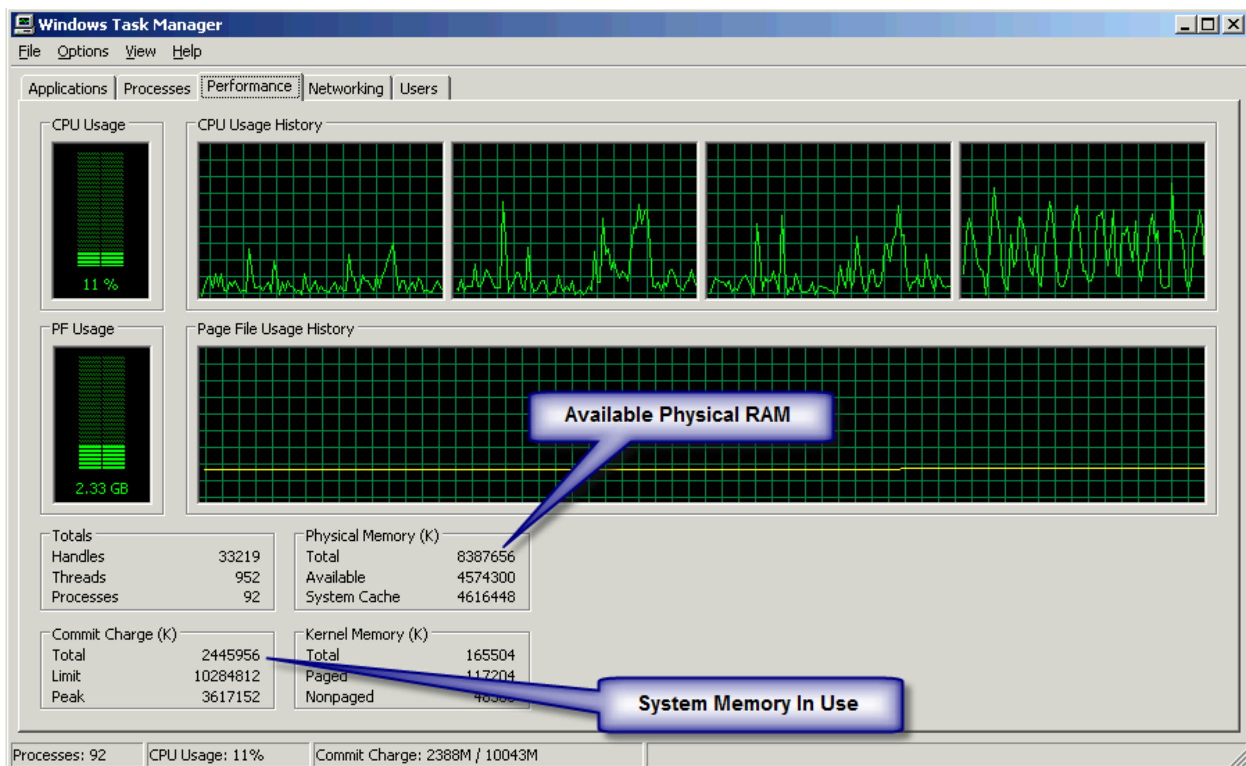
Tech Tip: Interpreting Server Memory Counters

Written by Bill Bach, President of Goldstar Software Inc.

This tech tip is the conclusion of our in a series of tips designed to help you understand the way that your Pervasive PSQL Summit v10 database engine utilizes memory on the database server, and thus provide you with insights on optimizing that memory usage to get the best performance out of your system. In the previous Tech Tips, we reviewed some server memory counters and explained how to find those values on your own system. In this tip, we extend that information to show how we can use those counters to answer some critical questions about your server's environment, and to also locate some problems that may occur.

Do I Have Enough Memory in my Server?

Let's first examine the **Available Physical RAM** and **System Memory In Use** values from the PerfMon screen that we used in our previous Tech Tip.



According to this screen, the **Available Physical RAM** is 8GB, and the **System Memory In Use** is 2.4GB. Note that because of the way that virtual memory works, if the **System Memory In Use** is larger than the **Available Physical RAM**, the OS will be forced to write memory pages out to the swapfile, and severe performance degradation will usually follow. The OS allows such a configuration to enable a server to "do more with less", but due to the performance penalty of disk (easily 1000x slower than memory), the server will be notably sluggish. This same problem is also very common on workstations with less than 1GB of RAM, so you may wish to check your workstations, too!

The server indicated in the picture is using well below is rated maximum memory, and we can use that fact to add more work or increase memory usage on existing processes (if we want to efficiently use our hardware). If your picture shows the opposite, say 3GB of System Memory In Use and only 2GB of Available Physical RAM, then you have a few choices to avoid swapping:

1. **Add Memory**: Physically adding more RAM (up to the maximum supported memory of the OS) is a cheap solution that will solve performance woes for a lot of servers. However, the cost of the memory is only a small part of the real cost of this solution. You must also consider the cost in time to investigate the hardware to see what kind of memory can go in the box (and if there are open DIMM slots), purchase the RIGHT memory for your server, install the memory (within a requisite downtime window), and deal with any problems that may result, especially if the memory turns out to be bad.
2. **Disable Unused Services**: We find many cases where servers are running loads of services that are either not configured or not being used. These services should be disabled.
3. **Upgrade Services and Applications**: In some cases, applications may have bugs that lead to a memory leak, and you may be able to easily upgrade these applications to address the leak. If you see a process using an abnormally high amount of memory, especially if memory usage increases over time, check with the vendor about any updates.
4. **Decrease Memory Usage in Existing Processes**: You can sometimes adjust memory usage, like decreasing the Pervasive database L1 and L2 caches, or changing memory allocations in other applications (like SQL Server), to free up memory. Remember that decreasing memory for applications (like your PSQL database) will usually result in lower performance for that application, but should leave more free memory for the server as a whole.

How Much memory Is My Database Engine Really Using?

Next, let's compare **Process Memory Usage** to **Process Virtual Memory Size**. We get this information from the *Processes* tab of Task Manager, as explained in the last Tech Tip.

Image Name	PID	User Name	CPU	CPU Time	Mem Usage	VM Size	I/O Read Bytes	I/O Write Bytes
ntdbsmgr.exe	2560	SYSTEM	00	0:03:19	155,164 K	454,552 K	78,857,925	445,778,984
SBAMSvc.exe	2988	SYSTEM	00	7:48:40	69,876 K	299,572 K	239,714,227,856	79,823,678,896
QBDBMgrN.exe				0:07:43	155,688 K	148,068 K	196,831,299	79,516,338
QBW32.EXE				0:00:23	141,588 K	82,944 K	342,465,717	262,667
QBDBMgrN.exe				0:00:02	35,412 K	72,244 K	11,934	11,416
lsass.exe	504	SYSTEM	00	0:18:33	60,456 K	45,676 K	901,678,114	117,585,483
dsm_om_connsvc32.exe	3048	SYSTEM	00	0:05:16	66,276 K	45,584 K	14,827,580	2,619
sqlservr.exe	2308	SYSTEM	00	0:03:29	63,196 K	44,564 K	76,528,210	127,798,548
dns.exe	2088	SYSTEM	00	0:00:24	62,796 K	39,004 K	3,831	676
java.exe	1864	SYSTEM	00	0:04:17	66,376 K	34,080 K	9,017,333	1,372

Why is the actual memory usage lower than the VM Size? When the PSQLv10 database engine starts up, it requests memory from the OS for data structures to maintain your database, as well as a sizeable block of memory equivalent to the L1 cache size (the *Cache Allocation Size* setting in the PCC). However, the OS is smart -- it knows that no data has been put in those cache memory pages yet, so it creates a "virtual allocation" and lets the engine think it has the memory without having to go through the work

of physically allocating the memory pages. When the database engine performs a memory write to one of these pages for the first time, the OS traps the memory access (via a *page fault*), allocates "real" memory to the page, writes "0" values to every byte, and then allows the memory write to continue. As this happens, the *Mem Usage* value increases. As the engine continues to work, loading data into L1 cache, all of this memory is eventually allocated, and *Mem Usage* will be nearly equal to *VM Size*.

Is My Database Engine Running Slow Due to OS Thrashing?

We can use these numbers to monitor for swapfile "thrashing" in the OS. Thrashing occurs when the OS makes a decision to swap a process out of memory (by writing it to the swap file) and then is forced to reverse its decision and re-allocate the pages in physical RAM again. This problem often occurs when a server's memory gets low (or when the large file system cache is enabled). When the OS decides to page out the database engine, the *Mem Usage* value decreases dramatically, because the OS has written the memory pages to the swap file and marked the physical memory as "free for use". When the database services its next request, it needs to access its memory, which is now unavailable. The OS generates a page fault, finds that the page is still in memory, and marks the page as in-use again. This is a fairly fast process, but it does take real time, and the database runs slowly in the meantime due to the overhead. Thrashing can be confirmed by monitoring disk writes to the swapfile, which will see a large number of disk writes, but very few disk reads. You can also see this in Task Manager by adding Page Faults/Sec to the column list, and you'll see that *Mem Usage* drops, then quickly increases, accompanied by a large number of page faults whenever this thrashing is evident.

What Happens When a Process Reaches the Process Maximum Addressing Space?

The **Process Maximum Addressing Space** is one of the most critical, and least understood, settings. What is not documented very well is the fact that this is a hard, upper limit. You can call it the "glass ceiling" of memory use, but a more appropriate term is a thick, concrete wall -- you are NOT going to get past it! If a process attempts to allocate memory beyond the **Process Maximum Addressing Space** (again, this is 2GB for a 32-bit process on a 32-bit server), then the memory allocation will fail. Some code (pointing fingers at OS vendors is bad form, so we won't do that here) may not properly handle situations where the memory allocations fail, and this can cause the application to fail, overwrite memory, or even halt completely. In a Pervasive environment that has run out of memory, we usually see the database engine stop responding to SQL requests, slow down, and then eventually stop responding to Btrieve requests shortly thereafter.

Can My Database Engine Really Be Out of Memory?

There is one very common question that confuses a lot of people: I have 16GB of RAM on my Win32 server. How can my database be running out of memory?

To answer this, we need to peer a bit deeper into what is included in this process memory space. Of course, you know that the L1 database cache (*Cache Allocation Size*) is included. But what else? Lots! The **Process Addressing Space In Use** value also includes the memory for L2 cache (*Max Microkernel Memory Size*), as well as memory structures for tracking things like files, handles and clients (which used to be configured via settings for *Maximum Files*, *Maximum Handles* and *Maximum Clients*). What is NOT readily apparent is that the addressing space value ALSO includes a minimum of 1MB stack space for

EVERY thread spawned by the process, *in addition to* the memory actually needed by the thread. This means that a database engine configured with 100 I/O Threads is losing 100MB of addressing space right off the bat. Configuring the server for 128 Communications Threads (used by Btrieve communications) sacrifices another 128MB of addressing space. Add in a SQL application that spawns another 250 SQL sessions (each SQL session is implemented in its own thread), and you've now lost over 1/4 of your 2GB address space (500MB) to thread overhead alone! Add in a healthy L1 cache (800MB), and then turn on the L2 cache to use up a bunch of what's left, and you'll quickly exceed the **Process Maximum Addressing Space** and crash the engine. Ugh!

How Can I Use More Memory on a 32-bit Server?

If you are running a 32-bit server, you're limited to the 2GB addressing space limit imposed by the operating system, so let's look at some options and their respective trade-offs:

1. **Use the /3GB Switch**: This switch increases the Process Maximum Addressing Space from 2GB to 3GB. This can greatly increase the amount of memory space available to your database engine. However, there is a trade-off -- the OS is now limited to 1GB of addressing space, and this can have adverse effects on a large number of other processes. In short, we do NOT recommend this option. If you go this option and start getting Windows Status 1450 Resource Allocation Errors, then you need to turn this option off immediately.
2. **Reduce L2 cache and Enable System Cache**: For servers with LOTS of memory, this may make sense, as the *Use System Cache* setting allows the operating system to provide caching for the database files above and beyond what the database can do. This uses memory above the 2GB limit for database files, and can really improve disk read performance. The trade-off here is that the setting ALSO enables the use of write caching, which can actually *slow down* the disk writes sent to the disk by the database, hampering database write performance. Users with a hefty amount of disk writes should avoid this option. If you do enable the System Cache, you should decrease or disable the L2 cache, since it will likely cache the same files as the OS cache. You may also be able to further increase the size of the L1 cache, though, if you have the available memory space to do so.
3. **Use the Xtreme I/O Cache with PSQLv10**: When PSQLv10 is installed on a 32-bit server with more than 4GB of RAM, you can optionally install the Xtreme I/O (XIO) kernel-level cache driver for your database files. This cache driver allows you to leverage the memory that lies outside of the 2GB addressing space for database file cache. If you enable XIO, be sure to disable the OS cache by setting *Use System Cache* to OFF and set *Max Microkernel Memory Usage* to 0 to disable the L2 cache. One trade-off with XIO is a documented issue with NTFS volumes that are dynamically attached to the server (like portable/USB hard disks). If you are using detachable hard disks, do not use the XIO cache, or you may experience server hang when a drive is connected or disconnected.

Are There Other Solutions?

If you don't like any of the above solutions, or if your database is sufficiently large that you still don't have enough memory to cache most of your database, then you are a great candidate for moving to a 64-bit operating system. A 64-bit server operating system can leverage as much as 2TB of installed RAM

(depending on your OS version), and it increases the **Process Maximum Addressing Space** for 32-bit processes from 2GB to 4GB, which may be enough to cache all of your data and get you the performance and stability that you really want. Of course, the cost of a server with 2TB memory is prohibitive today, but some companies are successfully running servers with 128GB of RAM already, and larger memory configurations are certainly possible with the latest hardware.

Of course, a 64-bit operating system is only one part of the answer. The other part is to consider moving to a 64-bit Pervasive PSQL Summit v10 server engine. Using a 64-bit process on a 64-bit operating system enables a whopping *8TB* for the **Process Maximum Addressing Space**, eliminating just about anyone's memory concerns! With this 64-on-64 combination, it is quite easy to configure a server with 128GB of memory to use 100GB for the L1 database cache, and you can safely run hundreds of users (and forget any worries about the number of SQL connections or threads) on your Pervasive database.

Note, though, that even the 64-bit PSQLv10 engine still contains a 32-bit SQL engine. As such, while you'll have eliminated the problems of a large cache sucking up your addressing space on the NTDBSMGR64.EXE process, you may still need to watch out for the 32-bit NTDBSMGR.EXE process exceeding its 4GB maximum.

Summary

Users with small Pervasive databases supporting a handful of users can leverage the power of Pervasive's "no DBA required" installation process to get great performance out of their applications. In fact, Pervasive PSQL works with just about ANY configuration you want to throw at it, and it works pretty darned well. [Heck, I still have a laptop running NetWare and PSQLV8 with only ***128MB*** of RAM!] However, sites with larger databases and hundreds of users may want to do a little bit more work to tune their server, and understanding the memory allocation and usage on your database server is the key to getting started with that tuning.

Author Information:

Bill Bach is the Founder and President of Goldstar Software Inc., a Pervasive reseller in the Chicago area that specializes in providing Pervasive products, services, and training to its customers in North America and abroad. Bill has written numerous tools and utilities to help system administrators and database developers work with their Pervasive database environments, and his training classes for Pervasive PSQL and DataExchange are the most comprehensive classes available. Get more information from http://www.goldstarsoftware.com.