Understanding Threading in ArcInfo™ 8

An ESRI® Technical Paper • September 2001

ESRI 380 New York St., Redlands, CA 92373-8100, USA • TEL 909-793-2853 • FAX 909-793-5953 • E-MAIL info@esri.com • WEB www.esri.com

Understanding Threading in ArcInfo 8

An ESRI Technical Paper

Contents **Page**

Appendix

Understanding Threading in ArcInfo 8

Introduction

This document is primarily intended for C++ programmers; threading, as it specifically applies to Visual Basic, will not be discussed. The reader is assumed to have some experience in Windows and Component Object Model (COM)programming and at least a basic understanding of threading concepts.

Writing robust multithreaded applications is much more difficult than writing single threaded applications and requires a thorough understanding of threading under COM and Windows. Given this, it should be obvious that a small technical paper by itself is inadequate preparation for writing reliable multithreaded applications or extensions to ArcInfo™ 8 software. This document contains useful information on what multithreading means in the context of COM, some guidelines on how to correctly use threading and COM together, and a few specific rules that must be followed to properly integrate threading into ArcInfo 8 applications. In addition, an accompanying code example has been provided, written in C++ using Microsoft's ActiveX Template Library (ATL). Finally, a few of the more notable texts on threading, Windows, and COM have been listed in the appendix for your convenience. Each of these resources should prove invaluable to developers interested in multithreading.

It is important to understand that multithreading is not a magic bullet for making faster or more responsive applications. In many cases it is ill suited for the problem at hand or may introduce complexities or overhead that actually reduces the speed at which a task can be done. Obviously, multithreading should be used only when it is clearly advantageous to do so and only with a good understanding of the objects that will be involved in its application.

What Is Multithreading?

Since the word multithreading has become an increasingly ambiguous term, even among programmers, a brief review is presented here. In general, multithreading refers to a software configuration where independent paths of execution are in use simultaneously in an application. Each thread has its own stack and its own CPU state. In interactive graphic user interface (GUI)-based applications, the CPU is inactive most of the time because interactivity is mostly concentrated in short bursts separated by long delays while the user digests the resultant information at "human" speed. Common applications of multithreading involve low-priority threads that perform calculations during these idle periods, providing a result that is helpful to the user but not necessarily needed before the user can continue. The sample provided with this document is an example of this technique, as are the many examples present in Windows itself such as print spooling, image file previews in Explorer, and directory file counts on directory property pages. A related approach takes advantage of the large periods of CPU inactivity that can occur when a thread sleeps while waiting for a comparatively slow hardware device. If several devices are connected and several requests are coming in to

use these devices, threads can be spawned to maximize CPU usage by applying other threads to available devices while others are waiting. Finally, if multiple CPUs are available on the same machine, several threads can be scheduled simultaneously—called multiprocessing—to divide and conquer a large problem. For this technique to be beneficial, you must be able to break the problem into independent pieces, a task that can be very difficult or impossible. In the best-case scenario, this technique can approach the total time normally required divided by the number of available processors.

Concurrency

When several threads act within an application, it is usually necessary to control how global data and other sorts of global resources are shared between these threads. This control, called concurrency control, is used to serialize method calls made from different threads, which might otherwise result in data corruption if both tried to access the same data simultaneously. When such concurrency control is in place, the code is said to be thread safe.

Classic thread concurrency control is normally accomplished with system objects such as mutexs, critical sections, semaphores, and events that act to suspend all but one of several threads vying for a particular function. However, making function bodies thread safe is not always sufficient because the persistent state of an object instance must also be considered when that instance is shared with more than one thread. Consider, for example, an object shared between two threads (we will call them thread A and thread B). This shared object represents a database cursor with reset/next semantics and, as such, must internally maintain state that keeps track of the current record position between calls to next(). If thread B were to call reset() after thread A had already enumerated through several records, thread A would then unexpectedly be back at the beginning on its own subsequent call to next(). This is known as a race condition.

In ArcInfo 8, single instances of stateful objects should not be shared between multiple threads if the possibility exists that more than one of these threads will be modifying the state of the instance. In other words, you as the developer must intervene and orchestrate multithread access to such shared instances to ensure a coherent state for the lifetime of the instance.

Working and Playing Well with Others

One problem with threading in the context of COM stems from the fact that component-based systems are often complex aggregations of many smaller components, each possibly supplied by independent software vendors and each with varying degrees of support for multithreading. To reduce the ambiguity associated with this issue, the COM specification requires that all clients and servers explicitly state whether or not they are prepared to handle concurrent access. In addition, basic thread concurrency control (function concurrency) is integrated into the COM run time via marshaling and the apartment concept described in the next section.

Apartments

The COM threading model is based on a conceptual entity known as an apartment. As of this writing, three distinct apartment types have been defined: the single threaded apartment (STA), the multithreaded apartment (MTA), and the thread neutral apartment (TNA). STAs can contain one thread only, though there can be many independent

STAs, each containing a single thread. Every thread that plans to run under an STA must, as its first action, register itself with COM using CoInitialize. A new single threaded apartment will be created by the COM run time for each thread that registers itself in this way.

STA components are normally written without concern for concurrency issues, that is, without using critical sections in their functions to protect their class state against concurrent access. Simply associating your component with the STA will allow the COM run time to automatically serialize method invocations from other threads via a COM supplied proxy[i]. This alone, however, may still leave a component unprotected if instances of this component share some global resource. To illustrate, imagine a component implemented by class A; code within methods on class A commonly access a global pointer to a chunk of memory that all instances of class A use as a scratch area during calculations. Now imagine that one instance of the object implemented by class A is created on STA thread T1, and that a separate instance is created in a separate STA (on a separate thread) T2. If a method is called by T1 on its instance of A, it may at any time suddenly be preempted by T2. T2 might call the same method on its own instance of A and so the contents of the global chunk will be altered; when control is returned to T1, it will likely crash since its chunk has been altered from underneath it. The concurrency provided by STAs does not and was not meant to cover such cases. The protection provided by the STA was intended to protect single instances of objects shared between different threads.

STAs rely on windows messages to handle concurrency behind the scenes and so must periodically pump windows messages. If a thread in an STA must wait for another thread to become signaled, a special wait function must be utilized called ::MsgWaitForMultipleObjects (see Figure 1 below). This function allows messages to be pumped while waiting on any kernel object. Note that the use of local message pumps could cause the waiting thread to be reentered (on the same thread) if the user starts interacting with the user interface. One way of avoiding this problem is to simply disable the user interface while waiting if such reentry is undesirable, and reenable it afterward.

[i] See the marshaling section for more information on proxies.

Figure 1
Pseudo Blocking in STAs

```
void ThreadManager::WaitForObject(HANDLE hThread)
{
  // Wait until worker thread terminates.
  // Pump incoming messages while waiting if necessary.
  //
  while (true)
  {
    // This function will suspend this thread until hThread is signaled
    // (terminates), or until a message is received on this thread.
    // STA/UI threads are not permitted to block; they must pump messages while
    // waiting, or a deadlock will occur.
    DWORD dwCode = ::MsgWaitForMultipleObjects(1, &hThread, FALSE, INFINITE,
                                               QS_ALLINPUT);

    // This condition is true if the thread is signaled (has terminated).
    if ((dwCode - WAIT_OBJECT_0) == 0) break; // Exit while.

    // Otherwise, service the message queue.
    MSG msg;
    while(::PeekMessage(&msg, NULL, 0, 0, PM_REMOVE|PM_NOYIELD))
    {
       ::DispatchMessage(&msg);
    }
  }
}
```

STAs can also be entered when calling out of their own apartment. Whenever code in an STA makes an out-of-apartment call, COM will pump messages for the thread while waiting for the call to return; this feature enables the STA to receive callbacks. It also allows other apartments to call in during this time. To control this sort of reentry, Microsoft allows each STA to register a MessageFilter—an object that implements IMessageFilter—using the COM application programming interface CoRegisterMessageFilter. For more information, see the Microsoft Developer Network (MSDN) documentation on the IMessageFilter interface. In contrast, MTA components must handle all function concurrency within their implementation. There is only one MTA per process space, but any number of threads can join the MTA by calling CoInitializeEx, passing COINIT_MULTITHREADED.

Ideally, the performance of code executing in the MTA can be more efficient since any thread in the MTA can directly access any other thread in the MTA without using a proxy. Since all cross-apartment communication using COM interfaces must be performed through a proxy, the MTA may be a good apartment choice if objects will be shared between several collaborating concurrent worker threads.[ii] Since the MTA has

[ii] Such objects will still require cross apartment/proxy access when accessed by UI components in the main STA or other apartments.

weak thread affinity, it is not an appropriate apartment type for use with user interface components.

The TNA is a relatively new apartment type introduced with Windows 2000. Its purpose is to mitigate the expensive cost of interapartment communication by eliminating costly thread context switches. Components marked Neutral have no thread affinity whatsoever and always execute on the thread of the caller. Like the MTA, this makes them off limits for use in components associated with the user interface or any other resource that is tied to a particular thread. Unlike the other two apartment types, threads cannot CoInitialize into the TNA.

COM components exposed from dynamic link libraries (DLLs) must specify the apartment type under which they have been designed to run. COM servers in DLLs could potentially be used by any arbitrary client apartment and so must be explicit about their concurrency requirements. During component registration, each CoClass in the DLL must set its threading model key to the appropriate apartment type as shown in Figure 2.

Figure 2
Threading Model Keys

Value of Threading Model Key	Apartment Type
No Threading Model Designation	Single Threaded (ST)—All components activate in the main STA
Apartment	Single Threaded Apartment (STA)
Free	Multithreaded Apartment (MTA)
Both	Compatible with either (i.e., will always be created in the client's apartment)
Neutral	Thread Neutral Apartment (TNA)

Marshaling

To enforce the concurrency semantics set by each involved apartment, calls made between two apartments (even if in the same process) must be marshaled. Marshaling is the process of carrying a method invocation made from one apartment to another apartment. Standard marshaling can involve a fair amount of overhead due to packing, transmitting, and unpacking the method parameters into the target apartment.

Since objects specify their own apartment type, calls to instantiate an object from one apartment may not always result in a direct pointer to the object in the creator's apartment. ArcInfo 8 applications call CoInitialize on start-up and so establish the main thread of the process as the main STA thread. Components that have registered themselves as Apartment or Both will exist in this main STA thread if CoCreated from code within ArcInfo 8. Components registered as Free are not compatible with the main STA apartment and so will exist on a separate thread residing in the MTA. In this case, COM will return a proxy to the ArcInfo code running in the main STA that CoCreated the object. Obviously, this proxy must be connected to a thread in the MTA, but where did this thread come from? In fact, a thread pool is maintained by the COM run time for cases where an STA holds a proxy—obtained through CoCreateInstance—to an object in the MTA. When a method on such a proxy is invoked, the COM run time will use an available thread from this pool to execute the server code. Calls between an STA and a component marked Neutral (TNA) will execute through a proxy

but with a lighter weight variety that will not require a thread context switch. For a more complete discussion of threading in COM, consult *Essential COM* by Don Box, listed in the appendix, which provides a very detailed description of this topic.

Thread Affinity

Objects in an STA have thread affinity; that is, they can only be called in the same thread context as that in which they were created. In contrast, objects within the MTA have a lesser degree of thread affinity in that an object in one thread can be called directly by any other thread as long as both of these threads exist within the MTA. Objects within the TNA have no thread affinity.

Like COM objects in an STA, handles to certain Windows objects, such as HWNDs and HHOOKs, have thread affinity and normally should be used only from the thread that originally created them. For this reason, threads that directly make calls to or, in general, manipulate the user interface should always reside in an STA.[iii]

Passing Interfaces Between Threads

Given the previously stated constraints, a typical scenario might involve a separate worker thread residing in the MTA or a separate STA. This thread does some number crunching and reports the results back to the main STA so that it may update the UI and display the results to the end user. Worker threads typically need some hook from the main thread such as IApplication. You may be tempted to simply pass such a reference to your thread as the initialization parameter, but do not! As already established, raw interface pointers can never be passed across apartment boundaries because the target thread would then be able to call into the original apartment without restriction, violating the concurrency of the apartment. Instead, interface pointers must be packaged up into an IStream pointer that is treated specially by the COM run time. This IStream pointer should be passed as the thread procedure's initialization parameter instead of the raw interface. Within the thread procedure itself, the incoming IStream parameter can then be used to obtain a marshaled proxy to the packaged interface. Figure 3 demonstrates how the interface is packaged and passed to the worker thread; Figure 4 demonstrates how to unpackage this interface from within the worker thread.

[iii] Normally the main STA of the process.

J-8763

Figure 3
Passing a COM Interface to a Thread as an Initialization Parameter

```
void ThreadManager::StartWorkerThread()
{

  // "Package up" our reference to the application within a stream so that we
  // can pass it (indirectly) to the worker thread function.
  //
  IStream* pStrm;
  ::CoMarshalInterThreadInterfaceInStream(__uuidof(IApplication), m_ipApp,
                                          &pStrm);

  // Create thread, pass the IStream*.
  unsigned int threadID;
  m_hThread = (HANDLE)_beginthreadex(NULL, 0, CalcArea, pStrm, 0,&threadID);
 }
```

Figure 4
The Thread Procedure: Obtaining the COM Interface from the Stream Parameter

```
unsigned int WINAPI CalcArea(void* pStrm)
{
  ::CoInitialize(0)

  IApplication* pApp;

  // Unmarshal the application interface from the provided stream.

  ::CoGetInterfaceAndReleaseStream(reinterpret_cast<IStream*>(pStrm),
                                   __uuidof(IApplication), (void**)&pApp);

<rest of thread procedure (use interface here)>

  pApp->Release();

  ::CoUninitialize();

  return 0;
}
```

Normally, this thread initialization issue is the only place where you will need to become directly involved with marshaling an object reference.

Remember, you must release all local references to objects obtained within a worker thread before calling CoUninitialize, as calling methods on COM objects after calling CoUninitialize will result in the dreaded "undefined behavior." Also beware the use of smart pointers in such threads and take care to set them to NULL (which calls release) before calling CoUninitialize.

Marshaling's Influence on Performance

Interapartment calls (and thus marshaling) should be kept to a minimum for performance reasons. If an interface to an object from the main STA is handed to one of the threads in the MTA or a separate STA, all calls made on that interface will require marshaling. If possible, it is often more efficient to pass a thread neutral identifier, such as a file name or record ID, to worker threads instead of making many cross apartment calls. In this way, objects can be created from these identifiers in the worker's thread and apartment, and as a result, calls on these objects will be made directly.[iv] Even the TNA's lightweight proxy is much heavier than a direct call.

Your threaded code may not necessarily rely on external interfaces in separate apartments or may run in a low-priority thread where the overhead of marshaling may not be an issue. In the end, if your code exists in a separate thread from the main STA, it will have to manipulate ArcInfo objects that currently reside in the main STA through a proxy.

Threads and DLLs

Most opportunities for extending ArcInfo applications involve the usage of COM objects residing in external DLL modules; thus, multithreaded components will need to spawn threads. This situation turns out to be somewhat problematic due to the way that COM and Windows are designed. When executables shut down and CoUninitialize is called by the main thread, the operating system automatically unloads all DLLs attached to the main process. If some of these DLLs have spawned threads that are running at the time of shutdown, these threads will be unceremoniously terminated. They are not given the opportunity to shut down gracefully through CoUninitialize, and resources may not be freed as a result.

To address this issue and support users who wish to leverage multithreading, ArcInfo provides the IMultithreadedApplication interface obtainable from the application object in ArcMap™ and ArcCatalog™. This interface provides a simple callback mechanism for registering user-created thread manager objects, which will be notified prior to application shutdown so that all currently running threads can be exited cleanly before the process actually shuts down.

Threading Limitations Within ArcInfo 8

As of the current release, several conditions exist that complicate and/or limit the use of ArcGIS™ components on multiple threads. Most central is the presence of COM singletons such as any WorkspaceFactory, the StyleGallery, environment objects such as the SymbologyEnvironment, and others. These singletons are presently marked as Apartment and so reside permanently on the first STA that creates them. As a result, calls to such objects from other apartments/threads must be marshaled and this, as pointed out earlier, can be costly. In addition, a small number of components have been identified as thread unsafe under certain conditions.

As an alternative to threads, separate worker processes can be spawned, usually via custom local (exe) COM servers. Like threads, these separate processes will take advantage of multiple CPUs but will not share the same address space. Though this isolation can improve reliability, resource consumption will increase. In addition, information/results calculated by these processes will normally have to be marshaled

[iv] If their apartment types are compatible.

back to the main process or alternatively transformed into raw data and shared via Win32 Named Shared Memory.

ESRI is currently working on a number of improvements in this area aimed at better facilitating the use of threads. In the interim, programmers interested in using threads are encouraged to post specific queries via their support channel or through the user forum at ArcObjects™ Online.

Example—The Selection Area Command

In conclusion, a simple example has been provided to illustrate the points presented in this document. This example works in conjunction with the select features tool provided with ArcMap and simply calculates the area of the currently selected features in a background thread whenever the selection changes. The result is displayed on the left edge of the status bar.

Click here to download the Selection Area Command files (ZIP format, 625 KB).

To try it out, you will need to be running version 8.0.2 or greater of ArcInfo. Once the SelectionArea.dll is registered, start ArcMap, open the customize dialog, select the Multithreading Examples category, and drag and drop the SelectionAreaCmd to any convenient spot on any toolbar.

To enable selection area calculations, simply click on the SelectionAreaCmd button, and then click on the select features tool to make a selection. When a selection is made with the select features tool, the selection is highlighted on the map. Once the highlighted features appear, the worker thread will begin calculating the total area of the selection. While this thread is running, you should find the application responsive.

If another selection is made before the current area calculation is complete, the thread will automatically be canceled and another will begin calculating the new sum. If you decide to quit ArcMap while the thread is still running, the thread manager will terminate the thread cleanly, freeing all associated resources before ArcMap actually shuts down.

Appendix

Basic COM Apartment Rules

Adherence to the rules listed below will be crucial to any successful application of threading in ArcInfo.

- All threads must call CoInitialize/Ex on start-up and CoUninitialize on exit. Release all local pointers before calling CoUninitialize.
- Do not access raw interface pointers across apartment boundaries.
- User interface threads and objects must run in STAs.
- Access to shared instances of objects must be externally managed if threads participating in the share will alter the object state.
- Class shared resources, such as global variables or static class members, must be manually protected against concurrent access even if the STA is used.
- STA threads containing servers should pump messages and should not block.
- COM DLLs that spawn threads must contain a thread manager object that implements IDllThreadManager.
- Be aware of the possibility of reentry in STAs; the COM-provided concurrency control only protects functions from concurrent access by other threads.
- Use MessageFilters to control reentry when making outbound calls from STAs.

Recommended Resources on COM, Windows, and Threading

- The COM specification available from MSDN On-line at http://msdn.microsoft.com/.
- *Essential COM,* Don Box, 1998 Addison Wesley Longman, Inc.
- *Effective COM,* Don Box [et al.], 1999 Addison Wesley Longman, Inc.
- *Win32 Multithreaded Programming,* Cohen & Woodring, 1998, O'Reilly & Associates Inc.
- *Programming Applications for Microsoft Windows,* Jeffrey Richter, 1999, Microsoft Press.
- Several very helpful articles are available in back issues of *MSJ* (now called *MSDN* magazine). These issues are now available online at http://www.msj.com/.

J-8763

- The DCOM list server at http://discuss.microsoft.com/archives/dcom.html allows you to perform searches on the many thousands of postings stored in their database.