



CHAPTER 11

SQL Server Memory Management

Accustom a people to believe that priests, or any other class of men, can forgive sins, and you will have sins in abundance.

—Thomas Paine¹

In this chapter, we'll explore SQL Server's memory management architecture. The way that an application manages critical resources such as memory tells us a lot about how it is designed. It tells us what priority the application designers placed on efficient resource utilization and on maximizing the performance of the application. As you will see in the discussion that follows, efficient memory management and maximum system performance were both of paramount importance to the designers of SQL Server. A considerable portion of the complex code within the product is dedicated to managing memory efficiently and effectively. There are always trade-offs with memory management. Too little memory usage makes your app efficient but slow. Too much memory usage may make your app fast, but it may not play well with others and it may become a resource hog in general. As you'll see, SQL Server attempts to strike a balance between getting the most out of the resources available to it and running well alongside other applications on the system.

1. Paine, Thomas. "Worship and Church Bells." In *The Complete Writings of Thomas Paine, Vol. II*, ed. Philip S. Foner. New York: Citadel Press, 1945, p. 726.



Memory Regions

SQL Server organizes the memory it allocates into two distinct regions: the BPool (buffer pool) and MemToLeave (memory to leave) regions. If you make use of AWE memory, there's actually a third region: the physical memory above 3GB made available by Windows' AWE support. (Refer to Chapter 4 for details on AWE.)

The BPool is the preeminent region of the three. It is SQL Server's primary allocation pool. MemToLeave consists of the virtual memory space within the user mode address space that is not used by the BPool. The AWE memory above 3GB functions as an extension of the BPool and provides additional space for caching data and index pages.

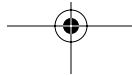
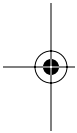
Sizing

When the server starts, it begins by computing the upper limit of the BPool. This upper limit is the maximum size to which the server will allow the BPool to grow. On a non-AWE system, this size will be set equal to the amount of physical memory in the machine or to the size of the user mode address space minus the size of the MemToLeave region, whichever is less. (If the `sp_configure` max server memory setting has been changed from the default and is less than or equal to the amount of physical memory in the machine, it will override this computation.) So, if the system has 1GB of physical memory installed, the BPool will be sized to 1GB, provided max server memory has not been adjusted.

On an AWE system, the BPool upper limit will be set to either the size of the total physical memory in the machine or to the max server memory setting, whichever is less. When AWE is used, the BPool isn't constrained by the size of the user mode address space or the size of the MemToLeave region.

By default, the MemToLeave region is sized at 384MB. Of this, 128MB is reserved for worker thread stacks (max worker threads = $255 \times .5\text{MB}$ for each thread stack), and 256MB is reserved for allocations outside of the BPool. Examples of the types of memory allocations that come from MemToLeave include OLE DB provider allocations, in-process COM object allocations, and *any memory allocation by the server code itself that is larger than 8KB*. This last item is important because it means that large procedure or execution plans can be allocated from the MemToLeave region.

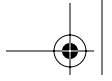
Although all allocations for a contiguous memory block larger than 8KB come from the MemToLeave region, the reverse is not always true. Because



the server will attempt to use the uncommitted portion of a reserved block for other allocation requests, it's possible that allocations smaller than 8KB could end up coming from the MemToLeave region, depending on the version and service pack of SQL Server you're using. In order to make an allocation request, a memory consumer within the server first allocates a memory allocation object, which it then uses to request the allocation (this object implements the COM IMalloc interface, as we'll discuss later in the chapter). If multiple requests are made via a given allocation object, it's possible that some of them will be fulfilled using MemToLeave memory, even if they request less than 8KB. For example, if a consumer within the server requests 10KB of contiguous storage, the allocation object will allocate this from the MemToLeave region. If necessary, it will first reserve a region sized to match the system's allocation granularity (64KB on 32-bit Windows), then commit two 8KB pages to satisfy the request. If the memory consumer then uses the same allocation object to request, say, 4KB of additional space, the system will see the unallocated 6KB of space at the end of the two-page region it just allocated and will fulfill the new request from this space. Thus, it's possible for an allocation that's less than 8KB in size to be satisfied outside of the BPool.

Because OLE DB providers, COM objects, and other external consumers that may reside within the SQL Server process will know nothing of SQL Server's BPool or its memory management facilities, it's essential that the server leave some amount of virtual memory free within the user mode space. That's why MemToLeave exists—it is basically unused memory within the SQL Server process space. If an in-process COM object or other external consumer calls `VirtualAlloc` or `HeapAlloc` itself, it will require virtual memory address space in order to satisfy the allocation request. If the BPool were to take up all of this user mode address space itself, allocation requests of this type would always fail.

The size of the MemToLeave region can be adjusted using the `-g` command line parameter. The total memory required for the worker thread stacks cannot be changed without changing the max worker threads `sp_configure` value or modifying the default thread stack size by hacking the SQL Server executable, which you should definitely not do. However, you can increase or decrease the 256MB set aside for allocations outside of the BPool by passing a different value for the `-g` parameter. This parameter can be handy in situations where you have a lot of linked server queries, in-process COM objects, or other memory consumers contending for space in the MemToLeave region. By making it larger, you give them a bigger sandbox in which to play. Conversely, shrinking it can provide more virtual memory space for the BPool and may improve performance in some situations.



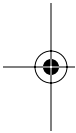
438 Chapter 11 SQL Server Memory Management

Even though the BPool can be sized based on the amount of physical memory in the machine, it is still based entirely in virtual memory, by default. The exception to this is when you make use of AWE memory. In that situation, part of the BPool is in virtual memory (but backed by pages locked in physical memory), and the rest is in physical memory above 3GB. AWE is the only way for a user mode process to access more than 3GB of memory. Since the maximum size of the user mode virtual address space is 3GB (even with /3GB or /USERVA enabled), AWE is the only means by which a process can access memory above the 3GB boundary. It just so happens that AWE memory is physical rather than virtual memory and must be mapped into the user mode space in order to be accessed.

The BPool

The vast majority of SQL Server's memory allocations come from the BPool. The BPool consists of up to 32 separate memory regions organized into 8KB pages. You'll recall from Chapter 4 our discussion of VirtualAlloc and the way it can be used to reserve contiguous blocks of memory. After the server has computed the maximum size of the BPool, it reserves the MemToLeave region in an attempt to ensure that this region will be a contiguous address range. If possible, it will reserve this using a single call to VirtualAlloc. If that's not possible (highly unlikely), the server will make multiple calls to VirtualAlloc to reserve the MemToLeave region. The server then attempts to reserve the BPool region from the user mode address space. (Put aside how AWE affects this for now; we'll return to it in just a moment.) With DLLs, memory-mapped files, and other allocations already in the user mode space, it's very unlikely that the entire BPool will be able to be reserved with a single call to VirtualAlloc. Instead, the BPool will likely have to consist of several fragments spread across the user mode space. The server will call VirtualAlloc repeatedly, each time with a smaller reservation request size until it succeeds. It will do this up to 32 times in order to reserve as much of the BPool's maximum size as possible. Once this completes, the server releases the MemToLeave region so that it will be available to external consumers. Because it was reserved before the BPool reservations were made and then released afterward, the MemToLeave region normally starts out as a single, contiguous block of free virtual address space.

When AWE is involved, the algorithm is slightly different. The repetitive calls to VirtualAlloc for the user mode portion of the BPool still occur. However, Windows does not support reserving and committing AWE mem-



ory using separate operations. As we discussed in Chapter 4, when using AWE, either the memory is allocated or it isn't. `AllocateUserPhysicalPages` does not support the concept of reserving, but not allocating, physical memory—memory is reserved and committed in one fell swoop. Once it is, it must be mapped into a window in the user mode space so that a 32-bit pointer can access it.

So, in this scenario, all of the memory set aside for the BPool is locked into physical memory. This applies to both the user mode portion as well as the AWE portion. Because physical memory is being locked, this can cause serious performance problems for other applications running on the same machine, including other instances of SQL Server. You typically set up a SQL Server this way when it is the only significant app running on a machine.

When AWE support is enabled in SQL Server, the user account under which the server is running must have the lock pages in memory privilege. SQL Server's setup program will automatically grant this privilege to the startup account you choose for the server. If you start the server from the command line or change the startup account, you have to take care of this yourself.

Hashing

In order to make locating particular pages faster, SQL Server hashes pages within the BPool. This basically amounts to creating a hash table over the pages in the BPool such that, given the database ID, file number, and page number of a data page, the server can quickly determine whether the BPool contains the data page and where it is located if it does.

When the server needs to access a particular data page, it hashes the page's database ID, file number, and page number such that the page maps to a particular bucket within the hash table. That bucket consists of a linked list of pointers to BPool pages. It is checked to see whether the page in question is on the list. If it is, it can be quickly accessed in memory. If it is not, it must first be loaded from disk.

Primitive Allocations

Before any pages can be allocated from the BPool, SQL Server must allocate the support structures required to manage it. The first of these that we'll talk about is a global variable to hold a reference to an instance of the class that defines the BPool. Because this variable has global scope, you can see it



yourself using WinDbg and the public symbols that ship with SQL Server. Exercise 11.1 takes you through locating both the global variable and its host data type.

Exercise 11.1 Using WinDbg to Find the Buffer Pool

1. Attach to your nonproduction SQL Server with WinDbg.
2. Make sure your symbol path is set correctly as described in Chapter 2.
3. Our next step will be based on two assumptions.
 - a. Due to its very nature and ubiquity within the server, the reference to the BPool is likely stored in a global variable or similar construct.
 - b. It is likely named BPool, BufferPool, or some variation thereof.
4. At the WinDbg command prompt, type:

```
.reload -f sqlservr.exe  
x sqlservr!*
```

This will list all of the public symbols included in sqlservr.pdb, the program database (symbol) file for SQL Server.

5. Scroll to the top of the command window and click above the start of the output from the x command. Press Ctrl+F, type BPool, and press Enter.
6. You should find the first occurrence of a reference to the BPool class. This is actually a reference to one of its methods. We can deduce from this that SQL Server has a class named BPool. It's a fair guess that this is the data type of the object that stores the SQL Server buffer pool, but we'll establish that with a good degree of certainty in just a moment.
7. Now let's look for the global variable that we suspect stores the reference to the buffer pool. Scroll to the top of the output and repeat your search, this time specifying "bPool" for the search string (no quotes). Be sure to specify a case-sensitive search in the dialog. Because C++ is a case-sensitive language, it's common for developers to name an instance of a variable after its type using different case. We'll start by checking for that.
8. If your search was case-sensitive, it should not have found any symbols named "bPool," so we need to keep looking. Another common tactic is to spell out a type name but abbreviate the names of instances of it or vice versa. Since we already know the type name is BPool, let's look for it spelled out as BufferPool. Repeat your search, this time specifying BufferPool as the search criteria.
9. You should find a symbol named sqlserver!BufferPool. Notice that this isn't prefixed by a class name and a pair of colons (::) as the BPool ref-

erence you found earlier was. This means that it's a global of some type. Based on the name alone, we can deduce that it's probably not a global function. Thus, it's likely a global variable and probably the one that stores the reference to the SQL Server buffer pool.

10. At this point, you could dump the contents of the BufferPool variable to the console using the `dd` command or something similar. The value of the variable itself isn't terribly useful to us at this point; I just wanted you to see that it was indeed a global. All of SQL Server's BPool functionality is wrapped up in the BPool class and in BufferPool, the single, global instance of it.
11. Scroll back to the top of the command window and repeat your search, this time using "DropCleanBuffers" as your search string. You should find an entry for BPool::DropCleanBuffers in the symbol list.
12. Readers of my previous books will recall the discussions of the DBCC DROPCLEANBUFFERS command. This command was once undocumented but is handy for releasing the clean buffers from the BPool in order to test a query with a cold cache without cycling SQL Server. Given that the BPool class has a member named DropCleanBuffers, might this be what the DBCC command calls? Let's set a breakpoint and find out.
13. At the WinDbg command prompt, type:

```
bp sqlserver!BPool::DropCleanBuffers  
g
```

14. Now, switch over to Query Analyzer, connect to your server, and run DBCC DROPCLEANBUFFERS in the editor. Switch back to WinDbg. You should see that execution has stopped at your breakpoint. This tells us definitively that DBCC DROPCLEANBUFFERS is implemented via a method off of the BPool class named DropCleanBuffers. It also reinforces our assertion that the BPool class we see in the debugger is the actual data type of SQL Server's global buffer pool instance.
15. Type `q` in the WinDbg command window and press Enter to stop debugging. You'll need to restart your SQL Server.

Page Arrays

I mentioned earlier that SQL Server makes up to 32 separate memory reservations to reserve the BPool. The BPool tracks these allocations in two parallel arrays—one array stores a list of pointers to the start of each region, the other stores a count of the 8K pages reserved in the region. Both arrays are private members of the BPool class.



BUF Array

SQL Server uses a special BUF structure to manage each page in the BPool. Before reserving the BPool, SQL Server calls VirtualAlloc to allocate an array of BUF structures from the MemToLeave region equal in size to the number of pages it will reserve for the BPool (including physical AWE pages). Each page in the BPool will have a corresponding BUF structure, as will each page of AWE memory allocated by the server, regardless of whether it has been mapped into virtual memory. Since each BUF structure is 64 bytes in size, this array isn't usually very large unless the server is using a significant amount of AWE memory.

Each page's BUF structure functions as a type of header for it. It stores information such as a pointer to the actual page in the BPool, the reference count for the page, the page's latch, and status bits that indicate whether the page is dirty, has I/O pending, is pinned in memory, and so on.

When the lazywriter traverses the pool looking for pages to free, this array of BUF structures is what it actually sweeps. We'll discuss the lazywriter further in just a moment.

Commit Bitmap

On startup, SQL Server allocates a bitmap from the default process heap that it uses to track committed pages in the BPool. The original reservations tracked by the page arrays are just that—reservations. As we discussed in Chapter 4, it's possible to reserve virtual memory address space without committing any physical storage to it. As each page in the BPool is committed, its corresponding bit in the commit bitmap is set.

AWE

Although the BPool keeps track of which pages in AWE memory it makes use of, it cannot access them directly due to the way that Windows' AWE facility works. It accesses them by mapping physical pages in and out of the user mode address space, as I mentioned in Chapter 4.

Note that if your server has less than 3GB of physical RAM and you enable AWE use via the `sp_configure awe enabled` option, SQL Server will ignore it. You must have 3GB or more of physical memory in order to use Windows' AWE facility with SQL Server, as we discussed earlier in the book.

The Lazywriter

The purpose of the lazywriter is twofold: (1) to keep a specified number of BPool buffers free so they can be allocated for use by the server and (2) to monitor and adjust the committed memory usage by the BPool so that enough physical memory remains free on the system to prevent Windows from paging (provided dynamic memory management is enabled so that the lazywriter can adjust the size of the BPool as necessary). SQL Server estimates the number of BPool buffers to keep free based on the system load and the number of stalls occurring (the number of times a memory consumer within the server has to wait on a free buffer page).

The amount of physical memory the lazywriter attempts to keep free usually varies between 4MB and 10MB. It is partially based on the computed page life expectancy for the pool (the number of seconds a page will stay in the buffer pool without being referenced). You can track the Buffer Manager:Page life expectancy Perfmon counter to see what this value is for a particular instance of SQL Server. As the page life expectancy increases (i.e., as the instance of SQL Server is less memory-pressured and, therefore, pages stay in the cache longer even when not being referenced), the amount of physical memory reserved for the OS climbs nearer to 10MB. As the page life expectancy decreases, the physical memory set aside for the OS continues to fall until it reaches approximately 4MB.

Keeping a minimum amount of physical memory available for use by the OS helps ensure that it doesn't page unnecessarily and helps keep it and the other processes on the SQL Server machine running smoothly. If the system begins to be pressured for physical memory—even if that pressure is not coming from SQL Server—the BPool will adjust its committed BPool memory downward so that more physical memory is available.

Computing Physical Memory

The manner in which the BPool determines the amount of available physical memory varies based on the OS. On Windows 2000, the Win32 API function `GlobalMemoryStatusEx` is called. If you're running SQL Server on Windows 2000, you can see this for yourself by attaching to SQL Server with WinDbg and setting a breakpoint in `kernel32!GlobalMemoryStatusEx`. If you list the call stack once your breakpoint is tripped, you'll see that `GlobalMemoryStatusEx` is being called by either `BPool::AvailablePagingFile` or `BPool::AvailablePhysicalMemory`. Obviously, the latter one is the call the



BPool is using to ensure that physical memory stays at or above the required threshold.

On Windows NT 4.0 and Windows 9x/ME, the Win32 API function `GlobalMemoryStatus` is called. This function is used rather than `GlobalMemoryStatusEx` because `GlobalMemoryStatusEx` is not supported on Windows NT 4.0 or Windows 9x/ME.

On Windows XP and Windows Server 2003, the BPool uses the `CreateMemoryResourceNotification` and `QueryMemoryResourceNotification` API functions to instruct Windows to notify the BPool when physical memory runs low. These APIs are not available on earlier versions of Windows, so their use is exclusive to Windows XP and Windows Server 2003.

Flushing and Freeing Pages

As I mentioned earlier, the BUF array contains an element for each page in the BPool. Each BUF structure functions as a kind of BPool page header and contains a reference count for its corresponding page. Each time a page is referenced, this reference count is incremented. Some more expensive pages—such as those that contain execution plans—begin with a higher reference count than other kinds of pages. This helps keep them in memory longer and helps the server avoid incurring the cost of recreating them or reloading them from disk unnecessarily.

Periodically, this BUF array is scanned. The reference count in each BUF structure is divided by four and the remainder is discarded. When the reference count for a page reaches zero, the page is checked to see whether it's dirty; if so, a write is scheduled via UMS to flush the dirty page to disk. (If a page's reference count reaches zero and the page is *not* dirty, it is simply freed—i.e., moved to the free list—without writing anything to disk.) Because SQL Server uses a write-ahead log, this flush to disk of the dirty page will be blocked while the transaction log is written to. Once a dirty page has been successfully written to disk, it is unhashed (removed from the hash table) and added to the free list. A dirty page is not usually decommitted in this scenario—it is merely moved on to a list of free buffers so that it can be reused. Avoiding unnecessary decommit/commit operations speeds up SQL Server's memory operations significantly and is a key design tenet of an effective memory cache.

The size of the free list is calculated internally by SQL Server based on the size of the BPool. The fact that SQL Server uses separate physical structures for a page and its header allows a page to be flushed to disk and to “move” from list to list without anything on the page actually changing.

When a dirty page is flushed to disk and freed, for example, information in the page's header (its BUF structure) changes, but the page itself does not have to be modified. Once flushed, its contents are considered disposable and can be overwritten when the page is reused. When a page needs to move from one list to another (e.g., when it's freed), the much smaller 64-byte BUF structure is moved from list to list rather than the 8KB page. The reason the size of the structure matters here has to do with the use of the processor's local cache. The smaller the structure, the more instances of it can be stored in the chip's onboard cache, and the more efficient the process of moving nodes between lists is. Leveraging the local cache on the CPU is another key tenet of an effective memory cache.

On the Windows NT family, this process of aging pages, flushing them to disk, and moving them to the free list is usually done by individual UMS workers. The dedicated lazywriter thread usually finds that it has very little to do. On Windows 9x and ME, though, the lazywriter thread plays a much bigger role. Because Windows 9x and ME do not support asynchronous file I/O, UMS is forced to perform all file I/O synchronously (see Chapter 10). This limits the ability of a UMS worker to perform lazywriter work; therefore, the dedicated lazywriter thread is much more active on Windows 9x/ME than it is on the Windows NT family.

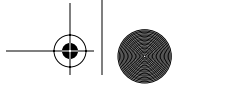
The lazywriter checks the free buffer count and the available physical memory once per second or when signaled. You can see this for yourself by attaching to SQL Server with WinDbg and setting a breakpoint in the BPool::AvailablePhysicalMemory method that we discovered earlier, as shown below. (Note that due to page width limitations this code appears printed on two lines, but you must type it all together as one line.)

```
bp sqlservr!BPool::AvailablePhysicalMemory  
".echo checking availphys; g"
```

Once you restart WinDbg, you'll see the message following .echo displayed once a second until you stop the debugger.

I should point out here that, until the BPool upper limit size is reached, the lazywriter doesn't free up buffers by flushing them to disk and putting them on the free list. Instead, it merely commits more reserved pages in the BPool each time the free list falls below the built-in threshold and changes the corresponding bits in the commit bitmap accordingly.

The lazywriter checks 16 BUF structures at a time. It keeps track of where it left off with each iteration and picks back up there on the next run (a second later or when signaled, as I've said). When the lazywriter reaches the end of the BUF array, it wraps back around to the start in a manner similar to



a clock—it sweeps continuously through the BUF array not unlike the way the hands on a clock sweep indefinitely.

Checkpoint

The checkpoint process also scans the BPool and flushes dirty pages to disk. It does not, however, move pages to the free list. Freeing pages is the job of the UMS workers and the dedicated lazywriter thread. Checkpoint's job is to shorten the amount of time required to recover the system by keeping the number of dirty pages to a minimum. It's common for a checkpoint not to find many pages to flush to disk because dirty pages have already been flushed by the lazywriter thread or individual UMS workers.

Partitions

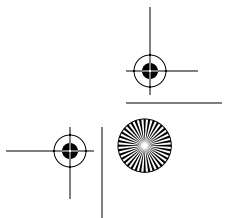
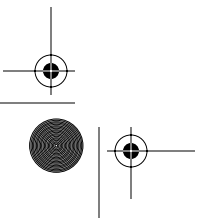
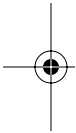
In order to provide for better scalability, SQL Server partitions the BPool free list by CPU. As we discussed in Chapter 10, a given UMS user is assigned to a particular scheduler when it first connects and remains with that scheduler until it disconnects. Each UMS scheduler is typically associated with a particular CPU. When a UMS user needs a free page, the partition associated with its CPU is checked first, followed by the other partitions if necessary. By partitioning the BPool free list by CPU, we make better use of the processor's local cache and improve the scalability of the server.

The Memory Managers

Rather than embed all of the complexities of memory management into the BPool itself, SQL Server spreads it across five major memory manager classes. This allows different memory consumers to allocate and manage memory independently of one another while still using a shared, managed memory pool.

Understand that a handful of allocations bypass these managers altogether. The transaction log implements its own caching mechanism, as does the backup/restore facility. There are other examples as well. Rather than go through a memory manager, these allocations come straight from the OS via calls to VirtualAlloc.

The five managers are the Connection, Query Plan, Optimizer, Utility, and General memory managers. Each is responsible for a different class of memory management within the server. Let's discuss each of these separately.



Connection

This memory manager is responsible for connection-related memory allocations. Each connection has a process status structure (PSS) and SRVPROC structure allocated for it, as well as one network send buffer and two network receive buffers. The Connection memory manager manages these types of allocations.

Query Plan

SQL Server uses the Query Plan memory manager mainly for allocations related to compiled plans and execution plans produced by the query optimizer (you can see these in the syscacheobjects system table). This memory manager also handles allocations related to cursors, RPC parameters, index creation, and certain DBCC commands that deal with indexes on computed columns.

Note that the Query Plan memory manager is the only memory manager that the lazywriter can cause to free up memory. A page whose BUF header has a status bit indicating that the page is related to a query plan can be aged out of the cache by the lazywriter when its reference count reaches zero.

Optimizer

SQL Server uses the Optimizer memory manager to allocate and manage metadata and tree structures related to query optimization. The server limits this manager to 80% of the server's total memory.

Utility

SQL Server has a special memory manager for use by utility functions. As the name suggests, it uses the Utility memory manager for various utility-related allocations, including buffers used by the server-side trace facility, log manager initialization, cluster- and log shipping-related functions, bitmap comparisons, hash table searches, and so on.

General

This memory manager is for allocations that don't fit any of the above categories. Allocations for several different types of internal storage engine structures, including locks, are made using the General memory manager.



OS Memory Allocations

As a rule, the five major memory managers attempt to allocate memory from the BPool unless the request is larger than 8KB. For allocations larger than 8KB of contiguous memory, the request is usually passed on to the memory manager responsible for allocating memory from the MemToLeave region. This memory manager is commonly known as the OS or Reserved memory manager. It is not a separate manager class in the same sense as the five major ones we just discussed; instead, it provides functionality that all of them can make use of for large allocations.

When it processes an allocation request, the OS memory manager calls VirtualAlloc to reserve a sufficiently large region of the MemToLeave pool. This reservation will be rounded up to the system's allocation granularity (64KB on 32-bit Windows), as we discussed in Chapter 4. Within this reservation, the memory manager will commit a sufficient number of pages to satisfy the request. If additional allocation requests are received before the region is released, they may also be fulfilled by committing pages in this region. Once all pages have been decommitted, the memory manager calls VirtualFree to release the region.

The Low Memory Manager

SQL Server provides a special emergency condition memory manager that is used in rare circumstances when the normal memory managers have failed and the server must have some memory in order to continue without severe consequences (e.g., corruption) during such things as logging or recovery. The normal managers do not automatically use this memory manager when allocations they attempt fail; it is used only by certain very critical code paths within the server.

When SQL Server starts, this memory manager commits a 64KB region of virtual memory. Only one consumer can use this memory at a time. Each new consumer must wait until the others have released the memory before the new consumer can use it.

When this memory manager is used, it writes a warning message to the SQL Server error log:

```
Warning: Due to low virtual memory, special reserved memory used
%d times since startup. Increase virtual memory on server.
```

Due to its very nature, you'll normally see other errors or warnings accompanying this message when it occurs.

IMalloc

Internally, SQL Server uses a custom implementation of the COM IMalloc interface to handle individual memory allocations. As I mentioned earlier in the chapter, when a consumer needs to make a memory request, it first allocates a memory allocation object. It then uses this allocation object to make the request. Multiple requests can be made via a single allocation object. This allocation object implements the standard COM IMalloc interface.

IMalloc includes all the features you'd expect to find in a standard memory allocator: allocation, deallocation, resizing, functions to determine allocation block sizes, and so on. COM provides a standard implementation of this interface that uses the Win32 heap facilities to carry out allocation requests. Since SQL Server performs its own memory management, it doesn't use the standard COM implementation—it uses an internal implementation that allocates requests from the BPool or MemToLeave regions as appropriate.

Pulling It All Together

Thus far, we've explored the individual components of SQL Server's memory architecture in some detail. In this section, we'll pull these elements together so you can better understand how SQL Server manages memory overall and why it behaves the way it does in certain circumstances.

When you start SQL Server, the BPool's upper limit is computed based on the physical memory in the machine, the max server memory `sp_configure` value, and the size of the MemToLeave region. Once this size is computed, the MemToLeave region is set aside (reserved) so that it will not be fragmented by the BPool reservations that are to follow. The BPool region is then set aside, using as many as 32 separate reservations in order to allocate around the DLLs and other allocations that may already be taking up virtual address space within the SQL Server process by the time the BPool is reserved.

Once the BPool is reserved, the MemToLeave region is released. SQL Server does not hold on to the MemToLeave region because this region is intended for "external" (i.e., outside the core SQL Server code) consumption. It is used for internal SQL Server allocations that exceed 8KB of contiguous space and for allocations made by external consumers such as OLE DB providers, in-process COM objects, and the like. As I've said, SQL Server reserves the entire MemToLeave region at startup, then releases it after the BPool reservations are made in order to keep it from being fragmented by BPool reservations.



450 Chapter 11 SQL Server Memory Management

So, once the server has started, the BPool has been reserved, but not committed, and the MemToLeave region is essentially free space within the virtual memory address space of the process. If you view the Virtual Bytes Perfmon counter for the SQL Server process just after SQL Server has started, you'll see that it reflects the BPool reservation. I've seen people become alarmed because this number is often so high—after all, it usually reflects either the total physical memory in the machine or the maximum user mode address space minus the size of the MemToLeave region. This is nothing to worry about, however, because it is only reserved, not committed, space. As I said in Chapter 4, reserved space is just address space—it does not have physical storage behind it until it is committed.

Over time, the amount of memory committed to the BPool will increase until it reaches the upper limit computed when the server was originally started. You can track this via the SQL Server:Buffer Manager\Target Pages Perfmon counter. As different parts of the server need memory, the BPool commits the 8KB pages it originally reserved until this committed size reaches the computed target. Since this is virtual space as opposed to physical space, and since the majority of virtual memory is backed by the system paging file, not by physical RAM, this does not necessarily equate to more physical memory usage. You can track the BPool's use of committed virtual memory via the SQL Server:Buffer Manager\Total Pages Perfmon counter. You can track the server's overall use of committed virtual memory via the Private Bytes counter for the SQL Server process.

Because most of SQL Server's virtual memory usage comes from the BPool, these two counters will, generally speaking, increase or level off in tandem. If the Total Pages counter levels off but the Private Bytes counter continues to climb, this usually indicates continued allocations from the MemToLeave region. These allocations could be completely normal—for example, they could be allocations related to thread stacks as additional worker threads are created within the server—or they could indicate a leak by an external consumer such as an in-process COM object or an xproc. If the process runs out of virtual memory address space because the MemToLeave region is exhausted due to a leak or overconsumption (or if the maximum free block within the MemToLeave region falls below the default thread stack size of .5MB), the server will be unable to create new worker threads, even if the sp_configure max worker threads value has not been reached. In this situation, if the server needs to create a new worker thread in order to carry out a work request—for example, to process a new connection request—this work request will be delayed until the server can create the thread or another worker becomes available for it to use. This can pre-

vent a user from connecting to the server because the connection may time out before a sufficient amount of MemToLeave space is freed or another worker becomes available to process the connection request.

A memory consumer within the server initiates a memory allocation by first creating a memory object to manage the request. This memory object is an implementation of the standard COM IMalloc interface. When the object allocates the request, it calls on the appropriate memory manager within the server to fulfill the request from either the BPool or the MemToLeave region. For requests of 8KB or less, the request is usually filled using memory from the BPool. For requests of more than 8KB of contiguous space, the request is usually filled using memory from the MemToLeave region. Because a single memory object may be used to carry out multiple allocations, it's actually possible for an allocation of less than 8KB to be allocated from the MemToLeave region, as I mentioned earlier.

Consumers of memory within the SQL Server process space are usually internal consumers, that is, consumers or objects within the SQL Server code itself that need memory to carry out a task, but they do not have to be. They can also be external consumers, as I've said. External consumers include OLE DB providers, xprocs, in-process COM objects, and so on. Usually, these external consumers use normal Win32 memory API functions to allocate and manage memory and, therefore, allocate space from the MemToLeave space since it is the only region within the SQL Server process that appears to be available. However, xprocs are a special exception. When an xproc calls the ODS `srv_alloc` API function, it is treated just like any other consumer within the server. Generally speaking, `srv_alloc` requests for 8KB of memory or less are allocated from the BPool. Larger allocations come from the MemToLeave space.

As the server runs, the lazywriter checks to make sure that a given amount of physical memory remains available on the server so that Windows and other apps on the server continue to run smoothly. This amount can vary between 4MB and 10MB (it trends closer to 10MB on Windows Server 2003) and is based on the system load and the page life expectancy for the BPool. If the physical memory on the server begins to dip below this threshold, the server decommits BPool pages in order to shrink its physical storage usage (assuming that dynamic memory configuration is enabled).

The lazywriter also ensures that a given number of pages remain free at any given point in time so that, as new allocation requests come in, they will not have to wait for memory to be allocated. By "free" I mean that the page is committed but not used. Unused committed BPool pages are tracked via a free list. As pages are used from this list, the lazywriter commits more



452 Chapter 11 SQL Server Memory Management

pages from the BPool reservation until the entire reservation has been committed. You will see the Process:Private Bytes Perfmon counter increase gradually (and usually linearly) due to this activity.

There is a separate free list for each CPU on the system. When a free page is needed to satisfy an allocation request, the free list associated with the UMS worker requesting the allocation is checked first, followed by the lists for the other CPUs on the system. This is done to improve scalability by making better use of the local cache for each processor on a multiprocessor system. You can monitor a specific BPool partition via the SQL Server:Buffer Partition Perfmon object. You can monitor the free list for all partitions via the SQL Server:Buffer Manager\Free Pages Perfmon counter.

So, throughout the time that it runs, SQL Server's lazywriter process (whether run from the lazywriter thread or via a UMS worker) monitors the memory status of the system to be sure that a reasonable amount of physical memory remains available to the rest of the system and that a healthy number of free pages remains available for use by new memory allocation requests.

Of necessity, some of this changes when AWE memory is used by the server. Because Windows' AWE facility does not support the concept of reserving but not committing memory (i.e., dynamic memory commitment/decommitment), SQL Server doesn't support dynamic memory management when using AWE memory. The BPool begins by acquiring and locking physical memory on the machine. The amount of memory it locks varies based on whether max server memory has been set. If it has, the BPool attempts to lock the amount specified by max server memory. If it has not, the BPool locks all of the physical memory on the machine except for approximately 128MB, which it leaves available for other processes. The BPool then uses the physical memory above 3GB (the AWE memory) as a kind of paging file for data and index pages. It maps physical pages from this region into the virtual memory address space as necessary so they can be referenced via 32-bit pointers.

Recap

So there you have it: SQL Server memory management in a nutshell. Understanding how an application allocates and manages memory is essential to understanding how the application itself works. Memory is such an important resource and its efficient use is such an integral element of sound application design that understanding how memory is managed by an application gives you great insight into the overall design of the application.



Knowledge Measure

1. The BPool is the pool from which most memory allocations are made within SQL Server. What region is set aside for external consumers such as OLE DB providers that run in-process with the server?
2. Which of SQL Server's five memory managers manages memory related to connections?
3. How often does the lazywriter process run?
4. What Perfmon counter reflects the total amount of committed virtual memory within a process?
5. Describe, in general terms, the two main functions of the lazywriter process.
6. True or false: On all releases of SQL Server 2000 and later, allocations of 8KB or less always come from the BPool.
7. True or false: Allocations made by an xproc via the ODS `srv_alloc` API function are always allocated from the MemToLeave region, regardless of size.
8. What standard COM interface is used within SQL Server to manage individual memory allocations?
9. Which of the five memory managers is the only one the lazywriter can cause to free allocated pages?
10. On a system with 512MB of physical memory and a SQL Server startup parameter of `-g768`, what will be the maximum size to which the BPool can grow?
11. What Win32 API function is used by the lazywriter process on Windows 2000 to check the available physical memory in the machine?
12. True or false: Because UMS workers are often allowed to perform the work of the lazywriter process, the dedicated lazywriter often finds little to do when it runs on an instance of SQL Server installed on the Windows NT family of the Windows operating system.
13. Why is the virtual memory address space for the MemToLeave region reserved at system startup?
14. On a four-way SMP system, how many BPool free list partitions will SQL Server create?
15. What command line parameter can you pass into SQL Server to adjust the size of the MemToLeave region?
16. What operating system privilege must the SQL Server startup account have in order to make use of AWE memory?
17. True or false: When you configure SQL Server to use AWE memory on a system with less than 3GB of physical memory, the server will refuse to start and log an error in the system error log.



454 **Chapter 11 SQL Server Memory Management**

18. What mechanism does SQL Server use to track committed pages in the BPool?
19. What mechanism does SQL Server use to locate a data page in the BPool using its database ID, file number, and page number?
20. How many total reservations can be made at system startup to reserve the BPool?
21. On a SQL Server with 2GB of physical memory, a startup parameter of -g512, and a max server memory setting of 384, what will be the maximum size to which the BPool can grow?
22. By default, how much of the MemToLeave pool is set aside for thread stacks?
23. True or false: Because memory allocations by in-process COM objects that are 8K or less come from the BPool, they are automatically freed when the object is destroyed.
24. True or false: When AWE memory is being used by SQL Server, the physical memory behind the BPool is locked and will not be available to other processes.
25. True or false: By default, the lazywriter process tries to keep at least 384MB available for the MemToLeave region within the SQL Server process.

