COSC 4600
Operating Systems Design

Spring 2019

Chapter 6: Process Synchronization in Programming

pthread Synchronization APIs (1)

- Mutex – a mutex variable acts like a "lock" protecting access to a shared data resource.
  - Create and initialize a mutex variable; `pthread_mutex_t mutex;`
    `pthread_mutex_init(&mutex, &attr);`
  - Several threads try to lock the mutex; `pthread_mutex_trylock(&mutex);`
    `pthread_mutex_lock(&mutex);`
  - Only one succeeds and owns the mutex;
  - The owner thread accesses shared data;

If lock is unavialbe, the current thread will be not be blocked

  - The owner thread unlocks the mutex; `pthread_mutex_unlock(&mutex);`
  - Another thread acquires the mutex and repeats the process;
  - Finally the mutex is destroyed `pthread_mutex_destroy(&mutex);`

Thread Synchronization APIs (2)

- Mutex (cont) – avoiding deadlocks
  - Deadlocks occur when a thread try to relock a mutex it owns

```
f( ) {
        pthread_mutex_lock ( & mutex);
        …
        g( );
        …
        pthread_mutex_unlock();
}
```

```
g( ) {
        pthread_mutex_lock ( & mutex);
        …
        pthread_mutex_unlock();
}
```

  - Deadlocks may also occur when mutexes are locked in reverse order

```
/* Thread A */
pthread_mutex_lock(&mutex1);
pthread_mutex_lock(&mutex2);
```

```
/* Thread B */
pthread_mutex_lock(&mutex2);
pthread_mutex_lock(&mutex1);
```

Notes: To avoid this, successive mutexes should be locked in the same order.

Thread Synchronization APIs (3)

- Condition variable
  - Condition variables allow threads to synchronize based upon the actual data value;
  - Without condition variables, the thread has to check the condition continuously. This can be very resource consuming;
  - With condition variables, the thread sleeps and waits on the condition variable;
  - A condition variable is always used in conjunction with a mutex.

Thread Synchronization APIs (4)

- Condition variable (cont)

standard "waiter" idiom:

```
pthread_mutex_lock( &mutex );
while ( students < 100 )
    pthread_cond_wait( &cond, &mutex );
do_something();
pthread_mutex_unlock( &mutex );
```

standard "signaler" idiom:

```
pthread_mutex_lock( &mutex );
students++;
if (students>100)
    pthread_cond_signal( &cond );
pthread_mutex_unlock( &mutex );
```

This function atomically releases mutex and causes the "waiter" to block on the condition variable "cond". Upon successfully return, the mutex has been locked and owned by the "waiter" again.

pthread_cond_wait( ) and pthread_cond_signal( ) should be called while mutex is locked. Why?

If more than one thread is blocked on "cond", the scheduling policy determines the order in which threads are unblocked. pthread_cond_broadcast ( ) unblocks all threads currently blocked on "cond".

Notes: It is a logical error to call signal before calling wait.

One More Example: Producer & Consumer

```
// lock for shared queue
pthread_mutex_t data_mutex = PTHREAD_MUTEX_INITIALIZER;

// flag to indicate queue is empty
int queue_empty = 1;

// queue_empty condition variable
pthread_cont_t queue_empty_cond = PTHREAD_COND_INITIALIZER;

// flag to indicate queue is full
int queue_full = 0;

// queue_full condition variable
pthread_cont_t queue_full_cond = PTHREAD_COND_INITIALIZER;
```

Producer

```
void *producer(void *)
 {
   Produce data

   Pthread_mutex_lock(&data_mutex);

   while( queue_full ) {
    /* sleep on condition variable – release lock implicitly*/
    Pthread_cond_wait(&queue_full_cond, &data_mutex);
    /* woken up – lock acquired implicitly */
   }

   Insert data into queue;

   queue_empty = 0;
   Pthread_cond_signal(&queue_empty_cond);

   if( queue is full ) queue_full = 1;

   Pthread_mutex_unlock(&data_mutex);
 }
```

Consumer

```
void *consumer(void *)
{
   Pthread_mutex_lock(&data_mutex);

   while( queue_empty ) {
    /* sleep on condition variable – release lock implicitly*/
    Pthread_cond_wait(&queue_empty_cond, &data_mutex);
    /* woken up – lock acquired implicitly */
   }

   Remove data from queue;

   queue_full = 0;
   Pthread_cond_signal(&queue_full_cond);

   if( queue is empty ) queue_empty = 1;

   Pthread_mutex_unlock(&data_mutex);

   consume_data();
}
```

Synchronization of Java Threads

- If a class has at least one synchronized method, each instance of it has a monitor. A monitor is an object that can block threads and notify them when it is available.

- Example:

```
public synchronized void updateRecord() {
    // critical code goes here ...
}
```

- Only one thread may be inside the body of this function. A second call will be blocked until the first call returns or wait() is called inside the synchronized method.

Synchronization of Threads (cont.)

- If you don't need to protect an entire method, you can synchronize on an object:

```
public void foo() {
    //...
    synchronized (this) {
        //critical code goes here ...
    }
    //...
}
```

- Declaring a method as synchronized is equivalent to synchronizing on *this* for all the method block.

Wait and Notify

- Allows two threads to cooperate
- Can be based on a single shared lock object

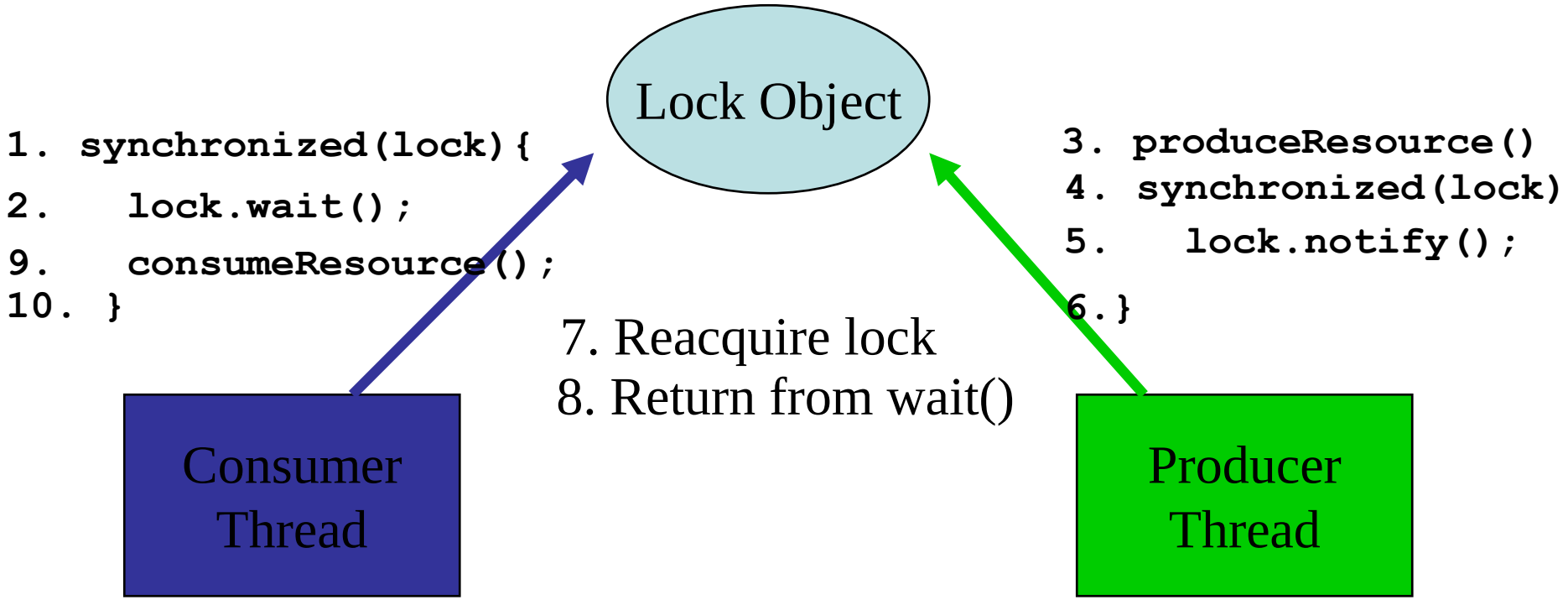
Consumer:

```
synchronized (lock) {
  while (!resourceAvailable()) {
    lock.wait();
  }
  consumeResource();
}
```

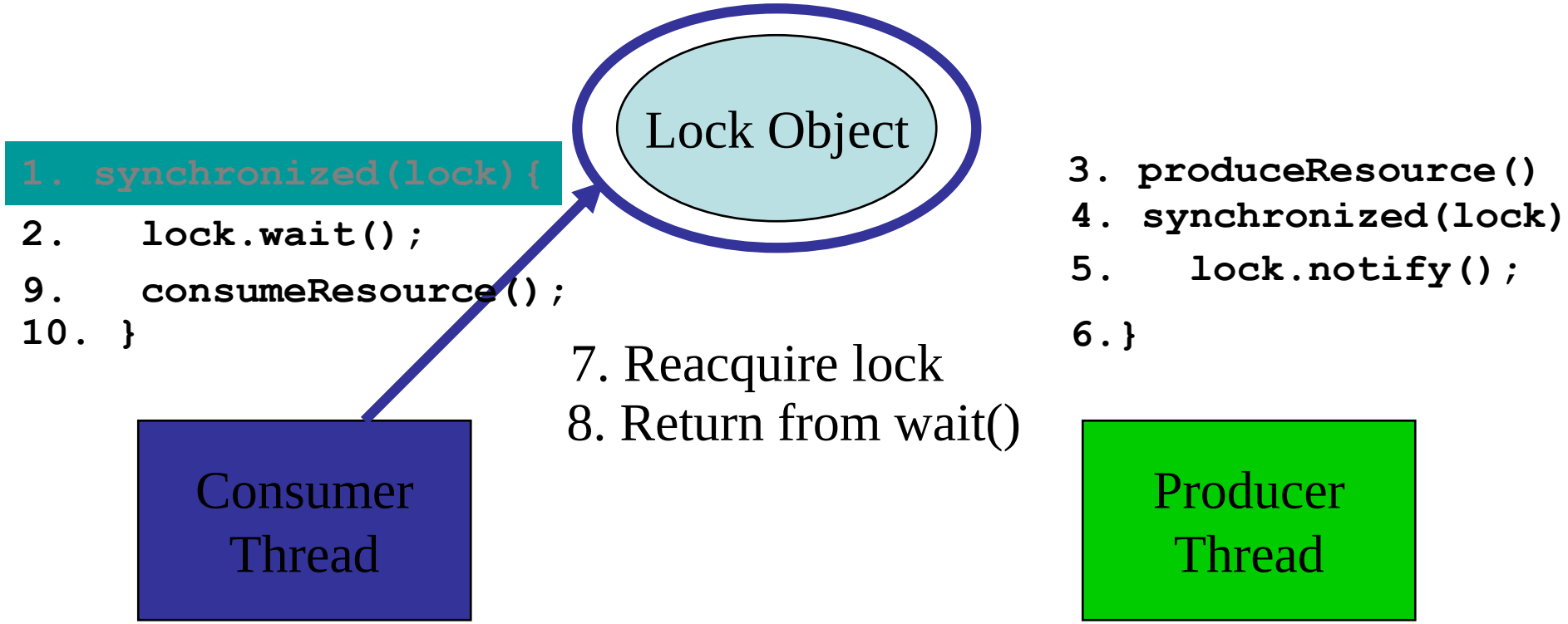
Producer:

```
produceResource();
synchronized (lock) {
  lock.notifyAll();
}
```

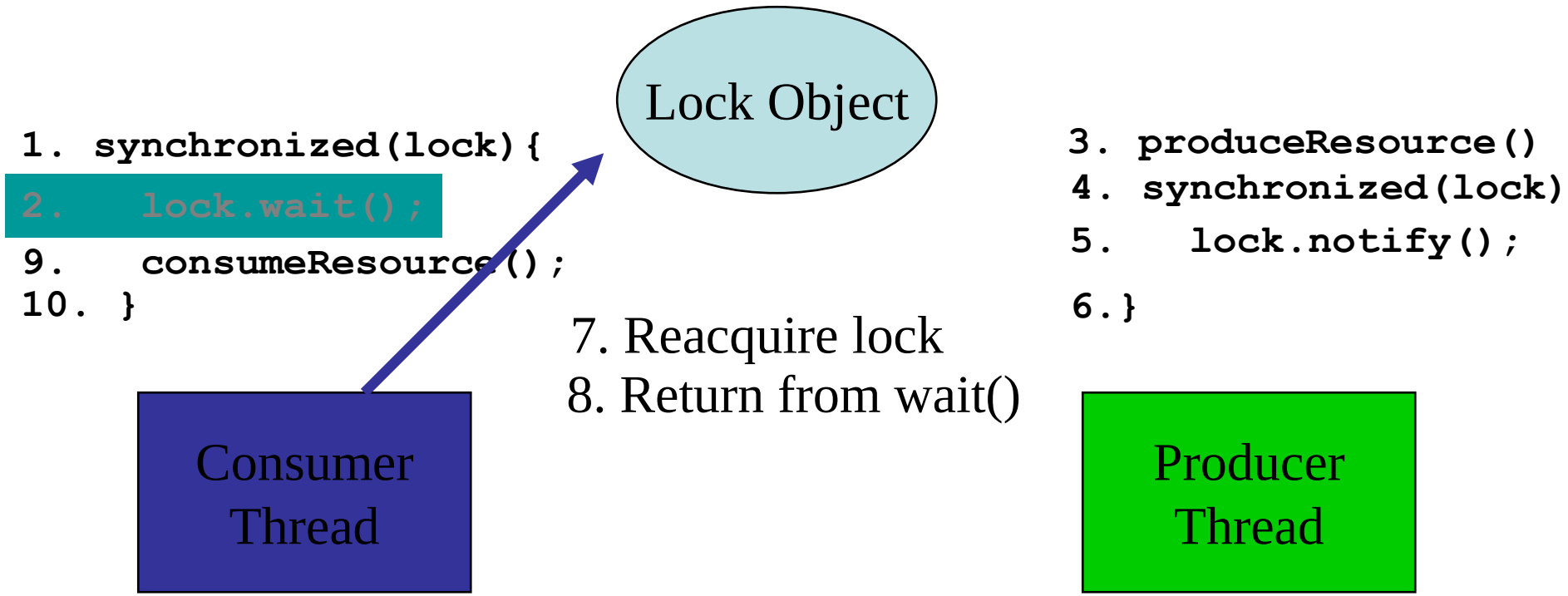
Wait/Notify Sequence



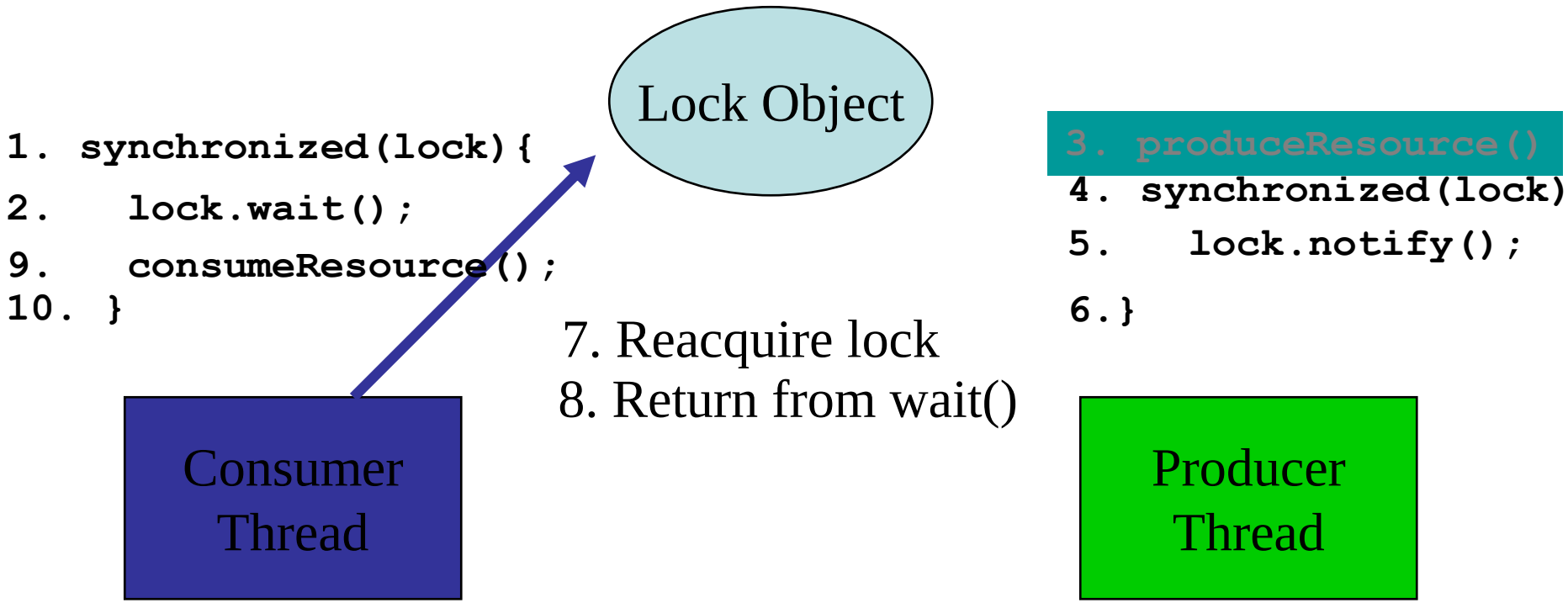
Wait/Notify Sequence



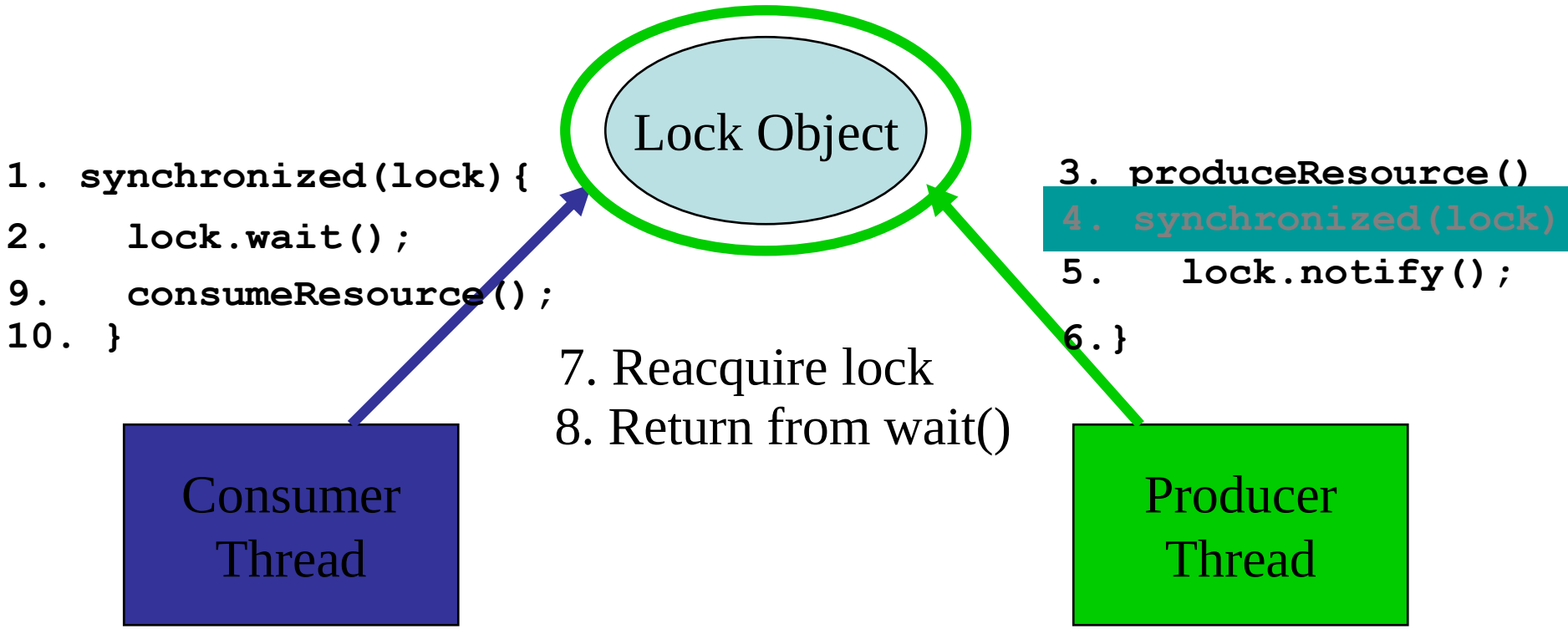
Wait/Notify Sequence



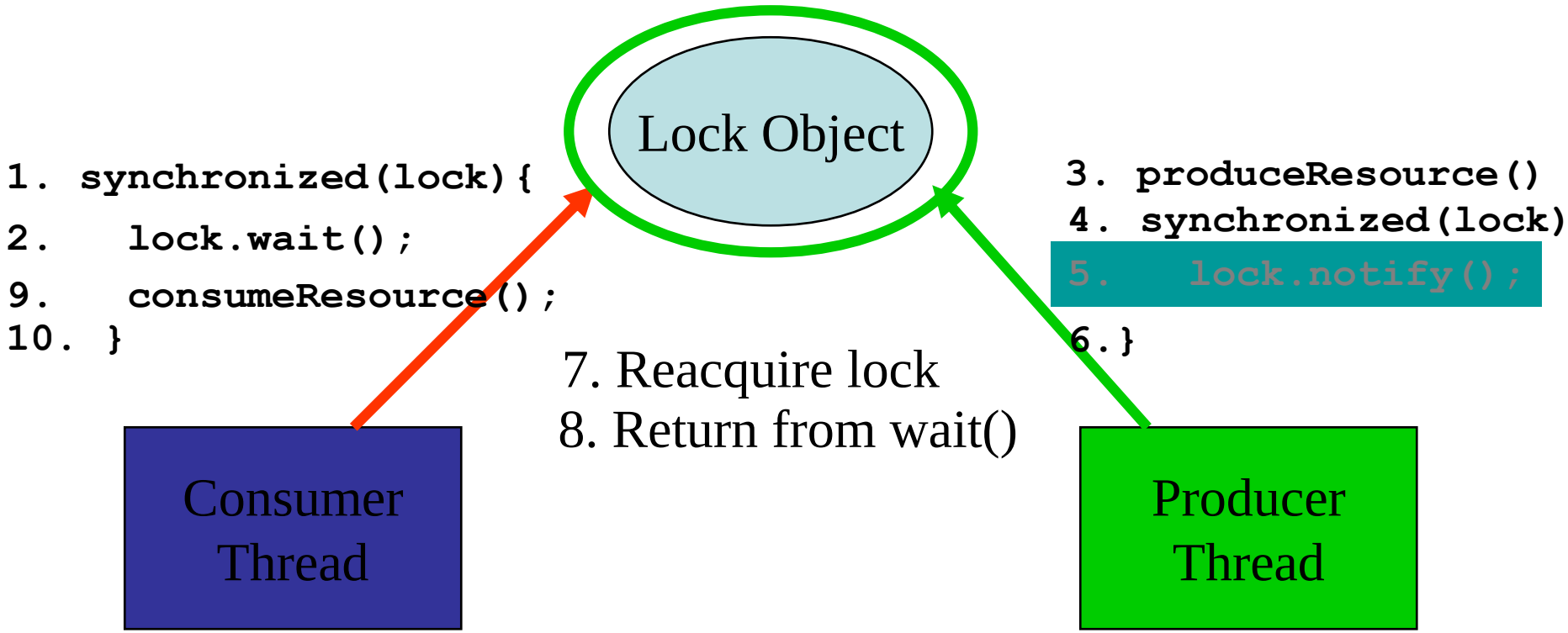
Wait/Notify Sequence



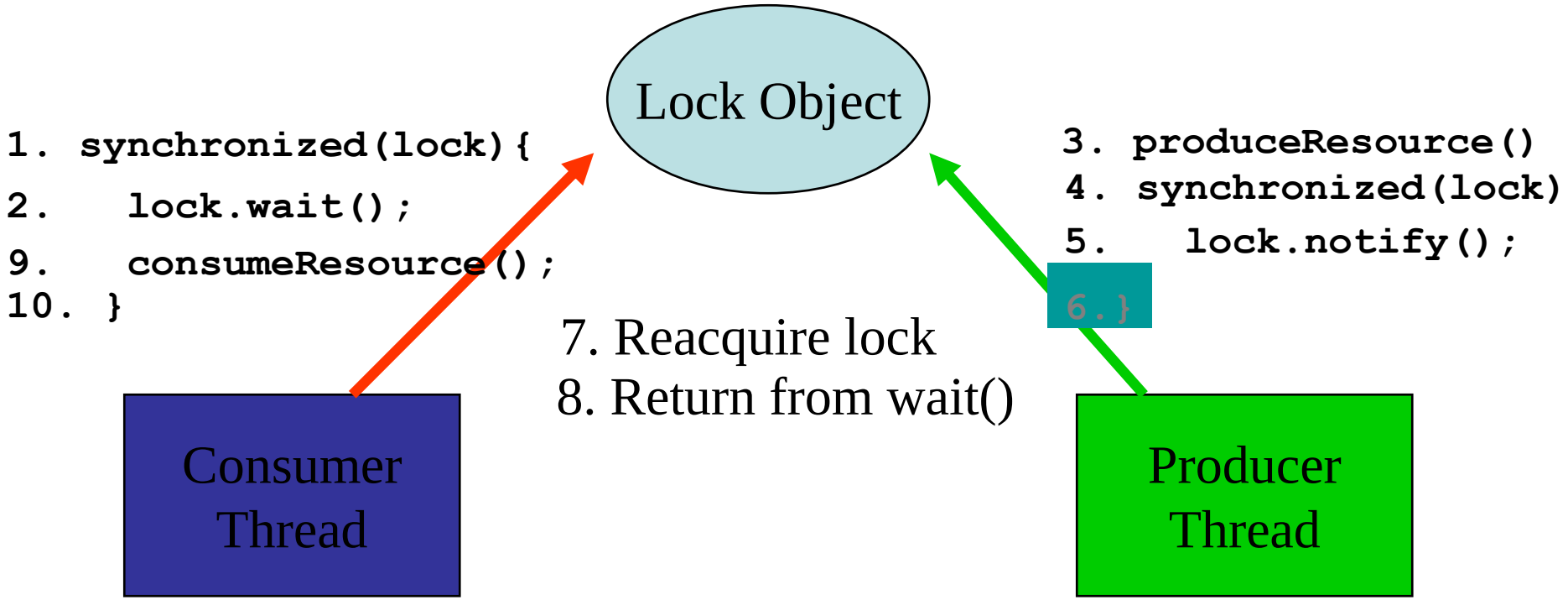
Wait/Notify Sequence



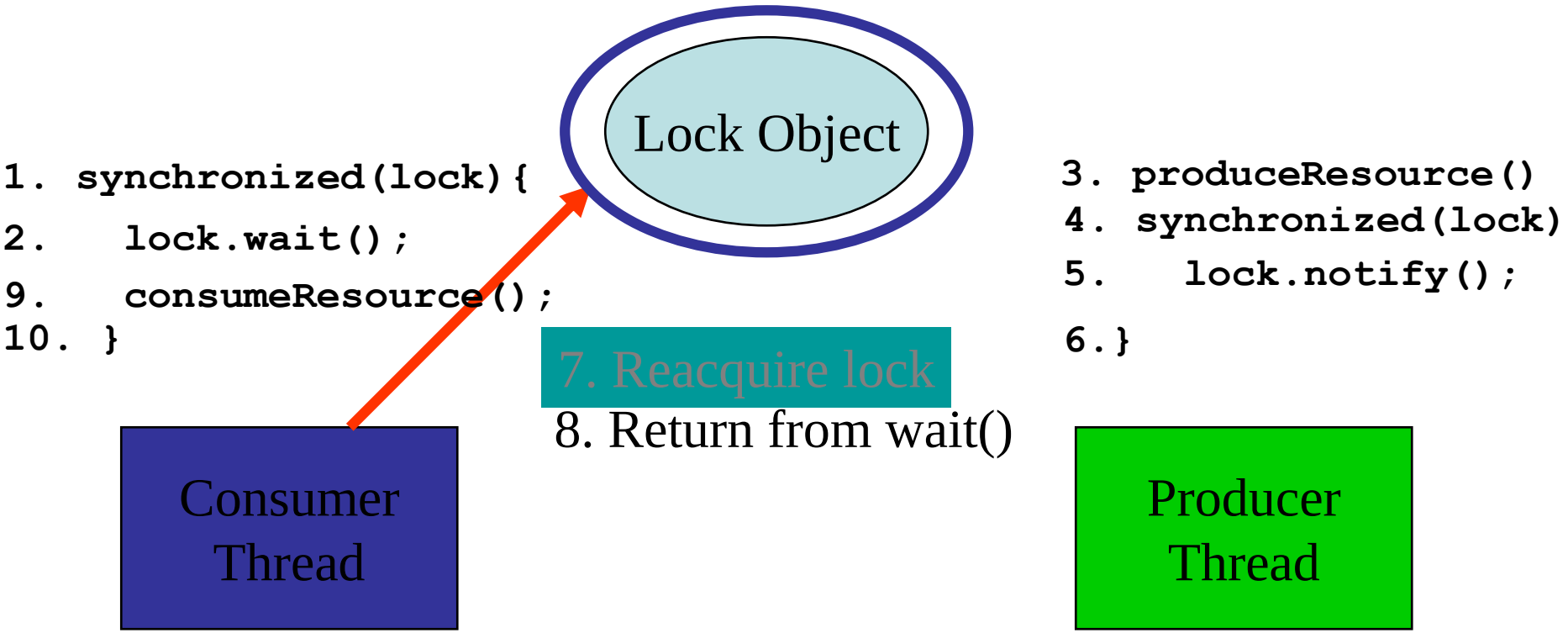
Wait/Notify Sequence



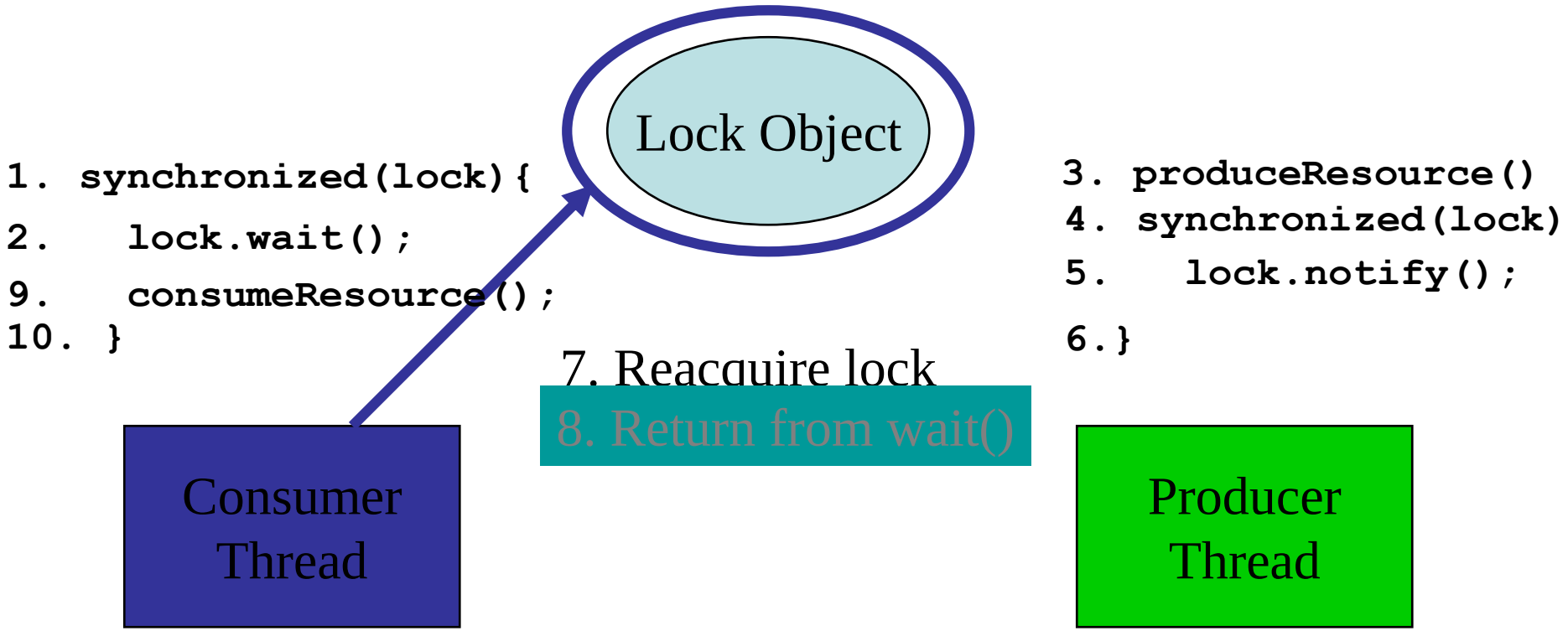
Wait/Notify Sequence



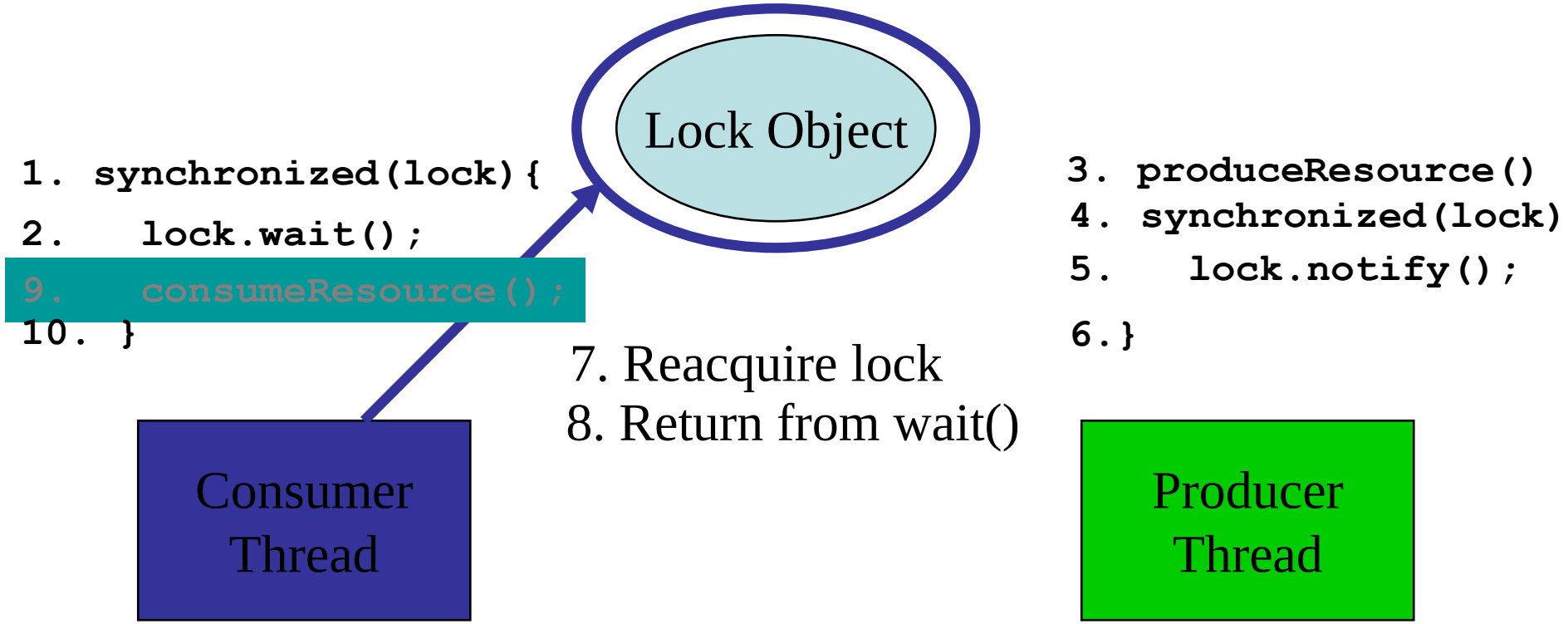
Wait/Notify Sequence



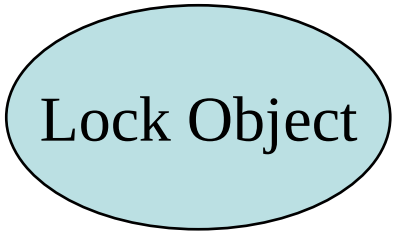
Wait/Notify Sequence



Wait/Notify Sequence



Wait/Notify Sequence




```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

7. Reacquire lock
8. Return from wait()

Consumer Thread

```
3. produceResource()
4. synchronized(lock)
5.    lock.notify();
6.}
```

Producer Thread

Wait/Notify: Details

- Must loop on wait(), in case another thread grabs the resource...
  - After you are notified
  - Before you acquire the lock and return to wait()
- Use lock.notifyAll() if there may be more than one waiting thread

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.
  - Protects every critical data item
  - With a multiprocessor system, it has two functions and if the data is locked
    1. If lock is held by running thread, then use a *spinlock* (busy loop).
    2. If lock is held by blocked thread, then go to sleep.
  - Single processor, uses only #2
- Uses *readers-writers* locks when longer sections of code need access to data.
  - Readers-Writers locks protect data is read often, but needs little updating.

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses *spinlocks* on multiprocessor systems.
  - Also protects spinlocks, by not allowing a process to be preempted when holding a spinlock.

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

Transaction

- Transaction - collection of instructions or operations that performs single logical function
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Effect: all or nothing

Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms provide serializability