Asynchronous Systems

Jim Fawcett

CSE681 – Software Modeling & Analysis

Fall 2015

References

- Java Concurrency in Practice, Brian Goetz, et al., Addison-Wesley, 2006
  - This is the best book I've seen on concurrency. Much of the material is language agnostic.
- Concurrent Programming in Java, Second Edition, Doug Lea, Addison-Wesley 2000
  - Several of the slides in this presentation closely follow the material in this book. In some cases the slides simply paraphrase material presented there.
- C# 5.0 in a Nutshell, Albahari & Albahari, O'Reilly, June 2012
  - See Chapters 14, 22, and 23.
- Evolution of Synchronization in Windows and C++, Kenny Kerr, MSDN Magazine, Nov 2012
- Evolution of Threads and I/O in Windows, Kenny Kerr, MSDN Magazine, Jan 2013

Agenda

- What is an asynchronous system?
- When should you use asynchronous methods?
- Operating system support – Demo code
  - Asynchronous methods
  - Asynchronous delegates
  - Asynchronous callbacks
- Design Forces

Synchronous - Definitions

- Orbital Satellites
  - Stationary relative to the earth
- Neurobiology
  - Mental processes that entrain to external stimuli as in epilepsy
- Communication Systems
  - Information is contained in frames with constant frame rate
- Radio and Radar Detection
  - Carrier is removed by an oscillator that locks onto the incoming carrier frequency
- Software
  - A function call that blocks the caller until finished
  - A component that collects input by waiting for data from a single sender at some point in the code, e.g., cin

Asynchronous Software

- Function call returns immediately without waiting for function completion.
- One of the following happens after return:
  - Caller needs no reply and ignores the callee.
  - Caller must poll for completion status, e.g., keep checking
  - Caller must provide a callback for the callee to use when finished
  - Caller deposits a message in a queue for the callee to process at some later time, without expecting or waiting for a reply.
    - The callee may, but is not required to, deposit a reply in a queue owned by the caller.
  - If a system is based on message passing, the callee can react to inputs from an arbitrary number of sources, arriving in any order, at any time.
    - Very flexible.

Synchronize (synonym – serialize)

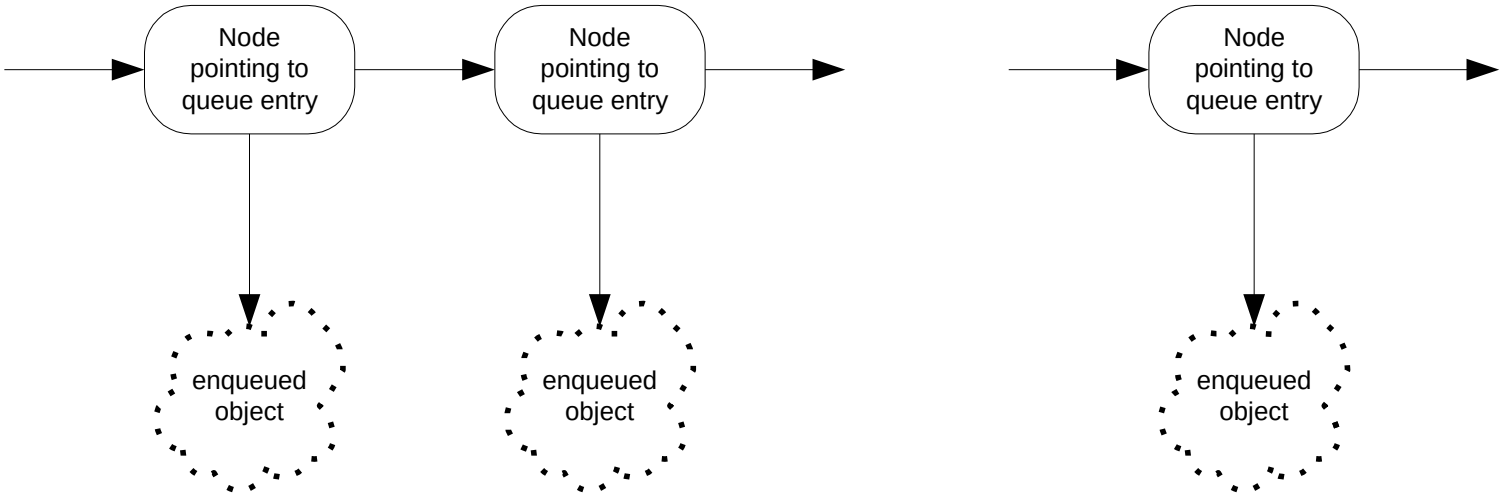
- Just to be confusing, the word synchronize, pronounced and spelled very like synchronous, means something entirely different.
- For software, synchronize means to control access to a resource shared between threads so that only one thread at a time is allowed to use the resource.
  - The resource is locked by a thread and no other thread gets access until the using thread releases the lock.
  - A lock may apply only to some specific code location or to an object accessed anywhere in the code.
  - A thread can lock a resource by using any of the following:
    - C# lock, .Net Monitor, Win32 critical section, mutex
  - The following resources are often shared between threads:
    - Queues, I/O streams, files, and windows
    - Static members of a class
    - C and C++ can share global variables

What is an Asynchronous System?

- Two parties communicate without being bound to a specific time.
  - Email is a classic example.
    - Message is sent, can be read at any later time.
    - No constraints on when message is sent.
    - No constraints on when message is read, as long as it is later than sending time.
  - Requires four things:
    - Sender
    - Receiver
    - Place to put messages
    - Transmission facility that does not require any action on part of sender or receiver, other than to send and collect message.

FIFO Queues

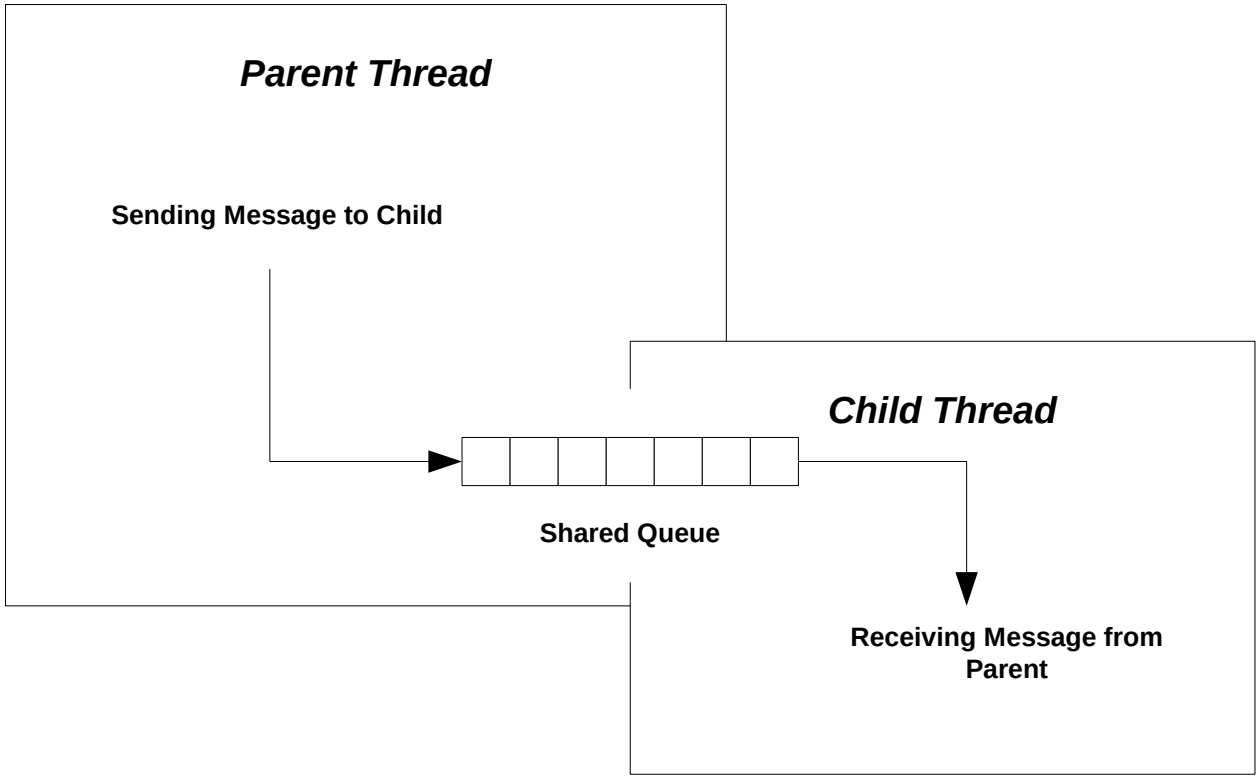
- First In First Out queues are often constructed with linked lists.
  - Objects enter the queue by getting linked to one end.
  - Objects leave the queue by getting unlinked from the other end.



Important Property of Queues

- Queues decouple a receiver from its sender.
  - Sender and receiver can be on different threads of a given process. That requires a thread-safe queue.
  - Receiver does not need to process an object when it is handed off by the sender.
  - Queues can eliminate timing mismatches between sender and receiver.
    - One might be synchronized in nature, requiring message passing at fixed time intervals – a radar signal processor for example.
    - The other might be free-running, handling messages with different service times, preventing synchronized operation.
  - Queues can support reliable communication over unreliable media, simply holding onto messages it can't send until the transmission link becomes available.

Message Passing Between Threads



Send and Receive Queues

- Essentially, a SendQ lets:
  - User thread create requests and post for sending.
  - Send thread dequeues messages and pushes into communication channel. It remembers requests it has not processed yet.
- A RecvQ lets:
  - Remote receive thread posts message without waiting for receiver processing thread to be ready to accept it.
  - Valuable remote resource need not block waiting for a hand-off to receiver.
- Send queue allows the client's main thread to service other important tasks as well as interact with the remote resource.
- Receive queue allows the server's main thread to process requests serially without blocking the sender.

What is an Asynchronous System?

- So, is every message-passing system asynchronous? No.
  - Exchange between a browser and web server is message-based, usually employing Get or Post HTTP messages.
  - That exchange is synchronous. The browser doesn't return until a page is delivered – either the page requested or an error page.
- Is every asynchronous system message-based? No.
  - .Net delegates and remoting proxies support asynchronous operations via BeginInvoke and EndInvoke procedure calls.
- Examples of asynchronous systems:
  - Windows operating system supports the ability to react to many kinds of events using each window's message loop.
  - Socket Listeners improve their availability by spawning client handler threads.
  - All the radar systems I worked on use asynchronous messaging between layers.
  - Email, your Project #4, many enterprise systems, …

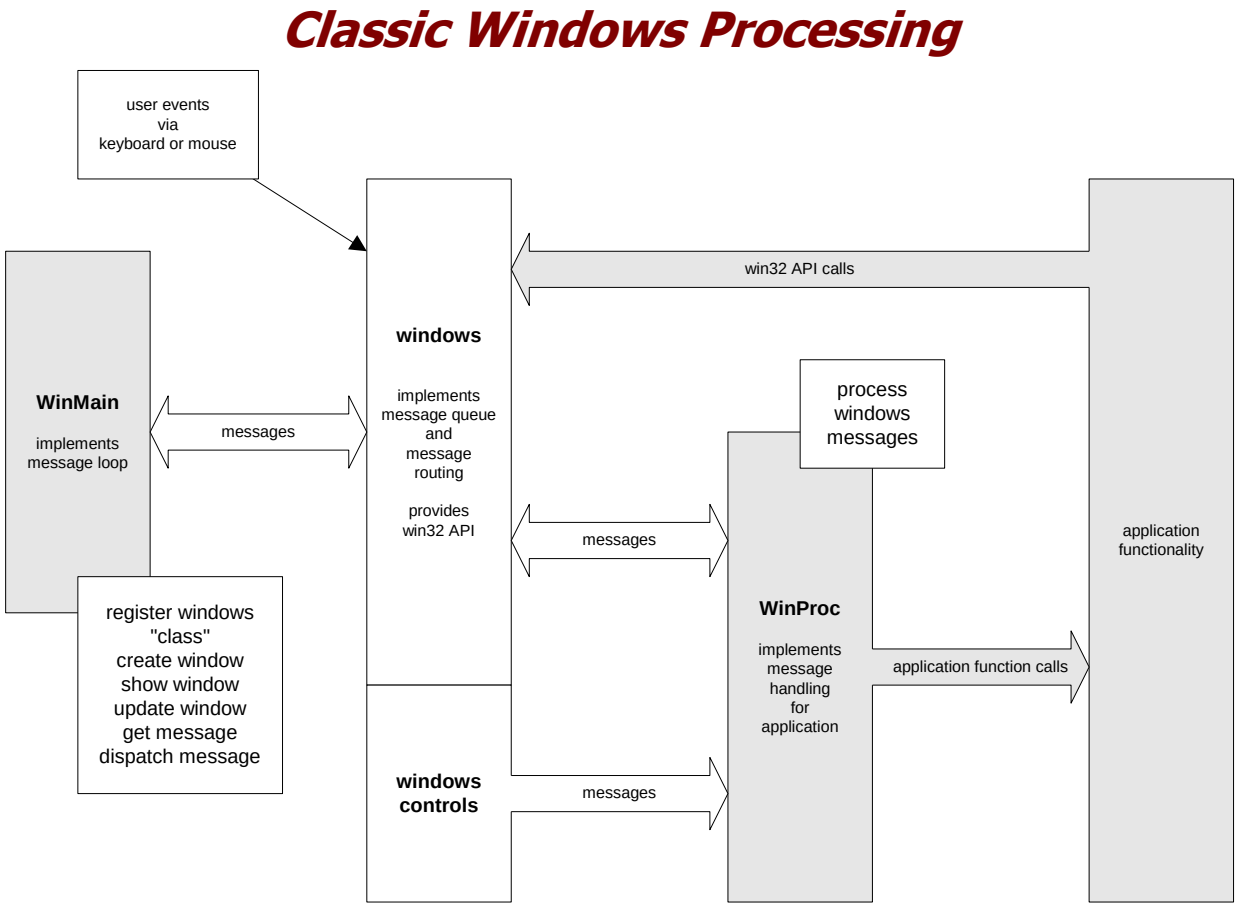
Why Use Asynchronous Systems?

Adapted from Concurrent Programming in Java, Doug Lea, Addison-Wesley, 1997

- System needs to be reactive
- System must have high availability
- Services must be controllable
- System needs to send asynchronous messages
- System may have to handle bursty events

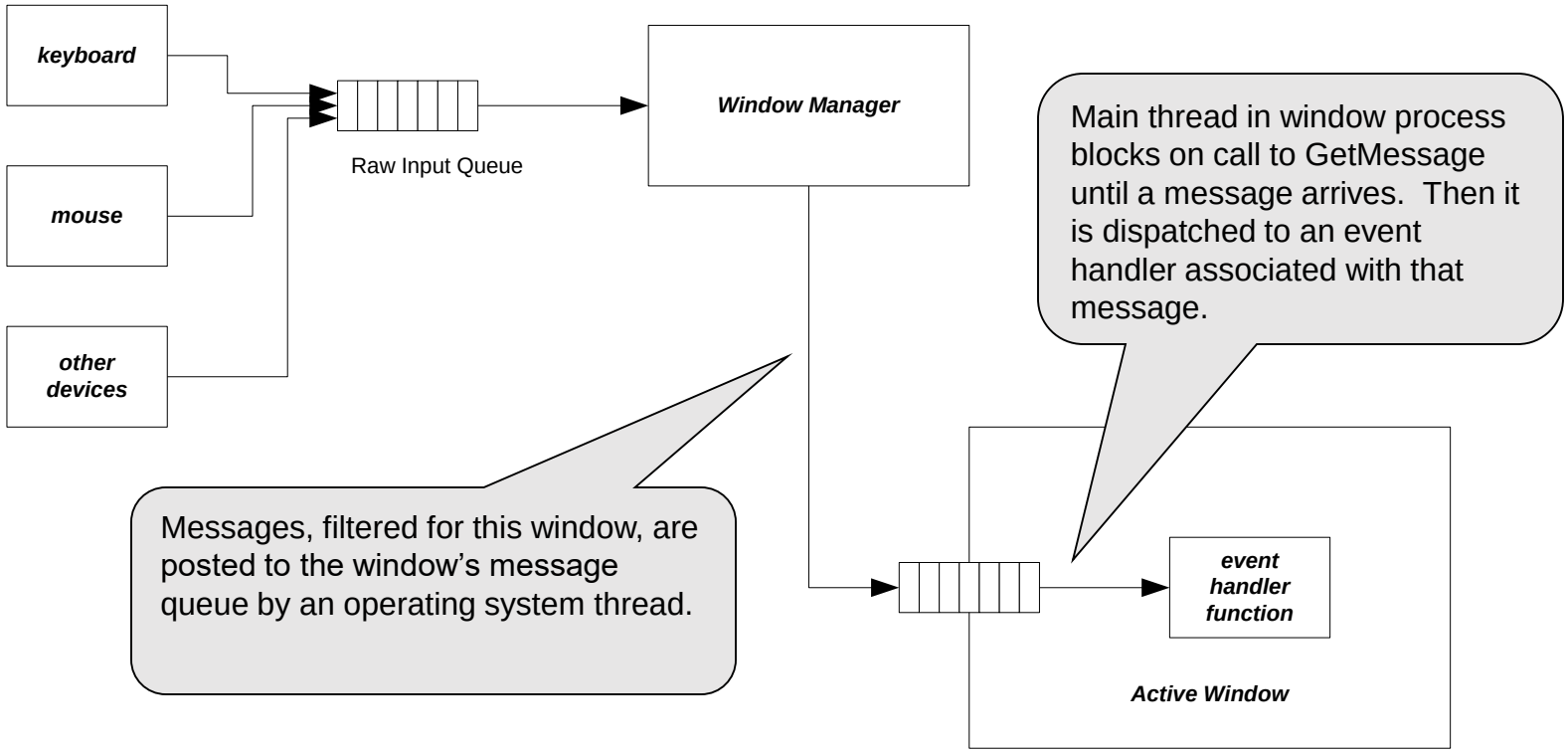
Why use Asynchronous Systems?

- The system needs to be **reactive**.
  - It does more than one thing at a time, each activity reacting in response to some input.
  - Each event results in a message in windows queue. Reponse happens later, if at all.



Windows, Queues, and Messages

- Graphical User Interfaces are the stereotype of message-passing systems using queues.

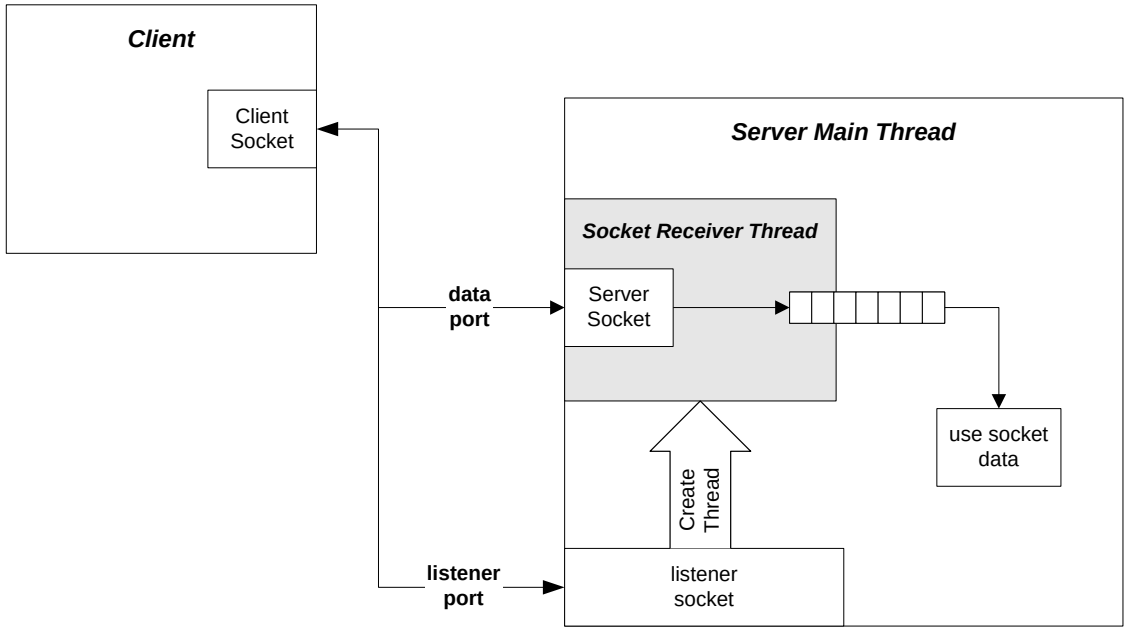


Windows Messaging

- With the architecture shown on the previous slide a window can respond to any of many different inputs, including:
  - User inputs from mouse movement, mouse buttons, and keyboard
  - System events, e.g., window being uncovered by an obscuring window, and so needing to repaint its region of the screen
  - Message generated from within the program running in that process, based on strategies put in place by the designer.
- Even if several messages arrive before a predecessor is processed, they won't be lost.

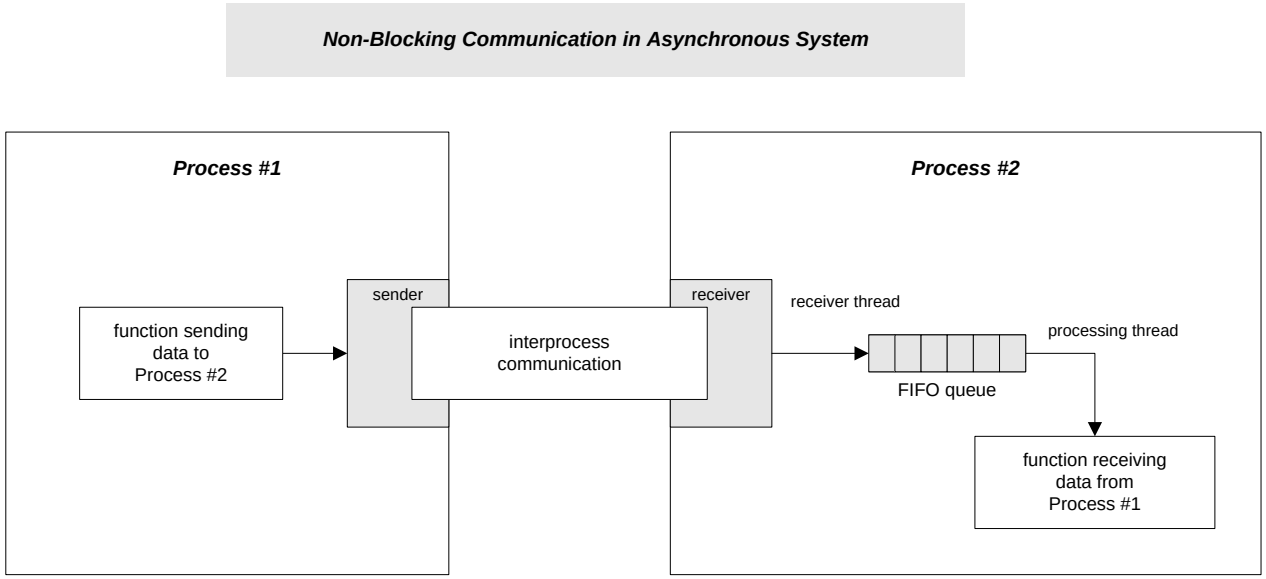
Why use Asynchronous Systems?

- The system must have **high availability**.
  - One object may serve as a gateway interface to a service, handling each request by constructing a thread to asynchronously provide the service.
  - Socket Listener must quickly dispose of a connection so that it can go back to listening for other requests to connect and clients find the server available.



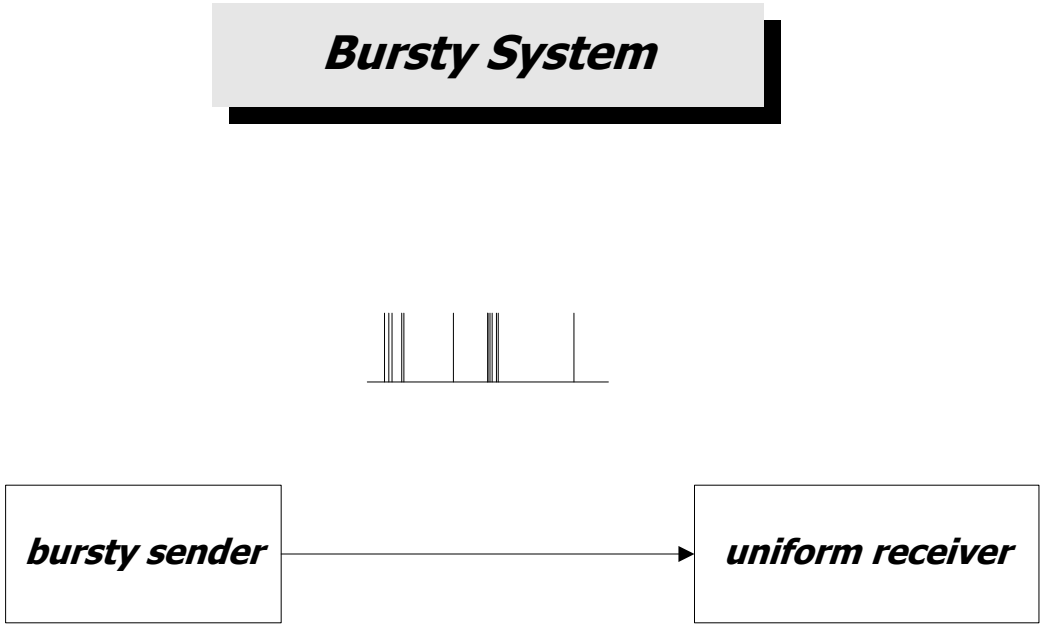
Why use Asynchronous Systems?

- The services need to be **controllable**.
  - Activities within threads can be suspended, resumed, and stopped by other threads. Controller issues command but does not wait for action.
- The system needs to **send asynchronous messages**.
  - The calling object may not care when a requested action is performed, at least within some reasonable bounds.



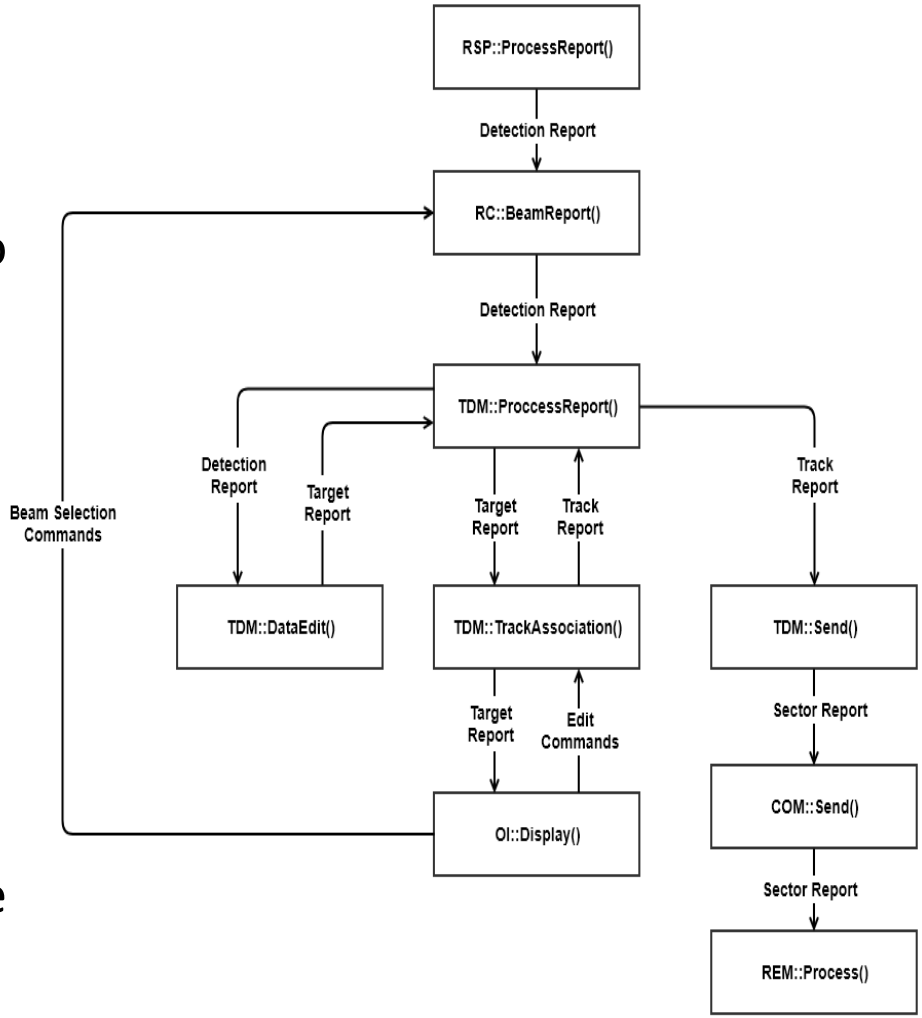
Why use Asynchronous Systems?

- The system may need to **handle bursty events**.
  - Some events may occur in rapid succession for brief periods, perhaps far faster than the system can react to them. When this happens an asynchronous system can enqueue requests to work on at a later time.



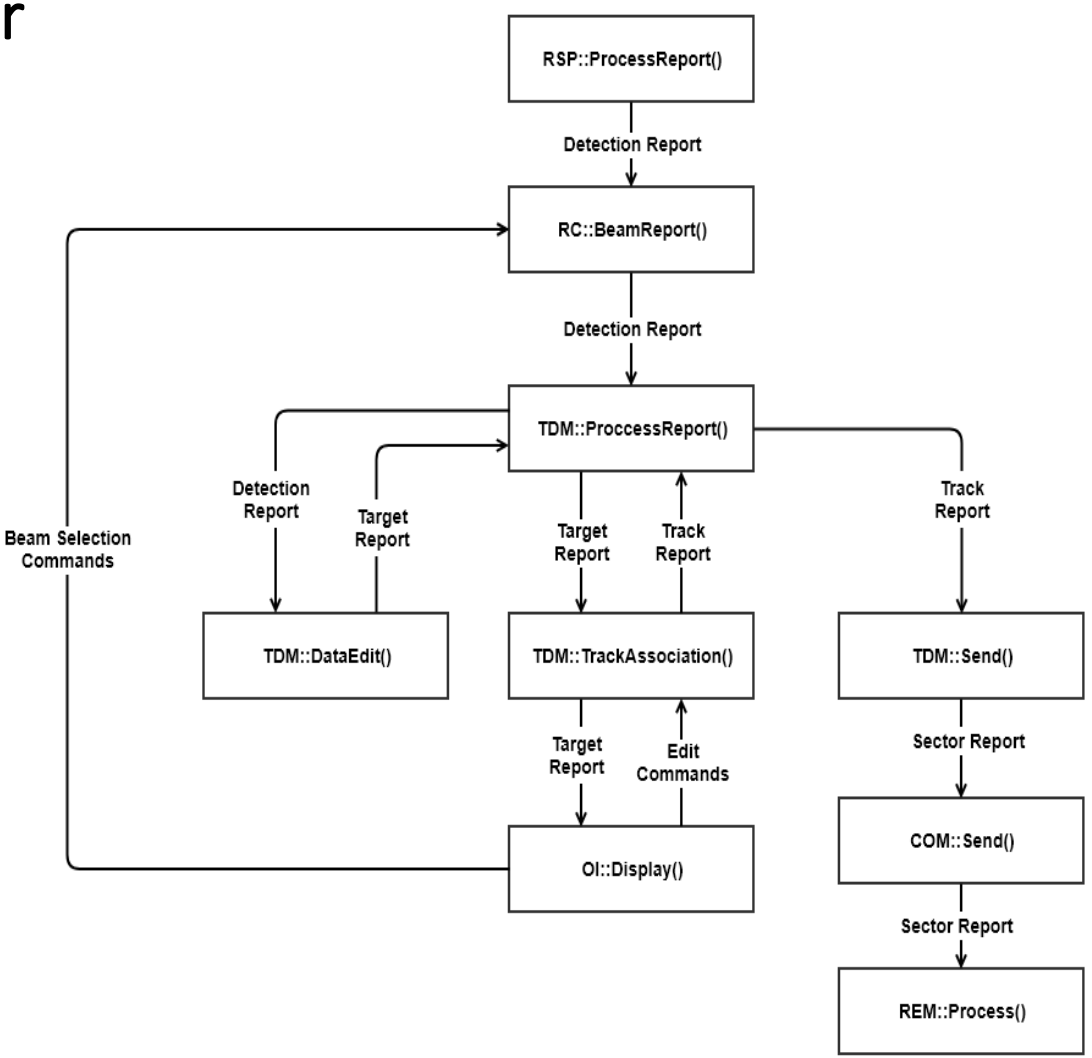
Synchronous Radar Just won't work!

- A sychronous radar would require the signal processor to wait, when it sends a report, for the report to be processed at several levels.
- This just will not work. The signal processor must operate at a high rate of speed to service all of the cells in coverage.
- Each radar component must function independently on the inputs provided to it.



Asynchronous Radar

- In the asynchronous radar each component computes its output and deposits it in a queue for processing by the next layer.
- The messages at the top of the chain are small and arrive at a high rate.
- Message lower in the chain are larger and less frequent.



Asynchronous Operation

- There are several ways to support Asychronous Operation. All of these use asynchronous methods:
  - Create a new thread
    - Pass it the method to run asynchronously, using a ThreadStart delegate.
  - Use a C# Task
    - Bind a lambda to the task, execute, and eventually wait for a result.
  - Use a thread from a ThreadPool
    - Queue a work item with a delegate pointing to the method to run asynchronously.
  - Use a BackgroundWorker in a Form
    - Has useful events for signaling progress and completion.
  - Use a delegate's BeginInvoke(), EndInvoke() methods
    - Runs the delegate's methods asynchronously.
  - Use a Form's BeginInvoke(), EndInvoke() methods.
    - Allows a worker thread to call a Form method running on the Form's UI thread.
  - Use a class that implements ISynchronizeInvoke interface
    - The ISynchronize interface declares methods: InvokeRequired(), BeginInvoke(), EndInvoke(), Invoke()

Create a Thread

- ```
  class QueuedMessageDemo {
      // declare and initialize member data used as
      // parameters for ThreadProc
      public void ThreadProc() { … }
  }
  ```
- Use System.Threading.ThreadStart delegate:

```
Thread child = new Thread(
          new ThreadStart(demo.ThreadProc)
     );
child.Start();
```

- Here, demo is an instance of QueuedMessageDemo. The ThreadProc function runs asynchronously and the call to Start returns immediately.
```

Use Task

- Tasks accept a delegate or lambda to define operation
- Tasks can return a value or handle to an instance

```
Task<int> t = new Task<int>(
 () => dt.methodWithResult(taskName, sleepTime)
);
t.Start();

// do some useful work here

int id = t.Result;  // blocking call
```

ThreadPool

- ```
  class TpDemo {
      // declare and initialize member data used as
      // parameters for ThreadProc
      public void ThreadProc() { … }
  }
  ```
- ```
  WaitCallback callforward
      = new WaitCallback(dem.threadProc);
  ```
- ```
  ThreadPool.QueueUserWorkItem(callforward,null);
  ```
- The instance dem is of type TpDemo. WaitCallback is a delegate defined in System.Threading. The call to QueueUserWorkItem is asynchronous, returning immediately.
  ```

BackgroundWorker

- class FormWithTask : Form
  {
  public delegate void TextInserter(string text);
  // …

- Start background ThreadPool thread:

```
backgroundWorker1.RunWorkerAsync();
```

- Define background task:

```
private void backgroundWorker1_doWork(
   object sender, DoWorkEventArgs e
)
{
 this.Invoke(textInserter, new object[] { “…” })
 // …
}
```

Asynchronous Delegate

- ```
  class Asynch
  {
      public delegate string SlowCallDelegateType(int secs);
      public SlowCallDelegateType callDelegate;
      public string SlowCall(int millisecs) { … }
     // more class functionality
  }
  ```

- ```
  asOp.callDelegate =
     new Asynch.SlowCallDelegateType(asOp.SlowCall);
  ```

- ```
  // begin asynchronous operation
  IAsyncResult ar
     = asOp.callDelegate.BeginInvoke(1000,null,null);

       // do some useful work here, then wait
       // on asynchronous call to finish

     string result2 = asOp.callDelegate.EndInvoke(ar);
  ```

- Here, asOp is an instance of Asynch.
  ```

Delegate's BeginInvoke Arguments

- The arguments of BeginInvoke depend on the signature defined by the delegate.
  - The first arguments are the signature arguments ordered as:
    - in parameters
    - out parameters
    - in/out parameters
    - ref parameters
  - These are followed by the final two arguments of type:
    - AsyncCallback
    - AsyncState
- The return value of BeginInvoke is an instance of IAsyncResult.

Delegate's EndInvoke Arguments

- The arguments of EndInvoke consist of the non-in parameters and the IAsynchResult instance returned by BeginInvoke:
  - The first arguments are the signature arguments ordered as:
    - in parameters ***omitted***
    - out parameters
    - in/out parameters
    - ref parameters
  - These are followed by the final argument of type:
    - IAsynchResult instance returned by BeginInvoke.
- The return value of BeginInvoke is the return value of the delegate's function.

Form BeginInvoke

```
if(ar != null)
    _form.EndInvoke(ar);

teva.msg = file;

if(_form.InvokeRequired)
{
  ar =
    _form.BeginInvoke(OnFileEvent,new object[] { this, teva });
}
else
    OnFileEvent(this,teva);
```

- `_form.BeginInvoke(…)` is an asynchronous call, e.g., returns immediately. teva is an instance of a class derived from EventArgs.
- Note that every call to BeginInvoke should be matched by an EndInvoke, otherwise you will have a memory leak – each BeginInvoke allocates a structure in the kernel which is deleted by EndInvoke.

Form's BeginInvoke Parameters

- The arguments of BeginInvoke are:
  - A System defined delegate accepting functions of the form:
    - `void OnMyEvent(object sender, EventArgs ev);`
  - An array of arguments matching the delegate:
    - `object[] args = new object[] { this, teva };`

      Here `teva` is an instance of a class derived from `System.EventArgs`.
  - The return value of `BeginInvoke` is an instance of `IAsyncResult.`
- The argument of EndInvoke is:
  - The instance of the `IAsyncResult` returned by `BeginInvoke(…).`

Design Forces

- The primary design forces are safety and liveness:
  - Safety
    - nothing bad ever happens to an object or resource
  - Liveness
    - Something eventually happens within an activity
  - Performance
    - How soon and quickly are services provided?
  - Reusability
    - How easy is it to use services in another programming context?

Safety

- Read/Write conflicts
  - One thread attempts to read a field while another writes to it.
- Write/Write conflicts
  - Two threads attempt to write to the same field.

Liveness

- In a live system we expect to make progress toward completion. This may not happen immediately for the following reasons:
  - Locking
    - A lock or synchronized method blocks one thread because another thread holds the lock.
  - Waiting
    - A thread blocks, on a join() or wait() method, waiting for an event, message, or result computed by another thread.
  - Input
    - An IO method waits for input that has not arrived yet.
  - Contention
    - A runnable thread fails to run because other threads are occupying the CPU or other needed resources.
  - Failure
    - A method encounters an exception or other fault.

Liveness (continued)

- A permanent lack of progress may result from:
  - Deadlock
    - Circular dependencies between locks
  - Missed event
    - A thread starts waiting for an event after the event occurred
  - Nested lockout
    - A blocked thread holds a lock needed by another thread attempting to wake it up.
  - Livelock
    - A continuously retried action fails continuously.
  - Starvation
    - Scheduler fails ever to allocate CPU time to a waiting thread.
  - Resource exhaustion
    - A thread attempts to access one of a finite set of resources, all of which are in use (file handles for example).
  - Distributed system failure
    - A remote machine, hosting a needed service, is unavailable.

Performance

- Permformance is usually described by the metrics:
  - Throughput
    - Number of operations per unit time, e.g., messages processed, files sent, …
  - Latency
    - Elapsed time between a request for service and its satisfaction.
  - Capacity
    - The number of simultaneous activities that can be supported for a given throughput or maximum latency.
  - Availability
    - Number of simultaneous requesters that can be supported without failures to connect.
  - Efficiency
    - Throughput divided by the amount of CPU resources needed, e.g., CPUs, memory, IO devices, …
  - Scalability
    - Rate at which throughput or latency improves when resources are added to the system.
  - Degradation
    - Rate at which latency or throughput decreases as more clients or activities are added without adding new resources

Performance (continued)

- Factors that affect permformance
  - Partitioning of distributed resources
    - Affects the amount and sizes of data sent between processes, machines, and networks.
  - Caching
    - Holds data that has been sent previously for possible use later. Attempts to avoid retrieving resources already held. May induce problems with consistency and staleness of the data.
  - Locking strategy
    - May trade off the use of immutable objects against use of mutable locked objects, e.g., create new immutable objects to achieve a state change, rather than locking and modifying a mutable object's state.
  - Use of background tasks
    - Define activities that can usefully proceed while main activities are blocked.
  - Use algorithms specifically designed for concurrency
    - Some efficient sequential algorithms do not lend themselves to efficent concurrent implementation, but there may be good concurrent versions available.

Threading Memory Model

- Both the compiler and the processor that your code runs on are allowed to:
  - Cache variables in registers or other cache memory
  - Rearrange instructions
  - For single threads, the compiler and processor are constrained to perserve sequential semantics, e.g., program order semantics.
  - For interactions between threads there are no such guarantees.
- The ***lock*** construct and ***volatile*** qualifier are intended to extend these guarantees to threads operating in a multi-threaded environment.
  - Any thread running within a lock body or in a synchronized function will enjoy the program order semantics guarantee.
    - Releasing a lock forces a flush of all writes from the thread's working memory.
    - Acquiring a lock forces a reload of all fields accessible to the acquiring thread.
  - If a variable is qualified as volatile:
    - Any written value is flushed by the writer thread before the writer performs any other memory operation.
    - Reader threads must reload the values of volatile variables for each access.

End of Presentation