Passing data from one thread to another (Inter-process Communication)

You need some form of a synchronization mechanism to modify objects between multiple threads. If you don't use a specialized thread safe collection (these are available in .NET 4), you need to lock using a monitor. Usually, a more appropriate collection type for the **producer/consumer** pattern is a **Queue** (a FIFO collection), instead of a List:

Plain Queue with explicit locking (System.Collections)

```
. . .
private readonly object _lock = new object();
private readonly Queue<Item> _queue = new Queue<Item>();
private readonly AutoResetEvent _signal = new AutoResetEvent();
. . .
void ProducerThread()
{
    while (ShouldRun)
    {
        Item item = GetNextItem();
        // you need to make sure only one thread can access the list
        // at a time
        lock (_lock)
        {
            _queue.Enqueue(item);
        }
        // notify the waiting thread
        _signal.Set();
    }
}

. . .
```

And in the consumer thread, you need to fetch the item and process it:

```
. . .
void ConsumerThread()
{
    while (ShouldRun)
    {
        // wait to be notified
        _signal.WaitOne();
        Item item = null;
        do
        {
           item = null;
           // fetch the item, but only lock shortly
           lock (_lock)
           {
               if (_queue.Count > 0)
                  item = _queue.Dequeue(item);
           }
```

```
            if (item != null)
            {
                // do stuff
            }
        }
        while (item != null); // loop until there are items to collect
    }
}

. . .
```

Starting with .NET 4, there is a ConcurrentQueue<T> collection, a thread-safe FIFO, which removes the need to lock while accessing it and simplifies the code:

ConcurrentQueue (System.Collections.Concurrent)

```
. . .
private readonly ConcurrentQueue<Item> _queue = new ConcurrentQueue<Item>();
. . .
void ProducerThread()
{
    while (ShouldRun)
    {
        Item item = GetNextItem();
        _queue.Enqueue(item);
        _signal.Set();
    }
}
. . .
void ConsumerThread()
{
    while (ShouldRun)
    {
        _signal.WaitOne();
        Item item = null;
        while (_queue.TryDequeue(out item))
        {
            // do stuff
        }
    }
}

. . .
```

Finally, if you only wish that your consumer thread gets items in chunks periodically, you would change this to:

ConcurrentQueue with threshold (10 sec. or 10 items)

```
. . .
```

```
private readonly ConcurrentQueue<Item> _queue = new ConcurrentQueue<Item>();
. . .
void ProducerThread()
{
    while (ShouldRun)
    {
        Item item = GetNextItem();
        _queue.Enqueue(item);
        // more than 10 items? panic!
        // notify consumer immediately
        if (_queue.Count >= 10)
           _signal.Set();
    }
}
. . .
void ConsumerThread()
{
    while (ShouldRun)
    {
        // wait for a signal, OR until
        // 10 seconds elapses
        _signal.WaitOne(TimeSpan.FromSeconds(10));
        Item item = null;
        while (_queue.TryDequeue(out item))
        {
           // do stuff
        }
    }
}

. . .
```

Exercise: This pattern is so useful that it's nice to abstract it into a generic class which delegates producing and consuming to external code. It would be a good exercise to make it generic.

You will also need a Stop method which will probably set a volatile bool flag indicating that it's time to stop, and then set the signal to unpause the consumer and allow it to end.