Programming Skills
C#/.NET

Threads
Process — execution of assembly.
Thread — program counter (+ stack).
Management.
Synchronization.
<table>
<thead>
<tr>
<th>Class/Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>manage one program counter.</td>
</tr>
<tr>
<td>ThreadPool</td>
<td>(provided by system.)</td>
</tr>
<tr>
<td>ThreadStart</td>
<td>delegate to wrap program counter.</td>
</tr>
<tr>
<td>ParameterizedThreadStart</td>
<td></td>
</tr>
<tr>
<td>ThreadInterruptedException</td>
<td>happens on <code>Interrupt()</code>.</td>
</tr>
<tr>
<td>Monitor</td>
<td>functions for Hoare-style synchronization.</td>
</tr>
<tr>
<td>Interlocked</td>
<td>functions for atomic operations.</td>
</tr>
<tr>
<td>Mutex</td>
<td>interprocess synchronization.</td>
</tr>
<tr>
<td>ReaderWriterLock</td>
<td>one writer, multiple readers.</td>
</tr>
<tr>
<td>Semaphore</td>
<td>counts calls and blocks, can be systemwide.</td>
</tr>
<tr>
<td>Timer</td>
<td>call method at fixed intervals.</td>
</tr>
</tbody>
</table>
### Thread

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentThread</td>
<td>static, returns object for current thread.</td>
</tr>
<tr>
<td>Sleep(msec)</td>
<td>static, immediately affects current thread.</td>
</tr>
<tr>
<td>Start()</td>
<td>activates program counter.</td>
</tr>
<tr>
<td>Interrupt()</td>
<td>sends exception once receiver is sleeping, waiting, or joining (not suspended).</td>
</tr>
<tr>
<td>Abort()</td>
<td>terminates receiver, can be intercepted.</td>
</tr>
<tr>
<td>Join()</td>
<td>waits for receiver to terminate.</td>
</tr>
<tr>
<td>IsBackground</td>
<td>application terminates if only background threads remain.</td>
</tr>
</tbody>
</table>
public class Many {
    public Many (int retry, string info) {
        this.retry = retry; this.info = info;
    }
    protected readonly int retry; protected readonly string info;
    public virtual void Run () {
        for (var n = 0; n < retry; ++ n) work();
        quit();
    }
    protected virtual void work () {
        Console.Write(info);
    }
    protected virtual void quit () {
        Console.WriteLine();
    }
    public static void Main (string[] args) {
        foreach (var arg in args)
            new Thread(
                new ThreadStart(new Many(args.Length, arg).Run)
            ).Start();
    }
}
public class Randy : Many {
    protected static int nRandy;
    protected Random random = new Random();
    public override void Run () {
        ++ nRandy; base.Run();
    }
    protected override void quit() { // deliberate race
        int nr = nRandy;
        Thread.Sleep(random.Next(1000));
        if (nr == 1) base.quit(); else nRandy --;
    }
    public static new void Main (string[] args) {
        if (args != null && args.Length > 0) {
            Thread[] t = new Thread [args.Length];
            int n = 0;
            foreach (string arg in args)
                t[n++] = new Thread(
                    new ThreadStart(new Randy(args.Length, arg).Run)
                );
            foreach (Thread thread in t)
                thread.Start();
        } }
}
public class Morse : Randy {
    // ...
    private static readonly object monitor = new object();
    public override void Run () {
        lock(monitor)
            ++ nRandy;
        // cannot call base.Run() for many.Run()
        for (int n = 0; n < retry; ++ n) work();
        quit();
    }
    protected override void quit() {
        lock(monitor)
            base.quit();
    }
}

Multi-thread access to the global nRandy is protected by lock(...)
but there is a glitch...

cd code\threads; nmake morse
## Monitor

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Enter(object)</code></td>
<td>one at a time, usually as <code>lock(object)</code>. Object must be a reference type.</td>
</tr>
<tr>
<td><code>TryEnter(object)</code></td>
<td>false if not possible.</td>
</tr>
<tr>
<td><code>Exit(object)</code></td>
<td>leave, let next one in.</td>
</tr>
<tr>
<td><code>Wait(object)</code></td>
<td>wait until pulsed, then let next one in.</td>
</tr>
<tr>
<td><code>Pulse(object)</code></td>
<td>pulse one waiting thread.</td>
</tr>
<tr>
<td><code>PulseAll(object)</code></td>
<td>pulse all waiting threads.</td>
</tr>
</tbody>
</table>
lock (object) statement

is syntactic sugar for

    Monitor.Enter(object);
    try {
        statement
    } finally {
        Monitor.Exit(object);
    }

object must have a reference type.
Timing demonstrates that `Sleep` does not release a monitor.


cd code\threads; nmake sleep
public static void Main (string[] args) { // single producer
    Thread[] threads = new Thread[args.Length];
    for (int n = 0; n < args.Length; ++ n)
        (threads[n] = new Thread(new ThreadStart(new Any(n).Run)))
            .Start();
    // produce information
    for (int n = 0; n < args.Length; ++ n)
        for (int m = 0; m <= args.Length; ++ m)
            lock(monitor) {
                while (info != null) Monitor.Wait(monitor);
                info = m == args.Length ? "" : args[m];
                Monitor.Pulse(monitor);
            }
    // wait for each consumer to terminate
    for (int n = 0; n < threads.Length; ++ n)
        threads[n].Join();
}

cd code\threads; nmake any
public void Run () { // multiple consumers
    while(true) {
        // acquire a private copy of the string
        string copy;
        lock(monitor) {
            while (info == null) Monitor.Wait(monitor);
            copy = info; info = null;
            Monitor.PulseAll(monitor);
        }
        // terminate on empty string
        if (copy.Equals("")) break;
        // reschedule threads
        Thread.Sleep(0);
        // publish information
        Console.WriteLine(n + " " + copy);
    }
}
**Semaphores (Dijkstra, 1970)**

**semaphore.cs**

\( P() \) (*passen*) decrements and blocks, waiting on a nonnegative value.

\( V() \) (*verlaten*) increments and releases.

**producer.cs**

uses semaphores to solve a one-producer-many-consumers problem.

cd code\threads; nmake producer
C# — nested classes

Outer static class is sealed, has only static methods and no instances.

Inner class is like a static inner class in Java, i.e., it can only access encompassing static things.

Inner class can simulate Java member class by passing context object as a parameter; however, access to context’s state must be arranged explicitly.
A philosopher is *(cogito ergo sum)* and eats. Five share five forks, but each needs two to eat. Deadlock problem.

One approach: yield fork if unsuccessful. One philosopher has alternate grabbing order.

philosopher.cs

cd code\threads; nmake philosopher