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 and monitors.
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

- Allowing multiple threads is the basic model for most forms of concurrent programming.
  - Threads are generally supported by the OS, or sometimes by the runtime.

- 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.
Threading libraries and abbreviations

- Support for threading is in the .NET namespace `System.Threading`.
- Some convenient abbreviations we’ll use in this unit for creating threads, and the basic monitor operations (coming):
  - These are all you need for the project, plus `lock` (in a few slides).

```ml
open System
open System.Threading

/// Make a thread from a function
let mkThread f = Thread (ThreadStart f)

/// Make a thread from a function and start it immediately
let startThread f = (mkThread f).Start()

/// wait until another thread signals the given object/ref has changed
let wait obj = ignore(Threading.Monitor.Wait obj)

/// wake up all threads waiting on a given object/ref
let wakeWaiters obj = Threading.Monitor.PulseAll obj
```
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
- Here we use mkThread, which includes converting to a ThreadStart
- The code then starts the thread running via the Start() method.
- Both threads then run concurrently, and interleave their effects.

```fsharp
let t =
    mkThread (fun () -> printfn "Thread %d" Thread.CurrentThread.ManagedThreadId)

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

    t.Join(); // Join waits for the thread to complete
    printfn "Thread %d: Done!" Thread.CurrentThread.ManagedThreadId
```

// These are the two possible interleaved outputs when the above is run

// Possibility 1
Thread 1: Waiting!
Thread 5
Thread 1: Done!

// Possibility 2
Thread 5
Thread 1: Waiting!
Thread 1: Done!
Race conditions

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

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

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

Here we could end up with \((20, 10)\) in \(x\) and \(y\), via the following:
- First the 1\textsuperscript{st} thread does \(\text{currX} \leftarrow 10\)
- Then the 2\textsuperscript{nd} thread does \(\text{currX} \leftarrow 20\) and \(\text{currY} \leftarrow 20\) (i.e. it interleaves)
- Then the 1\textsuperscript{st} thread does \(\text{currY} \leftarrow 10\)

This is called a \textit{race condition} – the exact timing affects the result.

We say that the result is not \textit{consistent} with either of the updates performed.
Interference, consistency & critical sections

- **Race condition**: when two threads run pieces of code that interfere.
  - *Interference* means they read/write related variables (or the same vars)
  - The result can be *inconsistent* – the expected relationships between values may be violated.

- **Critical sections (CS)**: parts of code which may interfere with each other

- If two threads interleave critical sections the result is unpredictable, and often wrong. E.g.,

```plaintext
type MutableCube(x) = // Intended to store a number and it’s cube
    let mutable currentX = x
    let mutable currentCube = x*x*x

member p.Value = (currentX, currentCube)

member p.Update(x) =
    currentX <- x           // A critical section: threads may
    currentCube <- x*x*x    // interfere in these two lines

let c = new MutableCube(0)
startThread <| fun() -> for i in 1..1000 do c.Update 10
startThread <| fun() -> for i in 1..1000 do c.Update 20;;
c.Value

// val it = (20, 1000) is a possible result, but is inconsistent
```
Locks cause threads to wait

- Locks prevent interference by guaranteeing that a thread will execute a whole sequence of steps without any other thread running code that requires the same lock (each CS may involve many locks).
  - Equivalently: certain interleavings between threads are prevented.
  - This is also called *mutual exclusion*: threads in a CS exclude others.
- Locks make threads wait when entering a CS until no other thread is in the CS.

```ocaml
type MutableCube(x) =
  let mutable currX = x
  let mutable currCube = x*x*x
  member p.Value = (currX,currCube)
  member p.Update(x) =
    lock p <| fun() ->
      currX <- x
      currCube <- x*x*x

let c = new MutableCube(0)
startThread <| fun() -> for i in 1..1000 do c.Update 10
startThread <| fun() -> for i in 1..1000 do c.Update 20;;
c.Value

// val it : int * int = (10, 1000) or (20,8000) each time
// hence, always consistent
```
Monitors

- Locks provide a natural way of preventing interference.
- But, what do we do when one thread needs to wait for another thread to do something (e.g. modify some object)?
- E.g. suppose a thread is writing to an array buffer, and the buffer is full.
  - It must wait until another thread takes the data from the buffer.

- A standard solution is to use monitors which combine locks with:
  - a way for threads to wait until another thread signals that it has modified an object
  - a way to wake up threads waiting for changes to an object.
- Threads waiting or waking waiters must hold the lock.

- The common usage is to wait within a while loop that checks whether a particular condition is true yet.
  - The lock is re-obtained before the call to wait returns.
  - This ensures that the condition remains true after the loop ends, provided that any code affecting the condition uses the lock.
  - So, after the loop the code can depend on the condition being true.
Monitor operations example: wait & wakeWaiters

```ml
type monitorBuffer() = // This is almost directly from Lab 6
  let n = 5 // The size of the buffer array
  let b = Array.create n 0 // Used as a “circular buffer”
  let inPos, outPos = ref 0, ref 0 // Buffer write/read positions
  let count = ref 0 // no. items currently in b

member this.append(value) = // locks this and waits when b full
  lock this <| fun () ->
    while(!count>=n) do
      wait this
      b.[!inPos] <- value // guarantee: !count < n
      inPos := (!inPos+1) % n // & no other thread
      count := !count+1 // modifying this
      wakeWaiters this

member this.take() = // locks this and waits when b empty
  lock this <| fun () ->
    while(!count=0) do
      wait this
      let returnValue = b.[!outPos] // guarantee: !count > 0
      outPos := (!outPos+1) % n // & no other thread
      count := !count-1 // modifying this
      wakeWaiters this
    returnValue
```