Principles of Concurrent Systems: Inter-process Communication

Messages and Message Queues

- The final method of effecting process communication using Win32 is based around the concept of a Message and a Message queue.
- A message is simply an integer (32 bit) value which is sent from one thread/process to another thread/process in the system.
- The message itself is usually a command, or order sent by one process to another asking it to do something. For example
    - The message 1000 could mean turn on the light bulb in the living room in an embedded home control system
    - In an operating system, the message 3000 could mean 'terminate process'
- What the message means and how it will be interpreted is entirely up to the sender and receiver to agree upon. There are no rules that say message 'X' has to mean do action 'Y'.
- Messages sent to a process/thread are sent to its message queue and sit there like a first in first out buffer, waiting to be read by the receiver, but there is no guarantee that just because a message has been sent to a process/thread, that the process/thread will read it. The choice of when to read a message (if at all) is down to the recipient, it cannot be forced to read the message.
- However, unlike a pipeline, a message queue can be searched. In effect a process/thread can say "have any messages with the value 'X' arrived", or have any message with values in the range 'X' to 'Y' been received.
- If so, those messages can be removed from the message queue ahead of their position and acted on earlier, thus messages can be assigned an informal priority.

Principles of Concurrent Systems: Inter-process Communication

Software Primitives to Support Messaging

- A number of software primitives exist to enable a process to
  - Create a message Queue
  - Send a message.
  - Read (and remove) a message from the queue if it is within a certain range of values
  - Test for the presence of a message within a certain range of values

- **CMessageQueue()** a class that encapsulates the functionality of a Win32 Message Queue. Each process/thread wishing to receive messages must create one of these objects to hold the messages sent to them. The queue is created by the constructor for the class.

- **Signal(int Message):** Posts the specified message to a thread message queue. The object that you are sending the message to is either a thread or a process (meaning the primary thread of that process). This object is obtained by creating a CProcess or CThread object and using that object with the Signal() function.

- **WaitForMessage()**: Forces the receiving thread to suspend itself until there is at least one message that can be read from the message queue. The function will thus return when there is a message that can be read

- **TestForMessage(Min=0, Max=0)**: Tests for the presence of messages in the message queue that lie between the Min and Max values. Returns true if at least one message in that range exists. If no Min or Max value are specified then test for the presence of any message in the message queue.

- **GetMessage(Min=0, Max=0)**: Retrieves the first message from the queue that lies within the range of values specified by Min and Max Value. If no Min or Max values are given, then the first message in the queue is retrieved regardless of its value. If no messages exist in the queue, then the thread will be suspended until one arrives.

Principles of Concurrent Systems: Inter-process Communication

- An example of a thread that creates a message queue and then reads and acts upon any messages it finds is shown below. This particular example uses a 'polling' approach within the child threads to *test* for the presence of message before reading it.

```
#include "rt.h"

UINT _ _stdcall ThreadFn1(void   *args)              // A child thread either a function or a class member function
{
      UINT    Message ;                             // A variable to hold a message read for message queue

      CMessageQueue    mq1 ;                        // Create a message queue object for this particular child thread
                                                    // Note each thread can create its own unique message queue
      do {
            printf("Child Thread 1: Polling Message Queue\n") ;
            Sleep(500) ;                                        // sleep for 1/2 second

            if(mq1.TestForMessage() == TRUE) {                  // see if any message available
                        Message = mq1.GetMessage() ;            // if so get the message
                        if(Message == 1000)                     // decide what message means
                                    printf("Child Thread 1 received Message 1000.....\n") ;

                        else if(Message == 1001)  {             // if message = 1001
                                    printf("Child Thread 1 received END Message.....\n") ;
                                    break ;                     // break from loop
                        }
            }
      }while(1) ;
      return 0 ;                                    // terminate child thread and message queue
}
```

Principles of Concurrent Systems: Inter-process Communication

```
int     main()
{
        UINT Message ;
        CMessageQueue           mainq ;                                       // create a message queue for main thread

        CThread   Child1(ThreadFn1, ACTIVE) ;                                 // Create a child thread in Active state

        do {
                printf("Main Thread: Polling Message Queue\n") ;
                Sleep(500) ;                                                  // sleep for 1/2 second

                if(mainq.TestForMessage() == TRUE) {                          // is there a message in main message queue
                        Message = mainq.GetMessage() ;                        // if so, get the message
                        if(Message == 1000)                                   // if message intended for thread1
                                Child1.Signal(Message) ;                      // send the message to it
                        if(Message == 3000)             {                     // if message from another process
                                printf("Main Thread GOT Message 3000, Killing Child Threads.....\n") ;
                                Child1.Signal(1001) ;                         // kill child by sending agreed END message
                                break ;                                       // end loop
                        }
                }
                // if no message, see if keyboard character pressed
                if(TEST_FOR_KEYBOARD() != 0)            {
                        if(getch() == 'x')              {                     // if keyboard character is 'x'
                                printf("Main Thread GOT X Key, Killing Child Threads.....\n") ;
                                Child1.Signal(1001) ;                         // kill child by sending agreed END message
                                break ;                                       // end loop
                        }
                }
        }while(1) ;

        Child1.WaitForThread() ;
        return 0 ;
}
```

Principles of Concurrent Systems: Inter-process Communication

- The program below shows how a parent process could send messages to a child process. These messages will therefore be sent to the main or primary thread within the child process, i.e. the message queue created in main() in the child process

```
int   main()
{

    CProcess    Child1(……) ;                    // Create a child process

    Child1.Signal(1000) ;
    SLEEP(2000) ;
    Child1.Signal(3000) ;

    Child1.WaitForThread() ;
    return 0 ;
}
```

Principles of Concurrent Systems: Timers

Timers

- Although not an inter-process communication mechanism, the concept of timers is introduced here as it is based on the concept of a message queue and thus it seems an appropriate point to discuss them.
- Timers are a particularly useful tool in real-time systems, because they can be used to trigger events and activities that need to take place at pre-defined points in time, say every second or perhaps every 5 mS
- Timers can also be used to record the passage of time between two events. Timers are characterised under Win32 by the following properties.

Characteristics of Win32 Timers

- A timer can be programmed to operate with a resolution of 1mS.
- When a timer expires it posts a pre-defined message to the destination thread message queue, thus each thread that makes use of a timer must therefore create a CMessageQueue object.
- Threads can therefore check for the existence of a timer message in their message queue.

Principles of Concurrent Systems: Timers

Creating and Using Timers

- To create a timer, a thread simply has to create a CTimer() object.
- The timer delay can either be specified when the CTimer object is created (in which case the timer starts immediately), or, the timer can be delayed until a time interval is specified later. (see example later).
- When a timer expires, a message is posted/sent to the threads message queue. A number of primitives exists to deal with these messages

▪ CTimer(int timeout)	- Constructor to create a timer with specified timeout period
▪ TestForTimer()	- interrogates the message queue to see if a timer has posted a message there.
▪ WaitForTimer()	- pauses the thread until a timer message is received.
▪ SetTimer()	- Changes the timer interval
▪ Kill Timer()	- Stops a timer so that it no longer posts timer messages

- A simple example program demonstrating the use of timers is give below which uses a single timer to carry out pre-defined actions at ½ and ¾ second intervals.
- A more advanced example can be found in the tutorial guide to Win32.
- Q10 in the tutorial demonstrates the use of messages message queues and timers.

Principles of Concurrent Systems: Timers

```
int main()
{
      CMessageQueue      mq1 ;                                        // Create a message queue for this child thread
      CTimer             t1(50) ;                                     // set up repetitive timer to go off at 50mS intervals

//    Use above timer delay to generate in software other timing info.
//    In this case, a timer that goes off at 1/2 sec and 3/4 second intervals

      int Timer1 = 10 ;                                               // 10 ticks of 50mS timer = 1/2 second
      int Timer2 = 15 ;                                               // 15 ticks of 50mS timer = 3/4 second

      for(int i = 0; i < 200; i ++)      {                            // wait for timer, do this 200 times
            t1.WaitForTimer() ;                                       // when this threads timer goes off
            if( --Timer1 == 0)           {                            // decrement timer1 until 10 ticks of real timer occur
                        printf("ThreadFn2 Got 1/2 sec Timer.....\n") ;
                        Timer1 = 10 ;
            }
            if(--Timer2 == 0)            {                            // decrement timer2 until 15 ticks of real timer occur
                        printf("ThreadFn2 Got 3/4 sec Timer.....\n") ;
                        Timer2 = 15 ;
            }
      } ;
      t1.KillTimer() ;                                                // stop the child threads timer
      return 0 ;                                                      // terminate child thread
}
```

Principles of Concurrent Systems: Inter-process Communication

Inter-process/thread communication in a distributed system

- In today's increasingly sophisticated system, more are more applications are being developed based on a distributed/networked architecture model, where the model is one of multiple processes running on multiple CPUs connected via a network, e.g. the internet, or some local private network.
- The problem faced here is how to write the application that allow a process on one computer communicate with a process on another computer in a seamless fashion.
- Basically there are three approaches

    - Unix style 'Sockets'
    - CORBA/COM Programming
    - Java's Remote Method Invocation (which is basically another protocol built on top of CORBA/COM for JAVA programmers)

Principles of Concurrent Systems: Inter-process Communication

Unix Sockets

- Basically this is a low level programming approach to inter-process communication in a distributed world which has been adopted by virtually all operating systems not just UNIX.
- Using sockets makes it hard to disguise the fact that you are communicating across a network interface perhaps to a remote machine. In other words you cannot easily hide the fact that your application is distributed across machines.
- Individual Sockets or communication paths can be created within your processes to facilitate communication over a network using either an IP addresses (e.g. 192.123.456.789 etc) in conjunction with a port number or, less commonly (because there are more efficient ways to do it as we seen in this material) using system specific pathlists to allow communication between processes on the same host machine.

The Listener

- Essentially socket programming involves the listener
  - Creating a socket using an IP/Port number address. The Port number effectively identifies the application sat at the listening end. When you connect to a port, you should thus know what to expect in terms of a protocol from the end applications. For example Port #23 is reserved for TelNet servers, Port #21 for FTP servers, while web server listens on Port #80.
  - Listens for incoming requests/data on that socket, i.e. other processes attempting to communicate with it.
  - Accepting an incoming request for communication.
  - And, because multiple processes may be attempting to communicate with you, you may have to create multiple threads to deal with each incoming request from remote threads on other machines.

Communicating with a remote process using Sockets

- Basically the talker process
  - Creates a socket
  - Connects to the host (listener) via the listeners IP address/port number.
  - Then communicates with the remote process using low level read/write functions similar to those for a pipeline
  - Closes the connection when communication has finished

- For a simple, quick and dirty primer on socket programming, and some sample C code

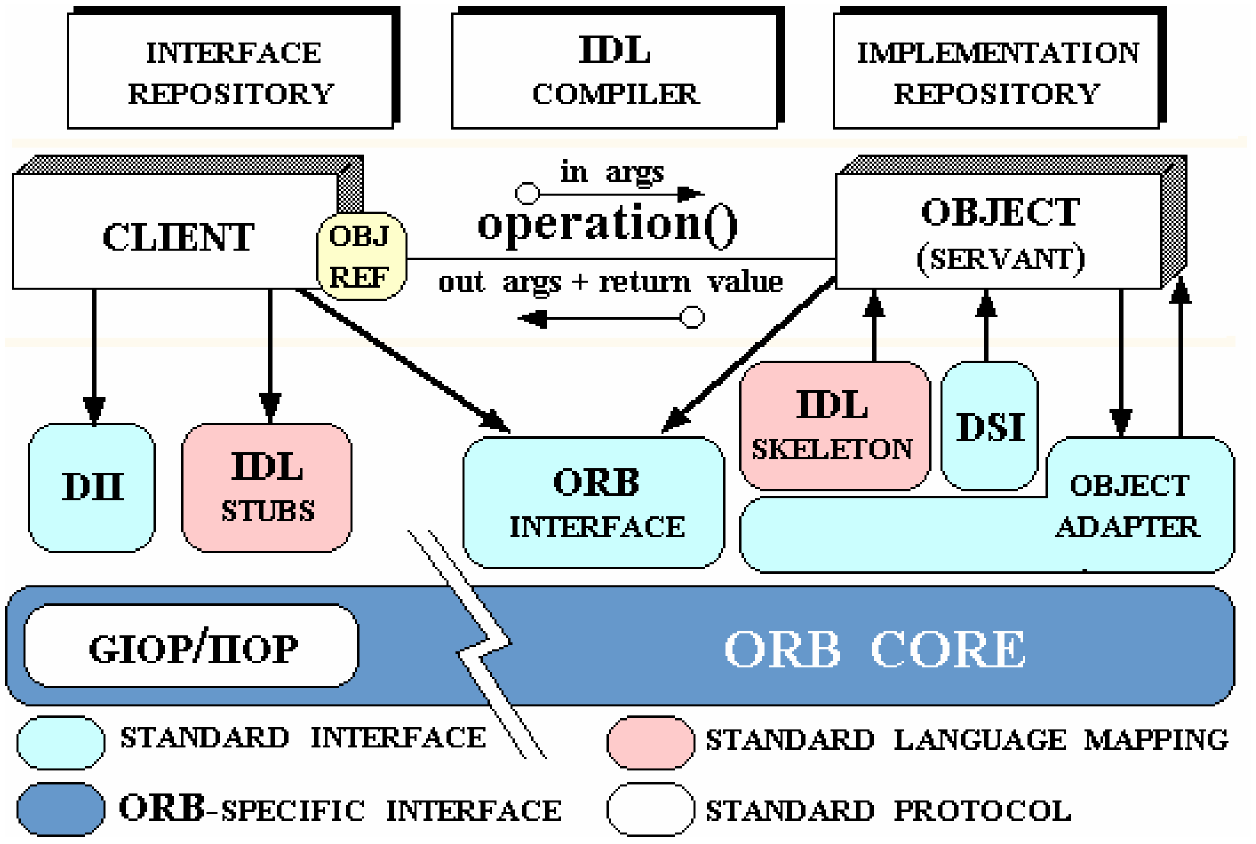
  search for “quick primer on socket programming”

Principles of Concurrent Systems: Inter-process Communication

Communicating with CORBA

- The Common Object Request Brokerage Architecture (CORBA) is an open standard defined and standardised by the Object Modelling Group (OMG). COM is Microsoft's own proprietary version of CORBA.
- CORBA Defines a model for communication between objects in a distributed object architecture.
- In traditional Object Oriented Programming languages like C++/Java etc, objects communicate through a process know as message passing, where one object directly invokes a member function/method of another object to get it to perform an action or operation.
- More and more applications are now being developed where the communicating objects that form the application are being distributed, perhaps around the world on different computers, making a direct message passing connection between objects impossible. One solution to this is to use sockets but it’s not very elegant and the cracks in the model are all to apparent.
- CORBA attempts to hide those cracks by building higher level protocols on top of something like sockets, so that code can be written in such a way that a process locates an object on another and appears to talk to it directly as if it were part of the same application.
- The client process communicates directly with IDL compiler generated stub code to gain access to an object request broker (ORB) which is responsible for locating the objects a process wishes to use (they could be on a machine half way around the world) and creating a reference to a proxy for the real object at the other end on another machine.
- The client communicates with the proxy which in turn communicates with the ORB to route the messages, parameters and returned data along the network between the communicating objects so as to hide the fact that objects are really communicating along a network

Principles of Concurrent Systems: Inter-process Communication



See (http://www.cs.wustl.edu/~schmidt/corba-overview.html)