Message passing is an important alternative to shared memory concurrency. It avoids the issues related to interference and locking by making each object the responsibility of only a single thread/task. Communication then occurs by explicitly sending messages, generally asynchronously via queues so that threads are never blocked.
## Constructs for computation expressions (including asynchronous blocks)

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
</tr>
</thead>
</table>
| `let! pat = expr` | Execute the async computation `expr` and bind its result to `pat` when it completes.  
If `expr` has type `Async<'a>`, then `pat` has type `'<a>. Equivalent to  
`async.Bind(expr,(fun pat -> ...)). |
| `let pat = expr`  | Execute an expression synchronously and bind its result to `pat` immediately.  
If `expr` has type `'<a>, then `pat` has type `'<a. Equivalent to an  
ordinary `let` that binds variables with scope until the end of the block. |
| `do! expr`  | Equivalent to `let! () = expr`. |
| `do expr` | Equivalent to `let! () = expr`. |
| `return expr` | Evaluate the expression, and return its value as the result of the  
containing asynchronous workflow. Equivalent to `async.Return(expr)`. |
| `return! expr` | Execute the expression as an asynchronous computation, and return its  
result as the overall result of the containing asynchronous workflow.  
Equivalent to `expr`. |
| `use! pat = expr` | Like `let!` but calls `Dispose` on each variable in the pattern when  
the enclosing async ends. Equivalent to  
`async.Bind(expr,(fun x -> async.Using(x,fun pat -> ...)))` |
| `use pat = expr` | Like `let` but calls `Dispose` on each variable in the pattern when  
the async ends. Equivalent to `async.Using(expr,(fun pat -> ...))`. |
Async static members (there are many more)

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Async.Parallel: seq&lt;Async&lt;'a&gt;&gt; -&gt; Async&lt;'a[]&gt;</td>
<td>Runs the async's in parallel and collects the results. Cancels all when exceptions.</td>
</tr>
<tr>
<td>Async.RunSynchronously: Async&lt;'a&gt; -&gt; 'a</td>
<td>Runs via the thread pool and waits for the result.</td>
</tr>
<tr>
<td>Async.Start: Async&lt;unit&gt; -&gt; unit</td>
<td>Queues without waiting.</td>
</tr>
<tr>
<td>Async.SpawnChild: Async&lt;unit&gt; -&gt; Async&lt;unit&gt;</td>
<td>As above, but inherits the cancellation handle from the current async.</td>
</tr>
<tr>
<td>StartWithContinuations : Async&lt;'T&gt; * ('T -&gt; unit) * (exn -&gt; unit) * (OperationCanceledException -&gt; unit) * ?CancellationToken -&gt; unit</td>
<td>As above, but then its result is available, executes the first callback, or one of the other two callbacks for exceptions or cancellations.</td>
</tr>
<tr>
<td>Async.Catch: Async&lt;'a&gt; -&gt; Async&lt;Choice&lt;'a, exn&gt;&gt;</td>
<td>Catches errors from an asynchronous computation and returns a Choice result indicating success or failure.</td>
</tr>
</tbody>
</table>

Catch makes use of: type Choice<'a,'b> = Choice2_1 of 'a | Choice2_2 of 'b

Message passing

- Shared memory is often a natural way to program, but it has some issues.
- Avoiding deadlocks can be difficult, particularly when putting together components that each use their own locks.
- As a result, shared memory does not scale well to large programs.
- Also, shared memory has performance costs:
  - The time to do the lock and unlock operations.
  - Many cores accessing the same address invalidates CPU caching - sharing memory leads to slower access times.
- When multiple computers are involved, shared memory is generally not an appropriate model.
- Message passing addresses these issues by making communication explicit.
- Asynchronous messages are generally preferred so that senders do not have to wait for receivers.
- Generally messages are queued in a mailbox – this means that receivers also do not have to wait unless the mailbox is empty.
Asynchronous messages – small example

```fsharp
open Microsoft.FSharp.Control.Mailboxes

let counter =
    MailboxProcessor.Create(fun inbox -> // returns an async
        let rec loop(n) =
            async {
                do printfn "n = %d, waiting..." n
                let! msg = inbox.Receive()
                return! loop(n+msg) }

        loop(0))

val counter : MailboxProcessor<int>

> counter.Start();
n = 0, waiting...
> counter.Post(1);;
n = 1, waiting...
> counter.Post(2);;
n = 3, waiting...
> counter.Post(1);;
n = 4, waiting...
```
Defining message processors/agents

- Message-processing components (also called agents) typically use sets of recursive functions, each defining an async.

- Each function corresponds to a “state” that determines how the agent will react to the next message.
  - For example, an agent may have a special function/state for when it is waiting for a reply to a message it has sent.
  - If an agent always operates the same way, one state is enough.

- Arguments can be passed between these functions to represent further information about the current state.

```csharp
// This is the general pattern for an agent
let agent =
    MailboxProcessor.Start(fun inbox ->
        // The states of the state machine
        let rec state1(args) = async { ... }  // Each state’s async
        and  state2(args) = async { ... }    // ends with a call to
        ...                                     // the next state.
        and  stateN(args) = async { ... }
        // Enter the initial state
        state1(initialArgs))
```
Mailbox members

- **Post: 'msg -> unit**
  
  Posts a message to a mailbox queue.

- **Receive: ?timeout:int -> Async<'msg>**
  
  Returns the next message, waiting asynchronously. If a timeout occurs, then raises a TimeoutException.

- **Scan: ('msg -> Async<'a option) * ?timeout:int -> Async<'a>**
  
  Scans the mailbox for a message accepted by the function. Returns the chosen result.

- **TryReceive: ?timeout:int -> Async<'msg option>**
  
  If a timeout occurs returns None. (Also TryScan, similarly.)

- **PostAndReply: (AsyncReplyChannel<'reply> -> 'msg) -> 'reply**
  
  Post a message and await a reply on the channel. The function must build a message containing the reply channel, and the (unique) reply on that channel will be returned.
Example: Labs from the project

- To compare with shared memory and locks, here's how the labs from the project would look as message passing agents – with 2 states “idle” and “busy”

```plaintext
type Agent<'T> = MailboxProcessor<'T>

type Cont<'a> = 'a -> unit

type delay = int

type LabMsg = Rules of Cont<labRules> // Alas, using Agent<labRules> is awkward.

| DoExp of delay * exp * clientID * Cont<bool> // Ditto
| ExpDone

let postLater (box: Agent<'a>) delay (msg:'a) = Async.Start (async { do! Async.Sleep delay; box.Post msg })

let lab (labID, rules) =
    MailboxProcessor.Start (fun inbox ->
        let rec idle() =
            async { let! msg = inbox.Receive()
                match msg with
                | Rules rulesCont -> rulesCont rules; return! idle()
                | DoExp (delay, exp, clID, cont) -> postLater inbox delay ExpDone; return! busy clID cont
                | ExpDone -> failwith "ExpDone received without an experiment in progress" }

        and busy usingClID cont =
            async { let! msg = inbox.Receive()
                match msg with
                | Rules rulesCont -> rulesCont rules; return! busy usingClID cont
                | DoExp (delay, exp, clID, expCont) ->
                    let str = sprintf "BANG! lab%d explodes - host%d is already using it" labID usingClID prStamp -1 str "" ; return! busy usingClID cont
                | ExpDone -> if random 2 = 1 then cont true else cont false; return! idle() }

        idle()
```
Example: Bank accounts

- To compare with the concurrent code we saw earlier (in Lab 6), we now consider code for bank accounts using messages.

- This example makes for quite a good comparison between message passing and shared memory with monitors/locks.
  - However, the message passing version doesn’t provide exactly the same guarantees – we’ll consider this after seeing the code.

- We have a union type with the different kinds of messages.

- We have an agent that does the message processing.

- We wrap the agent in a class here, to show how message passing can be combined with standard objects.
  - In particular, this demonstrates how to interface with ordinary synchronous code,
Example: Bank accounts with messages

type acctMsg = BalanceMsg of AsyncReplyChannel<int>
   | WithdrawMsg of int
   | DepositMsg of int
   | TransferMsg of account * int

and account(name:string) =
  let agent = MailboxProcessor.Start(fun inbox ->
    let balance = ref 1000
    let withdraw amt = balance := !balance - amt
    let rec loop() =
      async {
        let! msg = inbox.Receive()
        match msg with
        | BalanceMsg replyCh-> replyCh.Reply(!balance)
        | WithdrawMsg amt -> withdraw amt
        | DepositMsg amt -> balance := !balance + amt
        | TransferMsg (toAcct, amt) -> withdraw amt
          toAcct.Deposit amt
        return! loop()
      }
    loop() )

member this.Balance =
  agent.PostAndReply (fun replyChan -> BalanceMsg replyChan)
member this.Name = name
member this.Withdraw amount = agent.Post (WithdrawMsg amount)
member this.Deposit amount = agent.Post (DepositMsg amount)
member thisTRANSFER (toAcc:account) amount =
  agent.Post (TransferMsg (toAcc, amount))
Example: Bank accounts – main program

```ml
let doTransfers (acc:account) toAcc () =
    for i in 1..100 do acc.Transfer toAcc 100
    printfn "%s balance: %d Other balance: %d"
        acc.Name acc.Balance toAcc.Balance
// The handout included a sleep here, but in fact it isn’t needed
    printfn "%s balance: %d Other balance: %d"
        acc.Name acc.Balance toAcc.Balance

let main() =
    let acc1=account("Account1")
    let acc2=account("Account2")
    startThread (doTransfers acc1 acc2)
    startThread (doTransfers acc2 acc1)

main()

// The output is as follows
Account2 balance: -9000 Other balance: 1000
Account1 balance: -9000 Other balance: 1000
Account2 balance: 1000 Other balance: 1000
Account1 balance: 1000 Other balance: 1000
```
Notes on bank account example

- The balance and name methods are synchronous (they wait for the result) while the others are asynchronous.
  - For the balance, this requires a reply channel
  - The channel is sent in the message, and then PostAndReply waits.

- Inside the agent, we have strong guarantees about non-interference.
  - Only one thread can be running within the agent at a time.

- For a transfer, our guarantees are not quite the same as the version in Lab 6 which locks both accounts.
  - The total money in the accounts appears to change.
  - This is common in message passing: consistency properties involving multiple agents need to be reconsidered.
  - In this case, we need to include the money in messages waiting to be delivered – if we do, this will never change.
  - It is tricky to calculate properties involving many agents, like the total sum of money, however there are algorithms to do this by sending special “marker” messages to form a global snapshot which is consistent with the actual state.
  - This reflects the asynchronous nature of messages.