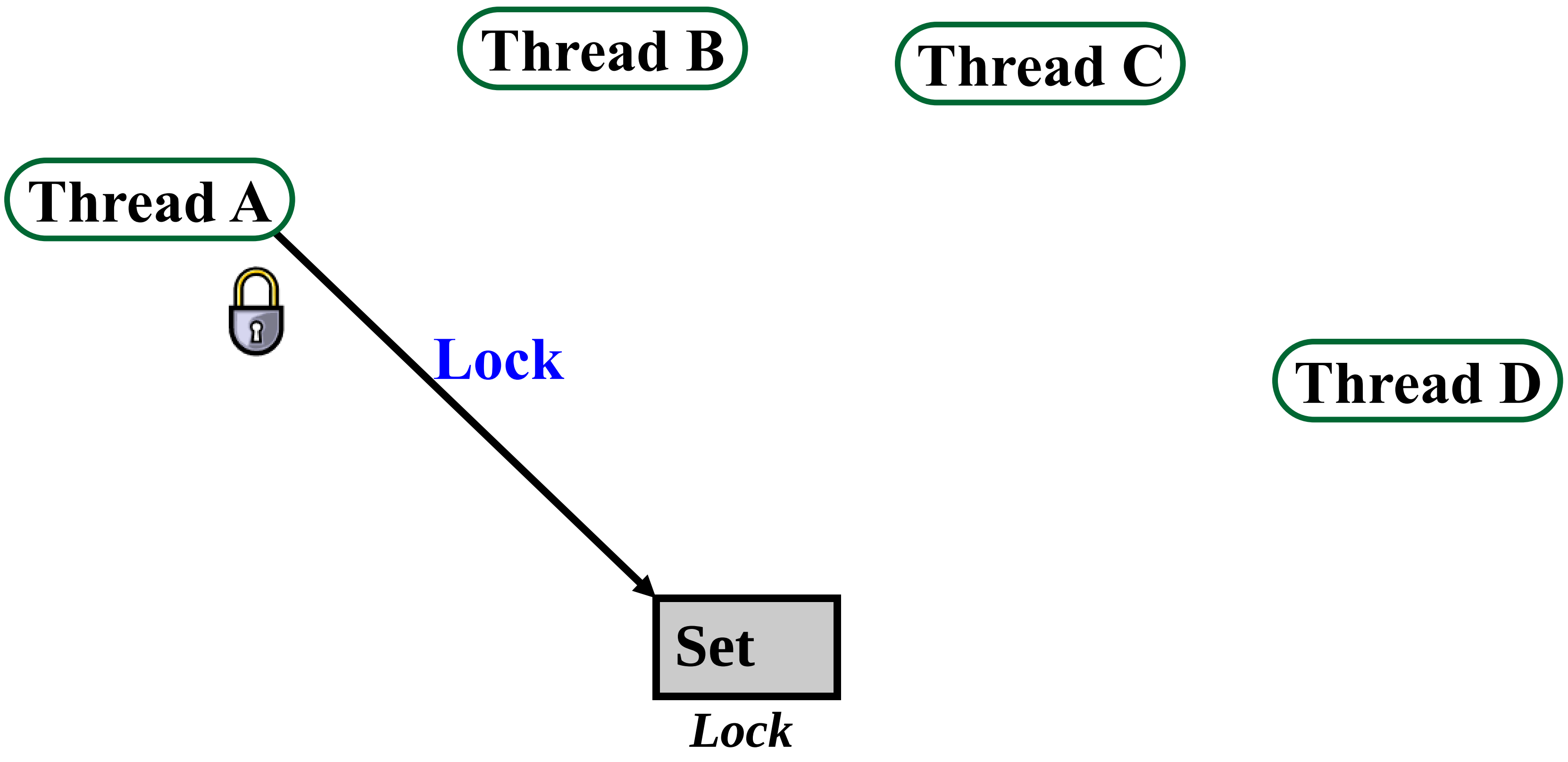
Acquiring and releasing locks



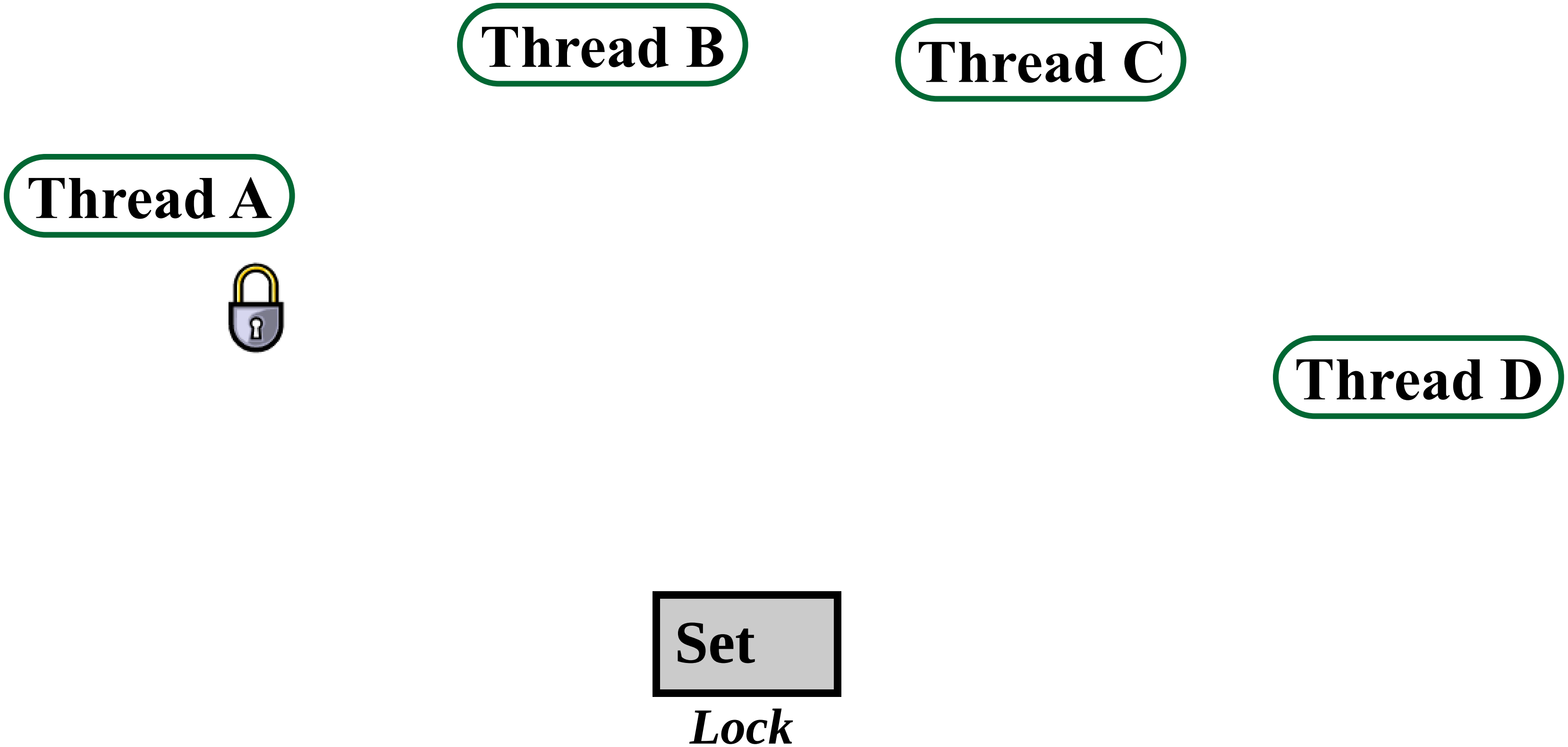
Acquiring and releasing locks



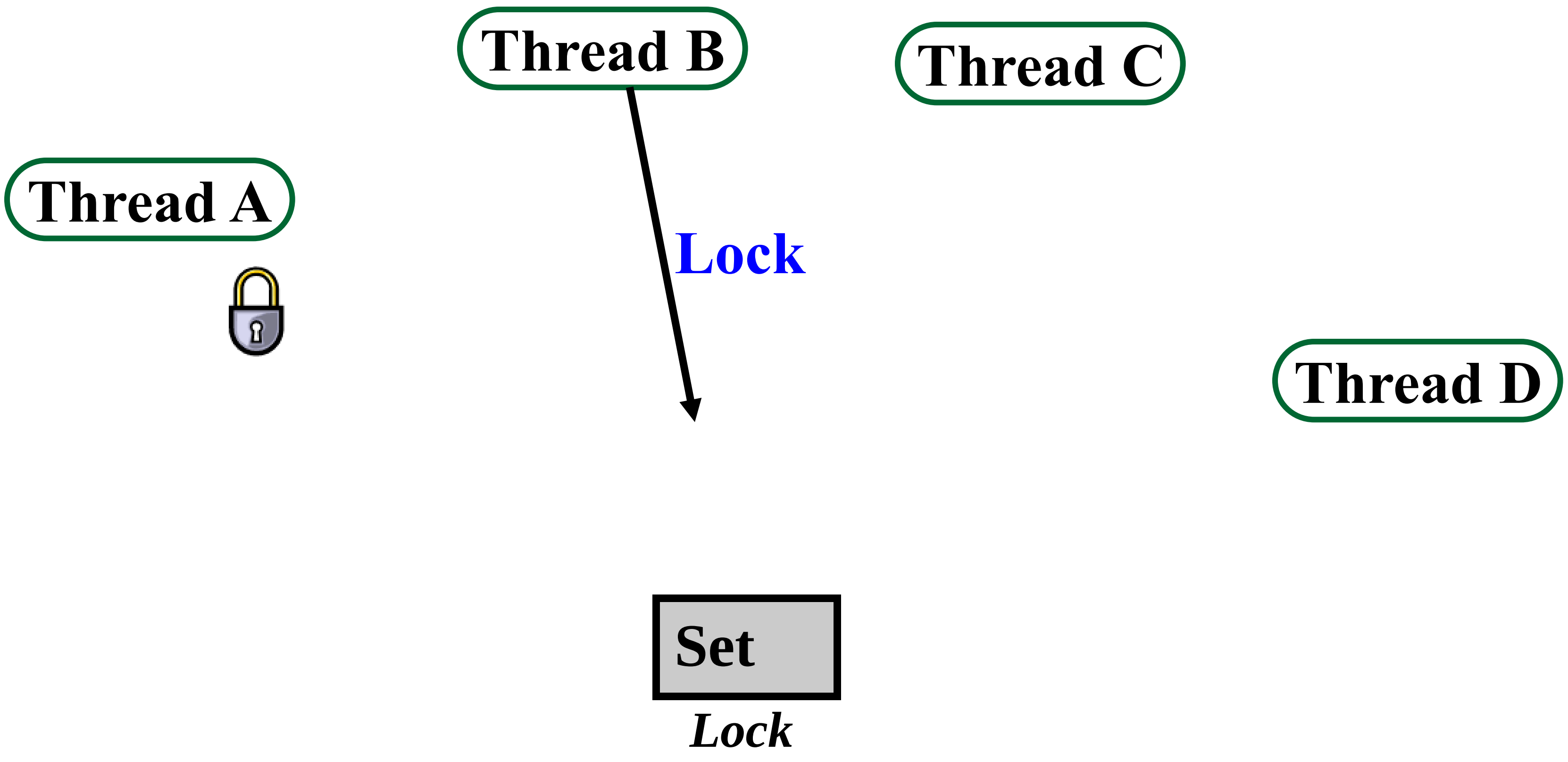
Acquiring and releasing locks



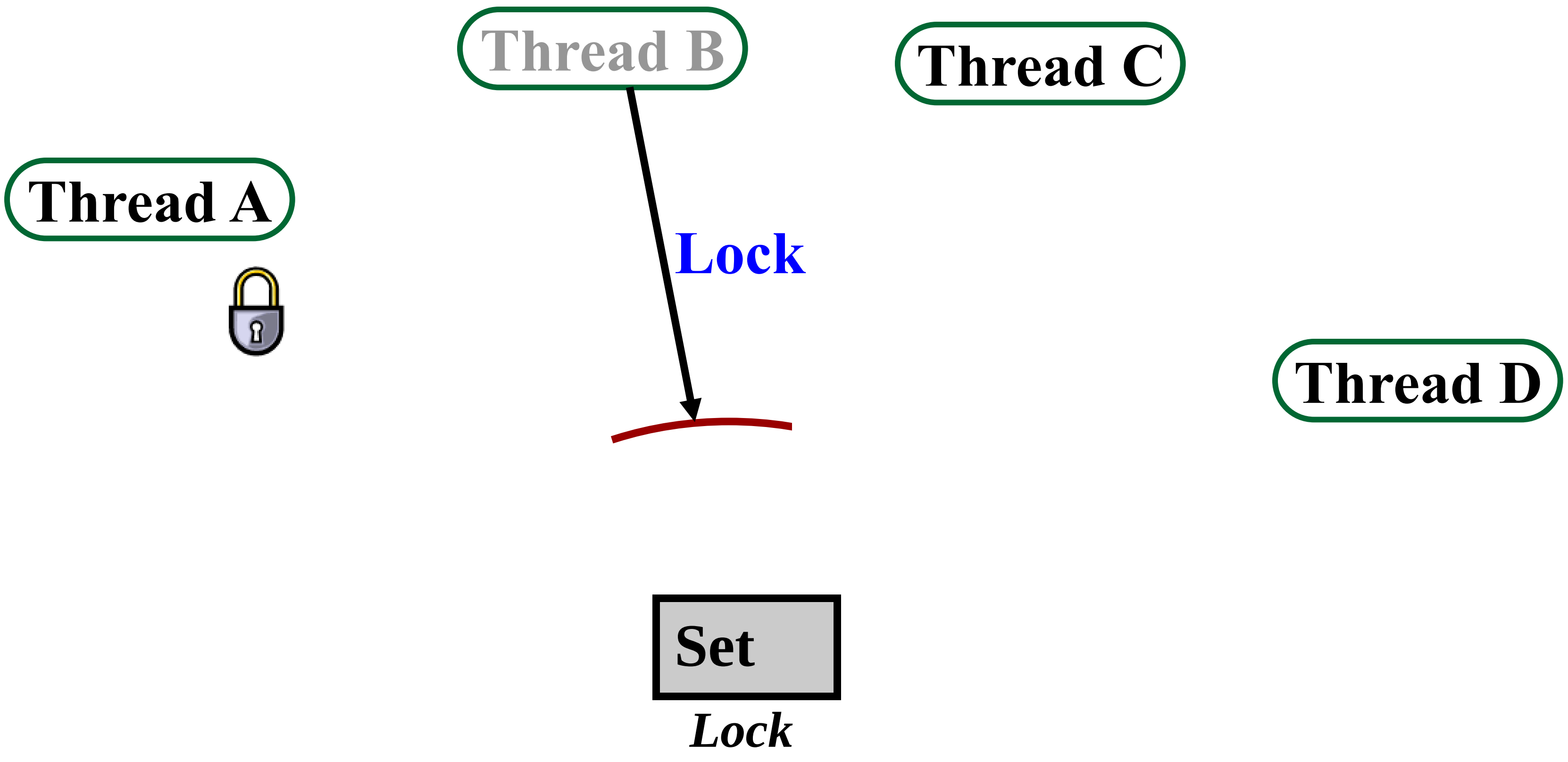
Acquiring and releasing locks



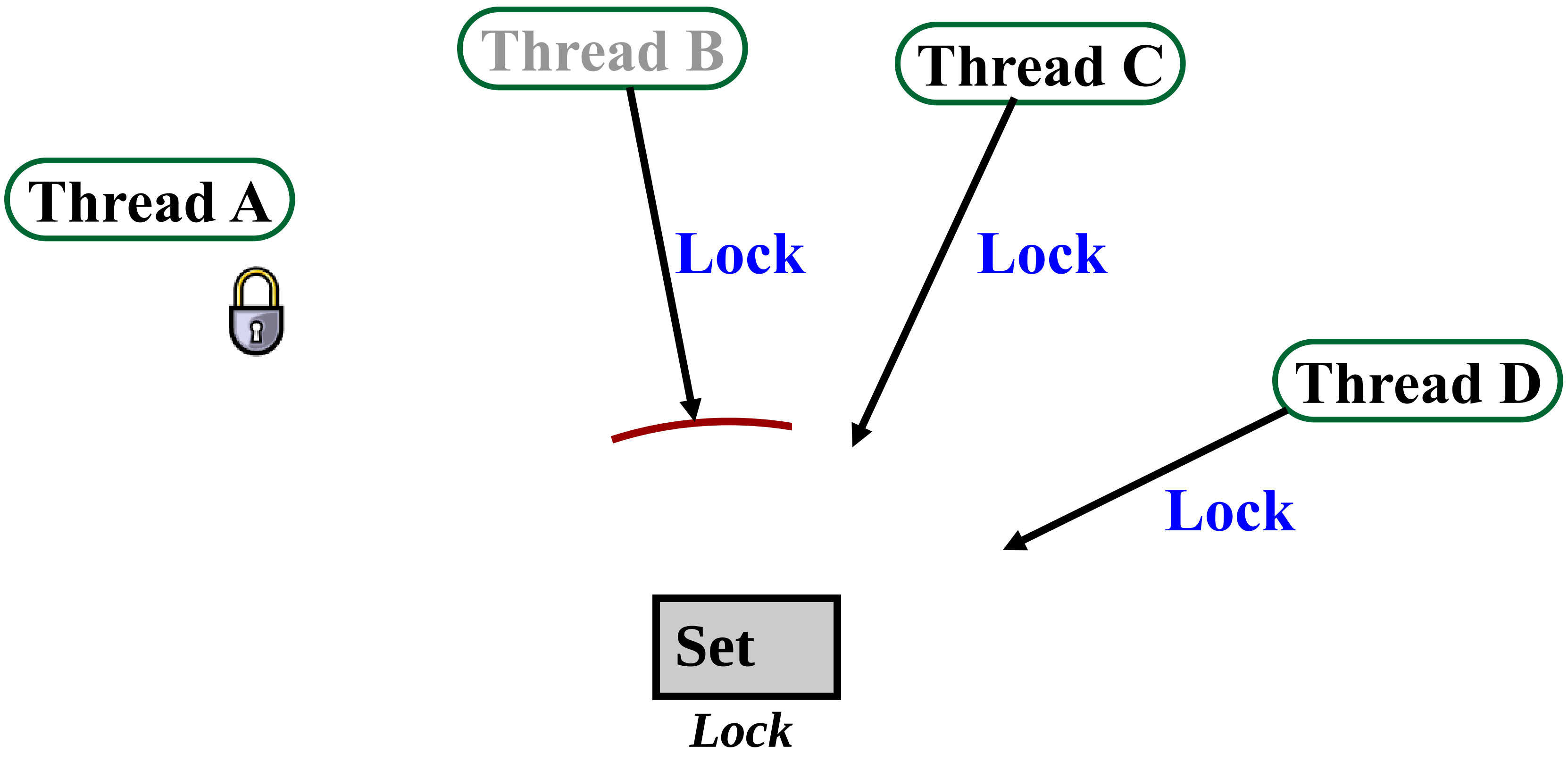
Acquiring and releasing locks



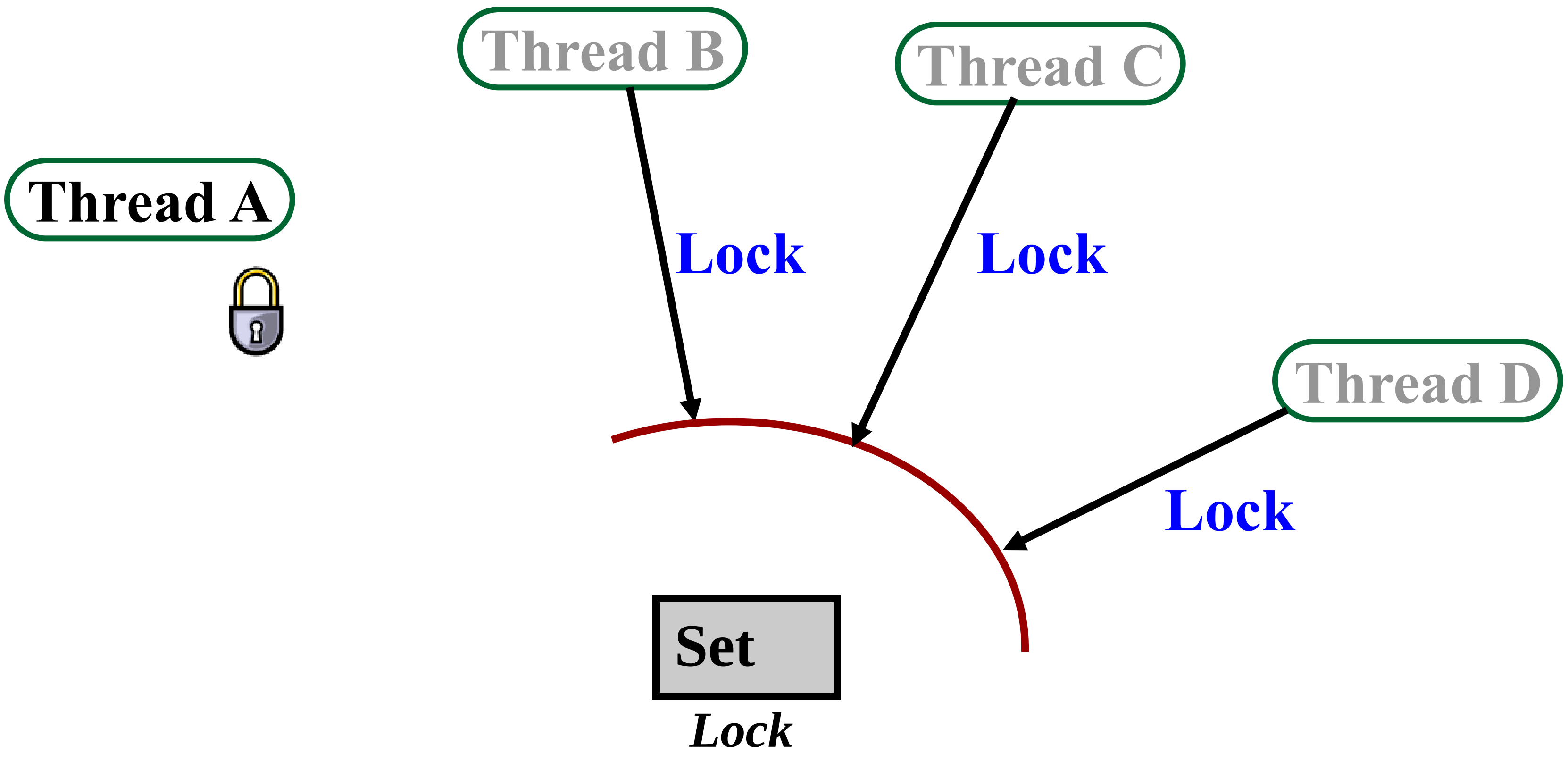
Acquiring and releasing locks



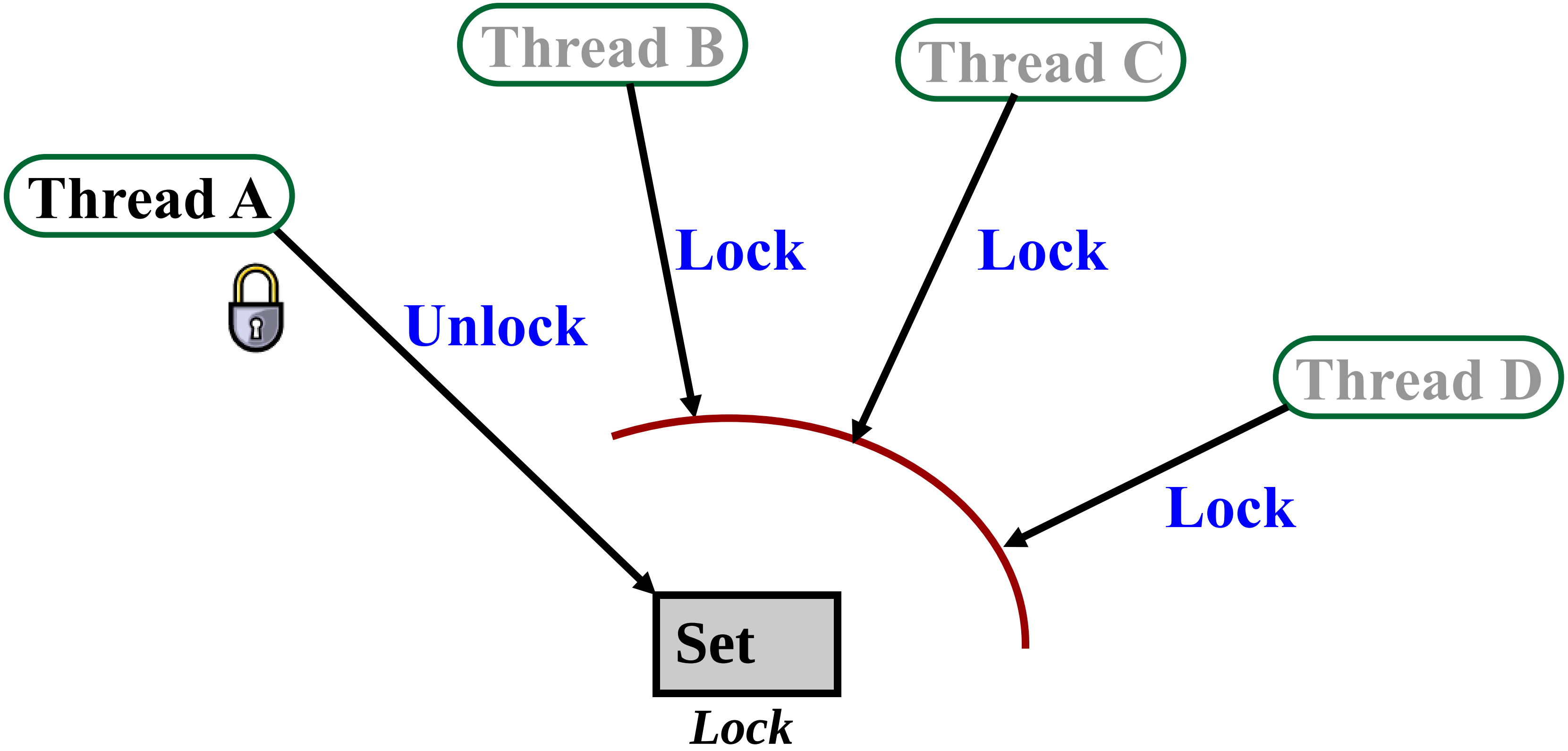
Acquiring and releasing locks



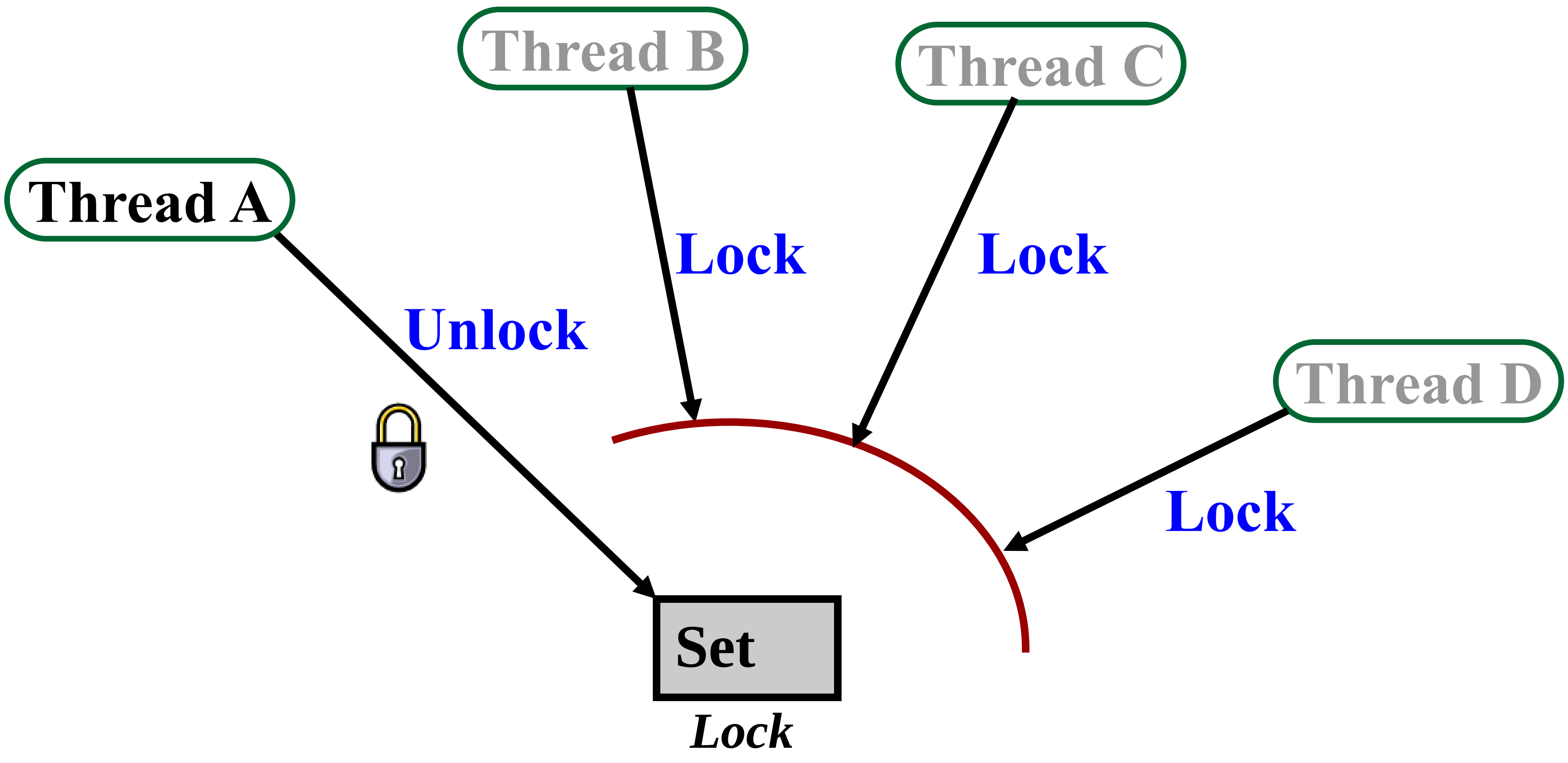
Acquiring and releasing locks



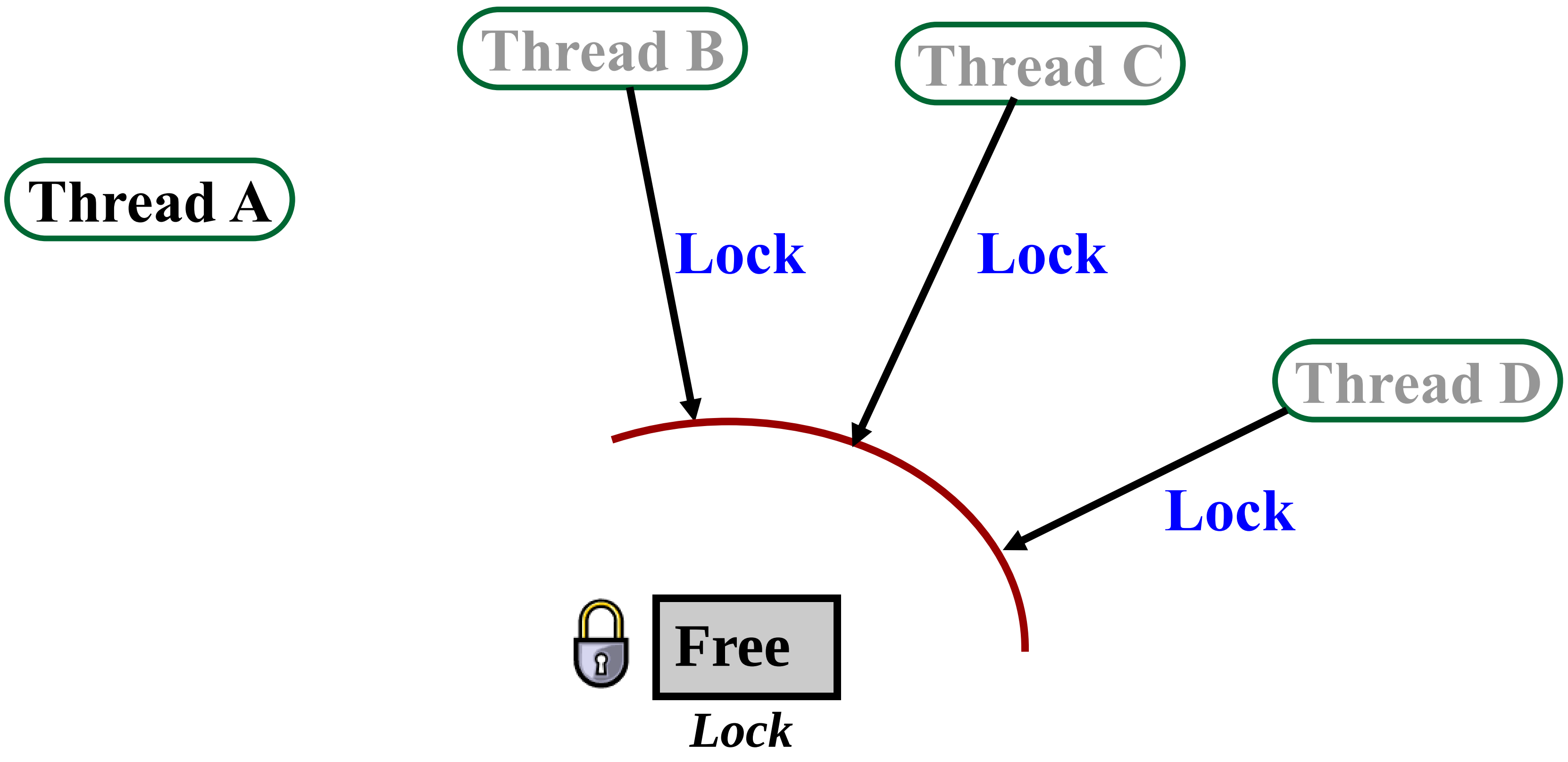
Acquiring and releasing locks



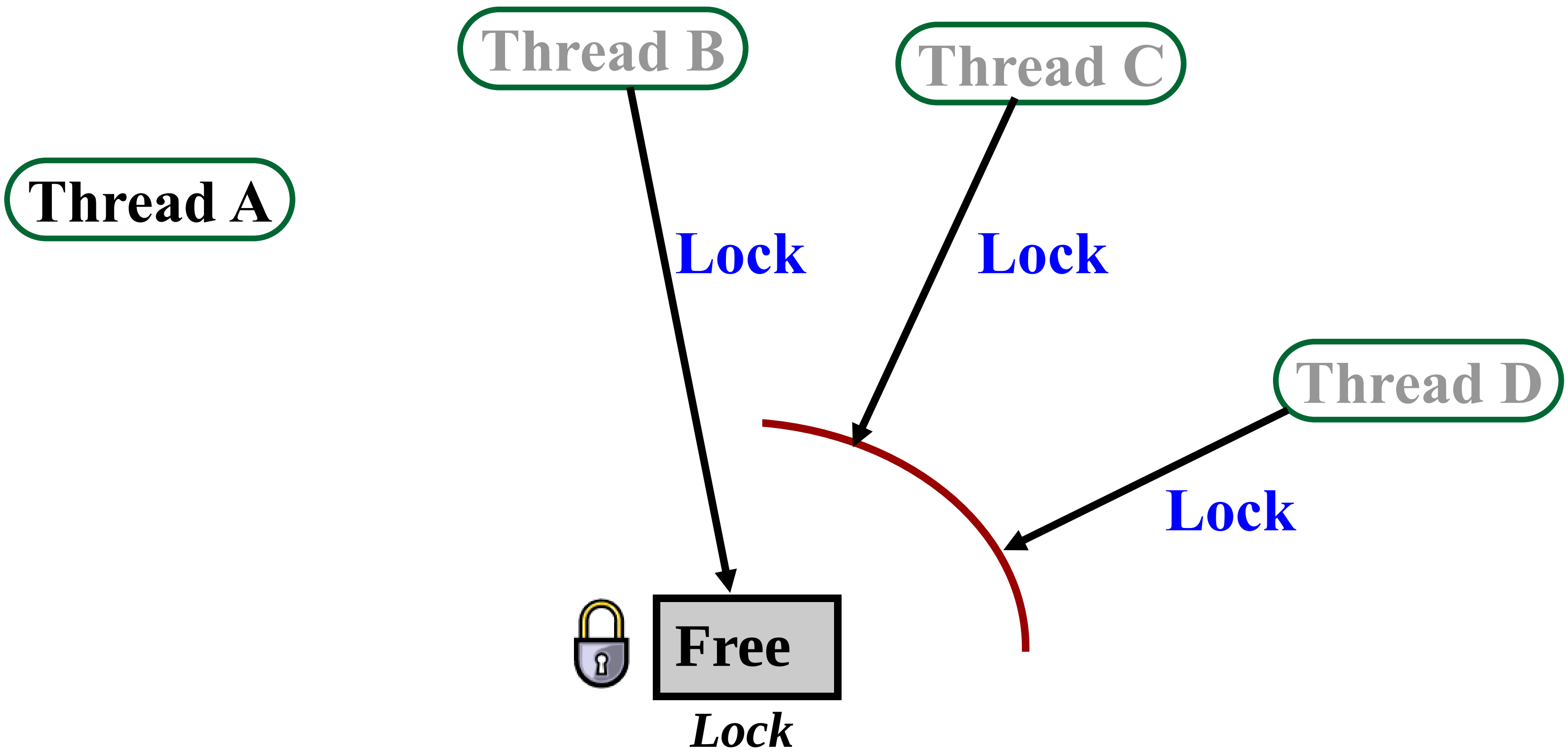
Acquiring and releasing locks



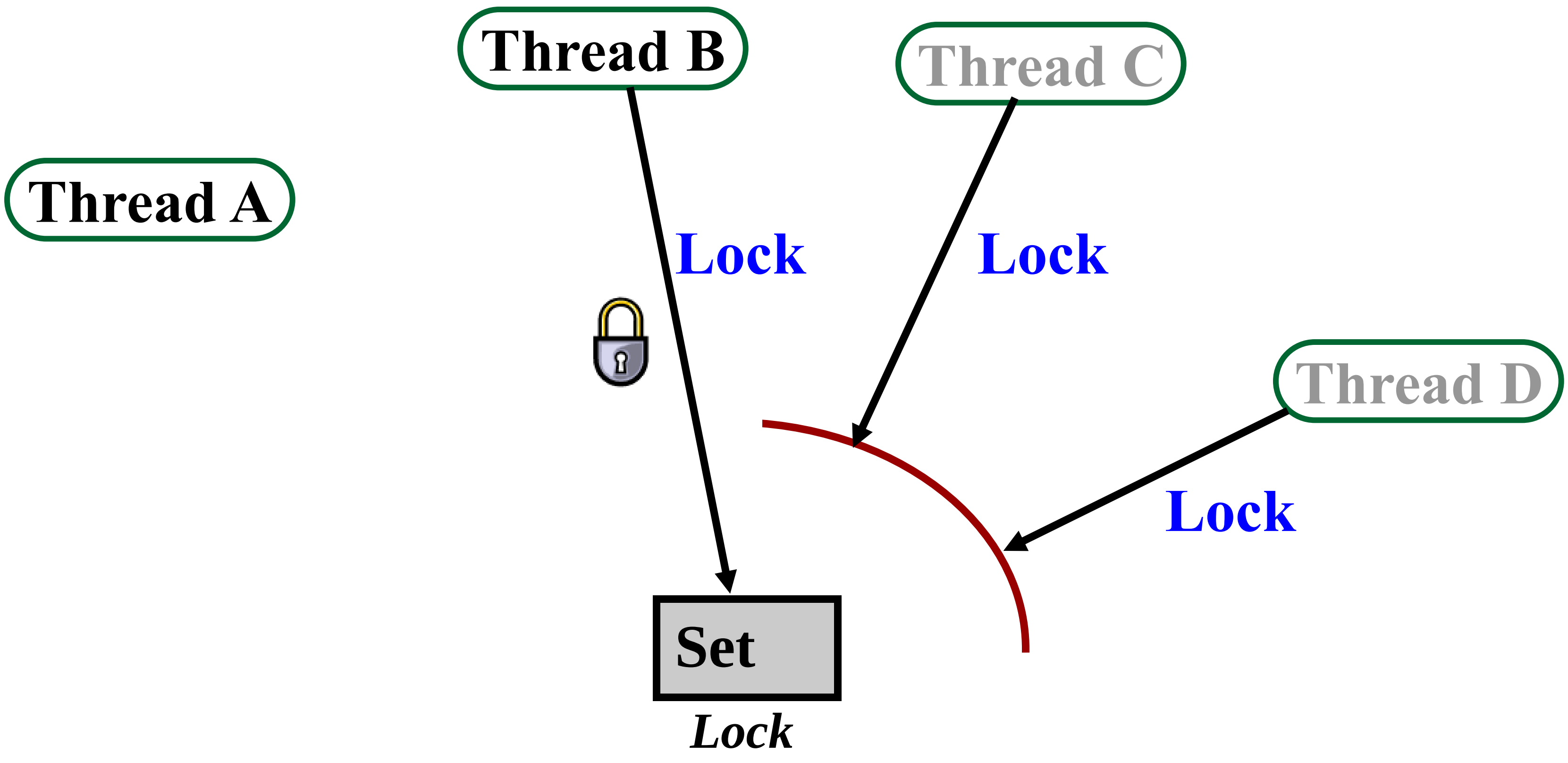
Acquiring and releasing locks



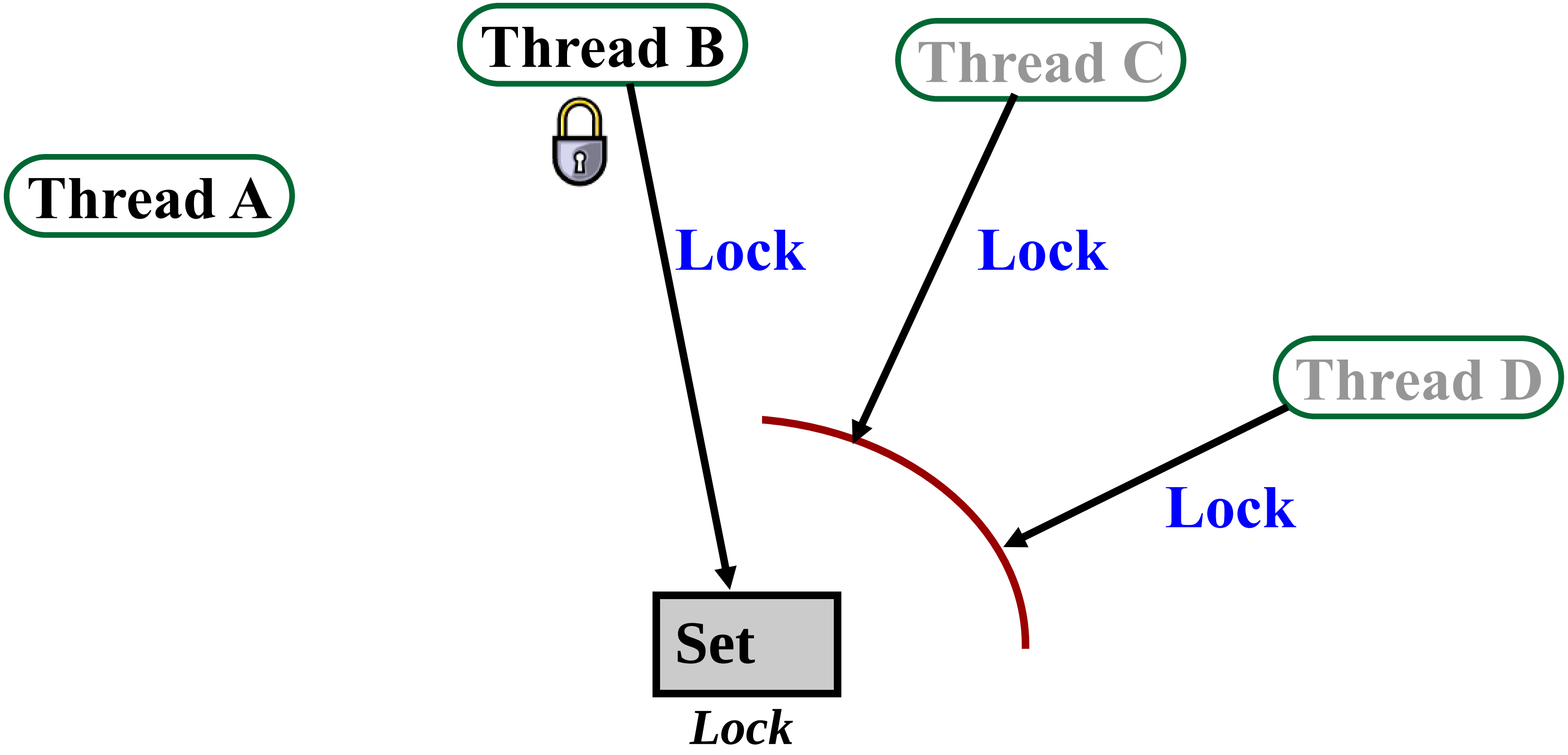
Acquiring and releasing locks



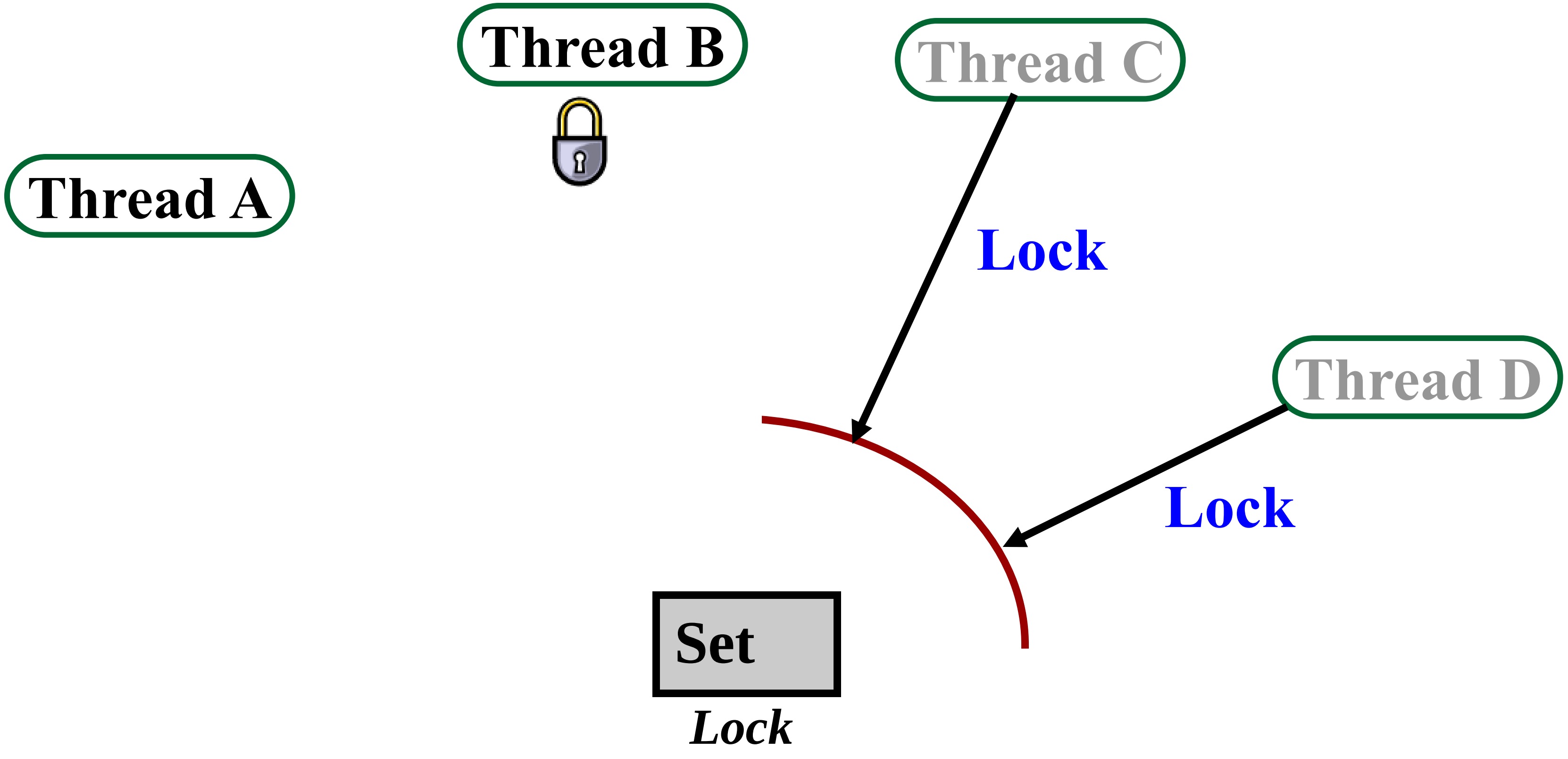
Acquiring and releasing locks



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?

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
```

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**

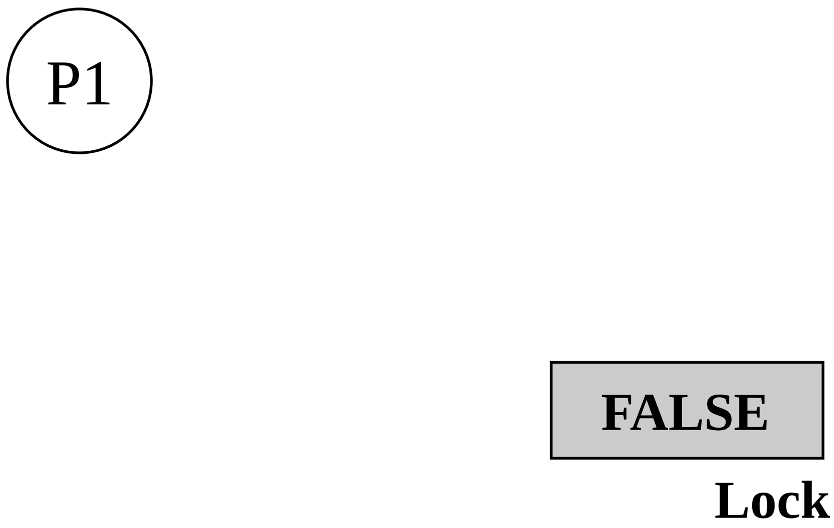
Test-and-set-lock instruction (TSL, tset)

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

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

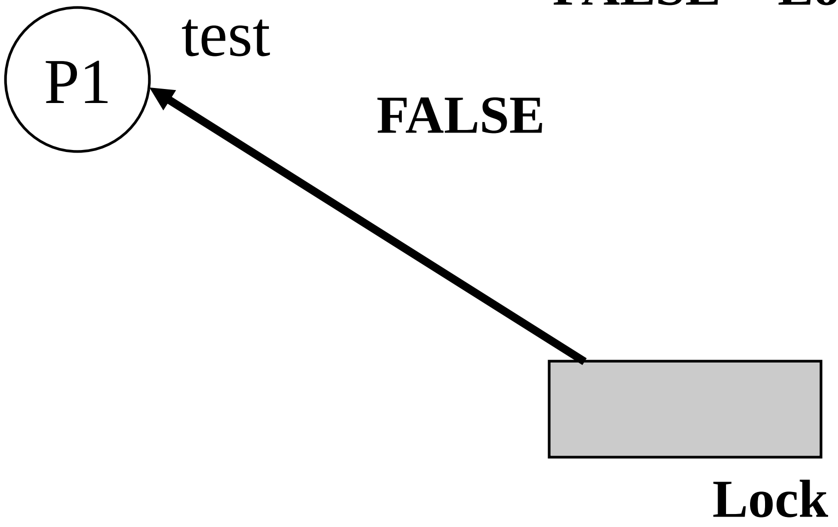
If the returned value was FALSE...
  Then you got the lock!!!
If the returned value was TRUE...
  Then someone else has the lock
  (so try again later)

Test and set lock

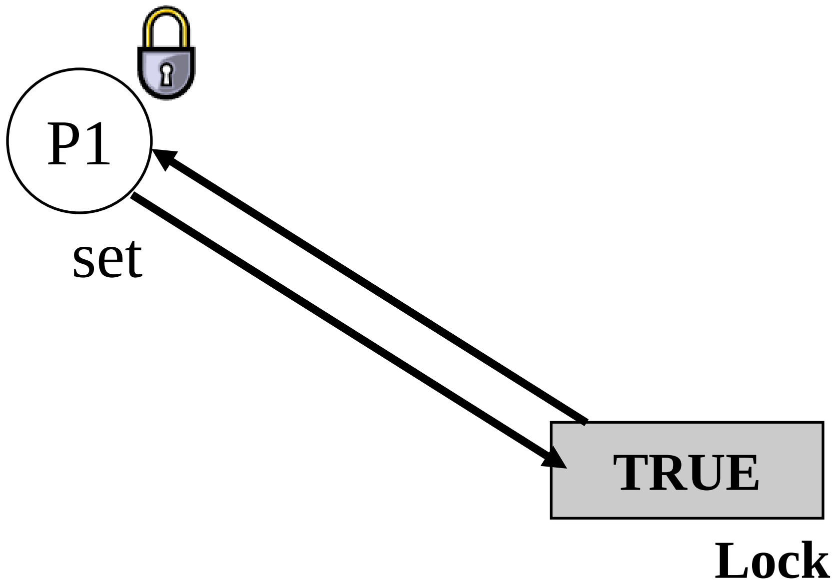


Test and set lock

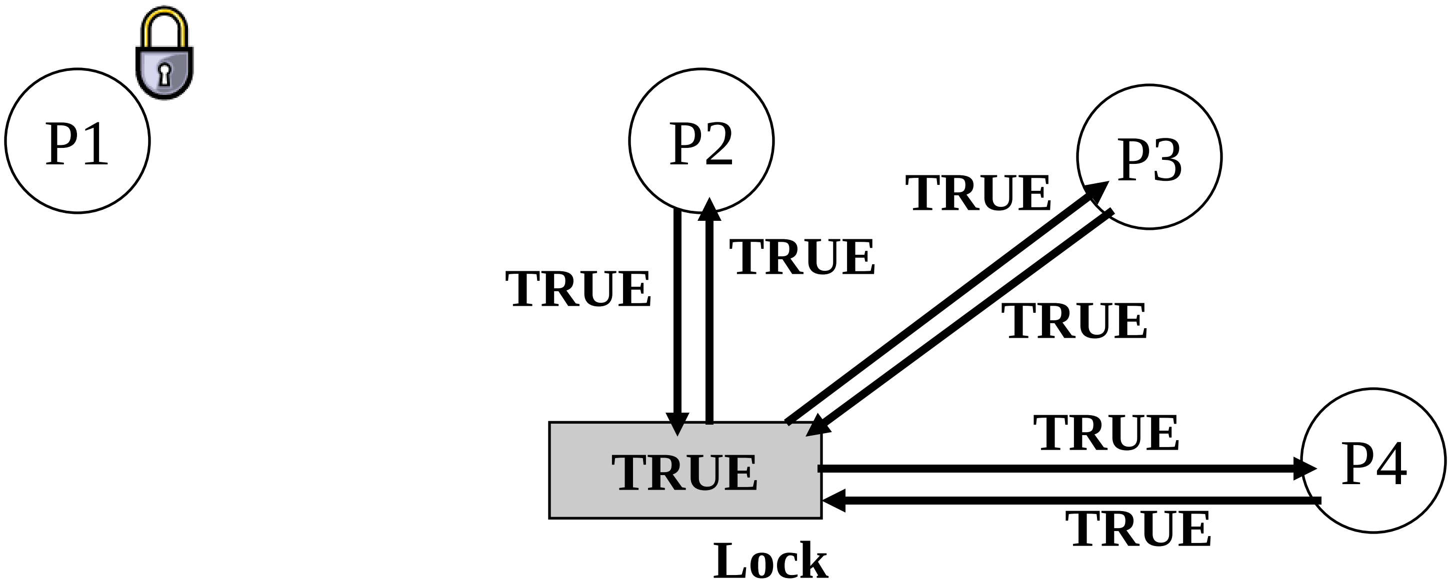
FALSE = Lock Available!!



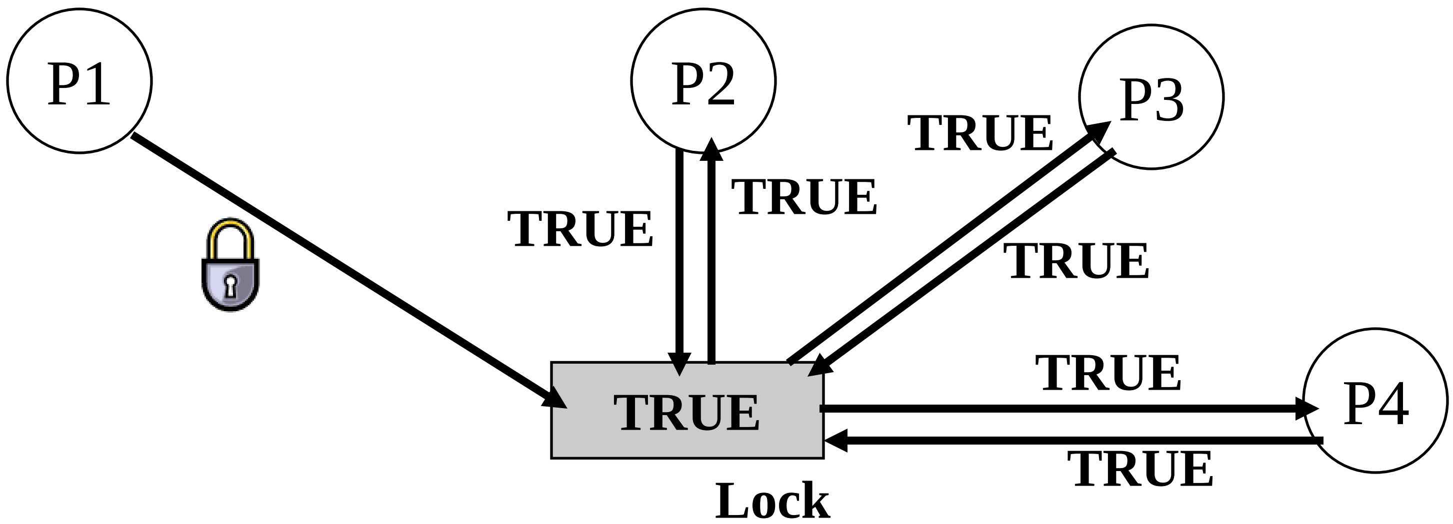
Test and set lock



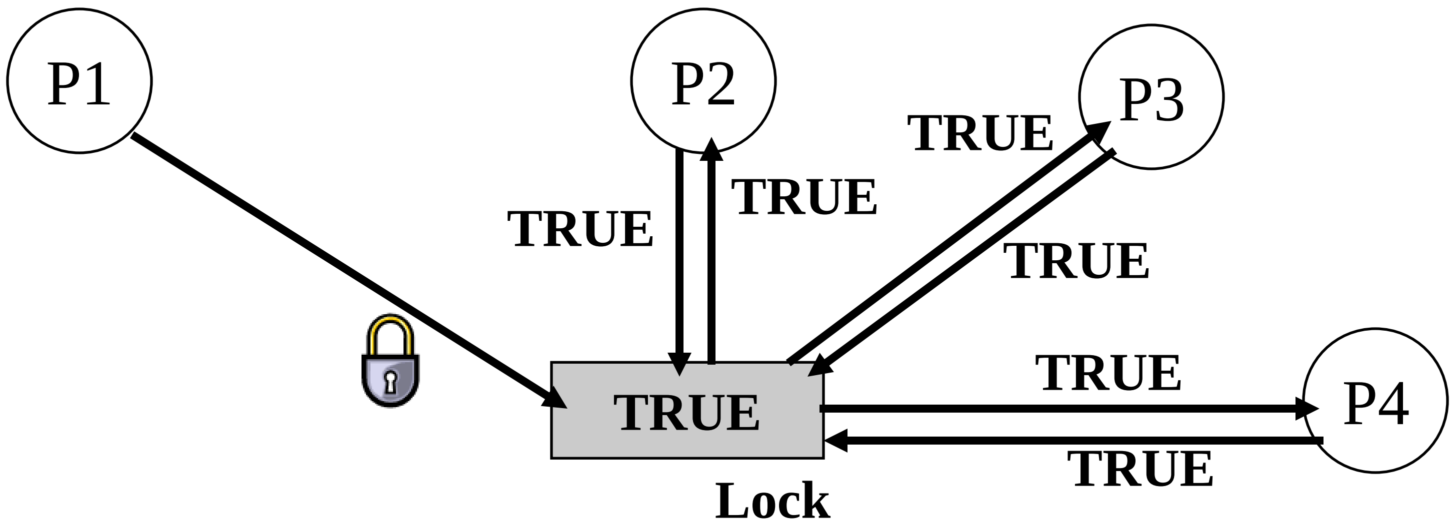
Test and set lock



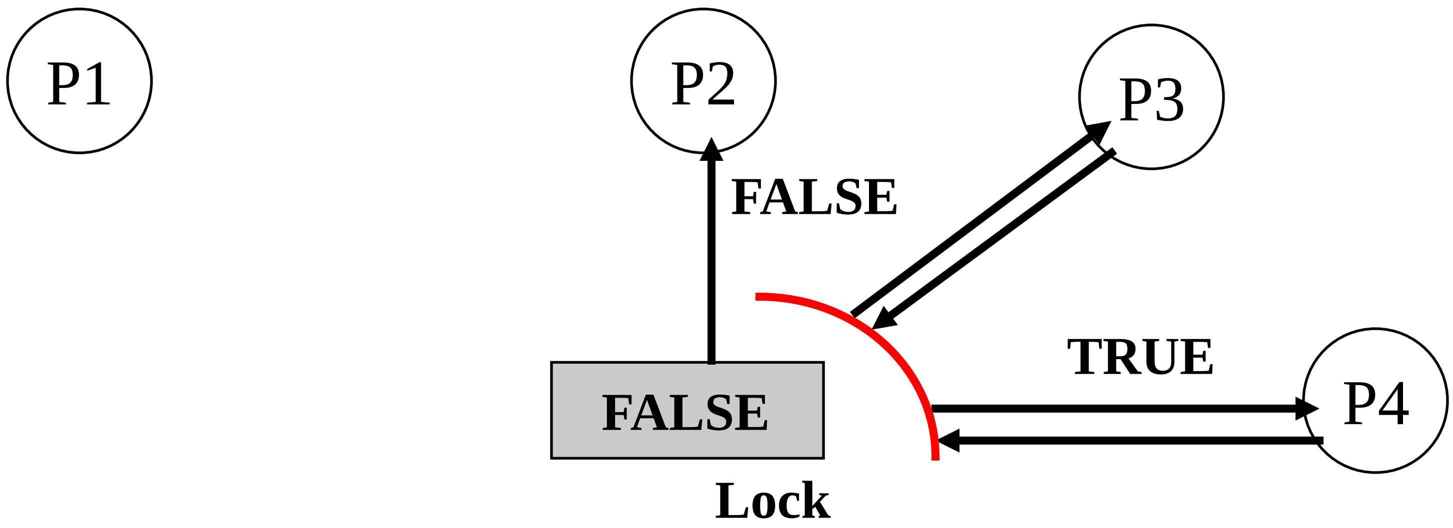
Test and set lock



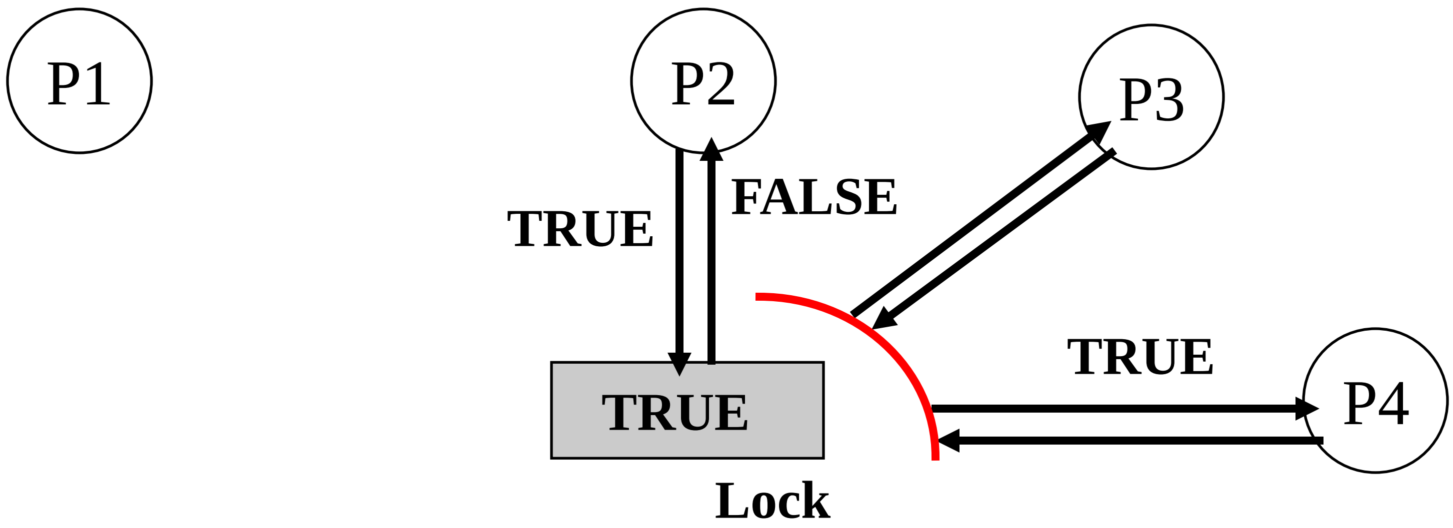
Test and set lock



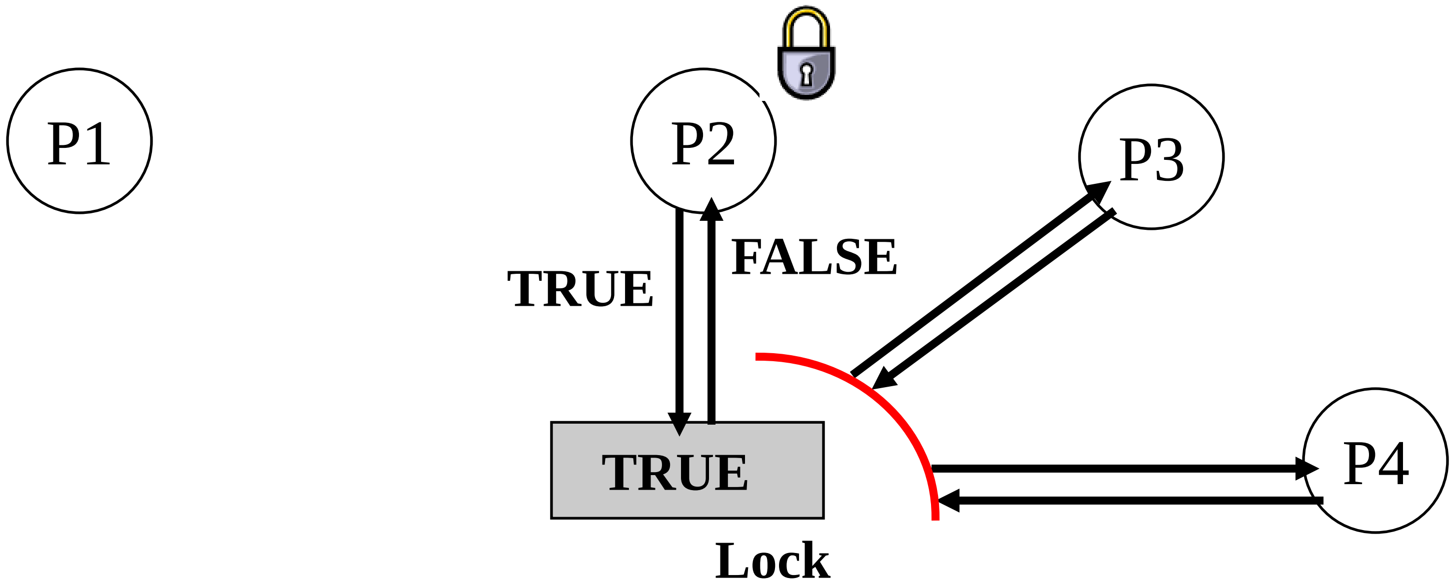
Test and set lock



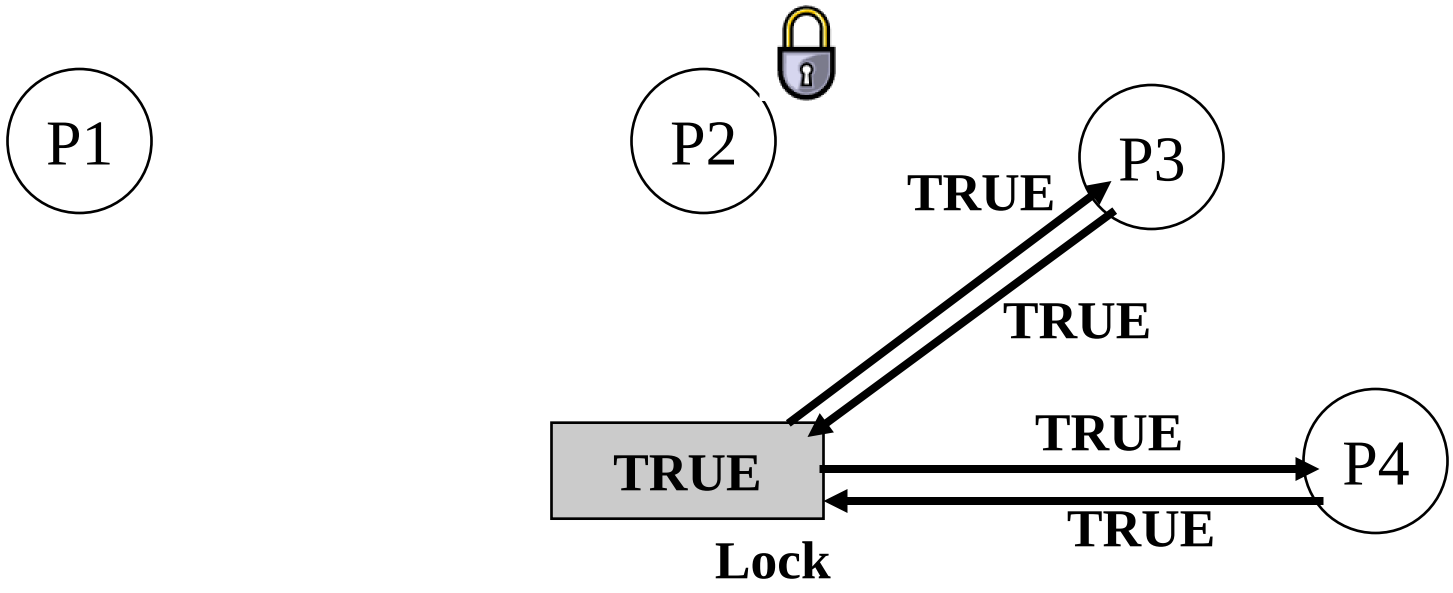
Test and set lock



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*

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*

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 !

An Example Synchronization Problem

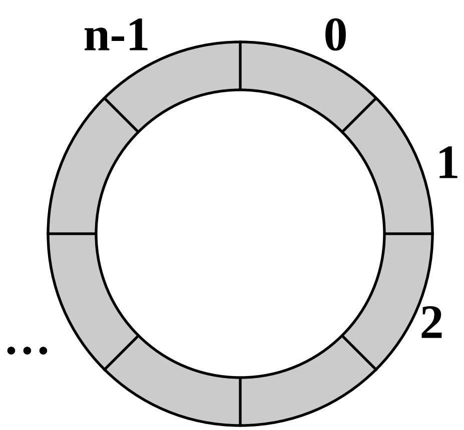
The Producer-Consumer Problem

- **An example of the pipelined model**
  - One thread produces data items
  - Another thread consumes them
- **Use a bounded buffer between the threads**
- **The buffer is a shared resource**
  - Code that manipulates it is a critical section
- **Must suspend the producer thread if the buffer is full**
- **Must suspend the consumer thread if the buffer is empty**

Is this busy-waiting solution correct?

```
thread producer {
    while(1){
      // Produce char c
      while (count==n) {
        no_op
      }
      buf[InP] = c
      InP = InP + 1 mod n
      count++
    }
}
```

```
thread consumer {
    while(1){
      while (count==0) {
        no_op
      }
      c = buf[OutP]
      OutP = OutP + 1 mod n
      count--
      // Consume char
    }
}
```




```
Global variables:
    char buf[n]
    int InP = 0    // place to add
    int OutP = 0   // place to get
    int count
```

This code is incorrect!

- **The "count" variable can be corrupted:**
  - Increments or decrements may be lost!
  - Possible Consequences:
    - **Both threads may spin forever**
    - **Buffer contents may be over-written**

- ***What is this problem called?***

This code is incorrect!

- **The "count" variable can be corrupted:**
  - Increments or decrements may be lost!
  - Possible Consequences:
    - **Both threads may sleep forever**
    - **Buffer contents may be over-written**

- ***What is this problem called?*** ***Race Condition***

- **Code that manipulates count must be made into a *???* and protected using *???***

This code is incorrect!

- **The "count" variable can be corrupted:**
  - Increments or decrements may be lost!
  - Possible Consequences:
    - **Both threads may sleep forever**
    - **Buffer contents may be over-written**

- ***What is this problem called? Race Condition***

- **Code that manipulates count must be made into a *critical section* and protected using *mutual exclusion!***

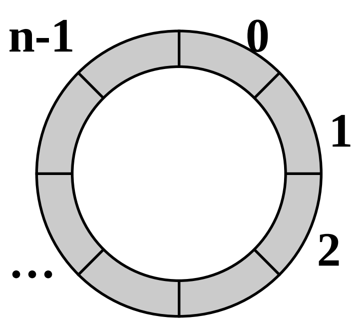
Some more problems with this code

- **What if buffer is full?**
  - Producer will busy-wait
  - On a single CPU system the consumer will not be able to empty the buffer
- **What if buffer is empty?**
  - Consumer will busy-wait
  - On a single CPU system the producer will not be able to fill the buffer
- **We need a solution based on blocking!**

Producer/Consumer with Blocking – 1st attempt

```
thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1)
      wakeup(empty)
    }
  }
```

```
thread consumer {
  while(1) {
    while (count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1)
      wakeup(full)
    // Consume char
  }
}
```

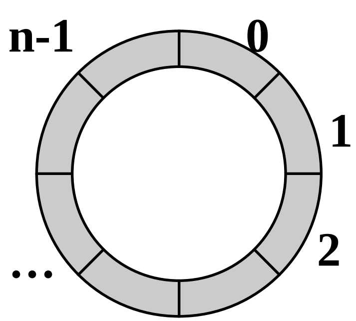



```
Global variables:
    char buf[n]
    int InP = 0    // place to add
    int OutP = 0   // place to get
    int count
```

Use a mutex to fix the race condition in this code

```
thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1)
      wakeup(empty)
    }
  }
```

```
thread consumer {
  while(1) {
    while (count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1)
      wakeup(full)
    // Consume char
  }
}
```




```
Global variables:
    char buf[n]
    int InP = 0   // place to add
    int OutP = 0  // place to get
    int count
```

Problems

- Sleeping while holding the mutex causes deadlock !
- Releasing the mutex then sleeping opens up a window during which a context switch might occur … again risking deadlock
- How can we release the mutex and sleep in a single atomic operation?
- We need a more powerful synchronization primitive

Semaphores

- **An abstract data type that can be used for condition synchronization and mutual exclusion**

What is the difference between mutual exclusion and condition synchronization?

Semaphores

- **An abstract data type that can be used for condition synchronization and mutual exclusion**

- **Mutual exclusion**
  - only one at a time in a critical section

- **Condition synchronization**
  - wait until invariant holds before proceeding
  - signal when invariant holds so others may proceed

Semaphores

- **An abstract data type**
  - containing an integer variable (S)
  - Two operations: Wait (S) and Signal (S)

- **Alternative names for the two operations**
  - *Wait(S) = Down(S) = P(S)*
  - *Signal(S) = Up(S) = V(S)*

- **Current version of Blitz uses Down/Up as names for its semaphore operations**

Semaphores

- Wait (S)
  - decrement S by 1
  - test value of S
  - if S is negative sleep until signaled

- Signal (S)
  - increment S by 1
  - signal/wakeup a waiting thread

- **Both Wait () and Signal () are assumed to be *atomic!!!***
  - A kernel implementation must ensure atomicity

Variation: Binary Semaphores

- **Counting Semaphores**
  - same as just "semaphore"
- **Binary Semaphores**
  - a specialized use of semaphores
  - the semaphore is used to implement a *Mutex Lock*

Variation: Binary Semaphores

- **Counting Semaphores**
  - same as just "semaphore"

- **Binary Semaphores**
  - a specialized use of semaphores
  - the semaphore is used to implement a *Mutex Lock*
  - the count will always be either

    <=0 = locked

    1 = unlocked

Using Semaphores for Mutex

semaphore **mutex = 1** **-- unlocked**

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread A

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread B

Using Semaphores for Mutex

```
semaphore mutex = 0          -- locked
```

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread A

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread B

Using Semaphores for Mutex

semaphore mutex = 0 --locked

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread A

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread B

Using Semaphores for Mutex

```
semaphore mutex = 0          -- locked
```

Thread A

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread B

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Using Semaphores for Mutex

semaphore **mutex = 0** -- locked

```
repeat
   wait(mutex);
   critical section
   signal(mutex);
   remainder section
until FALSE
```

Thread A

```
repeat
   wait(mutex);
   critical section
   signal(mutex);
   remainder section
until FALSE
```

Thread B

Using Semaphores for Mutex

`semaphore mutex = 1          -- unlocked`

This thread can now be released!

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread A

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread B

Using Semaphores for Mutex

```
semaphore mutex = 0          -- locked
```

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread A

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread B

Exercise: producer/consumer with semaphores

```
Global variables
  semaphore full_buffs = ?;
  semaphore empty_buffs = ?;
  char buff[n];
  int InP, OutP;
```

```
thread producer {
   while(1){
      // Produce char c...
      buf[InP] = c
      InP = InP + 1 mod n
   }
}
```

```
thread consumer {
   while(1){
      c = buf[OutP]
      OutP = OutP + 1 mod n
      // Consume char...
   }
}
```

Counting semaphores in producer/consumer

```
Global variables
  semaphore full_buffs = 0;
  semaphore empty_buffs = n;
  char buff[n];
  int InP, OutP;
```

```
thread producer {
  while(1){
    // Produce char c...
    wait(empty_buffs)
    buf[InP] = c
    InP = InP + 1 mod n
    signal(full_buffs)
  }
}
```

```
thread consumer {
  while(1){
    wait(full_buffs)
    c = buf[OutP]
    OutP = OutP + 1 mod n
    signal(empty_buffs)
    // Consume char...
  }
}
```

Continue next time …

Implementing semaphores

- Wait () and Signal () are assumed to be atomic

Implementing semaphores

- **Wait () and Signal () are assumed to be atomic**

  How can we ensure that they are atomic?

Implementing semaphores

- **Wait () and Signal () are assumed to be atomic**

  How can we ensure that they are atomic?

- **Implement Wait() and Signal() as system calls?**
  - how can the kernel ensure Wait() and Signal() are completed atomically?
  - Same solutions as before
    - **Disable interrupts, or**
    - **Use TSL-based mutex**

Blitz code for Semaphore.wait

```
method Wait ()
   var oldIntStat: int
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait‘
        operation")
   EndIf
   count = count – 1
   if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.wait

```
method Wait ()
   var oldIntStat: int
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait'
        operation")
   EndIf
   count = count - 1
   if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.wait

```
method Wait ()
   var oldIntStat: int
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait`
        operation")
   EndIf
   count = count – 1
   if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.wait

```
method Wait ()
   var oldIntStat: int
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait`
        operation")
   EndIf
   count = count – 1
   if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.signal

```
method Signal ()
   var oldIntStat: int
   t: ptr to Thread
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during
        'Signal' operation")
   endIf
   count = count + 1
   if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.signal

```
method Signal ()
   var oldIntStat: int
   t: ptr to Thread
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during
        'Signal' operation")
   endIf
   count = count + 1
   if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.signal

```
method Signal ()
   var oldIntStat: int
   t: ptr to Thread
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during
        'Signal' operation")
   endIf
   count = count + 1
   if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.signal

```
method Signal ()
   var oldIntStat: int
   t: ptr to Thread
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during
        'Signal' operation")
   endIf
   count = count + 1
   if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

But what is currentThread.Sleep ()?

- **If sleep stops a thread from executing, how, where, and when does it return?**
  - which thread enables interrupts following sleep?
  - the thread that called sleep shouldn't return until another thread has called signal !
  - … but how does that other thread get to run?
  - … where exactly does the thread switch occur?

- **Trace down through the Blitz code until you find a call to switch()**
  - Switch is called in one thread but returns in another!
  - See where registers are saved and restored

Semaphores using atomic instructions

- **As we saw earlier, hardware provides special atomic instructions for synchronization**
  - test and set lock (TSL)
  - compare and swap (CAS)
  - etc

- **Semaphore can be built using atomic instructions**
  1. build mutex locks from atomic instructions
  2. build semaphores from mutex locks

Building *yielding* mutex locks using TSL

```
Mutex_lock:
     TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
     CMP REGISTER,#0             | was mutex zero?
     JZE ok                      | if it was zero, mutex is unlocked, so return
     CALL thread_yield           | mutex is busy, so schedule another thread
     JMP mutex_lock              | try again later
 Ok: RET                         | return to caller; enter critical section

 Mutex_unlock:
     MOVE MUTEX,#0               | store a 0 in mutex
     RET                         | return to caller
```

Building *spinning* mutex locks using TSL

```
Mutex_lock:
     TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
     CMP REGISTER,#0         | was mutex zero?
     JZE ok                  | if it was zero, mutex is unlocked, so return
     CALL thread_yield       | mutex is busy, so schedule another thread
     JMP mutex_lock          | try again later
 Ok: RET                     | return to caller; enter critical section

 Mutex_unlock:
     MOVE MUTEX,#0           | store a 0 in mutex
     RET                     | return to caller
```

To block or not to block?

- **Spin-locks do *busy waiting***
  - wastes CPU cycles on uni-processors
  - Why?

- **Blocking locks put the thread to *sleep***
  - may waste CPU cycles on multi-processors
  - Why?

Building semaphores using mutex locks

Problem: Implement a counting semaphore

Signal ()

Wait ()

...using just Mutex locks

How about two "blocking" mutex locks?

```
var cnt: int = 0              -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = locked     -- Locked when waiting
```

Wait ():

```
Lock(m1)
cnt = cnt - 1
if cnt<0
  Unlock(m1)
  Lock(m2)
else
  Unlock(m1)
endIf
```

Signal():

```
Lock(m1)
cnt = cnt + 1
if cnt<=0
  Unlock(m2)
endIf
Unlock(m1)
```

How about two "blocking" mutex locks?

```
var cnt: int = 0              -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = locked      -- Locked when waiting
```

Wait ():

```
Lock(m1)
cnt = cnt – 1
if cnt<0
  Unlock(m1)
  Lock(m2)
else
  Unlock(m1)
endIf
```

Signal():

```
Lock(m1)
cnt = cnt + 1
if cnt<=0
  Unlock(m2)
endIf
Unlock(m1)
```

Contains a Race Condition!

Oops! How about this then?

```
var cnt: int = 0               -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = locked      -- Locked when waiting
```

Wait ():

```
Lock(m1)
cnt = cnt - 1
if cnt<0
  Lock(m2)
  Unlock(m1)
else
  Unlock(m1)
endIf
```

Signal():

```
Lock(m1)
cnt = cnt + 1
if cnt<=0
  Unlock(m2)
endIf
Unlock(m1)
```

Oops! How about this then?

```
var cnt: int = 0              -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = locked      -- Locked when waiting
```

Wait ():

```
Lock(m1)
cnt = cnt - 1
if cnt<0
   Lock(m2)
   Unlock(m1)
else
   Unlock(m1)
endIf
```

Signal():

```
Lock(m1)
cnt = cnt + 1
if cnt<=0
   Unlock(m2)
endIf
Unlock(m1)
```

Contains a Deadlock!

Ok! Lets have another try!

```
var cnt: int = 0            -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = unlocked -- Locked when waiting
```

Wait ():

```
Lock(m2)
Lock(m1)
cnt = cnt - 1
if cnt>0
  Unlock(m2)
endIf
Unlock(m1)
```

Signa;():

```
Lock(m1)
cnt = cnt + 1
if cnt=1
  Unlock(m2)
endIf
Unlock(m1)
```

… is this solution valid?

What about this solution?

```
Mutex m1, m2;   // binary semaphores
int C = N;      // N is # locks
int W = 0;      // W is # wakeups
```

```
Wait():
  Lock(m1);
  C = C - 1;
  if (C<0)
    Unlock(m1);
    Lock(m2);
    Lock(m1);
    W = W - 1;
    if (W>0)
       Unlock(m2);
    endif;
  else
    Unlock(m1);
  endif;
```

```
Signal():
  Lock(m1);
  C = C + 1;
  if (C<=0)
    W = W + 1;
    Unlock(m2);
  endif;
  Unlock(m1);
```

Quiz

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