
SYSC 3303 Real-Time Concurrent Systems

Synchronizing Java Threads

- Copyright © 2001-2004 D.L. Bailey, 2003-2012 L.S. Marshall Systems and Computer Engineering, Carleton University
- revised July 5th, 2012

Why Synchronize Threads?

- For practical purposes, threads that execute completely independently are of little use
- From time to time, threads must synchronize and communicate with one another
- Two types of synchronization are of interest
 - mutual exclusion
 - condition synchronization

Mutual Exclusion

- Recall that a sequence of statements that must appear to be executed *indivisibly* is called a *critical section*
 - e.g., the code that modifies a data structure shared by concurrent threads must be a critical section
- *Mutual exclusion* is the synchronization required to protect a critical section

Condition Synchronization

- *Condition synchronization* is required when one thread performs an operation that must be performed only after another thread has performed an operation or has reached a defined state

Example: A Container Shared by Threads

- An instance of class `Box` stores (a reference to) a single object.
- The `Box` object is shared by *producer* threads and *consumer* threads, and is used to transfer objects from producers to consumers
 - producers invoke `put (Object)` to store an object in an empty `Box` (i.e. this method fills the `Box`)
 - consumers invoke `get ()` to remove an object from the `Box` (i.e., this method leaves the `Box` empty)

Example: A Container Shared by Threads

```
public class Box {  
  
    /**  
     * The object stored in this Box.  
     */  
    private Object contents = null;  
  
}
```

Example: A Container Shared by Threads

```
/**
 * Stores its argument in the Box if
 * the Box is empty; otherwise, the
 * Box contents are not changed.
 *
 * @param obj the object that is to be
 *           stored in this Box.
 * @return true if obj was stored in
 *         this Box; false if another
 *         object was already stored
 *         in the Box (in which case,
 *         this Box was not changed by
 *         invoking this method).
 */
```

Example: A Container Shared by Threads

```
public boolean put(Object obj) {  
    if (contents != null)  
        // This Box is already full.  
        return false;  
  
    // This Box is empty, so store obj.  
    contents = obj;  
    return true;  
}
```

Example: A Container Shared by Threads

```
/**
 * Removes the object stored in this
 * Box, leaving the Box empty.
 *
 * @return the object stored in this
 *         Box, if there is one.
 *         If the Box is empty,
 *         returns null.
 */
```

Example: A Container Shared by Threads

```
public Object get() {  
    Object obj = contents;  
    // Mark the box as empty.  
    contents = null;  
    return obj;  
}  
}
```

Example: A Container Shared by Threads

- What problems might occur if
 - two producers invoke `put ()` concurrently?
 - two consumers invoke `get ()` concurrently?
 - a producer invokes `put ()` at the same time as a consumer invokes `get ()`?
- `Box` operations must be mutually exclusive

Java's Support for Thread Synchronization

- Java's thread synchronization model is based on the *monitor* construct proposed by P. Brinch Hansen and C.A.R. Hoare in the 1970s
 - an instance of a class containing `synchronized` methods has monitor semantics (method execution by multiple threads is guaranteed to be mutually exclusive)
 - these methods are called *thread-safe*

Thread-Safe Box

- To make the `put ()` and `get ()` methods thread-safe, all we have to do is declare them to be `synchronized`
- Methods declared with the `synchronized` modifier can be executed by only one thread at a time
 - the JVM enforces this
- Here is the revised class...

Thread-Safe Box

```
public class Box {  
  
    /**  
     * The object stored in this Box.  
     */  
    private Object contents = null;  
  
}
```

Thread-Safe Box

```
public synchronized
boolean put(Object obj) {
    if (contents != null)
        // This Box is already full.
        return false;

    // This Box is empty, so store obj.
    contents = obj;
    return true;
}
```

Thread-Safe Box

```
public synchronized Object get() {  
    Object obj = contents;  
    // Mark the box as empty.  
    contents = null;  
    return obj;  
}  
}
```

How Does Thread Synchronization Work?

- Every object has a lock that can be held by only one thread at a time
- When a thread invokes a `synchronized` method, it first attempts to obtain the lock
- If it obtains the lock, it starts executing the method
- If the thread cannot obtain the lock (because another thread holds the lock), the thread *blocks* (it is no longer eligible to run)
- When a thread returns from a `synchronized` method, it releases the object's lock

How Does Thread Synchronization Work?

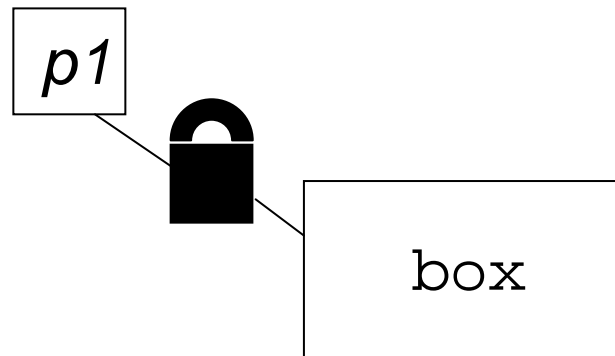
- The JVM grants the lock to one of the threads that is blocked on the object, and that thread is made eligible to run
- That thread executes when it is eventually scheduled for execution

Synchronized Method Execution

- Suppose we have an instance of `Box` (call it `box`), and producer threads `p1`, `p2`, and consumer thread `c1` each have a reference to this object
- `p1` sends the `put ()` message to `box`, and while `p1` executes the method, `p2` invokes `put ()` and `c1` invokes `get ()`
- The JVM ensures that only one thread at a time will execute `put ()` or `get ()` (because both methods are synchronized)
 - in other words, the JVM ensures that execution of `put ()` and `get ()` is mutually exclusive - these methods are critical sections

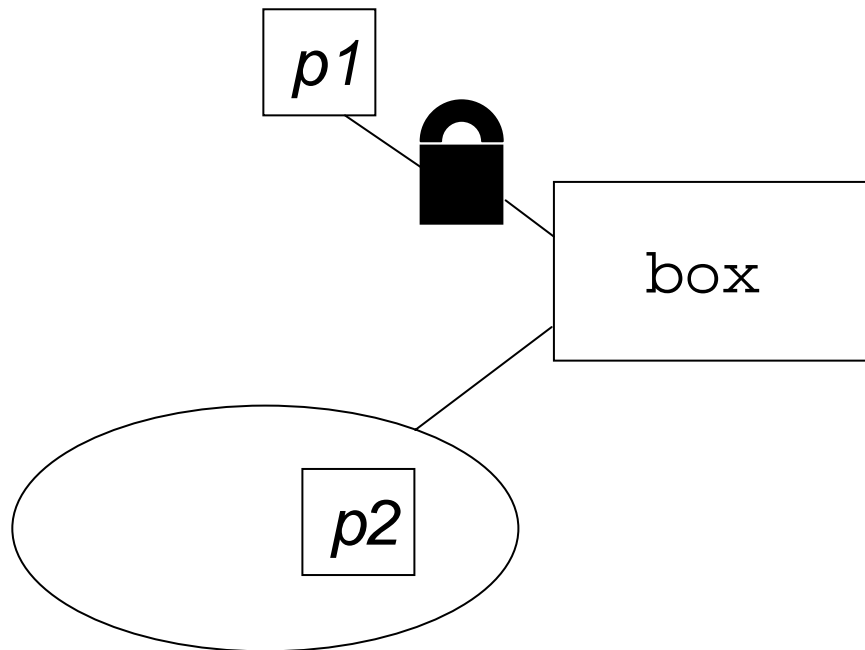
Synchronized Method Example, continued

- *p1* invokes `put ()`
 - *p1* obtains the lock and starts executing `put ()`



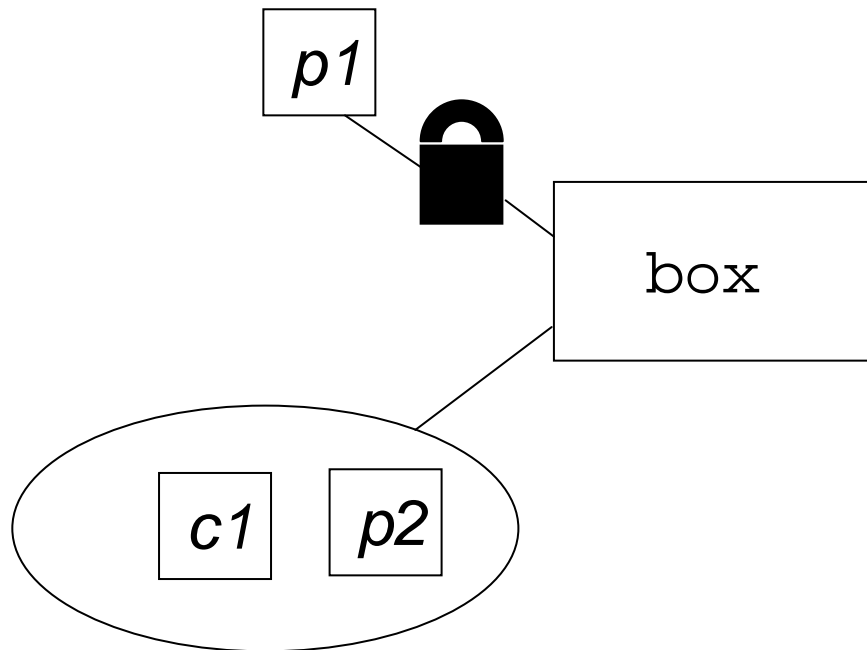
Synchronized Method Example, continued

- *p2* invokes `put ()`
 - *p2* cannot obtain the lock, so it blocks



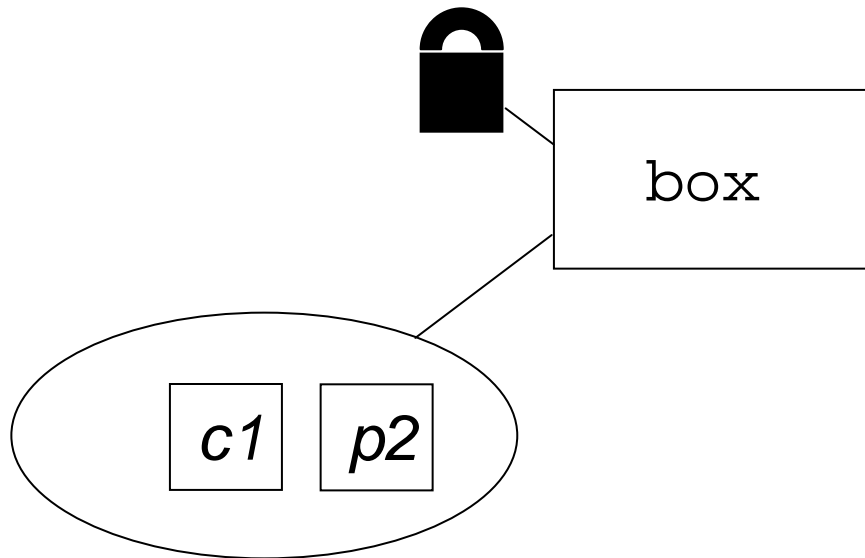
Synchronized Method Example, continued

- *c1* invokes `get ()`
 - *c1* cannot obtain the lock, so it blocks



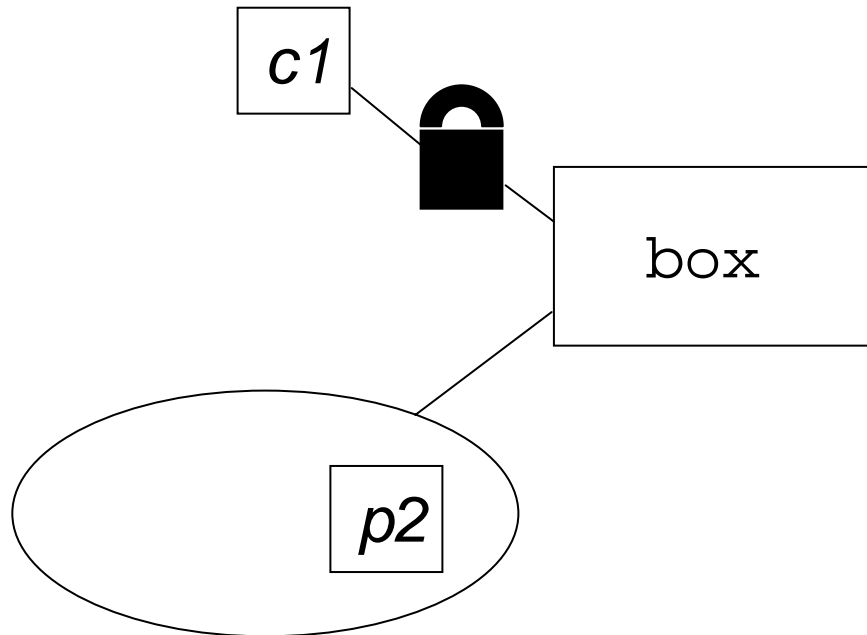
Synchronized Method Example, continued

- *p1* returns from `put ()`
 - *p1* releases the lock



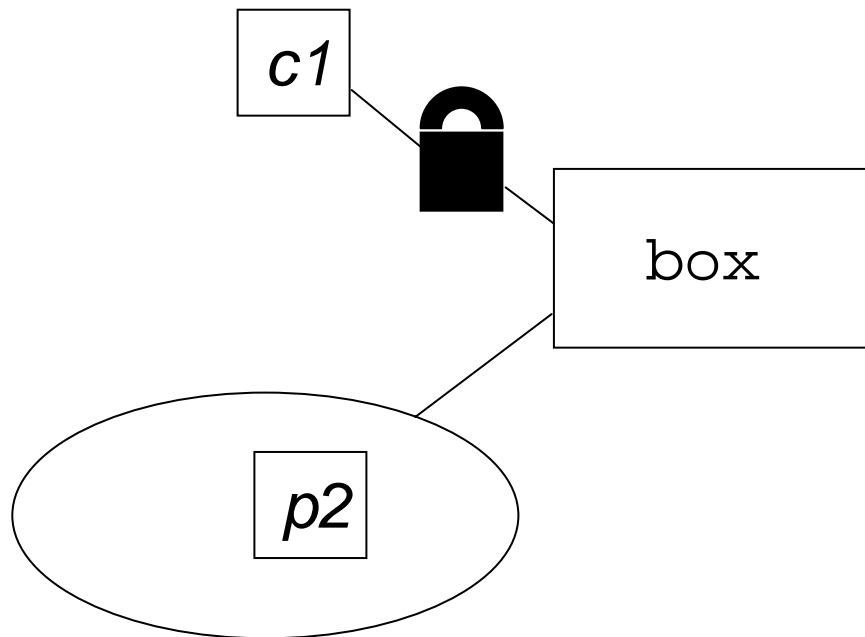
Synchronized Method Example, continued

- The JVM grants the lock to one of the threads blocked on the object (*c1* here: threads waiting for the lock are not necessarily handled FIFO)



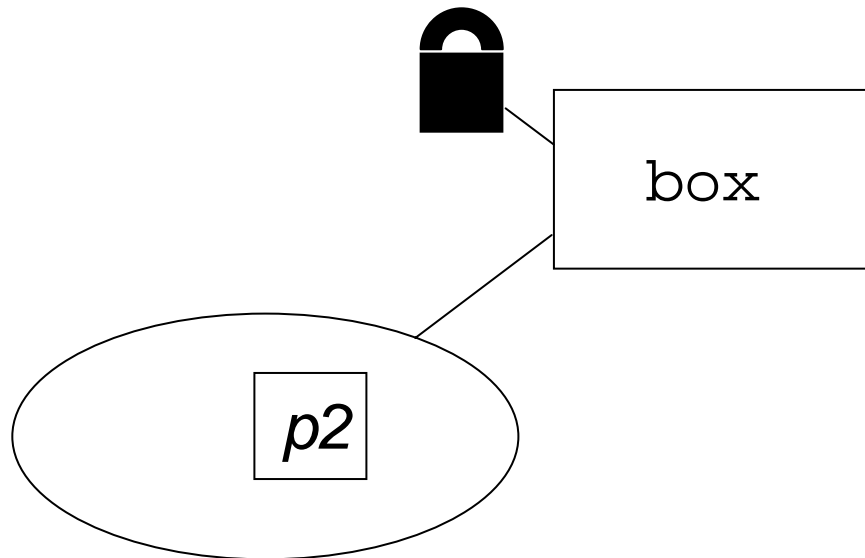
Synchronized Method Example, continued

- When *c1* is scheduled for execution, it starts executing `get ()`



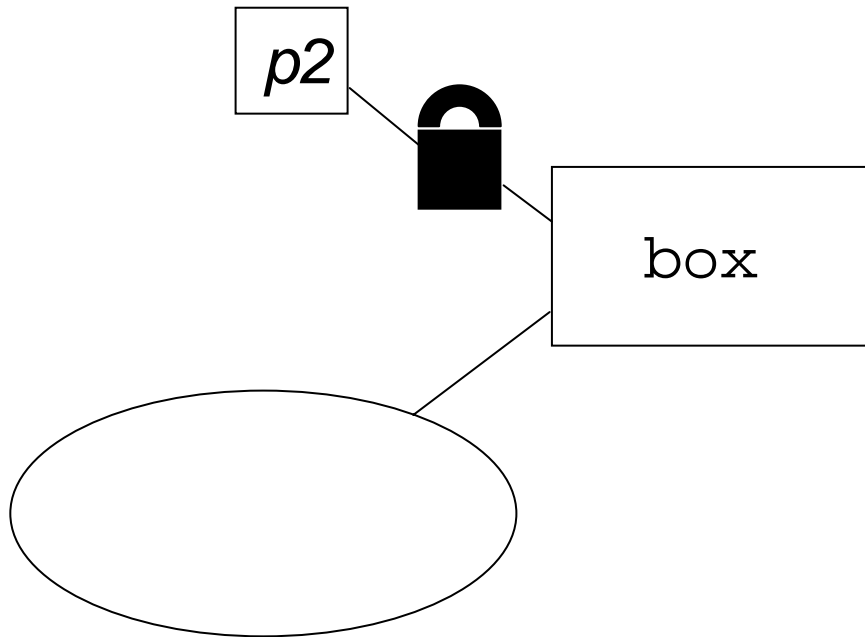
Synchronized Method Example, continued

- *c1* returns from `get ()`
 - *c1* releases the lock



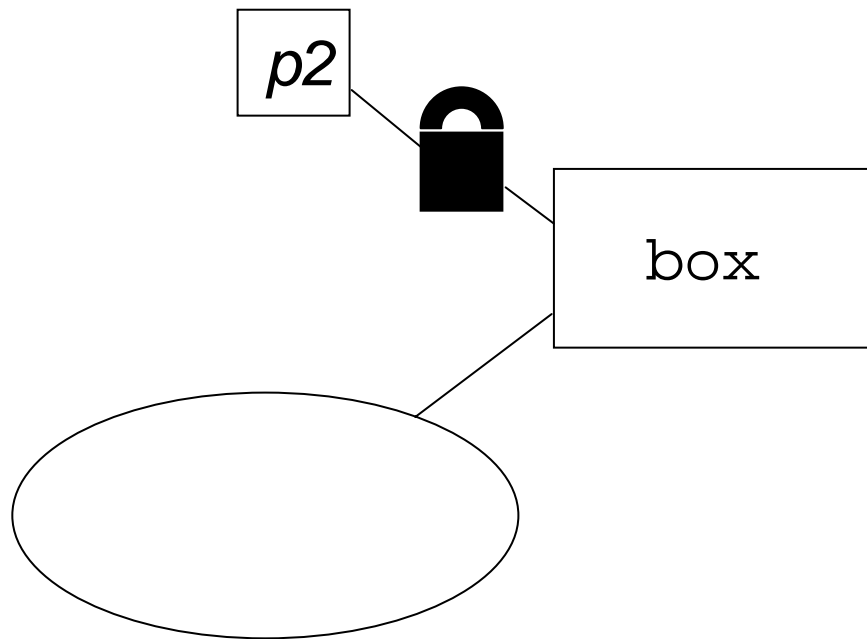
Synchronized Method Example, continued

- The JVM grants the lock to one of the threads blocked on the object (*p2* is currently the only one blocked)



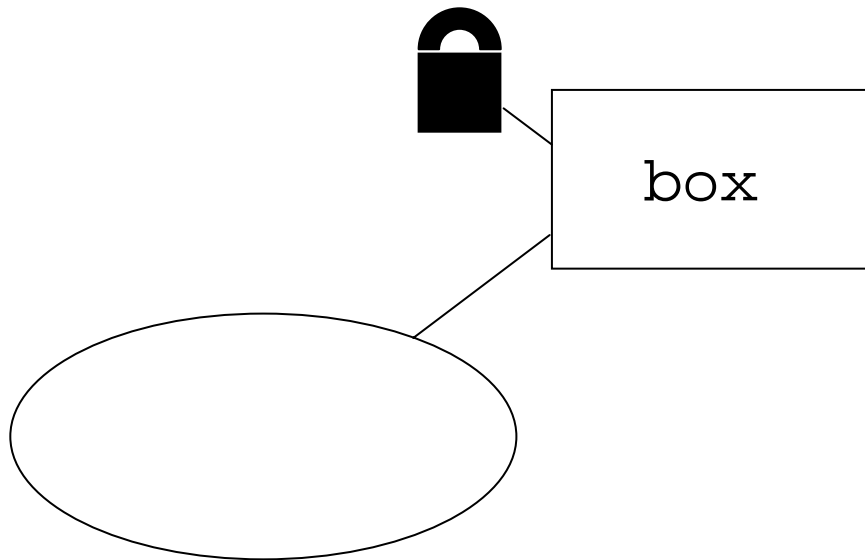
Synchronized Method Example, continued

- When *p2* is scheduled for execution, it starts executing `put ()`



Synchronized Method Example, continued

- *p2* returns from `put ()`
 - *p2* releases the lock
- the next thread that needs the lock will be able to get it immediately



Can Class (static) Methods be Synchronized?

- Yes
- Each class has a *class lock* that is completely separate from the object locks associated with the instances of a class
- When a thread invokes a `synchronized static` method, it must obtain the class lock before it can execute the method

Condition Synchronization

- What should a producer thread do if it determines that the `Box` is full (i.e., `put()` returns `false`)?
 - repeatedly invoke `put()` until it succeeds
- What should a consumer thread do if it determines that the `Box` is empty (i.e., `get()` returns `null`)?
 - repeatedly invoke `get()` until it succeeds
- We say that threads that do this are *busy-waiting*
- This is not a good approach - why?

Condition Synchronization

- We need a mechanism that will temporarily block producer threads if they attempt to put an object into a non-empty `Box`, and temporarily block consumer threads if they attempt to get an object from an empty `Box`
- In other words, producer threads must block until a *box-is-empty* condition becomes true, and consumer threads must block until a *box-is-empty* condition becomes false

Condition Synchronization

- Unlike Brinch Hansen's/Hoare's monitors, Java does not provide condition variables to allow threads to wait (block) for conditions to become true
- `wait()`, `notify()`, and `notifyAll()` methods (inherited from class `Object`) allow the programmer to build condition synchronization within monitor objects

wait()

- `public final void wait()` throws `InterruptedException`
- This method must be invoked by a thread that holds an object's lock; e.g., `wait()` must be invoked from within a `synchronized` method
- Every object has a *wait set*
- When a thread invokes `wait()` it releases ownership of the object's lock
- `wait()` places the thread in the object's wait set and disables the thread for thread scheduling purposes

wait()

- The waiting thread lies dormant until another thread reenables it for scheduling purposes, by invoking `notify()` or `notifyAll()`
- The reenabled thread then competes for the lock with other threads that are trying to obtain the lock
- When the thread regains ownership of the lock, it returns from `wait()` and resumes execution of the synchronized method

notify()

- `public final void notify()`
- This method must be invoked by a thread that holds an object's lock; e.g., `notify()` must be invoked from within a `synchronized` method
- `notify()` chooses one thread, *t*, from the object's wait set
 - if many threads are waiting on this object, only one of them is chosen
 - the choice is arbitrary and occurs at the discretion of the implementation

notify()

- Thread *t* is then reenabled for thread scheduling purposes, but it will not be able to proceed until the current thread (the thread that invoked `notify()`) relinquishes the object's lock (by exiting the synchronized method or by invoking `wait()`)
- After the notifier relinquishes the lock, *t* will compete for the lock in the usual manner with any other threads that might be actively competing to synchronize on this object
- When *t* obtains the lock, it returns from `wait()`

notifyAll()

- `public final void notifyAll()`
- This method must be invoked by a thread that holds an object's lock; e.g., `notifyAll()` must be invoked from within a `synchronized` method
- All threads in the object's wait set are reenabled for thread scheduling purposes
- The reenabled threads will not be able to proceed until the current thread (the thread that invoked `notifyAll()`) relinquishes the lock on this object

notifyAll()

- After the notifier relinquishes the lock, the reenabled threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object
- As each thread obtains the lock, it returns from `wait()`

Box with Condition Synchronization

```
public class Box {  
  
    /**  
     * The object stored in this Box.  
     */  
    private Object contents = null;  
  
    /**  
     * State of the box.  
     */  
    private boolean empty = true;  
}
```

Box with Condition Synchronization

```
public synchronized
void put(Object obj) {
    while (!empty) {
        wait();
    }
    contents = obj;
    empty = false;
    notifyAll();
}
```

- Notice that `put()` no longer needs to return a boolean value - why?

Box with Condition Synchronization

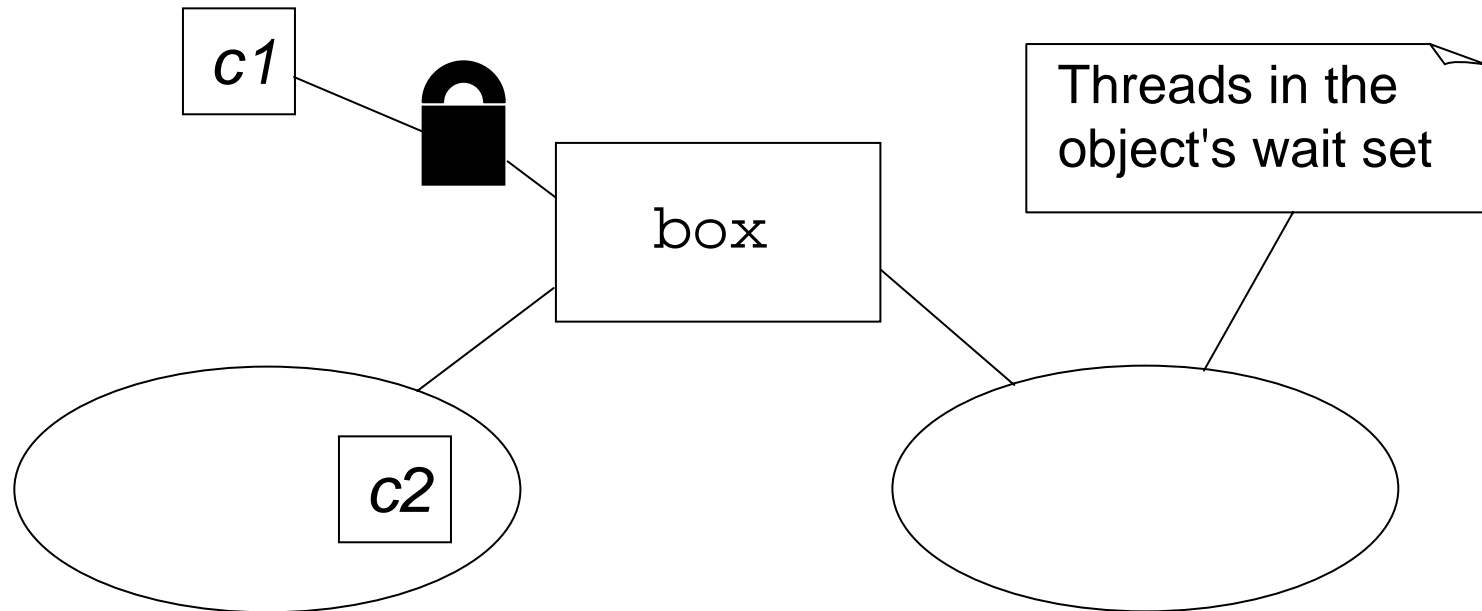
```
public synchronized Object get() {  
    while (empty) {  
        wait();  
    }  
    Object obj = contents;  
    empty = true;  
    notifyAll();  
    return obj;  
}  
}
```

Typical Producer/Consumer Execution

- Suppose `box` is empty, and consumer threads `c1` and `c2` invoke `get ()`
- `c1` obtains the object's lock
- Meanwhile, `c2` blocks until it can obtain the lock

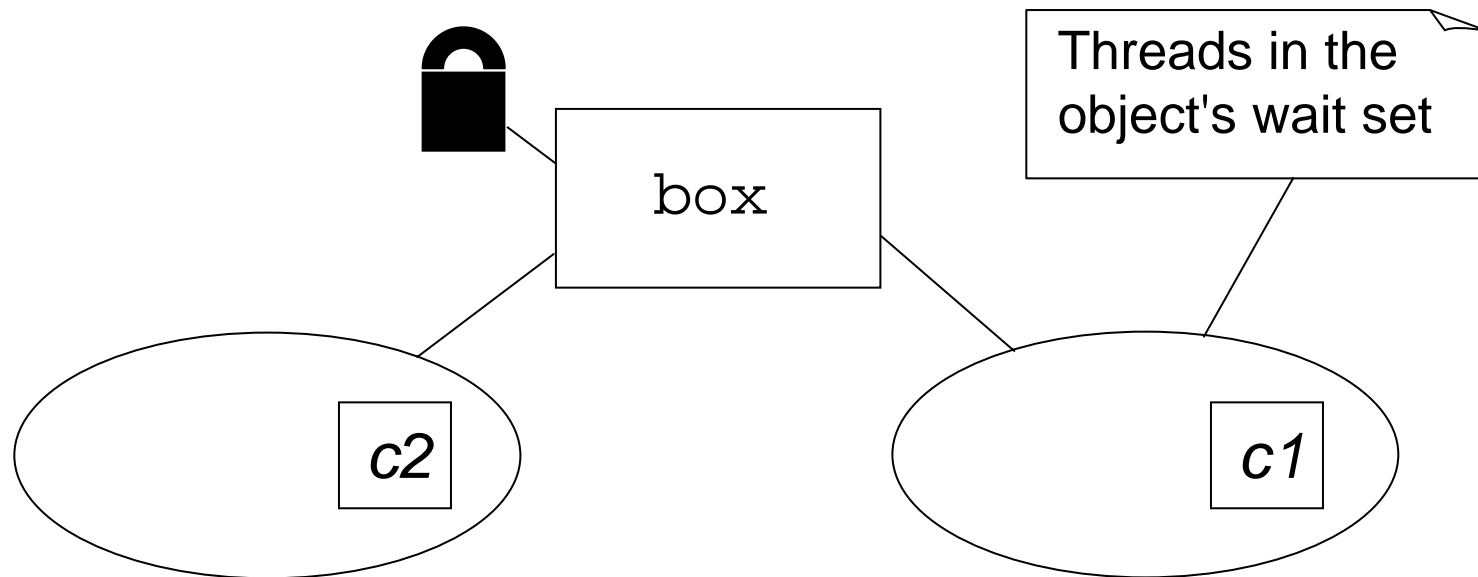
Typical Producer/Consumer Execution

- *c1* starts executing `get ()`



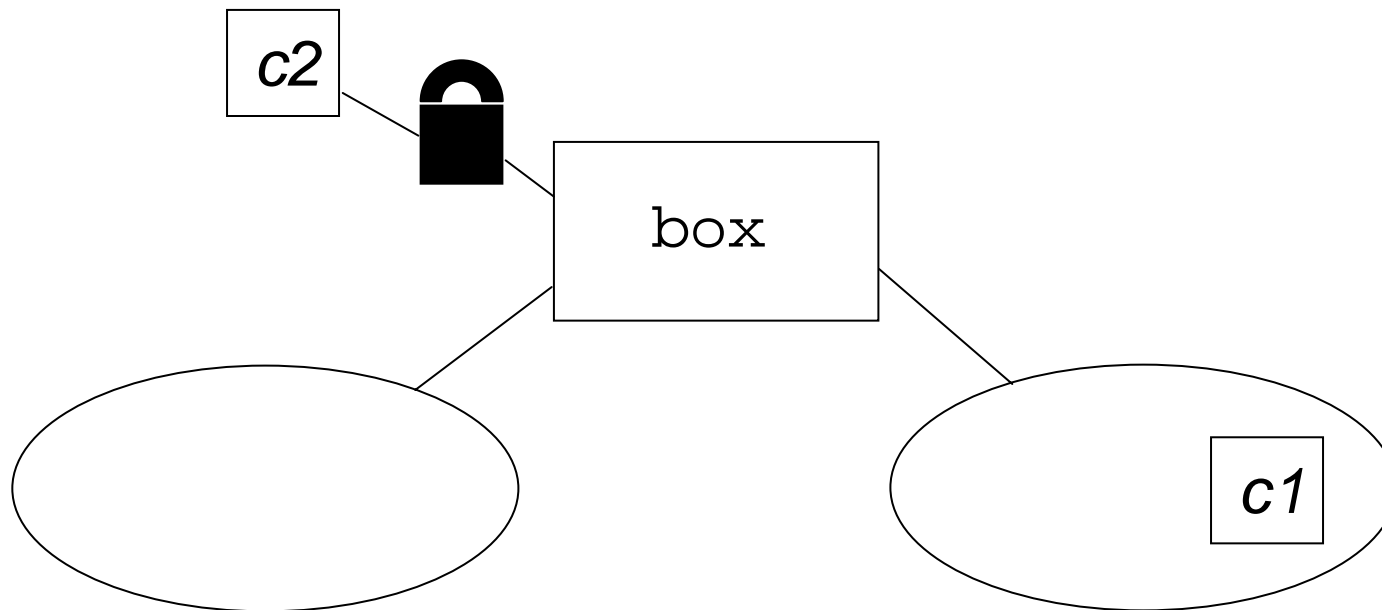
Typical Producer/Consumer Execution

- *c1* enters the `while` loop body (`empty` is `true`)
 - it invokes `wait()`, releases the object's lock, is placed in the wait set and blocks



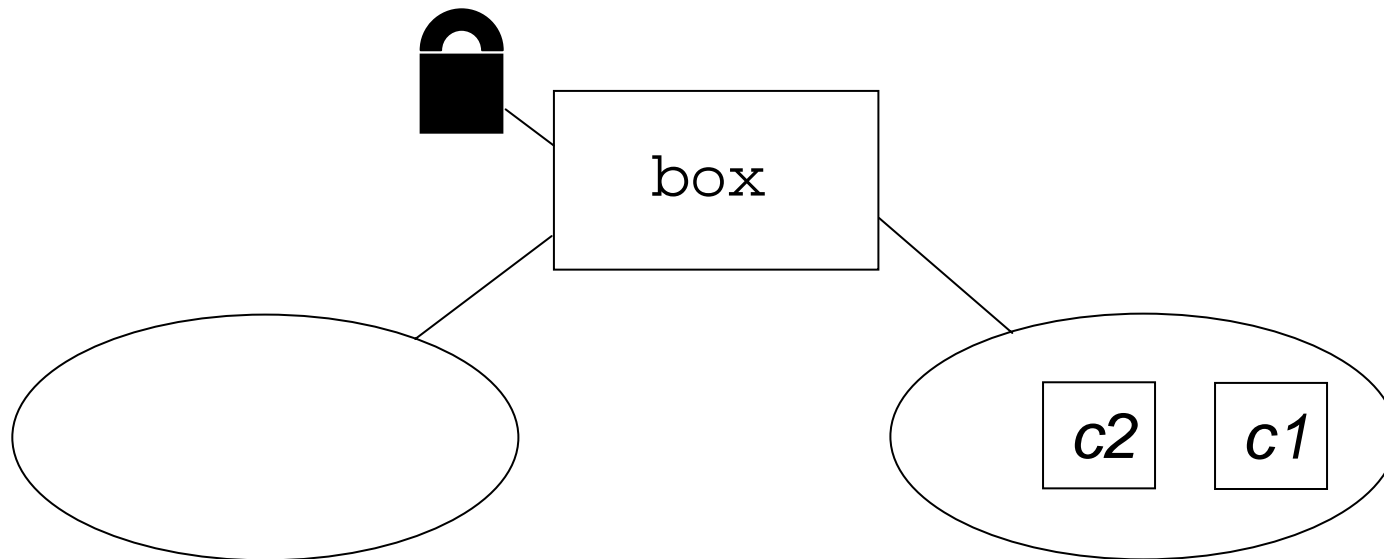
Typical Producer/Consumer Execution

- The JVM grants the lock to *c2*, which starts executing `get()`



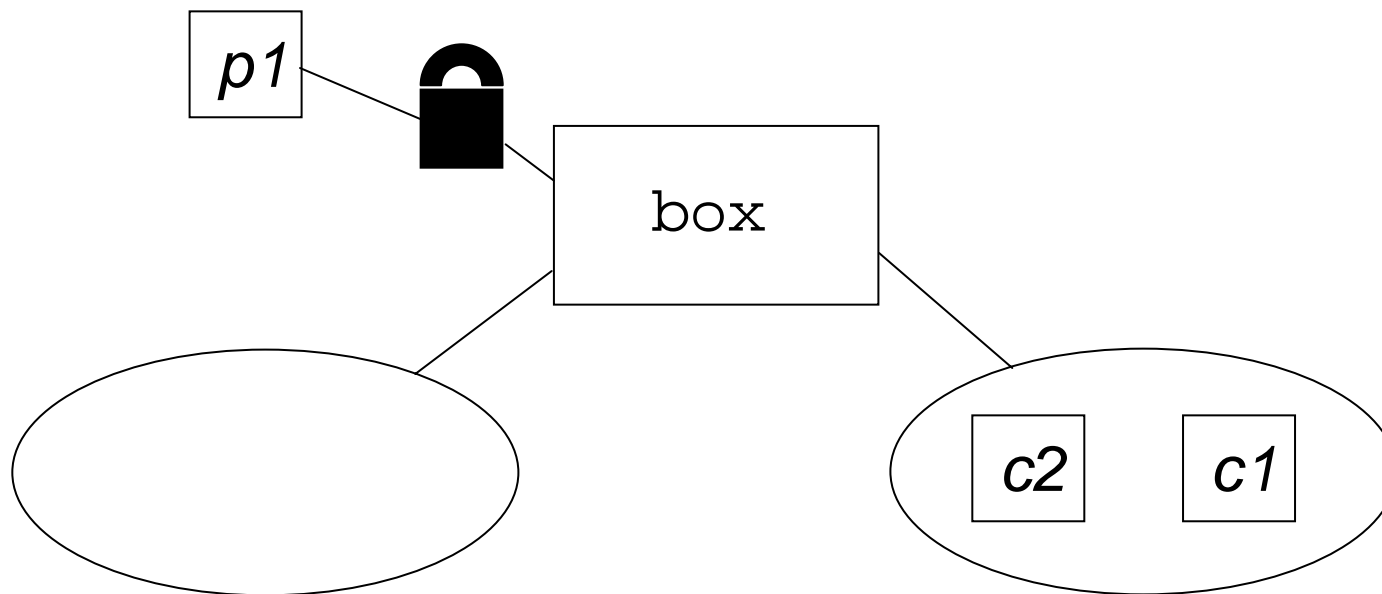
Typical Producer/Consumer Execution

- `c2` enters the `while` loop body (`empty` is `true`)
 - it invokes `wait()`, releases the object's lock, is placed in the wait set and blocks



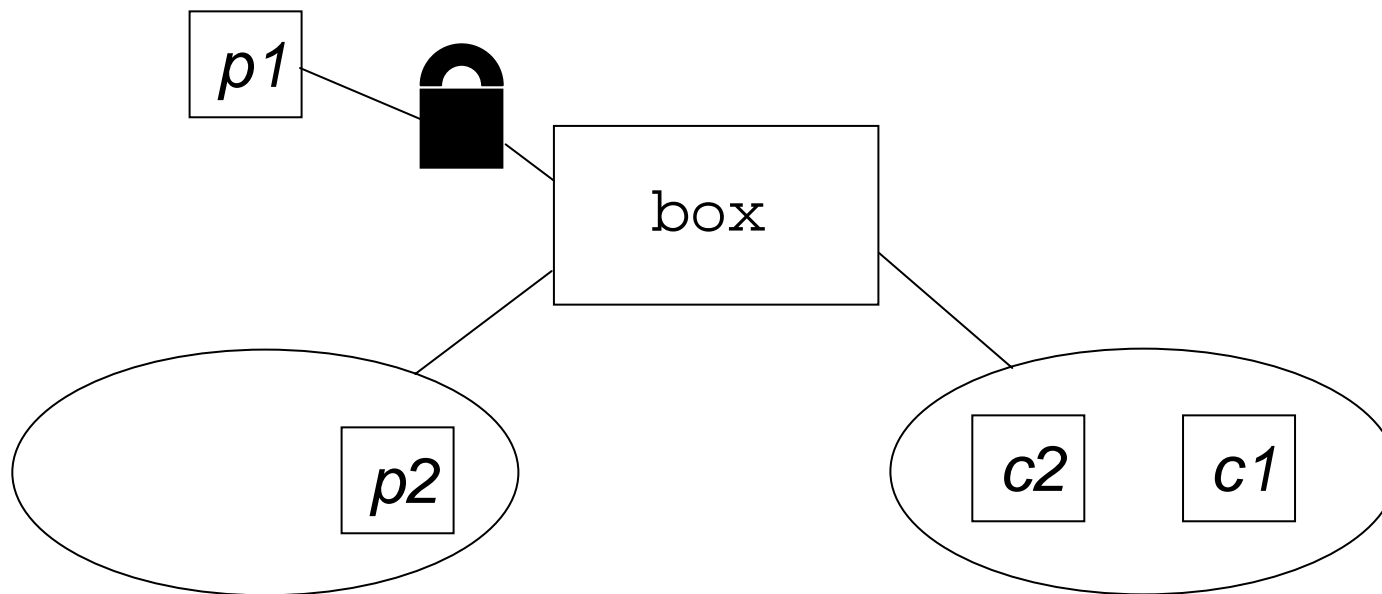
Typical Producer/Consumer Execution

- Producer *p1* invokes `put()`, obtains the object's lock, and starts executing `put()`
- `empty` is `true`, so it doesn't invoke `wait()`



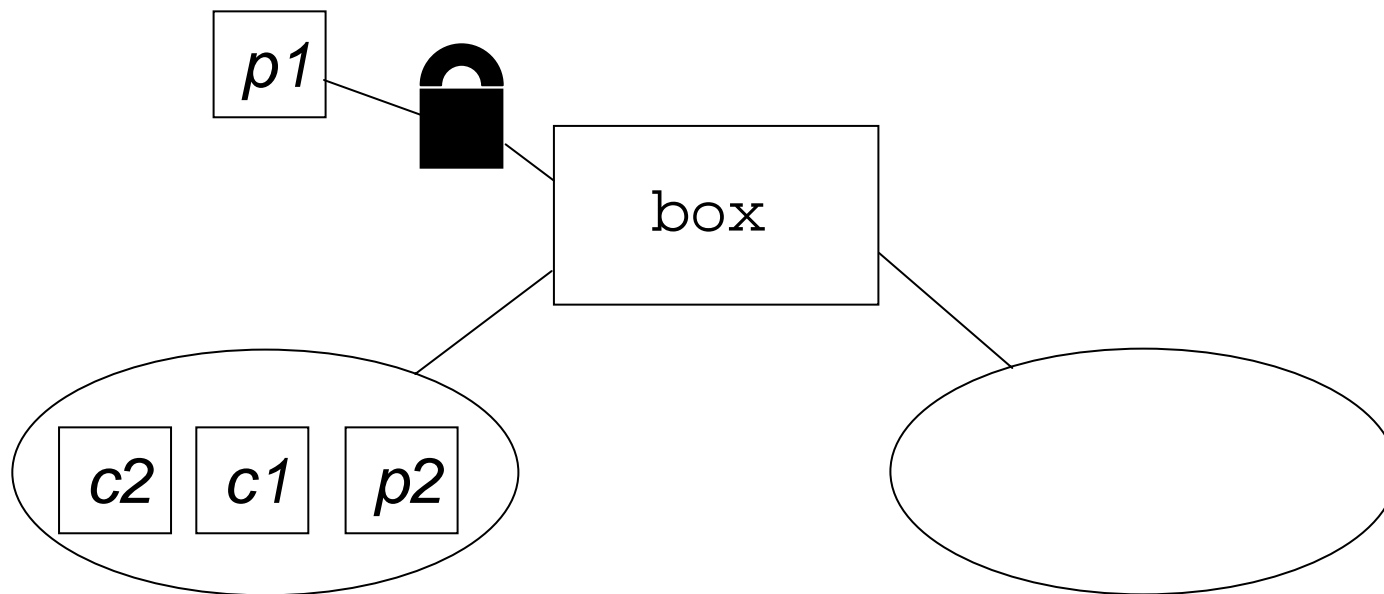
Typical Producer/Consumer Execution

- Producer *p2* invokes `put ()` and blocks until it can obtain the lock



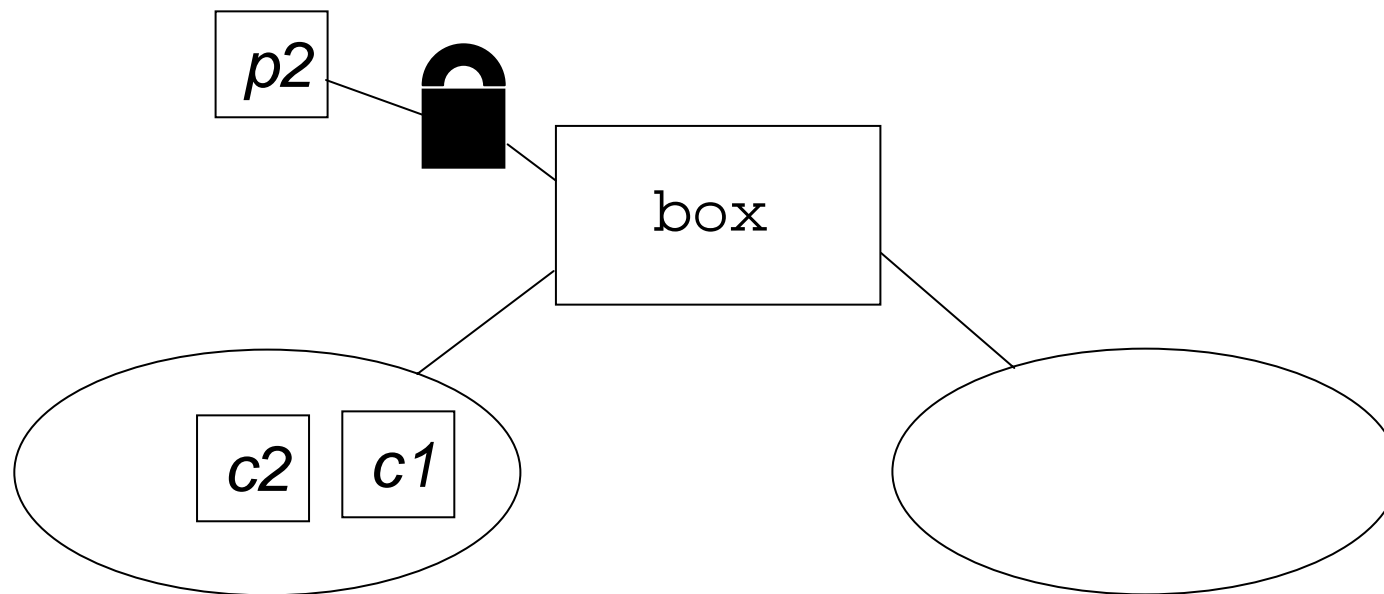
Typical Producer/Consumer Execution

- *p1* assigns `false` to `empty`, then invokes `notifyAll()`
 - *c1* and *c2* are reenabled for thread scheduling



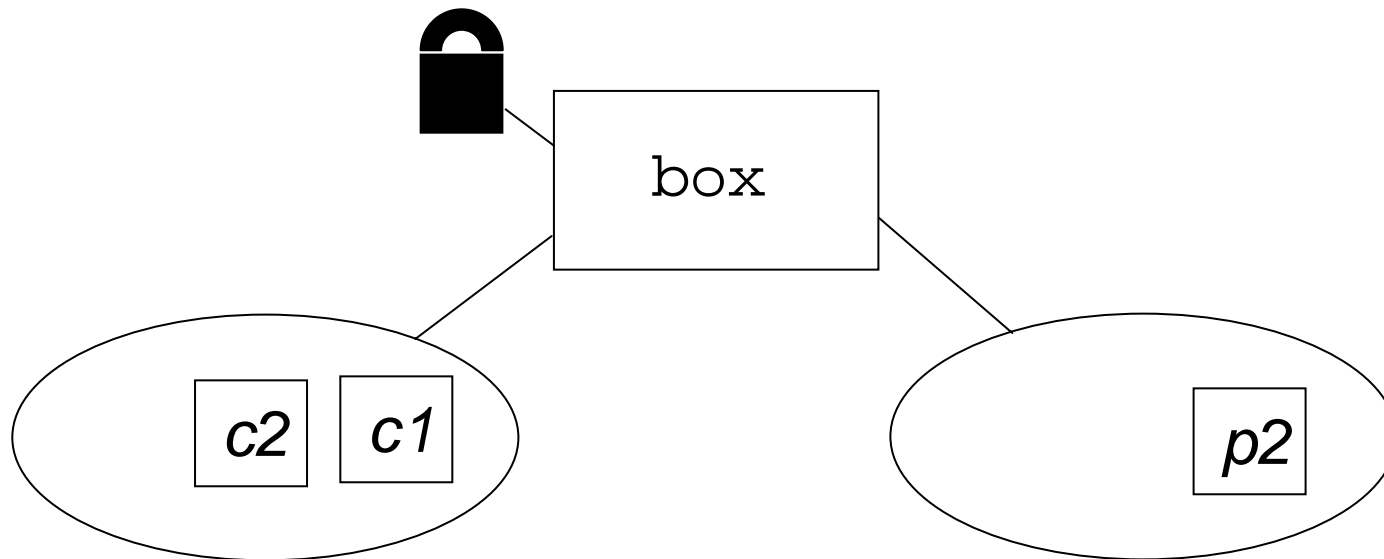
Typical Producer/Consumer Execution

- *p1* returns from `put ()` and relinquishes the lock
- The JVM grants the lock to *p2* (it could have granted it to *c1* or *c2*), which starts executing `put ()`



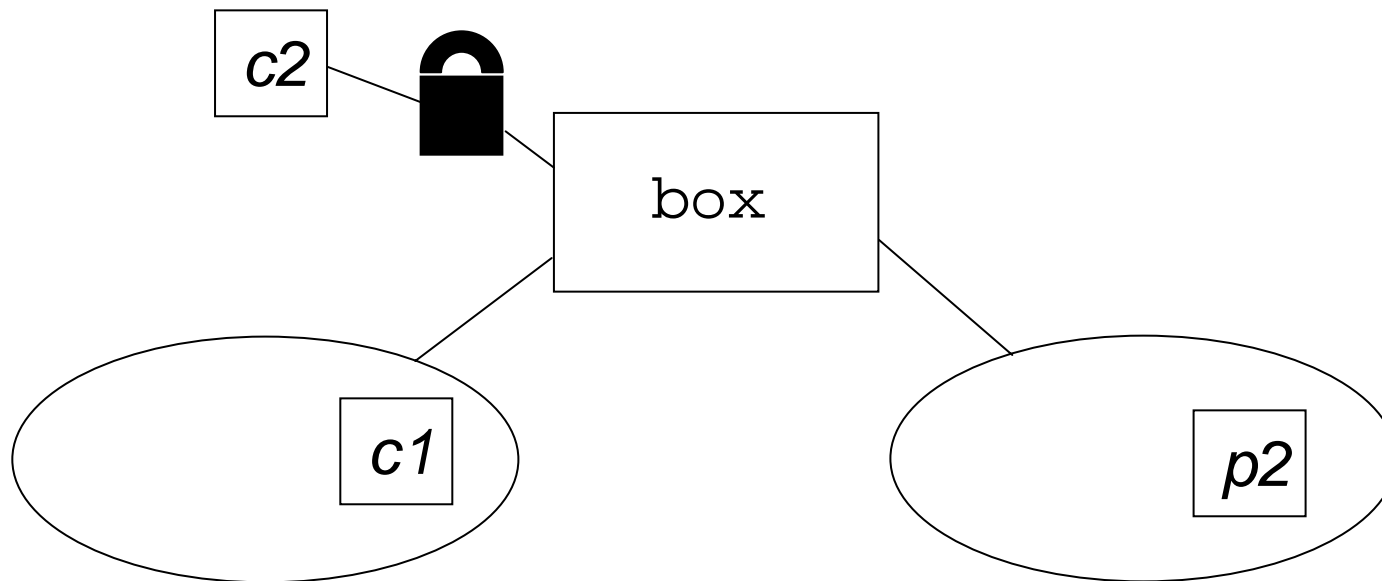
Typical Producer/Consumer Execution

- *p2* enters the `while` loop body (`empty` is `false`)
 - it invokes `wait()`, releases the object's lock, is placed in the wait set and blocks



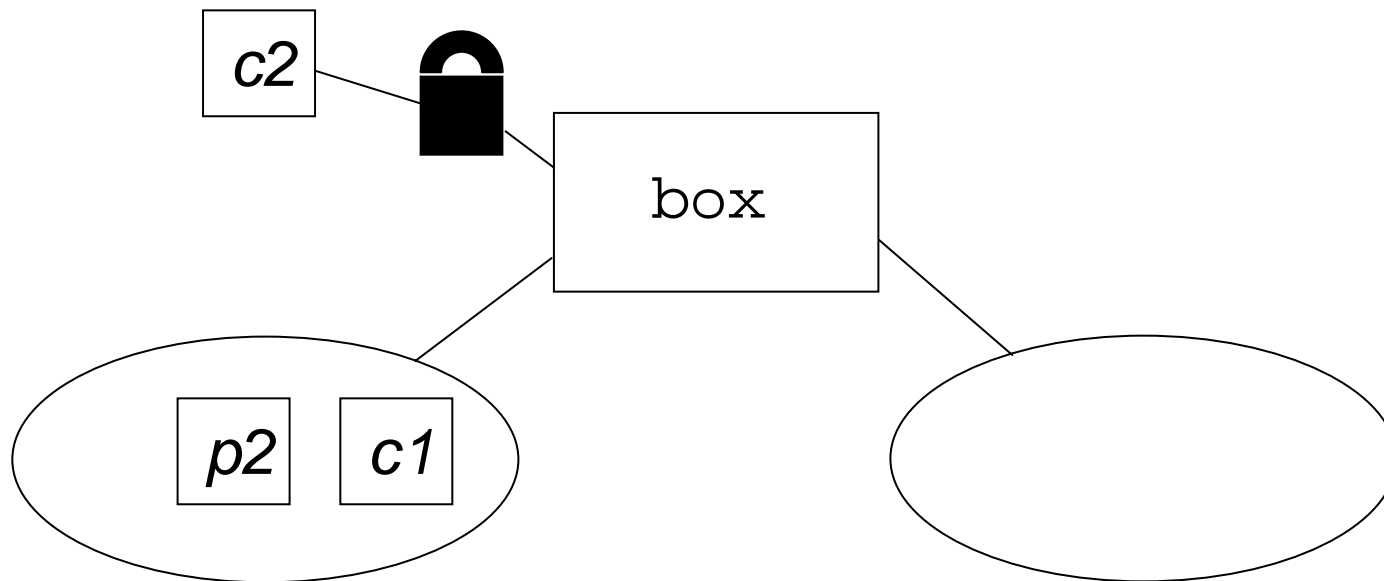
Typical Producer/Consumer Execution

- The JVM grants the lock to *c2* (it could have granted it to *c1*), which returns from `wait()`



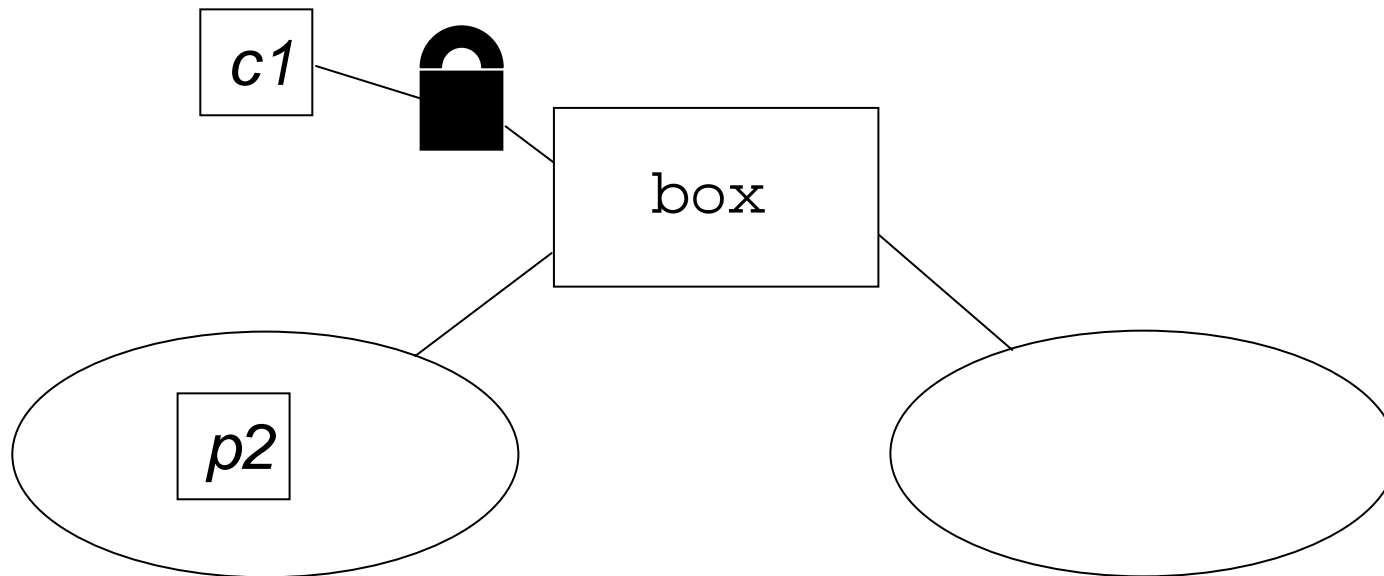
Typical Producer/Consumer Execution

- `c2` exits the `while` loop (`empty` is `false`), assigns `true` to `empty` and invokes `notifyAll()`
 - `p2` is reenabled for thread scheduling



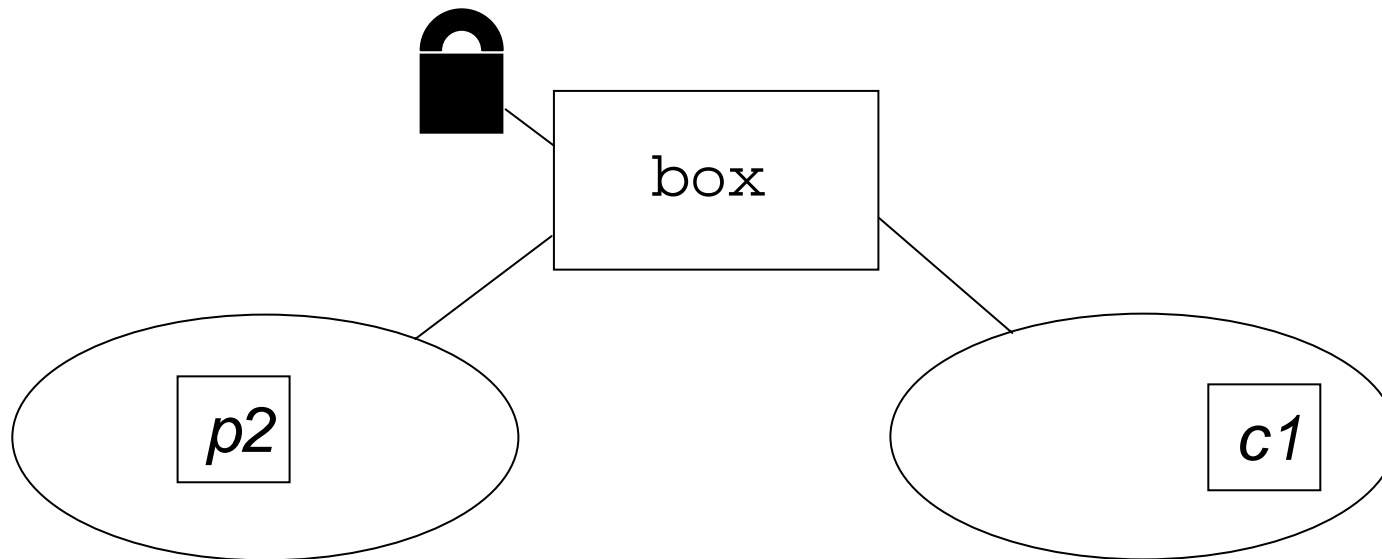
Typical Producer/Consumer Execution

- *c2* returns from `get ()` and relinquishes the lock
- The JVM grants the lock to *c1* (it could have granted it to *p2*), which returns from `wait ()`



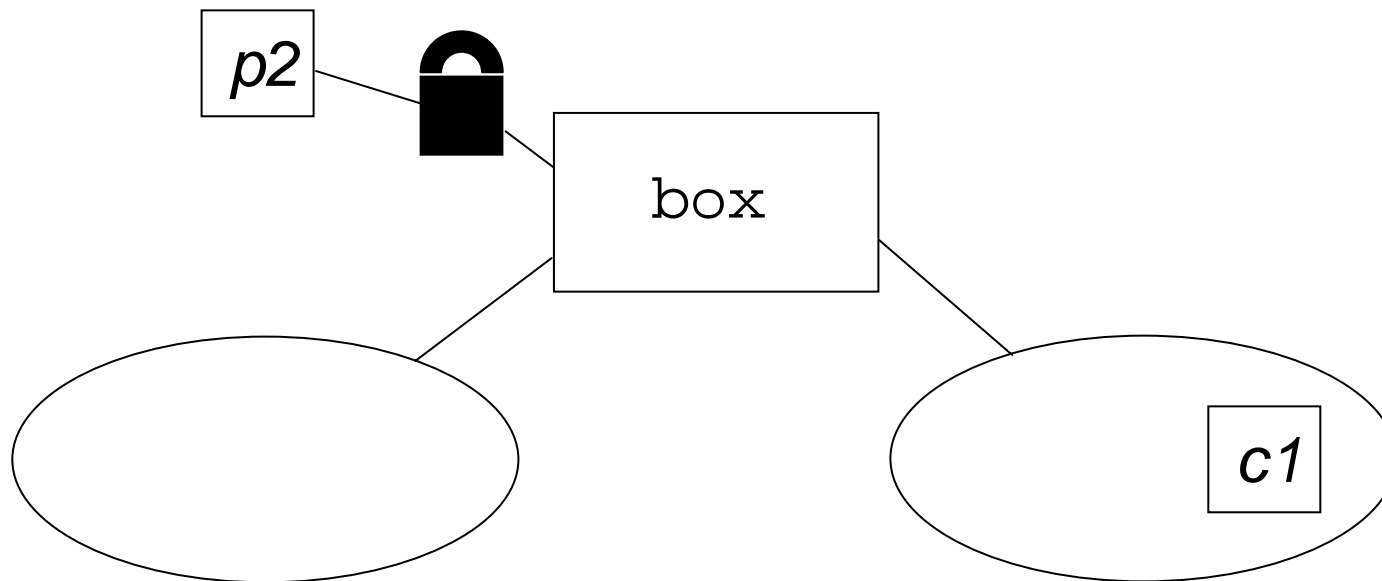
Typical Producer/Consumer Execution

- *c1* enters the `while` loop body (`empty` is `true`)
 - it invokes `wait()`, releases the object's lock, is placed in the wait set and blocks



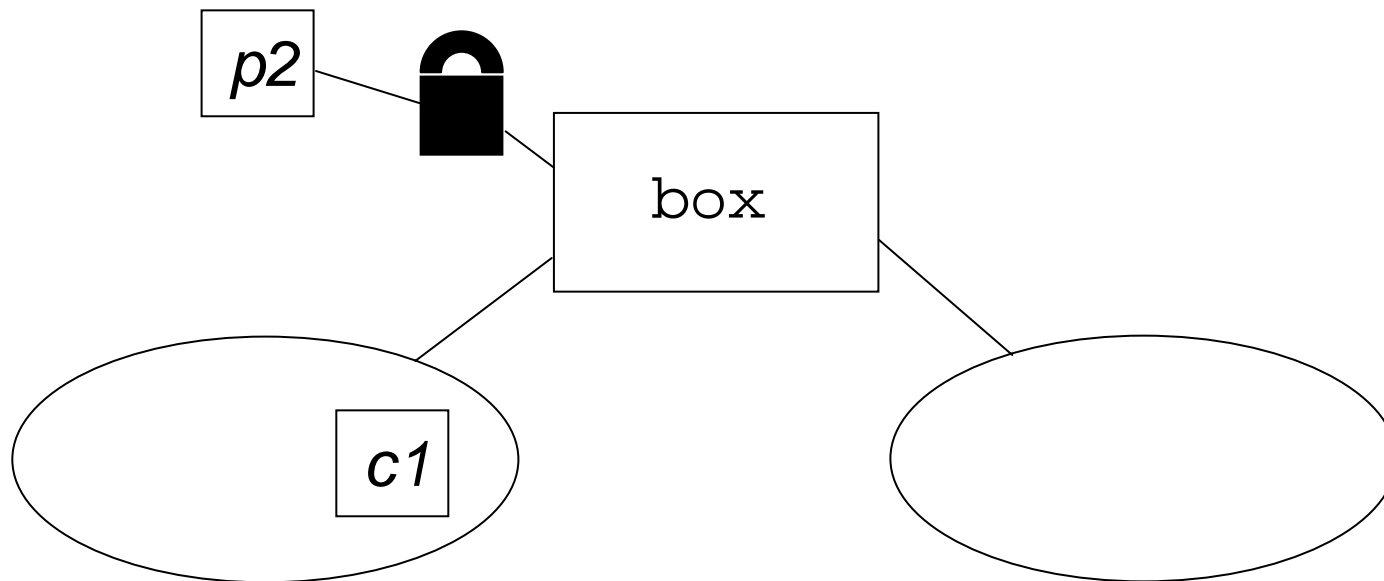
Typical Producer/Consumer Execution

- The JVM grants the lock to *p2*, which returns from `wait()`



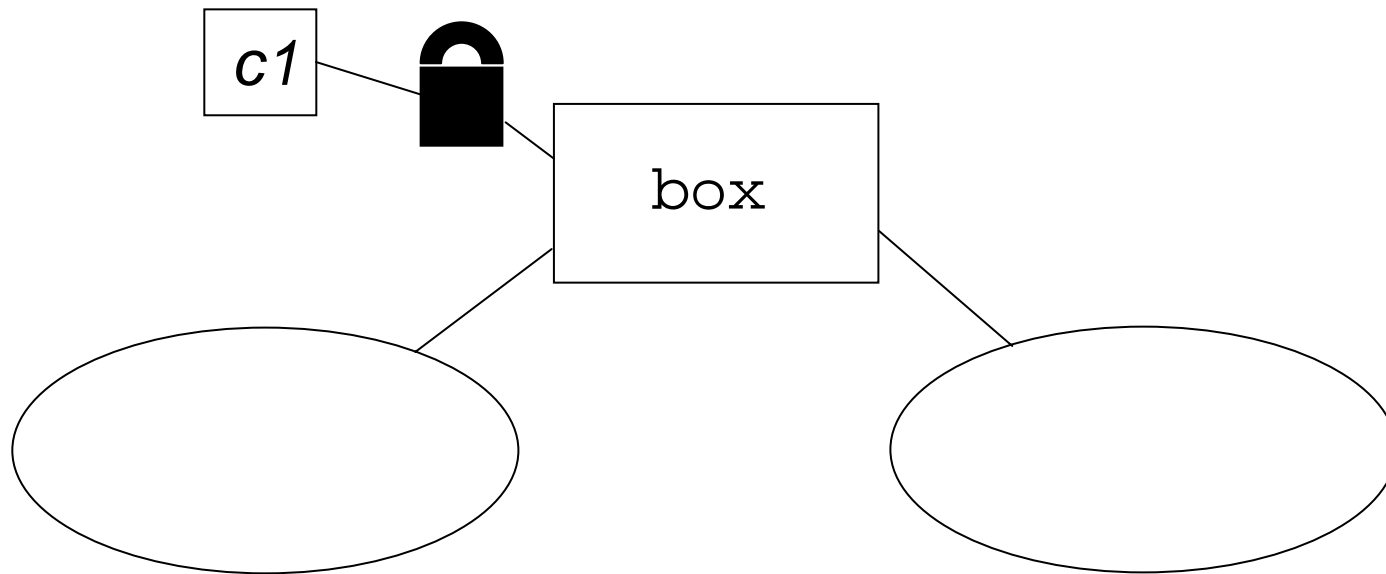
Typical Producer/Consumer Execution

- *p2* exits the `while` loop (`empty` is `true`), assigns `false` to `empty`, then invokes `notifyAll()`
 - *c1* is reenabled for thread scheduling



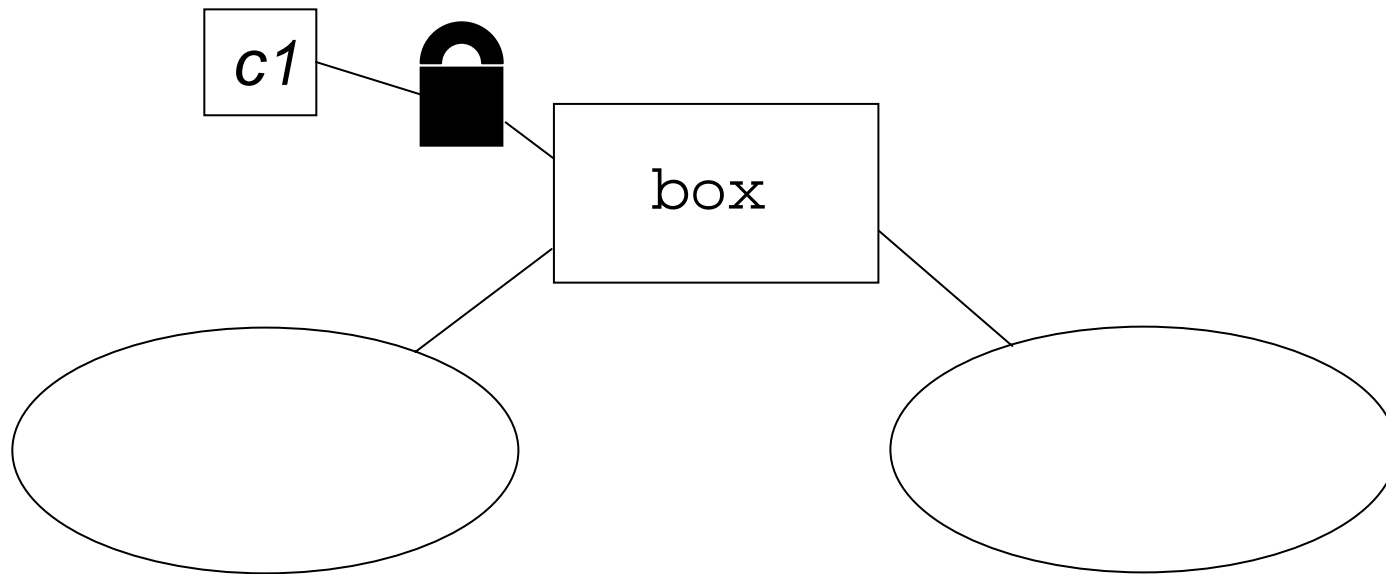
Typical Producer/Consumer Execution

- *p2* returns from `put ()` and relinquishes the lock
- The JVM grants the lock to *c1*, which returns from `wait ()`



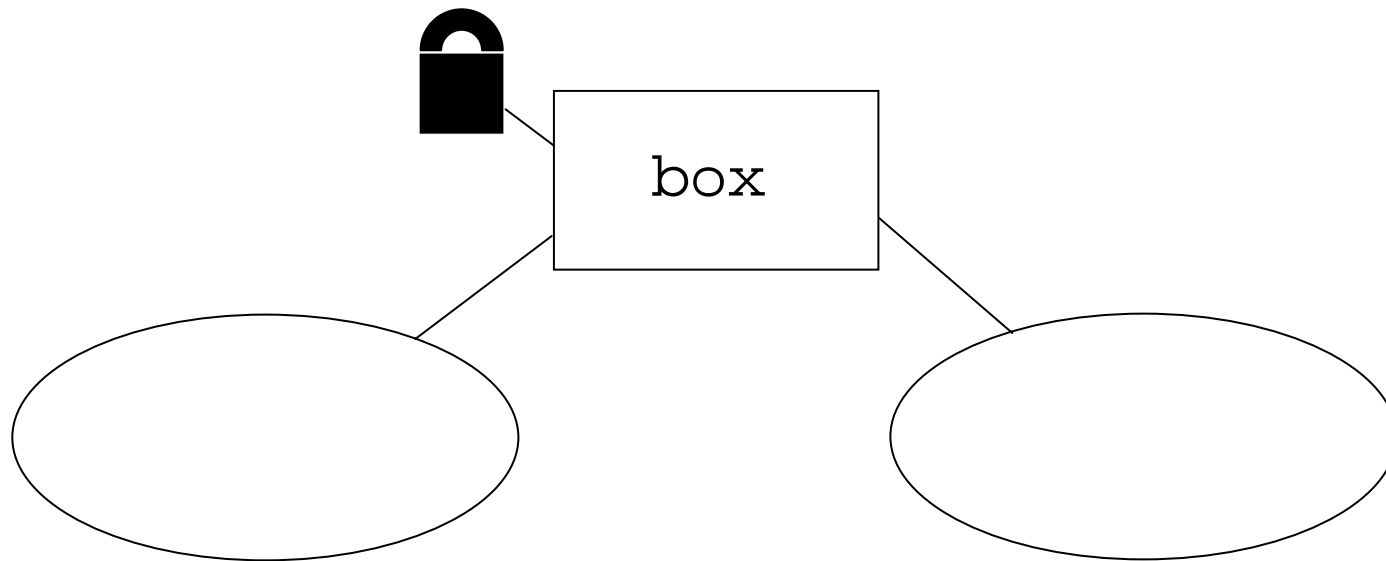
Typical Producer/Consumer Execution

- *c1* exits the `while` loop (`empty` is `false`), assigns `true` to `empty`, then invokes `notifyAll()`
- No threads are in the wait set, so this notification has no effect



Typical Producer/Consumer Execution

- *c1* returns from `get ()` and relinquishes the lock



The `wait()` Loop Idiom

- The standard idiom for using the `wait()` method is:

```
while (<condition does not hold>) {  
    try {  
        wait();  
    } catch (InterruptedException e) {  
        ...  
    }  
}  
perform action appropriate to condition
```

- Always use this idiom
- Never invoke `wait()` outside a loop

The wait() Loop Idiom

- The loop tests the condition before and after waiting
- Testing the condition before waiting and skipping `wait()` if the condition already holds ensures *liveness*
 - if the thread waits even though the condition already holds, the thread may never return from the wait (i.e., another thread may never invoke `notify()` or `notifyAll()` to reenable the thread)

The wait() Loop Idiom

- Testing the condition after waiting and waiting again if the condition does not hold ensures *safety*
 - if the reenabled thread proceeds when the condition does not hold, it can leave the object in an inconsistent state
- There are several reasons why a thread may be reenabled when the condition does not hold - see *Effective Java*, Item 50, for more information

Example: Producer/Consumer/Bounded Buffer

- Classic example for demonstrating mechanisms for condition synchronization and mutual exclusion between concurrent processes/tasks/threads
- Producer thread produces data
- Consumer thread consumes data
- To permit differing rates of data production and consumption, a bounded (fixed-capacity) buffer (a FIFO queue) shared by the threads is used to transfer data between them

Requirement for Mutual Exclusion

- Threads must not attempt to use the buffer simultaneously; otherwise, the buffer data structure may be damaged
 - buffer operations (e.g, add data, remove data) must be *mutually exclusive*
 - in other words, buffer operations must be *critical sections*
- Threads must synchronize to ensure that only one thread at a time executes the code in any critical section

Requirement for Condition Synchronization

- Producer must not attempt to add data to a full buffer
 - thread must wait for there to be room for at least one item of data
- Consumer must not attempt to remove data from an empty buffer
 - thread must wait for there to be at least one item of data in the buffer
- Waiting threads must relinquish the processor in order to allow other threads to run (no busy waiting!)

Classes for a Producer/Consumer/Bounded Buffer Program

- `class ProducerConsumer`
 - responsible for program initialization
- `class Producer`
 - code for the producer thread
- `class Consumer`
 - code for the consumer thread
- `class BoundedBuffer`
 - code for a thread-safe, fixed capacity, FIFO buffer

Program Initialization

```
public class ProducerConsumer
{
    public static void main(String[] args)
    {
        Thread producer, consumer;
        BoundedBuffer buffer;

        buffer = new BoundedBuffer();

        // Create the producer and consumer threads,
        // passing each thread a reference to the
        // shared BoundedBuffer object.
        producer = new Thread(new
            Producer(buffer), "Producer");
        consumer = new Thread(new
            Consumer(buffer), "Consumer");
    }
}
```

Program Initialization

```
        producer.start();  
        consumer.start();  
    }  
}
```

Class for Producer Thread

```
/**
 * Producer is the class for the producer thread.
 */
class Producer implements Runnable
{
    private BoundedBuffer buffer;

    public Producer(BoundedBuffer buf)
    {
        buffer = buf;
    }
}
```

Class for Producer Thread

```
public void run()
{
    for(int i = 0; i < 10; i++) {
        Integer item = new Integer(i);
        System.out.println(
            Thread.currentThread().getName()
            + " produced " + item);
        buffer.addLast(item);
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {}
    }
}
```

Class for Consumer Thread

```
/**
 * Consumer is the class for the consumer thread.
 */
class Consumer implements Runnable
{
    private BoundedBuffer buffer;

    public Consumer(BoundedBuffer buf)
    {
        buffer = buf;
    }
}
```

Class for Consumer Thread

```
public void run()
{
    for(int i = 0; i < 10; i++) {
        Object item = buffer.removeFirst();
        System.out.println(
            Thread.currentThread().getName()
            + " consumed " + item);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
}
```

Class for Bounded Buffer

```
public class BoundedBuffer
{
    // a simple ring buffer is used to hold the data

    // buffer capacity
    private static final int SIZE = 5;
    private Object[] buffer = new Object[SIZE];
    private int inIndex = 0, outIndex = 0, count = 0;

    // If true, there is room for at least one object
    // in the buffer.
    private boolean writeable = true;

    // If true, there is at least one object stored
    // in the buffer.
    private boolean readable = false;
```

Adding an Object to the Bounded Buffer

```
public synchronized void addLast(Object item)
{
    while (!writeable) {
        try {
            wait();
        } catch (InterruptedException e) {
            System.err.println(e);
        }
    }

    buffer[inIndex] = item;
    readable = true;
```

Adding an Object to the Bounded Buffer

```
    inIndex = (inIndex + 1) % SIZE;
    count++;
    if (count == SIZE)
        writeable = false;

    notifyAll();
}
```

Removing an Object from the Bounded Buffer

```
public synchronized Object removeFirst()
{
    Object item;

    while (!readable) {
        try {
            wait();
        } catch (InterruptedException e) {
            System.err.println(e);
        }
    }

    item = buffer[outIndex];
    writeable = true;
```

Removing an Object from the Bounded Buffer

```
        outIndex = (outIndex + 1) % SIZE;
        count--;
        if (count == 0)
            readable = false;

        notifyAll();

        return item;
    }
}
```

notifyAll() vs. notify()

- How do we choose which method to use?
- `notifyAll()` will always guarantee correct results because the threads that need to be awakened will be awakened
- It will also awake threads that didn't need to be awakened, but this doesn't affect correctness
 - they will determine that the condition on which they are waiting does not hold and go back to waiting

`notifyAll()` vs. `notify()`

- We can use `notify()` instead of `notifyAll()` if all threads in the wait set are waiting on the same condition and only one thread at a time can benefit from the condition becoming true
- But, using `notifyAll()` instead of `notify()` protects against accidental or malicious waits by unrelated threads
 - such waits could swallow a notification sent by `notify()`, leaving the intended recipient waiting indefinitely

notifyAll() vs. notify()

- On the other hand, `notifyAll()` can degrade the performance of certain data structures from $O(n)$ to $O(n^2)$ where n is the number of waiting threads
- If n is small, this is usually not a problem in practice

Exercise for the Student

- Could we replace the `notifyAll()` statements in `Box` with `notify()` statements? Explain why or why not. If not, give an example that demonstrates the problems that could result.
- Could we replace the `notifyAll()` statements in `BoundedBuffer` with `notify()` statements? Explain why or why not. If not, give an example that demonstrates the problems that could result.

sleep() VS. wait()

- A thread must never invoke `sleep()` as a substitute for `wait()/notify()/notifyAll()`
- `sleep()` does not relinquish any locks held by the thread
 - invoking `sleep()` within a synchronized method does not allow another thread to enter the monitor while the first thread sleeps

Deprecated Thread Methods

- The `stop()`, `suspend()` and `resume()` methods in class `Thread` have been *deprecated* and should never be invoked
 - `stop()` - forces the thread to stop, regardless of what it is doing
 - this method is dangerous
 - all locks held by the thread are unlocked, and the objects protected by those locks may be in an inconsistent state, but can now be accessed by other threads

Deprecated Thread Methods

- `suspend()` - suspends the thread
- `resume()` - resumes the suspended thread
- these two methods are deprecated because `suspend()` is deadlock prone
 - the suspended thread holds onto any locks that it owns when it is suspended
 - consider what would happen if the thread that invokes `resume()` to resume the suspended thread first tried to acquire the lock held by the suspended thread