This topic outlines the main features of used in one of the current mainstream styles of concurrent programming: multiple threads of execution, with interference controlled by enforcing mutual exclusion via locks.
What is concurrent programming?

- **Concurrent programming** broadly involves building programs that involve many things happening at the same time.

- There are many possible reasons for wanting concurrency.
  - In order to deal with events in the “real world” (the real world is naturally concurrent)
  - To allow a server to respond to requests from many sources.
  - To keep a user interface responsive while a program is busy.
  - To utilize the CPU while I/O is being performed.
  - To utilize multiple cores, CPUs, or computers (called parallelism).

- Many programs are not concurrent.
  - The basic imperative model of a program involves following a sequence of instructions, one after another.
  - This is usually called sequential programming, and is generally considered the opposite of concurrency.
  - The sequence of instructions is called a thread of execution.
Threads

- Creation of multiple threads is the basis for most forms of concurrent programming, and is supported by the OS.
- As each thread runs, it keeps track of its own position in the program, and the values of any local variables it has created, etc.
- Threads may run at different speeds, and their effects (I/O, memory writes, etc.) are interleaved.
  - A concurrent program is considered correct only when every possible interleaving produces a correct result.
- Generally, threads must communicate in some way to be useful.
  - Shared-memory concurrency allows threads to have some variables and data that many threads can read and write.
  - Message passing instead requires explicitly sending data to other threads. (If the other thread is on another computer, we have a distributed program.)
- We’ll see shared memory first, and see how to create threads in F#, coordinate them and avoid them interfering with each other.
  - Thread support is in the .NET libraries, hence not specific to F#
  - Java thread support is very similar – we’ll see it briefly later.
Creating threads

- Creating new threads in F# is quite simple via forking an existing thread:
  - An existing thread creates a thread object via System.Threading.Thread
  - It passes a function for the thread to run, converted to a ThreadStart
  - And then it starts it running via the Start() method.
  - Both threads then run concurrently, and interleave their effects.

```fsharp
open System.Threading
let t = Thread(ThreadStart(fun () ->
    // See topic 15 for abbrev. syntax
    printfn "Thread %d: Hello"
    Thread.CurrentThread.ManagedThreadId))

t.Start();
printfn "Thread %d: Waiting!" Thread.CurrentThread.ManagedThreadId

// Join waits for the thread to complete

val t : System.Threading.Thread

// This is the output when the above is run
Thread 1: Waiting!
Thread 5: Hello
Done!
```
Another way to start threads

- The thread class creates a raw .NET thread (an OS thread).
- `System.ComponentModel.BackgroundWorker` achieves the same thing, but integrates better with .NET events, GUIs, etc.
  - Adding functions to the events `DoWork` and `RunWorkerCompleted` causes them to be run when those events occur.

```csharp
open System.ComponentModel
open System.Windows.Forms
let worker = new BackgroundWorker()
let numIterations = 1000

worker.DoWork.Add (fun args ->
    let rec computeFibonacci resPrevPrev resPrev i =
        let res = resPrevPrev + resPrev
                // Compute next result
        if i = numIterations
           then args.Result <- box res
                // write the result
        else computeFibonacci resPrev res (i+1)
                // continue
    computeFibonacci 1 1 2)

worker.RunWorkerCompleted.Add
    (fun args-> MessageBox.Show(sprintf "Result = %A" args.Result)|> ignore)

worker.RunWorkerAsync() // Execute the worker
```
Race conditions

- Now, suppose we have a situation where two threads may modify some variables:

```csharp
type MutablePair<'a,'b>(x:'a,y:'b) =
let mutable currentX = x
let mutable currentY = y
member p.Value = (currentX,currentY)
member p.Update(x,y) =
    currentX <- x;                           // These updates are not atomic: other
    currentY <- y                            // threads may interleave between them

let p = new MutablePair<_,_>(1,2)
Thread(ThreadStart(fun args ->
    do (while true do p.Update(10,10)))).Start()
Thread(ThreadStart(fun args ->
    do (while true do p.Update(20,20)))).Start()
```

- Here we could end up with (10, 20) in x and y, when:
  - First the second thread updates x to 20
  - Then the first thread updates both x and y to 10 (i.e. it interleaves)
  - Then the second thread updates y to 20

- This is called a race condition – the exact timing affects the result.
- The result is not consistent with either of the updates requested.
Locks/Monitors

- Such issues can be fixed via monitors which allow threads to “lock” each another out while they complete a sequence of operations.

- Threads can request the lock associated with any object.
  - This includes objects, ref cells and records, but not int, float, char, ...

- Only one thread can hold the lock on a particular object at a time.

```csharp
type MutablePair<'a,'b>(x:'a,y:'b) =
    let mutable currentX = x
    let mutable currentY = y
    member p.Value = (currentX,currentY)
    member p.Update(x,y) =
        currentX <-x
        currentY <- y
        // These updates are “atomic” - other
        // threads see either both or neither

let p = new MutablePair<int,int>(1,2)
Thread(ThreadStart(fun args -> do (while true do
    lock p (fun () -> p.Update(10,10))
    ) ) ).Start()

Thread(ThreadStart(fun args -> do (while true do
    lock p (fun () -> p.Update(20,20))
    ) ) ).Start()
```