Technical Note TN1128

Understanding Open Transport Memory Management

CONTENTS

This Technote describes how Open Transport's interrupt-safe memory management system works, and how you can use it for best effect in your software.

This Technote is directed at advanced programmers writing Open Transport client or kernel code.

[Jan 09 2001]

Introducing OT Memory Management

Open Transport provides many different interrupt-safe memory allocation routines. These include `OTAllocMem`, `OTAllocMemInContext`, `OTAlloc`, `OTAllocInContext`, `OTAllocSharedClientMem` and `OTAllocPortMem`. In order to know which routines to use in which circumstances -- and how using these routines affects the memory available to Open Transport and the rest of Mac OS -- you need to understand the OT memory management system.

Pools of Power

OT memory management is layered on top of four classes of **memory pools**:

1. A **client pool** is allocated for each program that calls `InitOpenTransport` (or `InitOpenTransportUtilities`). It is used to satisfy OT memory allocations for that program and it is destroyed when that program calls `CloseOpenTransport` (either explicitly or by an application quitting).
2. The **shared client** pool (also known as the **native pool**) is allocated when the first program calls `InitOpenTransport` (typically this is the AppleTalk protocol stack, early in the boot sequence). This pool is used by the OT client-side libraries for the bulk of their allocation.
3. The **kernel pool** is allocated the first time the OT kernel is loaded. It is used by the OT kernel and its plug-ins (for example, STREAMS modules and drivers).
4. The **port pool** is allocated the first time a program calls `InitOpenTransport` or `InitOpenTransportUtilities`. The pool is used to hold information about ports. It is distinct from the kernel pool, because port scanners can run without loading the kernel, hence without creating the kernel pool.

Open Transport memory pools are currently implemented by the Apple Shared Library Manager (ASLM) `TStandardPool` class and inherit some attributes from that class:

1. A pool is always allocated within a Mac OS Memory Manager zone.
2. Each pool starts with an initial size.
3. Memory pools are interrupt-safe. You can allocate memory from the OT memory pools at any time. However, the pool can only grow at system task time. If you allocate memory at interrupt time, the allocation may fail even though there is

enough memory in the zone to grow the pool to meet the request.

4. When the pool runs low on memory, the pool expands by allocating memory from the Mac OS Memory Manager at system task time. The pool grows by a percentage factor, known as the **grow by** factor. The amount grown is bounded below by the **minimum grow** amount.
5. The pool starts to grow when the amount of free space in the pool drops below the **low mark**. There is also a **high mark**, which defines when the pool should start to shrink. This feature is used only by the kernel pool.

IMPORTANT:
Open Transport's use of ASLM memory pools is an implementation detail. In the future OT will not use ASLM at all; as a result, OT memory pools will be implemented by OT itself.

Note:
You can read more about the ASLM memory pool classes in the ASLM Developer's Guide, available as part of the ASLM SDK.

Pool Parameters

The following tables gives the basic parameters of the various Open Transport memory pools.

Pool Type	Zone	Initial	Grow By	Min Grow	Low Mark	High Mark
Client [1]	Appl	2K	20%	2K	1K	infinite
Client [2]	System	1K	20%	2K	512	infinite
Shared [3]	System	2K	20%	4K	2K+1	infinite
Shared [4]	System	3K	20%	4K	3K+1	infinite
Kernel [6]	System [9]	4K [5]	20%	34K	34K+1	[10]
Kernel [7]	System [9]	250K	20%	34K	34K+1	[10]
Kernel [8]	System [9]	16K [5]	20%	34K	34K+1	[10]
Port	System	2K	20%	1K	1K+1	infinite

Notes:

1. This row is for client programs that link with the OT application libraries (those whose names end with "App") and who have not called `InitLibraryManager`.
2. This row is for client programs that link with the OT extension libraries (those whose names end with "Extn") and who have not called `InitLibraryManager`.
3. Open Transport 1.3 and earlier.
4. Open Transport 2.0 and later.
5. This is discussed in more detail in below.
6. Open Transport 1.3 and earlier.
7. Open Transport 2.0 through 2.5.
8. Open Transport 2.6 and later.
9. OT explicitly holds (in the virtual memory sense) the memory in the kernel pool. OT does not guarantee to hold the memory in the other pools.
10. The OT kernel pool shrinks and grows depending on a number of factors, which are described below.

IMPORTANT:
As you can see by the long list of notes above, the various pool parameters are subject to change. You should try to avoid dependencies on these values.

Back to top

Using OT Memory Effectively

This section describes various hints and tips for using the OT memory management system effectively.

OT Routines and their Pool Usage

The following table is a summary of the common OT routines that allocate memory, the amount of memory they allocate, and the pool from which they allocate.

Routine	Pool	Approximate Amount
`OTAllocMem` [1], `OTAllocMemInContext`	Client	depends on `size` parameter
`OTAllocMem` [1]	Kernel	depends on `size` parameter
`OTAlloc`	Client	depends on `ref` and `fields` parameters
`OTAllocSharedClientMem`	Shared	depends on `size` parameter
`OTAllocPortMem`	Port	depends on `size` parameter
`OTOpenEndpoint`	Client Shared Port	16 bytes 150 bytes 1 KB
`OTStreamOpen`	Shared Kernel	40 bytes 1 KB
`OTCreateConfiguration`	Shared	100 bytes [2]
`OTSnd`	Kernel	`nbytes` [3]

IMPORTANT:
These amount are approximate. The values vary depending on the relative complexity of the protocol being used and between versions of Open Transport. **These values are only meant as a guide for analyzing your program's memory needs.**

Notes:

1. `OTAllocMem` behaves differently depending on the libraries with which you link. If you link with the OT client libraries (for example, `OpenTransportLib`), `OTAllocMem` allocates memory from the client pool. If you link with the OT kernel libraries (for example, `OpenTptModuleLib`), `OTAllocMem` allocates memory from the kernel pool. The "InContext" OT routines, for example `OTAllocMemInContext`, help to eliminate this confusion. See DTS Technote 1173 Understanding Open Transport Asset Tracking for more details.
2. The exact size depends on the complexity of the configuration. This value is a lower bound, based on a simple call to `OTCreateConfiguration("serial")`.
3. This memory is consumed only if the endpoint is copying sent data (ack sends is off), which is the default setting. If no-copy sends are enabled, the routine allocates a much smaller amount of housekeeping memory.

Examining Memory Pools in MacsBug

The above analysis was done empirically, by calling each routine repeatedly while recording the effect on each memory pool. While OT provides no programming interface for measuring the usage of its memory pools, you can easily find the pools in MacsBug.

First, you will need to find the debug version of Open Transport -- available via links on the OT web page -- and extract the "OT Debugger Prefs" file, included as part of the install package.

```
Open Tpt Debug Installer
   Open Transport Installer
     Open Transport Files
       OT Debugger Prefs
```

You should copy the "OT Debugger Prefs" file to your MacsBug Preferences folder, then restart your machine.

> **IMPORTANT:**
> It is vital that you use the "OT Debugger Prefs" file from a debug install of OT whose version number matches the version of OT you have installed. The "OT Debugger Prefs" file contains MacsBug templates that are automatically generated by the OT build system to match the layout of the fields in the OT data structures. This layout changes from version to version of OT. If you have the wrong version of the "OT Debugger Prefs", you will not get accurate results in MacsBug.

Once you have the "OT Debugger Prefs" file installed, you can use it to find and display the various OT memory pools. The first step is to dump the OT globals. This is done differently on 68K and PowerPC, and is explained in the following sections.

Dumping OT Globals on PowerPC

On PowerPC, you can dump the OT globals using the following command:

```
>>> dm __gOTGlobal OTGlobal
  Displaying OTGlobal at 0006BDA0
   0006BDA0  fGestaltValue      0000003F
   0006BDA4  f68KDeferredProc   00000000
   0006BDA8  fVersion           01308000
   [... other stuff deleted ...]
                                0006BF04  fClientGlobal
   0006BF04    fClientList
   0006BF04      fHead          005F1714
   [... other stuff deleted ...]
                                0006BF30    fNativePool      00095320
   [... other stuff deleted ...]
                                0006BF8C  fKernelGlobal
   0006BF8C    fKernelPool      0039A4A0
   0006BF90    fKernelPoolMaxSize  #13421772
   [... other stuff deleted ...]
                                0006BFD4    fPortPool        0037AA90
   [... other stuff deleted ...]
```

OT exports the address of the OT globals as a CFM symbol, `__gOTGlobal`. The above command dumps that address using the `OTGlobal` template from the "OT Debugger Prefs" file. As far as memory usage is concerned, there are three fields of interest:

1. `fNativePool` -- This is the address of the shared client pool.
2. `fKernelPool` -- This is the address of the kernel pool.
3. `fClientList.fHead` -- This is the head of the OT client list. You can dump out the first client using the command:

```
>>> dm 5f1714 RegisteredClient
  Displaying RegisteredClient at 005F1714
   005F1714  fLink
   005F1714    fNext            005F15BC
   005F1718  fProviders
   005F1718    fHead            005F13D0
   005F171C  fStreams
   005F171C    fHead            00000000
   005F1720  fWhoAmI            070A7134
   [... other stuff deleted ...]
```

You can examine the next client by following the `fLink.fNext` field. Your application will be the one whose `fWhoAmI` field points into your application heap. One you've found your application, you can display its connection to ASLM by dumping its `fWhoAmI` pointer using the `TLibraryManager` template, as shown below:

```
>>> dm 70a7134 TLibraryManager
  Displaying TLibraryManager at 070A7134
   070A7134  __vptr             003873B0
   070A7138  fPool              070A6780
   070A713C  fLibraryFile       00000000
   070A7140  fDefaultPool       070A6780
   [... other stuff deleted ...]
```

The address of your client pool is held in the `fDefaultPool` field.

Given the address of a pool, you can do a number of things with it:

- The following MacsBug command will display some basic information about the pool:

```
>>> dm 70a6780 TMemoryPool
  Displaying TMemoryPool at 070A6780
   070A6780  __vptr             00386F40
   070A6784  fMemList           070A6770
   070A6788  fSize              #2408
   070A678C  fLowMark           #1797
   070A6790  fHighMark          #4294967295
   070A6794  fMaxUsed           #352
   070A6798  fCurFree           #2056
   070A679C  fZone              06F7CF00
   070A67A0  fMemType           #1
   [... other stuff deleted ...]
```

The `fSize` field is the total amount of memory in the pool. The `fCurFree` field is the amount of free memory left in the pool.

- The `dumppool` dcmd will display the list of memory blocks in the pool, for example:

```
>>> dumppool 70a6780
  Allocated Memory
   ----------------
     70a7000(  #16)   70a7010(  #16)   70a7020( #168)
     70a70c8(  #64) "!$plnt"
     70a7108(  #40) "!$slst"
     70a7130(  #48) "!$lmgr"
   Free Memory
   -----------
     70a67f8(#2056)
```

- The `dumprawpool` dcmd will display a more detailed list, for example:

```
>>> dumprawpool 70a6780
  Allocated Memory
   ----------------
   F:   70a67f8(#2056)
   A:   70a7000(  #16)   70a7010(  #16)   70a7020( #168)
   A:   70a70c8(  #64) "!$plnt"
   A:   70a7108(  #40) "!$slst"
   A:   70a7130(  #48) "!$lmgr"
```

Dumping OT Globals on 68K

On 68K, the procedure is slightly more complex. The first step is to find the address of the OT global. You do this using the following MacsBug command:

IMPORTANT:
For this to work you will need to install the debug version of OT so that MacsBug can find the `FetchOTGlobal` symbol.

```
>>> hx 2800
  The target heap is the System heap at 00002800
>>> il FetchOTGlobal
  Disassembling from FetchOTGlobal
   FetchOTGlobal
      +00000 0015D5E2   LINK       A6,#$0000        | 4E56 0000
      +00004 0015D5E6   MOVE.L     $00092434,D0       | 2039 0009 2434
      +0000A 0015D5EC   UNLK       A6             | 4E5E
      +0000C 0015D5EE   RTS                         | 4E75
      [... other stuff deleted ...]
```

The first command switches the current MacsBug target zone to the system heap. The next command disassembles a function that returns the address of the OT globals. The line at `FetchOTGlobal + 4` moves the address of the OT globals into register D0. In this case, the address of the OT globals is stored in memory location `$00092434`. You can dump the globals using the following MacsBug command:

```
>>> dm 92434^ OTGlobal
  Displaying OTGlobal at 000B5050
   000B5050  fGestaltValue      0000000F
   000B5054  f68KDeferredProc   00238164
   000B5058  fVersion           01306007
   [... other stuff deleted ...]
```

After dumping the OT globals, you can proceed as in the PowerPC case.

Controlling Client Pool Parameters

As described above, the OT client pool for an application is allocated in the application heap when you call `InitOpenTransport`. The pool starts very small and grows on demand. However, this behavior is not always optimal. Specifically, if you want to exclusively use the OT memory allocator in your application, you should dedicate your entire application heap to its use. Having the allocator consume your application heap piecemeal is much less efficient than giving it to them in one big chunk. Moreover, if you regularly allocate memory at 'interrupt time', you will find that the OT allocator often fails at inopportune times; if the client pool is exhausted OT can not grow it at interrupt time.

You can get more control over your client pool by

- using an OT memory reserve, or
- manipulating the pool directly via ASLM APIs.

These techniques are described in the following sections.

IMPORTANT:
Apple strongly recommends that you use the first technique. While OT currently uses the ASLM client pool as its client pool, this will not always be true. Furthermore, as OT moves away from ASLM in general, there's no guarantee that ASLM will be installed on the system.

Using an OT Memory Reserve

The most compatible way to use your entire application zone for OT memory allocations is to use an OT memory reserve. When your applications starts up you create the memory reserve by allocating a number of large chunks of memory. When you need memory, first try to allocate the memory from OT. If that fails, free one of the chunks in the memory reserve and try again.

This technique is implemented by the OTMemoryReserve module of the OTStreamLogViewer sample code (version 1.0.1b1 and later). You can easily borrow the code and reuse it in your application.

Note:
One approach that does not work is to preflight the OT memory pool by allocating a large amount of memory and then freeing it immediately. This does not work because the OT memory allocator is too smart; if freeing a block of memory creates a large unused chunk in the pool, OT will return that chunk back to the Mac OS Memory Manager.

Controlling Client Pool Parameters via ASLM

IMPORTANT:
If you use the technique described in this section your application will be dependent on ASLM, and on the fact that OT uses the ASLM client pool as its client pool. This is not recommended.

IMPORTANT:
If you use `InitOpenTransportInContext` (implemented by either Carbon or the OTClassicContext sample code), you can not use this technique to control your OT client pool parameters.

You can obtain more control over your client pool by using the ASLM programming interface. If you have already initialized a connection to ASLM, `InitOpenTransport` will use it (and its client pool) instead of creating its own. You can use this to control how large your client pool is, where it is allocated, and how it grows.

Note:
To program with ASLM, you need the ASLM SDK from the Mac OS SDK CDs.

IMPORTANT:
To call ASLM from 68K C or C++ code, you must be building with the 4-byte integers.

To use this technique, you must call `InitLibraryManager` before calling `InitOpenTransport`. In addition, you must call `CleanupLibraryManager` after calling `CloseOpenTransport`. The prototypes for these routines are defined in "LibraryManager.h", but they are given below for your convenience.

```
OSErr InitLibraryManager(size_t poolsize, int zoneType, int memType);
void  CleanupLibraryManager(void);
```

The additional parameters to `InitLibraryManager` allow you to specify the initial size for your client pool (in bytes), the location of the client pool (typically `kSystemZone`, `kApplicZone`, or `kCurrentZone`), and the type of memory for the client pool (typically `kNormalMemory`; however, if you access the memory when paging is unsafe, `kHoldMemory` may be useful).

The following code snippet demonstrates this technique. It first creates a subsidiary zone within the application heap (whose size is calculated to consume the entire heap, minus some memory for use by the toolbox). It then calls `InitLibraryManager` to connect to ASLM (and establish the client pool in the subsidiary zone) before calling `InitOpenTransport`.

```
// IMPORTANT:
// If you use this code you will take a hard dependency
// on the presence of ASLM. See the text for reasons why
// this is bad.

                            static OSStatus
InitOpenTransportWithMemoryLimit(void)
{
     OSStatus err;
     SInt32 junkTotalFree;
     SInt32 contigFree;
     SInt32 zoneSize;
     Ptr subsidiaryZone;
     THz oldZone;

     // First call the system Memory Manager to determine the largest
     // contiguous block in the heap.

     PurgeSpace(&junkTotalFree, &contigFree);

     zoneSize = contigFree - kBytesReservedForToolboxInApplicationZone;

     // Allocate the memory for our zone and create a zone in that
     // block.  Then init ASLM, telling it to create a pool that
     // takes up the entire zone (minus the ASLM overhead factor)
     // in the current zone, i.e., the zone we just created.  Finally,
     // initialize OT.  OT will see that we've inited ASLM and use
     // the pool that ASLM created (in the zone we created) for
     // satisfying OTAllocMem requests.

     subsidiaryZone = NewPtr(zoneSize);
     oldZone = GetZone();

     // InitZone sets the current zone to the newly created zone,
     // so I don't have to do it myself.

     InitZone(nil, 16, subsidiaryZone + zoneSize, subsidiaryZone);
     err = InitLibraryManager(zoneSize - 2048, kCurrentZone, kNormalMemory);
     if (err == noErr) {
         err = InitOpenTransport();
         if (err != noErr) {
             CleanupLibraryManager();
         }
     }
     SetZone(oldZone);

     return err;
}
```

IMPORTANT:
This code is a simplified version (less error checking) of the code used by an older version of the OTStreamLogViewer sample code. Current versions of OTStreamLogViewer have been updated to use an OT memory reserve, as described above.

Note:
The above technique is by no means the only one available to you using the ASLM API. You should read the *ASLM Developer's Guide* for more information.

Back to top

Advanced Topics

This section describes some of the more advanced issues in the realm of OT memory management. Specifically, the section describes how the OT shared client and kernel pools grow and shrink over time. Before tackling this, you need to learn about another API call you can make to alter the behavior of the OT memory system.

Note:
This section of the note is intended for those with an intimate knowledge of the Open Transport architecture. Do not be alarmed if you do not understand it!

OTSetMemoryLimits

The `OTSetMemoryLimits` routine allows software to directly affect the behavior of the OT memory pools. The prototype for the routine is:

```
#ifdef __cplusplus
extern "C" {
#endif

extern OSStatus OTSetMemoryLimits(size_t growSize, size_t maxSize);

#ifdef __cplusplus
}
#endif
```

The `growSize` parameter is the amount by which OT should grow the kernel pool right now. When you call the routine, OT immediately tries to grow the kernel pool by this amount. The `maxSize` parameter is the new maximum size of the kernel pool. OT will never grow the kernel pool larger than this amount.

`OTSetMemoryLimits` also implicitly sets an internal Open Transport variable called `fServerMode`. If you call `OTSetMemoryLimits` with a positive `growSize` value, `fServerMode` is incremented. If you call it with a zero value, `fServerMode` is decremented. If `fServerMode` is non-zero, OT will never downsize the shared client or kernel pools. To be a good citizen, server software should call `OTSetMemoryLimits` with a positive `growSize` when it starts up, and a zero `growSize` when it shuts down.

Finally, if you grow the kernel pool by more than 20 KB, `OTSetMemoryLimits` will also grow the shared client pool by 10% of the `growSize` value.

OTSetMemoryLimits is only of use to server software which must handle extremely 'bursty' connection patterns or many connections in parallel. By increasing the maximum size of the kernel pool, the server can handle more connections in parallel. By growing the kernel pool immediately (rather than as each connection is created), the server can handle these parallel connections as soon as it's started, rather than waiting for the kernel pool to grow through usage. By never downsizing the kernel pool, the server can handle many connections simultaneously even after a long period of inactivity.

OTSetMemoryLimits must be called at system task time and returns an error result if it can't grow the kernel pool by the specified amount.

IMPORTANT:
Using OTSetMemoryLimits will never increase the performance of a single connection. It is only useful for software with dozens of parallel connections. DTS strongly recommends that client software never call OTSetMemoryLimits.

Note:
The OTSetMemoryLimits routine does not appear in any Open Transport header files. If you use the routine, you must declare the prototype yourself. This is a consequence of the above policy -- general application programs should not call this routine.

Note:
There is an earlier incarnation of OTSetMemoryLimits, called OTSetServerMode. This routine has been completely subsumed by OTSetMemoryLimits.

Growing OT Memory Pools

When OT attempts to grow a pool, it uses a binary back-off algorithm to do so. It starts by attempting to grow the pool by getting one big block of memory from the Mac OS Memory Manager. If a block of that size is not available, it halves the size requested and tries again. This process terminates when either OT has grown the pool the requested amount, or the block size shrinks below 10 KB.

Shrinking OT Memory Pools

OT memory pools have the ability to shrink. A memory pool is made up of a number of discontiguous memory blocks that have been allocated from the Mac OS Memory Manager. When a pool is **downsized**, each Mac OS memory block is examined to see if it is empty. If it is, that memory block is released back to the Mac OS Memory Manager.

OT memory pools are downsized at the following points:

- Whenever a client dies (it calls CloseOpenTransport or an application quits without calling CloseOpenTransport) **and** OT is not in server mode, OT downsizes the shared client and kernel pools.
- Whenever OT unloads the client libraries, it downsizes the shared client pool.
- Whenever OT unloads the kernel (which happens when there are no remaining clients who called InitOpenTransport), it downsizes the kernel pool if OT is not in server mode.
- Whenever OT unloads the kernel utilities library (which happens when there are no remaining clients who called InitOpenTransportUtilities), OT downsizes the port pool.
- OT downsizes the port pool after it runs port scanners.
- OT downsizes the shared client pool immediately after it runs through the list of configurators calling their OTSetupConfigurator or OTStartupConfigurator entry points.

More Kernel Pool Trivia

OT maintains a hard limit on the upper bound of the size of the kernel pool. Clients can set this limit using the OTSetMemoryLimits routine. This poses the question: What is the initial value for this limit? Out of the box, OT sets

this limit to 10% of the physical memory on the machine (as returned by `gestaltPhysicalRAMSize`). This strikes a balance between providing enough buffer space for networking while preventing OT from consuming all the user's memory.

The initial size of the kernel pool is specified by two parameters. The first parameter is the required initial pool size (described above). If OT cannot allocate a kernel pool of this size it will fail to load. Additionally, OT will try to grow the kernel pool to at least 96K every time it loads the kernel (including the first time); however, it does not require that this memory be available for the kernel to load. This mechanism allows the kernel pool to be small while the kernel is unloaded, but grow quickly when the kernel loads.

Back to top

Summary

Open Transport provides a reliable, flexible, and interrupt-safe memory management system. By understanding how it works, you can avoid some common pitfalls and still write code that allocates memory at interrupt time. *Finally.*

Back to top

References

Inside Macintosh: Networking With Open Transport

Inside Macintosh: Memory

Apple Shared Library Manager Developer's Guide

DTS Technote 1173 Understanding Open Transport Asset Tracking

OTStreamLogViewer sample code

OTClassicContext sample code

Open Transport Web Page

Back to top

Change History

11-May-1998	First released.
09-Jan-2001]	Updated to warn about the dangers of depending on ASLM and to offer an alternative. Also updated the pool parameters section to account for OT releases since 1998.

Back to top

Downloadables

Acrobat version of this Note (K). Download

Back to top