**Memory Management**

**Memory management in SQL Server has a three-level structure**. At the bottom are memory nodes, which are the lowest-level allocators for SQL Server memory. The second level consists of memory clerks, which are used to access the memory nodes, and cache stores, which are used for caching. The top level contains memory objects, which provide a smaller degree of granularity than the memory clerks allow directly.

**Only clerks can access memory nodes to allocate memory, so every component that needs to allocate substantial amounts of memory needs to create its own memory clerk when the SQL Server service starts.**

NEW MEMORY MANAGER FOR SQL SERVER 2012

Previous versions of SQL Server required VAS space outside of SQL Server’s memory allocation for Multi-Page Allocations (MPA) and CLR memory requirements.

The MPA was used whenever a component required a single allocation greater than 8KB and a single page allocator dealt with anything less than or equal to 8KB. In SQL Server 2012, there is only one page allocator for all requests and they all come directly from SQL Server’s memory allocation.

CLR allocations also come directly from SQL Server’s memory allocation in SQL Server 2012, which makes it much easier to size SQL Server’s memory requirements .

**1. Memory Nodes**

Memory nodes map directly onto NUMA nodes , and you can view details about these nodes on your server using the sys.dm\_os\_memory\_nodes DMV. You will always have at least one memory node, which has a memory\_node\_id of 0, and you may have several if your CPU architecture supports NUMA.

Each memory node has its own memory clerks and caches, which are distributed evenly across all the nodes (although some objects will only be found in node 0). SQL Server’s total usage is calculated using the sum of all the nodes.

**2. Clerks, Caches, and the Buffer Pool**

Memory clerks are the mechanism by which memory caches are used,. Buffer pool is the largest consumer of memory in SQL Server.

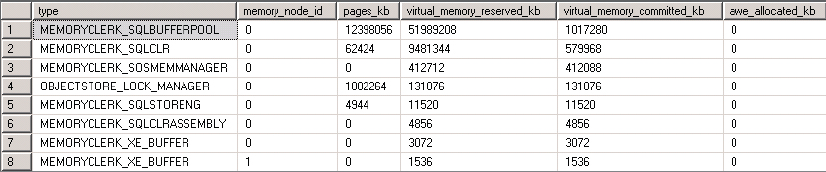
**Memory Clerks**

Whenever a memory consumer within SQL Server wants to allocate memory, it needs to go through a memory clerk, rather than going straight to a memory node. There are generic memory clerks like MEMORYCLERK\_SQLGENERAL, but any component that needs to allocate significant amounts will have been written to create and use its own memory clerk.

The buffer pool for instance has its own memory clerk (MEMORYCLERK\_SQLBUFFERPOOL), as do query plans (MEMORYCLERK\_SQLQUERYPLAN), which makes troubleshooting much easier because you can view the memory allocations made by each clerk and see who has what.

You can view details about all the memory clerks using the **sys.dm\_os\_memory\_clerks** DMV. For example, running the following query against a SQL Server 2012 Enterprise Edition instance running a production workload produced the results shown in Figure 1:

FIGURE 1



SELECT [type],  
 memory\_node\_id,  
 pages\_kb,  
 virtual\_memory\_reserved\_kb,  
 virtual\_memory\_committed\_kb,  
 awe\_allocated\_kb  
FROM sys.dm\_os\_memory\_clerks  
ORDER BY virtual\_memory\_reserved\_kb DESC;

The query orders the results by **virtual\_memory\_reserved\_kb**, so what you see in the figure are the top eight memory clerks ordered by the amount of VAS that they have reserved.

**Caches**

SQL Server uses three types of caching mechanism: object store, cache store, and user store.

Object stores are used to cache homogeneous types of stateless data, but it’s the cache and user stores that you’ll come across most often. They are very similar in that they’re both caches — the main difference between them is that user stores must be created with their own storage semantics using the development framework, whereas a cache store implements support for the memory objects mentioned previously to provide a smaller granularity of memory allocation.

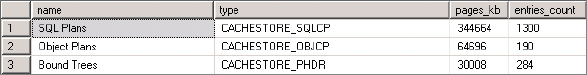
Essentially, the user stores are mainly used by different development teams within Microsoft to implement their own specific caches for SQL Server features, so you can treat cache stores and user stores the same way.

To view the different caches implemented on your SQL Server, use the **sys.dm\_os\_memory\_cache\_counters** DMV. For example, running the following query will show you all the caches available, ordered by the total amount of space they consume:

SELECT [name],  
 [type],  
 pages\_kb,  
 entries\_count  
FROM sys.dm\_os\_memory\_cache\_counters  
ORDER BY pages\_kb DESC;

Sample output showing the top three caches by size is shown in Figure 2.

FIGURE 2



Here, the caches you see are all related to query processing . These specific caches are used for the following:

* CACHESTORE\_OBJCP — Compiled plans for objects such as stored procedures, functions, and triggers
* CACHESTORE\_SQLCP — Cached plans for SQL statements or batches that aren’t in stored procedures. If your application doesn’t use stored procedures, then the plans are cached here. However, they are much less likely to be reused than stored procedure plans, which can lead to a bloated cache taking a lot of memory .
* CACHESTORE\_PHDR — Algebrizer trees for views, constraints, and defaults. An algebrizer tree is the parsed SQL text that resolves table and column names.

**Buffer Pool**

The buffer pool contains and manages SQL Server’s data cache. Information on its contents can be found in the sys.dm\_os\_buffer\_descriptors DMV. For example, the following query returns the amount of data cache usage per database, in MB:

SELECT count(\*)\*8/1024 AS 'Cached Size (MB)'  
 ,CASE database\_id  
 WHEN 32767 THEN 'ResourceDb'  
 ELSE db\_name(database\_id)  
 END AS 'Database'  
FROM sys.dm\_os\_buffer\_descriptors  
GROUP BY db\_name(database\_id),database\_id  
ORDER BY 'Cached Size (MB)' DESC

Monitoring SQL Server’s buffer pool is a great way to look out for memory pressure, and Performance Monitor provides numerous counters to help you do this for quick insight, including the following:

* MSSQL$<instance >:Memory Manager\Total Server Memory (KB) — Indicates the current size of the buffer pool
* MSSQL$<instance >:Memory Manager\Target Server Memory (KB) — Indicates the ideal size for the buffer pool. Total and Target should be almost the same on a server with no memory pressure that has been running for a while. If Total is significantly less than Target, then either the workload hasn’t been sufficient for SQL Server to grow any further or SQL Server cannot grow the buffer pool due to memory pressure, in which case you can investigate further.
* MSSQL$<instance >:Buffer Manager\Page Life Expectancy — Indicates the amount of time, in seconds, that SQL Server expects a page that has been loaded into the buffer pool to remain in cache. Under memory pressure, data pages are flushed from cache far more frequently. Microsoft recommends a minimum of 300 seconds for a good PLE; this threshold continues to be debated within the SQL Server community, but one thing everyone agrees on is that less than 300 seconds is bad. In systems with plenty of physical memory, this will easily reach thousands of seconds.

**Plan Cache**

Execution plans can be time consuming and resource intensive to create; therefore, it makes sense that if SQL Server has already found a good way to execute a piece of code, it should try to reuse it for subsequent requests. The plan cache (also referred to as the procedure cache) is used to cache all the execution plans in case they can be reused.

You can view the contents of the plan cache and determine its current size by using the sys.dm\_exec\_cached\_plans DMV or by running DBCC MEMORYSTATUS and looking for the “Procedure Cache” section, where you’ll find the number of plans in cache and the cache size, in 8KB pages.

NOTE

DBCC MEMORYSTATUS provides a lot of useful information about SQL Server’s memory state but you’ll find that DMVs provide far more flexibility with the output, so try to get used to finding the same information from DMVs whenever possible. The following DMVs are a good place to start:

* sys.dm\_os\_memory\_nodes
* sys.dm\_os\_memory\_clerks
* sys.dm\_os\_memory\_objects
* sys.dm\_os\_memory\_cache\_counters
* sys.dm\_os\_memory\_pools

The following example script uses sys.dm\_exec\_cached\_plans to show the number of cached plans and the total size in MB:

SELECT count(\*) AS 'Number of Plans',  
sum(cast(size\_in\_bytes AS BIGINT))/1024/1024 AS 'Plan Cache Size (MB)'  
FROM sys.dm\_exec\_cached\_plans

Running this on a production SQL Server 2012 instance with Max Server Memory set to 32GB produced the following results:

Number of Plans Plan Cache Size (MB)  
14402 2859

The maximum size for the plan cache is calculated by SQL Server as follows:

* 75% of server memory from 0–4GB +
* 10% of server memory from 4GB–64GB +
* 5% of server memory > 64GB

Therefore, a system with 32GB of RAM would have a maximum plan cache of 3GB + 2.8GB = 5.8GB.

**Query/Workspace Memory**

In SQL Server, query memory (also known as workspace memory) is used to temporarily store results during hash and sort operations when executing a query. If you look at an execution plan for a query and you see hash and/or sort operators, that query needs to use query memory to complete execution.

Query memory is allocated out of the buffer pool, so it’s definitely something to be aware of when you’re building a picture of the memory usage on a server.

You can find out how much query memory an individual query uses by looking at the properties of an actual execution plan in Management Studio, as opposed to an estimated execution plan. The estimated plan contains information about how SQL Server will run the query, and it shows any hash or sort operators; but the actual plan reflects what SQL Server used to execute the query, and it contains additional runtime data, including how much query memory was used.

You can view the details of any queries that already have an allocation of query memory (memory grant) and those that are waiting for a memory grant using the sys.dm\_exec\_query\_memory\_grants DMV.

Query memory also has its own memory clerk, which means you can view the sizing information for outstanding memory grants by querying the sys.dm\_exec\_query\_memory\_grants DMV where type = ’MEMORYCLERK\_SQLQERESERVATIONS’.

The memory requirements for all hash and sort operators in a plan are added together to get the total query memory requirement.

The amount of space available as query memory is dynamically managed between 25% and 75% of the buffer pool but it can grow larger than that if the buffer pool is not under pressure.

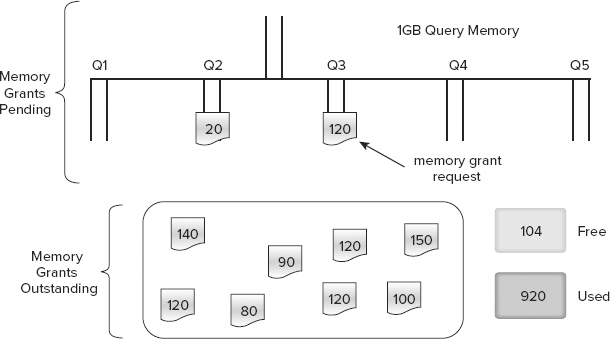
Five percent of query memory is reserved for small queries that require less than 5MB of memory and have a “cost” of less than 3. SQL Server assigns a cost to queries based on how many resources will be needed to run the query.

No individual query will get a grant for more than 20% of the total query memory, to ensure that other queries can still be executed. In addition to this safeguard, SQL Server also implements a query memory grant queue. Every query that contains a hash or sort operation has to pass through the global query memory grant queue before executing, which is organized as five queues organized by query cost

Each query is put into the appropriate queue based on cost, and each queue implements a first-come first-served policy. This method enables smaller queries with lower memory requirements to be processed even if larger queries are waiting for enough free memory.

Figure 3 shows a representation of the five queues based on query cost that make up the global memory grant queue on a server with 1GB of query memory. The box at the bottom of the picture contains eight existing memory grants totaling 920MB, leaving 104MB free. The first request to arrive was for 120MB and went into Q3. This request can’t be allocated immediately because only 104MB are free. The next request is only for 20MB and goes into Q2. This request can be fulfilled immediately because having multiple queues means that it isn’t stuck behind the first request that is still waiting.

FIGURE 3



**The Query Wait Option**

Queries can time out if they spend too much time waiting for a memory grant. The time-out duration is controlled by the Query Wait option, which can be modified either using sp\_configure or on the Advanced page of Server Properties in Management Studio. The default value is -1, which equates to 25 times the cost of the query, in seconds. Any positive value for Query Wait will be used as the time-out value in seconds.

It is possible for a transaction that contains a query waiting for a memory grant to hold locks open and cause a blocking problem before it times out. In this situation, a lower Query Wait value would reduce the impact on other tasks by causing the query to time out sooner.

However, first try to reduce the memory requirement for the query or increase the available memory to avoid the timeout before changing a global option like Query Wait because doing so affects all queries running on the server. The default setting allows for a dynamic time-out value that varies according to the query cost, so it’s generally the best option.

**Query Memory Diagnostics**

There are a number of different ways to get information on query memory usage on your SQL Server. Performance Monitor provides the following counters, all of which are found within the instance’s Memory Manager:

* Granted Workspace Memory (KB) — Total amount of query memory currently in use
* Maximum Workspace Memory (KB) — Total amount of memory that SQL Server has marked for query memory
* Memory Grants Pending — Number of memory grants waiting in the queue
* Memory Grants Outstanding — Number of memory grants currently in use

The RESOURCE\_SEMAPHORE wait type is a wait on a memory grant, so if you see this near the top in your results from the sys.dm\_os\_wait\_stats DMV, then your system is struggling to provide memory grants fast enough.

You can also encounter performance issues other than just a query timing out while it waits for a memory grant. Within an execution plan or when analyzing a SQL trace, you may notice hash warning or sort warning messages if you have selected the relevant events. These occur when the memory grant was insufficient for a query’s requirements.

A hash warning occurs when the hash build doesn’t fit in memory and must be spilled to disk (its actually written to tempdb). A sort warning occurs when a multi-pass sort is required because the granted memory was insufficient. Both warnings generally occur because the SQL Server Query Optimizer made the wrong choice, usually because of inaccurate statistics or a lack of useful statistics.

Now, We will discuss how to diagnose memory pressure in SQL Server, where SQL Server is forced to operate with an insufficient amount of memory, using memory related performance counters and Dynamic Management Views.

**Diagnosing Memory Pressure**

If SQL Server is forced to operate with an insufficient amount of memory, it will be able to store fewer data pages in the buffer pool. As a result, higher physical I/O is needed to bring data pages from disk into the buffer pool as they are requested, and performance will degrade. These data pages may soon be subsequently flushed from the buffer pool, to make room for other pages, only to be then read back into the buffer pool, in a repetitive cycle known as **buffer pool churn**.

The heightened disk I/O observed during buffer pool churn often means that it's easily mistaken for a disk I/O subsystem problem, rather than a memory pressure problem. If a system consistently performs read operations that require more memory than is available to SQL Server, each read of the data will result in flushing the buffer cache, and cause physical I/O against the disk subsystem. In addition, if the workload requires significant amounts of data in sort or aggregate operations during query processing, the data can spill over to tempdb work tables due to a lack of available memory to accommodate that processing, again further increasing the I/O demand on the system.

The best way to tell if SQL Server is under memory pressure and needs more memory is to review a number of critical performance counters related to the health of the SQLServer instance's buffer pool.

**Memory-related counters**

We'll review some of the critical performance counters related to memory usage. It's important to realize that there is no single performance counter that can be used in isolation to determine if SQL Server is under memory pressure, and there is no single point-in-time snapshot of the counters that will allow you to diagnose the "health" of the server in regard to memory use. Diagnosing memory pressure requires tracking several counter values across a period of time (a day, week, or even a month) that represents the normal operations of the database.

**SQL Server: Buffer Manager**

There are a number of useful counters belonging to the **SQL Server:Buffer Manager** object that, when monitored collectively, can help uncover problems relating to buffer pool churn.

**Buffer Cache Hit Ratio**

The **SQL Server:Buffer Manager\Buffer Cache Hit Ratio** counter provides adatabase-wide measure of how often SQL Server gets data from memory (buffer cache) as opposed to disk. There are a ton of references on the Internet suggesting that this counter should be greater than 95% for OLTP systems and greater than 90% for OLAP systems.

**Page Life Expectancy**

The **Page Life Expectancy** counter provides the time in seconds that a page exists in cache before being aged out to allow reuse of the cache space. Again, there are numerous references out there on the Internet, suggesting that the value for this counter should generally be greater than 300 (i.e. five minutes).

However, most of those references are based on a recommendation made by Microsoft over a decade ago, for SQL Server 2000, in a time when 4 GB of RAM was considered to be a lot of memory, and SQL Server was only allocating 1.6 GB of memory to the buffer pool. If SQL Server has to read 1.6 GB of data from disk every five minutes, the impact is minimal.

However, fast-forward ten years, and servers today commonly have 64 GB of RAM, or more, and SQL Server is commonly configured to use a majority of this memory for the buffer pool. If SQL Server has to read 50 GB or more of data from disk every five minutes, then the impact to the I/O subsystem is going to be substantially greater than it would have been when Microsoft made the original recommendation.

The value of this counter reflects the Page Life Expectancy (PLE) at that exact point intime, and it is not uncommon to see periodic drops in the value returned by this performance counter, especially when a large query is executing a table scan, reading new pages into the buffer pool from disk. This counter must be monitored over long periods of time in order to properly identify normal trends.

**Free Pages**

The Free Pages counter reflects the total number of free pages that exist for the SQL Server buffer pool, allowing for immediate allocations by an executing request without having to release additional pages from cache to satisfy the request. The number of free pages in the system should never reach zero. This counter should be monitored in conjunction with the **Page Life Expectancy** counter and the **Free List Stalls** counter, to gauge whether the system is actually under memory pressure.

If the **PLE** experiences drops in value that correlate to low or zero values for Free Pages, and the system is also experiencing Free List Stalls at the same time, then these are a sure sign that the instance is experiencing memory pressure and could benefit from having addition memory allocated to the buffer pool.

**Free List Stalls/sec**

**Free List Stalls** occur whenever a request has to wait for a free page in the buffer pool. If the number of stalls exceeds zero frequently or consistently over a period of time, this is sign of memory pressure.

**Lazy Writes/sec**

The **Lazy Writes/sec** counter reflects the number of buffer pages that have been flushed by the Lazy Writer process, outside of a normal checkpoint operation, allowing the buffer to be reused for other pages. If you observe Lazy Writes occurring in conjunction with a low PLE, a low number of free pages, and the occurrence of Free List Stalls, this is a sign that the workload is exceeding the amount of memory that is available to the buffer pool, and additional memory needs to be added to the server.

**SQL Server:Memory Manager**

The counters relating to the **SQL Server:Memory Manager** object provide useful insight into overall memory consumption and memory management issues on the server.

**Total Server Memory (KB) and Target Server Memory (KB)**

Respectively, these counters represent the total amount of memory that has been allocated by SQL Server and the amount of memory that SQL Server wants to commit.

When the Target Server Memory (KB) counter exceeds the Total Server Memory (KB) counter, the SQL Server process wants to commit more memory than is available on the server, which can be a sign of memory pressure.

**Memory Grants Outstanding**

This counter measures the total number of processes that have successfully acquired a workspace memory grant. Low values for this counter, under periods of high user activity or heavy workload, may be a sign of memory pressure, especially if there are a high number of memory grants pending.

**Memory Grants Pending**

This counter measures the total number of processes that are waiting for a workspace memory grant. If this value is non-zero, it is a sign that tuning or optimization of the workload should be performed if possible, or that additional memory needs to be added to the server.

**Memory-related DMVs**

There is also some information regarding memory-related waits and non-buffer pool memory allocations and so on, that can be extracted from the DMVs, such as the sys.dm\_os\_memory\_\* objects in the Operating System-related DMVs, or the sys.dm\_exec\_query\_memory\_grants DMV. For example:

• **sys.dm\_exec\_query\_memory\_grants** can be used to find queries that are waiting(or have recently had to wait) for a memory grant, especially those requesting relatively large memory grants.

The following query is one we frequently run to identify memory usage for the queries running on our systems.

SELECTsession\_id,request\_id,requested\_memory\_kb,required\_memory\_kb,

used\_memory\_kb,ideal\_memory\_kb,sql\_handle,plan\_handle

FROM sys.dm\_exec\_query\_memory\_grants

• **sys.dm\_os\_memory\_cache\_counters:** provides a snapshot of current usage of the memory cache. Includes the multi\_pages\_kb column showing the amount of memory allocated by the multiple-page allocator.

select(sum(pages\_kb)+sum(pages\_in\_use\_kb))\* 8 /(1024.0 \* 1024.0)asplan\_cache\_in\_GB

fromsys.dm\_os\_memory\_cache\_counters

wheretype='CACHESTORE\_SQLCP'ortype='CACHESTORE\_OBJCP'

* CACHESTORE\_OBJCP. These are compiled plans for stored procedures, functions and triggers.
* CACHESTORE\_SQLCP.  These are cached SQL statements or batches that aren't in stored procedures, functions and triggers.  This includes any dynamic SQL or raw SELECT statements sent to the server.
* CACHESTORE\_PHDR.  These are algebrizer trees for views, constraints and defaults.  An algebrizer tree is the parsed SQL text that resolves the table and column names.

SELECTTOP 6

LEFT([name], 20)as [name],

LEFT([type], 20)as [type],

[pages\_kb] + [pages\_in\_use\_kb] AScache\_kb,

[entries\_count]

FROMsys.dm\_os\_memory\_cache\_counters

orderbypages\_kb+pages\_in\_use\_kbDESC

• **sys.dm\_os\_sys\_memory** summarizes the overall memory condition of the system,including current levels of memory in the system, the cache, and so on.

Step-1:

select\*fromsys.dm\_os\_sys\_memorywheresystem\_memory\_state\_desc='Available physical memory is low’'

We can use this script to get the notification whenever memory use by SQL server is high and take some proactive action before database stop response.

**Step 2**: If we are getting alert like that SQL server Available physical memory is low then we can run the below script to know which database using most memory.

**–Memory Occupied by each Database**

SELECT

(CASEWHEN ([is\_modified] = 1)THEN'Dirty'ELSE'Clean'END)AS'Page State',

(CASEWHEN ([database\_id] = 32767)THEN'Resource Database'ELSEDB\_NAME(database\_id)END)

AS'Database Name',

COUNT(\*)AS'Page Count'

FROMsys.dm\_os\_buffer\_descriptors

GROUPBY [database\_id], [is\_modified]

ORDERBY [database\_id], [is\_modified];

GO

**Step 3: To free up the memory we can use the below script together.**

DBCC FREESYSTEMCACHE (‘ALL’) WITH MARK\_IN\_USE\_FOR\_REMOVAL;

DBCC FREESESSIONCACHE WITH NO\_INFOMSGS;

GO

DBCC DROPCLEANBUFFERS

DBCC FREEPROCCACHE;

* **sys.dm\_os\_memory\_clerks** provides information related to memory clerk processes that manage SQL Server memory. For example, significant memory allocation in the buffer pool associated with the MEMORYCLERK\_SQLQERESERVATIONS may indicate insufficient memory in the buffer pool for certain queries to execute.

To see how SQL is using memory internally we can query the sys.dm\_os\_memory\_clerks DMV to view currently active memory clerks within SQL Server. A memory clerk sits between memory nodes and the memory components within SQL Server. Each component has its own memory clerk that interfaces with the memory nodes to allocate memory; these clerks can then be used to track resource consumption. This architecture also means that threads cannot directly interface with the low level memory allocators but must go to the clerks for memory requests.

* The test instance that will use has 16Gb of RAM in the Server and I have allocated 8Gb to SQL Server, by running the following query I can see the top 5 memory consumers by clerk type and see how much they are using.
* 

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT TOP(5) [type] AS [ClerkType],  SUM(pages\_kb) / 1024 AS [SizeMb]  FROM sys.dm\_os\_memory\_clerks WITH (NOLOCK)  GROUP BY [type]  ORDER BY SUM(pages\_kb) DESC |

As I would expect the Buffer Pool is the largest consumer of memory within the instance with just over 4.5Gb allocated. The lock manager is next with just over 1Gb allocated for lock resources and the remaining clerks relate to allocations for the query plan. The CACHESTORE\_OBJCP allocation refers to plans for stored procedures and functions. The CACHESTORE\_SQLCP are plans not within those object types and refer to statements executed directly against SQL Server whilst the CACHESTORE\_PHDR row shows algebrized trees for various objects.

On a busy SQL Server this information is really useful for us to capture at regular intervals so we can closely monitor memory allocation under normal workloads. If we were to experience performance problems where we suspect memory pressure we can repeat the query to see if memory is being allocated differently.

As an example here’s the same query taken when a full database consistency check is being ran against one of my test databases. We can see here that there’s a new memory clerk that is now in our top 5 allocations list, this particular clerk, SQLQERESERVATIONS is related to Memory Grant allocations within SQL Server.

* Upon seeing the SQLQERESERVATIONS we can query the current memory grants using the sys.dm\_exec\_query\_memory\_grants DMV and by using the CROSS APPLY function to sys.dm\_exec\_sql\_text we can return the query text that is associated with the process. 

The query returns the following single result and with only one process running we know the consistency check has had a direct effect on our memory allocations.

SELECTsession\_id,requested\_memory\_kb/ 1024 asRequestedMemMb,

granted\_memory\_kb/ 1024 asGrantedMemMb,text

FROMsys.dm\_exec\_query\_memory\_grantsqmg

CROSSAPPLYsys.dm\_exec\_sql\_text(sql\_handle)

**Common Memory-Related Problems**

**Paging problems**

Starting with SQL Server 2005 Service Pack 2, if a working set trim occurs SQL Server will, as discussed earlier, write a message to the error log to the effect that "**a significant part of SQL Server process memory has been paged out."**

Whenever these messages appear in the error log, performance degradation has occurred as a result of the working set for SQL being paged out to disk. There are a number of causes of working set trims of the SQL Server process, but the most common ones are:

• Incorrect settings for the max server memory sp\_configure option, when LockPages in Memory is not being used

• A large system cache in Windows caused by caching of non-buffered I/O operations such as file copy operations

• Hardware driver issues that result in memory leaks or excessive memory allocations by the driver.

The fastest way to resolve problems related to the OS trimming the working set of the SQL Server process is to use brute force, and enable *Lock Pages in Memory* for the SQL Server service.

**OS instability due to Lock Pages in Memory plus unlimited max server memory**

When *Lock Pages in Memory*i s being used for a SQL Server instance, the default configuration for the max server memory sp\_configure option must be changed to limit the amount of memory that the SQL Server instance can allocate for the buffer pool.

If the max server memory configuration option is not set for the instance, under the default configuration the instance will allocate all of the memory available on the server for use by the SQL Server buffer pool. The problem with this scenario is that when the Windows OS gets into memory pressure (and it certainly will, as SQL Server commits all of the memory available on the server), it can't page out or trim the SQL

Server working set in response to the memory pressure. This leaves the OS at the mercy of SQL Server to respond to the memory pressure fast enough to prevent the Windows OS from crashing. The same scenario can occur if an inappropriately high value has been set for max server memory for the instance. For this reason, it is critical that when Lock Pages in Memory is set for a SQL Server instance, the max server memory configuration option should be set low enough to ensure that the Windows OS never gets into memory pressure.

**Memory settings for multiple instances**

One of the selling points that Microsoft marketing uses for SQL Server is the ability to install multiple named instances of SQL Server on a single, large server. Consolidating multiple SQL Server instances onto a single server or a failover cluster can lead to a substantial reduction in SQL Server licensing costs. However, when multiple instances are being run on the same physical server, it is very important that the min server memory and max server memory sp\_configure options are set appropriately for each instance, based on workload, in order to avoid competition for memory resources, and one instance starving others of memory. Each of the instances should be monitored using the performance counters to determine where the max server memory value should be set. The aggregate total of the max server memory sp\_configure options for all of the instances on the server should be low enough to ensure that the WindowsOS still has sufficient available memory, as tracked by the **Memory\Available Mbytes** counter, to prevent the OS low memory notifications.It is also recommended that min server memory be configured for each instance, to guarantee a minimum amount of memory to the buffer pool if the Windows OS sets thelow memory notification, and the SQL Server instances reduce their memory allocationsin response. If the min server memory configuration option is not set, a single instance of SQL Server may voluntarily reduce its memory usage to the point that it experiences min server memory for all of the instances will prevent this from occurring, and instead cause the other instances to reduce memory usage appropriately, in response to the OS low memory notification.