Data sharing on Mac OS

Miroslav Jurišić, <meeroh@mit.edu>

Abstract

This paper explores different possibilities for data sharing on Mac OS. The applications for data sharing include communication between different processes as well as communication between processes and standalone code. The mechanisms explored cover Apple Events, Code Fragment Manager, Process-to-Process Toolbox, Gestalt, Multiprocessing Queues, Mach IPC, and others. The environments considered are Mac OS 8 and 9, Carbon on Mac OS 8 and 9, Carbon on Mac OS X, Mac OS X, and the Classic environment on Mac OS X. The paper compares different methods of data sharing based on their runtime requirements, operating system requirements, ease of programming, user experience, and appropriateness for different applications.

Introduction

Frequently, there's a need to share data between different pieces of code which comprise one package. For example, if you are writing an application and a Control Strip module, you might want to display application status information in the Control Strip module. This means that you need to share some data between the control strip and the application.

Matters can get much more complicated as your product grows. You might have more than one application, a Control Strip module, a contextual menu module, a faceless background application, and they might all need to share some data.

This article explores different ways to share data among applications and standalone code under Mac OS. For each of the techniques, the description includes a summary of runtime environments in which the technique can be used. The runtime environments considered are:

- Mac OS 8 and 9 interrupt time

  Techniques in this category can be used from interrupt time under Mac OS 8 and 9. Typically, there is some setup code which has to run as system task time before the interrupt time code can run.

- Mac OS 8 and 9 standalone code

  Techniques in this category can be used from standalone code (code resources, plugins, etc.) under Mac OS 8 and 9. These techniques do not require an event loop, but they are in general not interrupt-safe.

- Mac OS 8 and 9 68K

  Techniques in this category can be used on 680x0-based computer running Mac OS. (Note that 680x0 code running on PowerPC can take advantage of additional techniques available on PowerPC by using the Mixed Mode Manager.)

- Mac OS 8 and 9 InterfaceLib

  Techniques in this category can be used from InterfaceLib code under Mac OS 8 and 9. Some of them may have additional system requirements.

- Mac OS 8 and 9 CarbonLib

  Techniques in this category can be used from CarbonLib code under Mac OS 8 and 9. Some of them may have additional system requirements.

- Mac OS X CFM

  Techniques in this category are directly available to Carbon CFM code running under Mac OS X. Although it's possible to call between CFM and Mach-O code under Mac OS X, it can be tedious, and thus the distinction is made between facilities available to CFM code and to Mach-O code.

- Mac OS X Mach-O

  Techniques in this category are directly available to Mach-O code running under Mac OS X. They are also available to CFM code via one of the methods described in the "Calling Mach-O from CFM" section.

- Mac OS X Classic environment

  Techniques in this category, when used in code running inside the Mac OS X Classic environment, enable communication with applications running natively on Mac OS X.

General Considerations

Data Sharing Architecture

If you need to use data sharing, you have a product which consists of several components, and you need to share some data among them. Some of the components will generate data (*data producers*), some will just use it (*data consumers*), and some will do both.

In most cases, you will have one component which is charged with storing the data. This component is your *data provider*; data producers hand data off to the data provider, while data consumers retrieve data from the data provider.

When possible, it's common to make one of your existing components the data provider. This avoids adding more components to the system, but it is only possible if the type of component you need for a data provide (a faceless background application, a library, etc.) already exists in a suitable place in your product.

Some literature refers to data providers as servers, and other components as clients; this paper avoids that terminology because it's overloaded.

Designing a Data Sharing API

The first thing you should do when you need to deal with data sharing is design an API which provides the necessary accessors functions. This allows you to change the underlying data sharing implementation much more easily. This helps even if all you need to share are a few num-

bers, and is especially important if your data becomes complex.

Your API design will normally have very little influence on which data sharing mechanisms you can use. The mechanisms available to you are mostly determined by the complexity of your data, the complexity of communications you need, and your runtime and system requirements. Therefore, concentrate on designing a simple, robust API, and you will be able to find an appropriate implementation.

Calling Mach-O from CFM

If your code is running on Mac OS X, you may need to access system services which are not provided in the form of CFM libraries. In order to do that, you will need to call Mach-O code from your CFM code – unless you want to rebuild your code as Mach-O code.

The basic principle of calling Mach-O code from CFM is to use the Bundle Services API in Core Foundation. You can get a reference to a Mach-O framework using the Bundle Services, load the framework, and then retrieve addresses of various symbols in the framework.

Apple provides an example of this in the Carbon SDK, as well as in the sample code archive.

If you need to do this for many symbols, you may want to look into using the CocoaLib CodeWarrior plugin, which automates the process.

Synchronization and Data Consistency

Whenever you are sharing data between different pieces of code which can run concurrently, you have to worry about synchronizing access to shared data.

The standard example of this is the concept of interrupt-safe code on Mac OS 9. For example, code running at interrupt time cannot call the Memory Manager, because at the time the interrupt is triggered, the internal Memory Manager data structures might not be consistent – for example, if the interrupt is triggered when the Memory Manager is in the middle of relocating a block.

If you are sharing data in your code, you must worry about the same problem. If you do not understand the issues surrounding data consistency, you will find yourself tracking down bugs which are extremely hard to reproduce, because they depend on the exact timing of your code.

The first step in understanding the issues of data consistency is understanding the difference between cooperatively scheduled code and preemptively scheduled code.

Cooperatively scheduled code runs until it explicitly relinquishes control. For example, any Mac OS 9 application is cooperatively scheduled with respect to any other Mac OS 9 application, because the system will only switch from one to the other when WaitNextEvent is called. (It might seem that Carbon Events complicate this, but on Mac OS 9 Carbon Events use WaitNex-

tEvent themselves.)

Code which is scheduled preemptively can gain control of the CPU without that control being explicitly relinquished by the code currently running. For example, an interrupt task under Mac OS 9 is preemptively scheduled relative to all applications running under Mac OS 9.

When all you are dealing with is cooperatively scheduled code, it's easy to make sure that your data is always consistent: never relinquish control when the data is not consistent. For example, imagine you are sharing two integers, who sum always must be zero. When you change one of them, you have to change the other, and therefore there is a small window of time when their sum is not zero – when one has been updated but not the other. If you never relinquish control during that window of time, your data will always be consistent.

When your code is preemptively scheduled, you must use other techniques. There are many different ways in which code can be preemptive on Mac OS 9 and X, and therefore dealing with preemptively scheduled code is depended on what environment the code is running in. However, some basic rules apply:

- On Mac OS 9
  - Applications and other code running at system task time are cooperatively scheduled with respect to each other. Control is relinquished by calling `WaitNextEvent()` or `SystemTask()`.
  - All code running at interrupt time is preemptively scheduled with respect to system task time code, but only interrupt time code can interrupt system task time code – not the other way around.
  - Deferred time tasks are scheduled cooperatively with respect to each other. Likewise, secondary interrupt time tasks are cooperatively scheduled among themselves.
  - Other code running at interrupt time is preemptively scheduled with respect to interrupt time code.
  - Multiprocessing threads are preemptively scheduled with respect to all other code.
- On Mac OS X
  - Applications are preemptively scheduled with respect to each other.
  - POSIX, Mach, and Multiprocessing threads are likewise scheduled preemptively with respect to each other.
  - Some asynchronous Carbon callbacks (e.g., Open Transport or Time Manager callbacks) are scheduled preemptively with respect to the main thread, however, they can also be preempted by the main task, unlike on Mac OS 9. It is extremely important that you are aware of this change in behavior, because code is commonly written to assume the opposite.

When you are dealing with preemptive scheduling on Mac OS X, your life is usually relatively simple. You can use an appropriate synchronization API (Multiprocessing queues if you use the Multiprocessing APIs, POSIX mutexes if you use POSIX threads, etc.).

On Mac OS 9, you need to use one of the Mac OS 9 synchronization APIs; they exist as part of Driver Services, Open Transport, and Multi-Processing API. The Driver Services and Open Transport APIs work on a lower level than the other APIs. However, the Driver Services and Open Transport APIs are your only choice on Mac OS 9 unless you need to synchronize data among Multiprocessing threads.

Details of using those APIs are beyond the scope of this article, but the bibliography lists references to appropriate API documentation.

Sharing Mechanisms

Once you've decided what you want from of your API, you will need to implement the API to use a data sharing mechanism, or even several mechanisms, if your needs are very complex.

You should get a general feeling of which mechanisms are appropriate simply by reading the overview of every mechanism. For the ones that apply to you, read the detailed description and the sample code, to familiarize yourself with the mechanism, and then write your own code. References to Apple documentation are included for every mechanism, as the full documentation of the respective Mac OS APIs is beyond the scope of this paper, and you should read at least the introductory sections of the relevant documents, to gain a general understanding of the APIs you will be calling.

Gestalt

Overview

Gestalt is a system-wide registry which maps selectors (32-bit integers) to values (32-bit integers). A selector can have either a specific value (which can be changed), or a callback function which returns the current value.

On Mac OS 8 and 9, Gestalt can be used for data sharing among 68K and PowerPC applications and standalone code. On Mac OS X, Gestalt can be used for data sharing within one application, which is occasionally useful, but it cannot be used to share data between applications.

Gestalt primarily works for one-way communication. Trying to implement two-way communication in Gestalt is tricky and you are probably better off using one of the other methods in that case.

Details

There are two basic ways to use Gestalt for data sharing: Gestalt values and Gestalt callbacks.

When you use Gestalt values, your data provider will install a Gestalt value when it is initialized, and change the value whenever the data changes.

Your data consumers will read the gestalt value as necessary. Data consumers depend on the provider generating the information in a timely manner.

Gestalt values are limited to 4 bytes. If you need to share more information, the simplest way to do it is to allocate a block of memory in your data provider, and put a pointer to that block in the Gestalt value. The data consumer can dereference the pointer and access the data.

If you are sharing a block of data by putting a pointer in a Gestalt value, remember to remove the selector value when your data provider quits or is otherwise unloaded, so that the data consumer does not dereference a stale pointer.

If your data provider needs to leave the data around even after it quits or is otherwise unloaded, you will need to allocate the block in the System heap. You should avoid doing this, as the System heap is never eligible for virtual memory disk paging and therefore its size has visible impact on system performance.

You might prefer to generate your data when the data consumer requests it, rather than generating it in advance. If your data changes frequently, but data consumers only ask for it infrequently (for example, CPU load), it might be more efficient to only recalculate when the data is requested. If your data provider can't reliably get processing time and therefore can't update the data frequently enough, your data might be more current if it's generated on demand rather than updated periodically.

If so, you need to use a Gestalt callback. However, Gestalt callbacks are trickier than Gestalt values, because the code has to be loaded in the System heap.

When you really need a Gestalt callback, compile your code as a code resource (preferably making it fat if you support 68K and PPC computers), load the code resource into the system heap and lock it there, and install it as a Gestalt callback.

Summary

Mac OS interrupt time	no
Mac OS standalone code	yes
Mac OS 68K	yes
Mac OS InterfaceLib	yes
Mac OS CarbonLib	yes
Mac OS X CFM	no
Mac OS X Mach-O	no
Mac OS X Classic	no

PPC Toolbox

Overview

The Process-to-Process Communication Toolbox allows applications and standalone code to send and receive arbitrary messages, locally or via AppleTalk. Sending and receiving can be done at either interrupt time or system task time.

The main disadvantage of the PPC Toolbox is that it is not available under Carbon, and therefore any code using the PPC Toolbox has to be rewritten for Mac OS X. However, the PPC Toolbox has a number of powerful features which make it a good choice for code which will not be ported to Mac OS X, or code which will use a different data sharing mechanism on Mac OS X.

The main benefits of the PPC Toolbox are that

it is available at interrupt time and does not require access to an event loop. It is therefore an excellent choice for communication between an application and standalone or interrupt-time code.

The programming interface of the PPC Toolbox is significantly more complex than Gestalt. If you need simple one-way communication at system task time, Gestalt will serve you better than the PPC Toolbox.

The PPC Toolbox is a good choice for two-way communication in standalone code and interrupt time code. However, it is not the best choice for communication between applications, since other techniques are available to them which work as well, but support a wider range of operating systems.

Details

Your data provider will become a PPC Toolbox server. The server will be called at interrupt time to handle incoming requests from clients, which will be your data consumers.

Your server will handle the request at interrupt time, and this makes it hard to do many useful things in the PPC Toolbox server code. This is usually the biggest burden in writing PPC Toolbox code.

If you can generate your data at interrupt time, then your problem is easily solved – you can directly return your data. However, this mostly applies only to simple data, and if your data is so simple maybe you should just be using Gestalt.

Summary

Mac OS interrupt time	yes
Mac OS standalone code	yes
Mac OS 68K	yes
Mac OS InterfaceLib	yes
Mac OS CarbonLib	no
Mac OS X CFM	no
Mac OS X Mach-O	no
Mac OS X Classic	no

CFM shared data

Overview

Code Fragment Manager handles loading shared libraries and applications on PowerPC. Although iIt has been retrofitted to 68K, it is sometimes cumbersome.

Every CFM shared library carries its global data, and can choose whether the data will be shared per process or globally. Per process sharing means that if several applications load the library, the library will get a separate copy of its data in each application. This is the default setting, and the one that is most natural to use – there is no interaction between different applications as far as the library is concerned.

Global sharing means that there is no one copy of the data shared among all applications. The library, loaded in one application, can modify the shared data, and the changes will be seen by other applications in which the library is loaded.

CFM shared data is not supported under Mac OS X. This technique for sharing data can only be used on Mac OS 8 and 9. It is available to standalone code and code running at interrupt time, and it tends to be simpler to use than the PPC Toolbox.

Details

In this model, your code might not be as clearly divided into data producers and data consumers, since all of your code will access the shared data in exactly the same way. Nonetheless, when thinking about your code, it might be helpful to think about some of your code as producing data and mostly writing to the shared memory, and other as consuming the data and mostly reading from the shared memory.

You can divide all the data you will be sharing into fixed-size data (such as integers and fixed-size strings), and variable-size data (such as arrays).

Any fixed-size data can be declared as global variables in a shared library. Any variable-size data will have to be declared as pointers, and you will have to allocate the actual memory for the data on the fly. You will have to allocate the memory for variable-size data in the System heap; otherwise, it will be destroyed when the application in which it was allocated quits, causing all other applications to dereference stale pointers and crash when they try to use the data.

In general, you should not put any code in a shared library with globally shared data. More precisely, you can put code in a shared library with globally shared data if that code does not link against another library whose data is not globally shared. This means you can't call into InterfaceLib from your shared library with globally shared data, so there's really not much you can do. To avoid writing some safe code now and then forgetting about the restriction later, I recommend just not putting any code into your global shared library. Since you should have designed an API to access your shared data before you got this far, the safest approach is to put all the code which implements your API into a shared library with per-process data, and all the data into a globally shared library, and link the former against the latter.

You can treat your shared data just as any other variable, except that when you change it in one application, you can see that change from other applications.

Summary

Mac OS interrupt time	yes
Mac OS standalone code	yes
Mac OS 68K	yes
Mac OS InterfaceLib	yes
Mac OS CarbonLib	yes
Mac OS X CFM	no
Mac OS X Mach-O	no
Mac OS X Classic	no

AppleEvents

Overview

The Apple Event Manager allows applications to exchange arbitrary application-defined mes-

sages. The structure of AppleEvents is defined by the application.

Receiving AppleEvents requires an event loop. Sending AppleEvents doesn't, but it requires that the application be AppleEvent-aware. Because of that, AppleEvents are in general unsuitable for data sharing in standalone code or in libraries which can't guarantee they will only be loaded in AppleEvent-aware applications.

The Apple Event Manager is not interrupt safe, so Apple Events can't be used from interrupt task time.

The Apple Event Manager is available in Carbon. AppleEvents can leave and enter the Classic compatibility environment on Mac OS X.

Details

In order to use AppleEvents for data sharing, you will almost certainly need a faceless background application. A faceless background application is an application which is invisible to the user, and does not show up in the Application menu, but can send and receive AppleEvents.

Your faceless background application will be the caretaker of your shared data. Your data producers will send the data to the faceless background application, and your data consumers will retrieve the data from it.

Normally, your code will be simpler if you make the faceless background application be the data producer, but that is not necessary.

The biggest problem with AppleEvents is performance. Parsing and unparsing AppleEvents structures is more expensive than most other data sharing methods, and therefore you will run into performance problems if you try to send a large amount of data or perform a large number of transactions using AppleEvents. However, they are one of the few methods which allows your data to escape from the Classic environment on Mac OS X, so you should seriously consider AppleEvents if you need to exchange your data between classic and native applications of Mac OS X.

You can improve performance of your AppleEvent parsing code by not using AppleEvent Manager structures for your private data. For example, if you need to transfer 2 integers, you don't need to make an AppleEvent record with two fields, you can simply put the two integers in a struct and stuff the handle into an AEDesc. This is not such a good idea if you are using AppleEvents to make your code scriptable, but if you are only using them for data sharing, you can treat all AppleEvents as private and not bother with such niceties as coercion handlers. Simply treat an AppleEvent as a way to get a blob of data to another application.

Summary

Mac OS interrupt time	no
Mac OS standalone code	yes
Mac OS 68K	yes
Mac OS InterfaceLib	yes
Mac OS CarbonLib	yes
Mac OS X CFM	yes
Mac OS X Mach-O	yes
Mac OS X Classic	yes

Mac OS 9 File Mapping

Overview

Mac OS 9.1 introduces a new API for directly mapping disk files to memory addresses. This allows you to treat a file as a block of memory and read to it and write from it in the same way you would read and write memory. The changes you make to the memory range to which the file is mapped are reflected on disk.

Unfortunately, Mac OS 9.1 has a bug in the file mapping implementation which prevents an application from using more than one memory-mapped file on the same volume.

As of this writing, there are no Carbon file mapping APIs.

The main advantage of file mapping over other data sharing APIs is that the data is persistent – the data is written to the disk, and will be available to your application even after quitting and relaunching.

However, system requirements are steep, the Mac OS 9.1 implementation is limited, and the API can only be used to map a fixed file range to a memory range, so if your data needs to grow and shrink over time, you have to implement your own memory manager on top of the file mapping API.

This makes Mac OS 9 file mapping APIs suitable for only a very small number of problems.

Details

All of your data producers and consumers need to agree on which file to map to use for data sharing. Then, they all map the file into their respective address spaces, and use the shared memory however they want.

Since you get a fixed-size block of shared memory, if memory needs for your shared data grow, you will need to map additional file ranges. Your additional file ranges may come from the same file or a different file, and it is up to you to communicate among your components which additional ranges need to be mapped.

Before you can start using a mapped file, all of your code has to agree on which file to map, and in order to do that you will have to use some other method to communicate – probably one of the other data sharing methods available to you.

Since file mappings are quite limited in what systems they work on and how they can be used, and you have to use a different method to communicate the initial information, you are almost certainly better off just using a different method of data sharing. This is not to say that file mappings are useless, just that for data sharing you are better off using something else in most cases.

Summary

Mac OS interrupt time	no
Mac OS standalone code	yes
Mac OS 68K	no
Mac OS InterfaceLib	yes
Mac OS CarbonLib	no
Mac OS X CFM	no
Mac OS X Mach-O	no
Mac OS X Classic	no

Mach IPC

Overview

The Mach kernel is the foundation of Mac OS X. It has powerful inter-process communication APIs. The most fundamental API is the Mach IPC layer; other inter-process communication APIs in Mac OS X use Mach IPC.

Writing directly to the Mach IPC layer is in general discouraged by Apple. HIgher-level APIs, such as the Foundation Kit (in Cocoa) and Apple Event Manager (in Carbon) should be used instead.

However, there are situations where you may want to use Mach IPC directly. For example, if you are unable to use Java or Objective C, you cannot take advantage of Foundation Kit. If you do not want to rely on Carbon, you cannot use the Apple Event Manager. In such cases, you might want to resort to writing directly to the Mach IPC layer.

Mach IPC is only available on Mac OS X.

Details

If you are using Mach IPC you will typically want to have a faceless background application which stores all the shared data, and make other components talk to that application to retrieve or store the data.

In Mach parlance, the faceless background application will be an IPC server, and the remaining applications will be IPC clients.

Mach IPC works by having the client assemble a message it wants to send to the server, and then calling the Mach IPC interfaces. The Mach IPC interface suspends the client, sends the message to the server, and wakes the server. The server receives the message, and crafts a response, which is returned to the client. The client wakes and resume processing and the server is suspended again.

The functions which create structures passed into Mach IPC (in the client) and retrieve data from those structures (in the server) follow a standard form. All the parameters which need to be sent across are stored as fields of a structure; the pointer to that structure is passed to Mach IPC. The pointer comes out on the server end, where the fields are read out to create the response.

Because of this, Mac OS X provides the “mig” utility to automatically generate that code for you. The input to mig is a text file in C-like syntax which describes messages you want to send to the server. The output consists of client functions, which assemble messages from their arguments, and server functions, which disassemble

the messages and assemble the replies.

In order to establish a Mach IPC connection, you need to create a Mach IPC port which connects a client to a server. This is done by using the bootstrap server. The bootstrap server is a process which maintains an association between service names (which are descriptive strings) and servers. Your IPC server registers with the bootstrap server, with a name of your choosing. Your IPC clients contact the bootstrap server to establish a connection with your server.

The bootstrap server is an IPC server as well; the only thing special about it is that every application automatically has a connection to the bootstrap server, which allows it to establish connections to arbitrary other IPC servers.

Summary

Mac OS interrupt time	no
Mac OS standalone code	no
Mac OS 68K	no
Mac OS InterfaceLib	no
Mac OS CarbonLib	no
Mac OS X CFM	no
Mac OS X Mach-O	yes
Mac OS X Classic	no

Multiprocessing Queues

Overview

Mac OS 8, 9, and X support the Multiprocessing API which provides preemptively scheduled threads on PowerPC. This API includes facilities for sharing data between threads.

MP threads run in the address space of the application which created them, and therefore on Mac OS X only threads within one application can directly exchange data (even if you use kMPAllocateGloballyMask). However, on Mac OS 8 and 9, including the Classic environment on Mac OS X, MP threads from different applications can exchange data directly, and therefore can be used as a form of data sharing.

Details

The system requirements for the Multiprocessing API are listed in the API documentation. In short: Mac OS 8.6, PowerPC only, excluding PowerMac 6100, 7100, 8100, 5200, and 6200.

Prior to Mac OS 9.1, the list of Mac OS functions you can call from an MP thread was very short. Therefore, if you want to use MP threads for data sharing before Mac OS 9.1, you will need to delegate all the real work to an ordinary application or interrupt time task, and only use MP threads as a storage vehicle.

When using MP threads for data sharing, start by creating a thread and its request queue. The request queue is used to relay data requests to the thread; the requests might be either read requests, made by data consumers, or write requests, made by data producers.

Each data consumer and data producer will also have a response queue, which is where the thread will deposit the data when it's ready.

When a data consumer needs some data from

the thread, it puts a request on the request queue; the request includes a pointer to the response queue. Then it waits for the thread to put the response on the response queue.

After the thread is started, it handles requests simply by waiting for one to appear in the request queue, extracting it, producing the response, and putting the response on the appropriate response queue.

The tricky part of this system is that under Mac OS 8 or 9, the main Mac OS task is waiting for the thread to return the data, and therefore the thread cannot attempt to do anything that might require cooperation from the main Mac OS task. For example, if your thread needs to allocate memory to complete a request, it must use `kMPAllocateNoGrowthMask` to avoid a deadlock.

Summary

Mac OS interrupt time	yes
Mac OS standalone code	yes
Mac OS 68K	no
Mac OS InterfaceLib	yes
Mac OS CarbonLib	yes
Mac OS X CFM	yes
Mac OS X Mach-O	yes
Mac OS X Classic	no

Unix Pipes

Overview

A pipe is a first-in first-out communication channel. Two Unix processes can communicate by creating two pipes, one for each direction of communication.

A pipe can be associated with a file system entry (a named pipe), which allows processes to easily find the pipe.

Pipes work well for bidirectional communication among Mac OS X native processes. They do not exist on Mac OS 9, and are not available to processes running inside the Classic environment.

Details

Typically you will want to use named pipes, because they allow other processes to attach to the pipe after the pipe has been created. Unnamed pipes are only useful if your application creates the pipes and then uses `fork()` to start another application. Then the two applications can communicate using the pipes, but no other applications can join in.

To create a named pipe, you use the `mkfifo()` function. Given a file system path (usually to a temporary file), `mkfifo` will associate a pipe with that path. Then any process can open the pipe for reading or writing, using the standard C library (`open()`).

Usually you will only want one program reading from a pipe; if you need bidirectional communication, you should use two pipes, one in each direction. If you need to have multiple programs writing to a pipe, you have to implement some kind of synchronization mechanism, otherwise they will stomp over each other and the data on the pipe will be garbled. The simplest

way to do that is to use a “lock file”, which is a temporary file which is created by a process before it writes to the pipe and destroyed after it’s done writing. Since file creation is atomic, this guarantees that access to the pipe will be exclusive. Processes which need to write to the pipe need to wait until the lock file is released, using a `select()` call. When using bidirectional communication, the lock file can’t be deleted until the request has been sent on one pipe and the response read from the other.

Summary

Mac OS interrupt time	no
Mac OS standalone code	no
Mac OS 68K	no
Mac OS InterfaceLib	no
Mac OS CarbonLib	no
Mac OS X CFM	no
Mac OS X Mach-O	yes
Mac OS X Classic	no

Loopback Network Interface

Overview

Every computer capable for TCP/IP networking has a special ”loopback” network interface. The data sent over this interface never leaves the computer. Programs can communicate over the loopback interface just like they would communicate over any other TCP/IP connection; the reserved IP address 127.0.0.1 always corresponds to the loopback interface.

This method of data sharing is available on Mac OS 8 and 9 as well as Mac OS X. Processes inside the Classic environment can’t communicate with native Mac OS X processes using this method. However, the performance of this method on Mac OS 8 and 9 can be unsatisfactory, because of Mac OS’s cooperative scheduling.

Details

To use the loopback interface you will need a background application which listens for network connections and responds to them. This application will usually be your data producer as well, and data consumers will connect to it over the loopback interface.

Your server will need to listen on a specific port. Since you will in general not have access to ports below 1024, because they require administrator access on Mac OS X, you will need to use a port above 1023. However, any application can listen on any of those ports, sp you will probably want to chose your port on the fly, rather than assuming a particular port will always be free for you to use. You will need to communicate the port number via some other means (e.g. AppleEvents) in order to be able to establish the communication.

You will also need to design a protocol which your clients (data consumers) will communicate with your server (data producer). The simplest protocol would include at the beginning of each message the protocol version number, the message type, and the message length, followed by a variable size message.

This allows you to preserve compatibility with older clients, and to skip over messages

which your server does not understand (thereby allowing the client to try using a different protocol version if the first one it tries is not understood by the server).

You have to remember to connect to the IP address 127.0.0.1, rather than trying to connect to the real IP address of the computer you are running on, because there might be times when the computer does not have an IP address.

The loopback interfaces of Classic and Mac OS X are not connected to each other, so a server listening on the loopback interface in Classic will not see any traffic sent to the loopback interface from outside of Classic, and vice-versa.

Once you've determined how you will negotiate the port and what protocol you will use to communicate between data consumers and producers, the rest is a simple matter of Open Transport or BSD sockets programming.

Summary

Mac OS interrupt time	yes
Mac OS standalone code	yes
Mac OS 68K	yes
Mac OS InterfaceLib	yes
Mac OS CarbonLib	yes
Mac OS X CFM	yes
Mac OS X Mach-O	yes
Mac OS X Classic	no

Component Manager

Overview

Component manager is the plugin/shared library architecture of the Classic Mac OS which predates CFM. It's mainly used for QuickTime plugins, but it's still alive in several other areas as well. It is not supported on Carbon, and therefore using the Component Manager restricts you to Mac OS 9 or older.

Components can be used from any environment of Classic Mac OS.

Details

Component manager works similarly to CFM shared data. Your data provider needs to be implemented as a component, and the component has to be loaded on the system. This can be done from an extension, in order to make a component available to every application.

Allocations done in your component which need to persist among all applications must be done in the System heap, and therefore you should strive to minimize them.

Your data consumers and data providers will call the Component Manager to establish a connection with your component, and then use calls specific to your component to get the functionality you need.

Summary

Mac OS interrupt time	yes
Mac OS standalone code	yes
Mac OS 68K	yes
Mac OS InterfaceLib	yes
Mac OS CarbonLib	no
Mac OS X CFM	no
Mac OS X Mach-O	no
Mac OS X Classic	no

Revision History

- August 23, 2001: Reworded AppleEvents section to be more appropriate for Carbon (Eric Grant); removed stale todos from section on FBAs (John Selhorst)
- June 2001: First revision, for MacHack 2001

References

General Documentation

[1] Background-Only Applications
<http://developer.apple.com/technotes/tn/tn1070.html>

[2] Background-Only Applications in System 7
<ftp://ftp.apple.com/developer/Periodicals/develop/develop09/develop_Issue__9/Background-Only_Apps_in.sit.hqx>

[3] Interrupt-Safe Routines
<http://developer.apple.com/technotes/tn/tn1104.html>

[4] Carbon documentation
<http://developer.apple.com/techpubs/macosx/Carbon/carbon.html>

[5] CocoaLib
<http://www.eagrant.com/CocoaLib.hqx>

Specific APIs

[6] Gestalt Manager documentation
<http://developer.apple.com/techpubs/macos8/OSSvcs/GestaltManager/gestaltmanager.html>

[7] Gestalt Manager sample code
<http://developer.apple.com/samplecode/Sample_Code/OS_Utilities.htm>

[8] PPC Toolbox documentation
<http://developer.apple.com/techpubs/macos8/InterproCom/PPCToolbox/ppctoolbox.html>

[9] PPC Toolbox sample code
<http://developer.apple.com/samplecode/Sample_Code/Interapplication_Comm.htm>

[10] Code Fragment Manager documentation
<http://developer.apple.com/techpubs/macos8/RuntimeSvcs/CodeFragmentManager/codefragmentmanager.html>

[11] Code Fragment Manager sample code
<http://developer.apple.com/samplecode/Sample_Code/Runtime_Architecture.htm>

[12] Apple Event Manager documentation
<http://developer.apple.com/techpubs/macos8/InterproCom/AppelEventManager/appleeventmanager.html>

[13] Apple Event Manager sample code
<http://developer.apple.com/samplecode/Sample_Code/Interapplication_Comm.htm>

[14] File Mapping documentation
<http://developer.apple.com/technotes/tn/tn2011.html>

[15] Mach Server Writer's Guide
<http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/publications.html>

[16] Mach Server Writer's Interface
<http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/publications.html>

[17] Multiprocessing Services documentation
<http://developer.apple.com/techpubs/macos8/OSSvcs/MultiPServices/multiprocessingservices.html>

[18] MP-Safe Routines
<http://developer.apple.com/technotes/tn/tn2006.html>

[19] UNIX Network Programming, Volume 1: Networking APIs - Sockets and XTI, W. Richard Stevens

[20] Open Transport documentation
<http://developer.apple.com/techpubs/macos8/NetworkCommSvcs/OpenTransport/opentransport.html>

[21] Open Transport sample code
<http://developer.apple.com/samplecode/Sample_Code/Networking.htm>

[22] Component Manager documentation
<http://developer.apple.com/techpubs/macos8/RuntimeSvcs/ComponentManager/componentmanager.html>