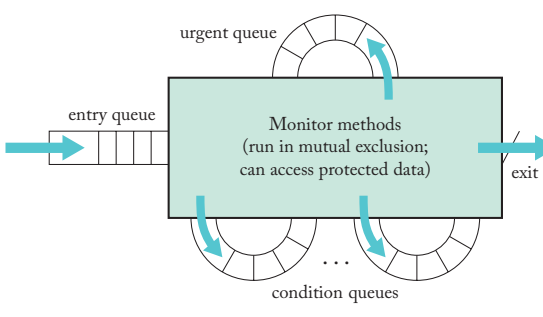
==========
Language-level Synchronization

Scheduler-based locks in many languages (C/C++, Rust, Scala, Ruby, ...)

Monitors

Attempt to address the two weaknesses of semaphores previously discussed. Suggested by Dijkstra, developed more thoroughly by Brinch Hansen, and formalized nicely by Hoare (a real cooperative effort!) in the early 1970s. Several parallel programming languages have incorporated monitors as their fundamental synchronization mechanism. None, to my knowledge, incorporates the precise semantics of Hoare's formalization.

A monitor is a shared object with operations, internal state, and a number of ***condition queues***. Only one operation of a given monitor may be active at a given point in time. A thread that calls a busy monitor is delayed until the monitor is free. On behalf of its calling thread, any operation may suspend itself by ***waiting*** on a condition. An operation may also ***signal*** a condition, in which case one of the waiting threads is resumed, usually the one that waited first.



The precise semantics of mutual exclusion in monitors are the subject of considerable dispute. Hoare's original proposal remains the clearest and most carefully described. It specifies two bookkeeping queues for each monitor: an ***entry queue***, and an ***urgent queue***. When a thread executes a signal operation from within a monitor, it waits in the monitor's urgent queue and the first thread on the appropriate condition queue obtains control of the monitor. When a thread leaves a monitor it unblocks the first thread on the urgent queue or, if the urgent queue is empty, it unblocks the first thread on the entry queue instead.

The two main semantic controversies:

(1) Should a `signal`-er keep going, rather than moving to the urgent queue and letting the `wait`-er in? That reduces context switches but requires that we treat signals as "hints" instead of "absolutes". The idiom

```
if not condition wait
```

becomes

```
while not condition wait
```

(2) The "nested monitor problem": A calls M1.e1, which calls M2.e2, which waits. Should A release exclusion on M1? If it does, the world may change before it returns, and it may not even be able to resume, if some other thread enters and locks M1. If A doesn't release M1, however, B may not be able to pass through M1 to reach M2 to signal A.
The most elegant solution I know of (but which I don't think anybody implements) was suggested by Wettstein: to make signals hints, release all monitors, re-acquire them all ***outermost first*** on wakeup, and require that invariants hold when making nested calls.
Java does not release outer monitors.

Building a correct monitor requires that one think about the ***monitor invariant***. (Everybody remember loop invariants?)
The monitor invariant is a predicate that captures the notion "the state of the monitor is consistent." It needs to be true initially, and at monitor exit. It also needs to be true at every `wait` statement. In Hoare's formulation, needs to be true at every `signal` operation as well, since some other thread may immediately run.

Hoare's definition of monitors in terms of semaphores makes clear that semaphores can do anything monitors can. The inverse is also true; it is trivial to build a semaphores from monitors (if you don't see how, you should figure it out :-).

Concurrency in Java

Explicit threads (in Java from the beginning)

```
class Foo extends Thread {
   public Foo ...  // constructor; does not start thread running

   public void run() {
      // this is where the thread starts running
   }
}

Foo f = new Foo;
// returns when constructor is done
f.start()
   // puts new thread on the ready list,
   //    set up to execute its run routine

...
f.join()
   // optional; waits for f to finish (return from its run method)
```

`start()` is implemented by `Thread`. It calls `run()`.
In classes derived from `Thread` you should always override `run`, and you should make threads begin execution by calling `start()`.
Never override `start()`. Never call `run()`.

`Executors` (introduced in Java 5). Allow caching of threads to avoid start-up / shut-down costs. Also abstract out the physical parallelism, so you can have N tasks run by M ≤ N threads under the hood.

```
class Foo implements Runnable {
   ...
   // constructor and run() method same as before
}
...
ExecutorService pool = Executors.newCachedThreadPool();
...
pool.execute(new Foo( constructor_args ));
...
// indicate that pool will never get any additional tasks
pool.shutdown();
// wait for all workers to complete
try {
   pool.awaitTermination(60, TimeUnit.SECONDS);
} catch(InterruptedException e) {}
```

`Runnables` are ***object closures***. They're useful for other things besides concurrency -- basically anything you want to package up for future execution. There's also a `Callable` that produces a value that can be picked up later.

Can also use `newFixedThreadPool(numThreads)`
or `newSingleThreadExecutor()`
These are all ***factory*** methods that create and manage `Executor` objects.

Java 2 synchronization (really should be generic; leaving that out for simplicity):

```
class BB {
   final private int SIZE = 10;
   private Object[] buf = new Object[SIZE];

   private int nextEmpty = 0;
   private int nextFull = 0;
   private int fullSlots = 0;

   synchronized public void insert(Object d)
         throws InterruptedException {
      while (fullSlots == SIZE) {
         wait();
      }
      buf[nextEmpty] = d;
      nextEmpty = (nextEmpty + 1) % SIZE;
      ++fullSlots;
      notifyAll();       // explain why!
   }

   synchronized public Object remove()
         throws InterruptedException {
      while (fullSlots == 0) {
         wait();
      }
      Object d = buf[nextFull];
      nextFull = (nextFull + 1) % SIZE;
      --fullSlots;
      notifyAll();       // explain why!
      return d;
   }
}
```

Several operations (e.g. `wait`, `join`, `sleep`) can throw `InterruptedException`. Any time you call one of these you have to be either (a) inside a try block with a handler (`catch` clause) for `InterruptedException`, or (b) inside a method whose header indicates "`throws InterruptedException`". Kind of a nuisance.

What is `InterruptedException` for? It's the way to get a waiting thread to wake up. If `t` is of class `Thread`, `t.interrupt()` will cause `t` to experience an `InterruptedException` the next time it calls a blocking operation. Note that there is no acceptable way to cause an exception in a non-blocked thread – all such mechanisms have been deprecated.

Notes:

1) You can also use a synchronized ***statement*** (alternative to synchronized method):

```
synchronized(my_obj) {
   // critical section
}
```

2) There is only a single queue associated with each object. If you have threads that may wait for more than one reason, you need to worry about the "wrong kind" of thread waking up:

```
if (!condition) {
   wait();
}
while (!condition) {
   notify();
   wait();
}
```

This can be expensive. In some cases you can get around it by waiting in multiple sub-objects, but this doesn't work in general because a waiting thread relinquishes only the ***inner*** lock, not any outer ones (leading to deadlock if the releasing thread has to get into the same outer objects).

3) A single thread can acquire the lock on a single object multiple times (doesn't exclude itself). If it waits, it releases the lock "the appropriate number of times." When it awakes, it will have re-acquired the lock "the appropriate # of times," and must leave that many synchronized methods or statements (or wait again) before anybody else can get in.

4) `my_obj.notifyAll()` will wake up ***all*** threads waiting on `my_obj`.

5) The lock on an object is associated with the data in the object only by convention. Java guarantees that if one thread releases a lock and a second then acquires it, the second sees all previous writes ***to all data*** by the first.

BB solution in Java 2 without using `notifyAll` is quite a bit harder:

```
class BB {
   final private int SIZE = 10;
   private Object[] buf = new Object[SIZE];

   private Object producerMutex = new Object();
   // waited upon only by producers; protects the following:
      private int nextEmpty = 0;
      private int emptySlots = SIZE;

   private Object consumerMutex = new Object();
   // waited upon only by consumers; protects the following:
      private int nextFull = 0;
      private int fullSlots = 0;

   public void insert(Object d) throws InterruptedException {
      synchronized (producerMutex) {
         while (emptySlots == 0) {
            producerMutex.wait();
         }
         --emptySlots;
         buf[nextEmpty] = d;
         nextEmpty = (nextEmpty + 1) % SIZE;
      }
      synchronized (consumerMutex) {
         ++fullSlots;
         consumerMutex.notify();
      }
   }

   public Object remove() throws InterruptedException {
      Object d;
      synchronized (consumerMutex) {
         while (fullSlots < 0) {
            consumerMutex.wait();
         }
         --fullSlots;
         d = buf[nextFull];
         nextFull = (nextFull + 1) % SIZE;
      }
      synchronized (producerMutex) {
         ++emptySlots;
         producerMutex.notify();
      }
      return d;
   }
}
```

Solution using Java 5 locks is efficient and arguably more algorithmically elegant, but syntactically more cluttered due to library-based synchronization:

```
class BB {
   final private int SIZE = 10;
   private Object[] buf = new Object[SIZE];

   private int nextEmpty = 0;
   private int nextFull = 0;
   private int fullSlots = 0;

   Lock l = new ReentrantLock();
   final Condition emptySlot  = l.newCondition();
   final Condition fullSlot = l.newCondition();

   public void insert(Object d) throws InterruptedException {
      l.lock();
      try {
         while (fullSlots == SIZE) {
            emptySlot.await();
         }
         buf[nextEmpty] = d;
         nextEmpty = (nextEmpty + 1) % SIZE;
         ++fullSlots;
         fullSlot.signal();
      } finally {
         l.unlock();
      }
   }

   public Object remove() throws InterruptedException {
      l.lock();
      try {
         while (fullSlots == 0) {
            fullSlot.await();
         }
         Object d = buf[nextFull];
         nextFull = (nextFull + 1) % SIZE;
         --fullSlots;
         emptySlot.signal();
         return d;
      } finally {
         l.unlock();
      }
   }
}
```