Jon's Performance Musings: Remember Memory?

Jon E. Schmidt
Janis K Kuritsubo
Transaction Design, Inc.
San Rafael, CA, 94901, USA
1.415.256.8369
inform@banbottlenecks.com

Jon is the founder of Transaction Design, Inc. (TDI), a consulting firm located in the San Francisco Area which specializes in performance and capacity studies with clients worldwide. He is the creator of the Ban Bottlenecks® service and has an extensive background in the implementation, testing, and tuning of high-availability systems. Janis is Chief Technical Officer for TDI.

Indispensible, Ignored Memory

Indispensible, but frequently ignored, memory is a key component of any performance study. Whether you call it Core, RAM, Main Memory, Internal Memory, or Primary Storage, it still performs the same function: Holding instructions and data for the processors to use. Not having enough of it is a sure recipe for slowness. Misuse of it is also a recipe for slowness. Underuse of it is a waste of money.

Memory in modern computers is quite complex. Virtual memory adds a second dimension to memory which makes it more versatile, and more complex. Add another layer, virtual machines, and the potential for problems grows even larger. I'm not going to talk about the additional levels of cache memory internal to a processor chip, since we have little control of how that is used unless we're writing compilers or OS code.

Different architectures manage memory differently. Linux/Unix will use as much available memory as possible and adjust memory toward disk cache and other system-related services where as on NonStop the system admin has more control over how much and where memory is used. Low available memory on Linux doesn't necessarily mean the system is short on memory. On NonStop, low available memory means low available memory.

What's in memory?

Here's a short list:

Hypervisor	Disk cache
OS kernel	Database services
OS services	Database cache
OS data structures	Application programs
Disk services	Application data

Our Job:

As programmers and system managers our job is to get the computer to do work as efficiently as possible. This includes making sure that when programs and data are needed they are in memory. Obviously this involves a lot of factors and trade-offs.

Where's the problem?

What are the indicators that a problem exists? Easy: The system is slower than we want it to be. That's the basic definition. I'm a purist, so I hate to see any system "out of whack." On the other hand, I've seen terribly-configured systems, hurting on any number of points, but the user is comfortable with the response s/he is getting. The customer is always right, so I merely suggest that things could be better.

When a computer is slow, and the CPU and disks aren't working hard, it's probably a memory problem. It could mean that there isn't enough memory, or that memory isn't being used properly.

Page Faults/Paging/Swaps

When a computer needs a piece of program or data, and it isn't in memory, someone has to go get it. Depending on the architecture (operating system) being used, this will generate a page fault, a swap, or (generically) paging. Each term has its definition for a particular architecture, and some architectures use multiple terms with differing definitions. For example, NonStop uses the term "swap" but it means a page fault. Linux/Unix makes a distinction between paging and swapping, swap outs are more detrimental than page outs. Windows refers to paging, but it can mean hard or soft page faults or disk I/O for reading data.

Page Faults Are Always Bad

Page faults are always bad. Some are worse than others. Some are unavoidable. Think about it: A page fault means that a memory reference, which should have taken nanoseconds, caused a call to the operating system, and perhaps a disk I/O, which takes milliseconds. That's several orders of magnitude slower.

Page Faults Types

So, what are the different types of page faults, and how can they be controlled?

First we have what I'll refer to as "initialization" page faults. These occur at process start time, when the process needs to load instructions and data into memory. Ideally they occur only once. Unless there is pressure to reuse memory, these pages will stay in memory for the duration of the process.

Next: Transaction page faults. These are the ones that are traditional, but I hate to see. These occur when the page fault rate for the box tracks the transaction rate. They're traditional, since a generation of programmers have been taught to allocate memory at the beginning of a procedure, and de-allocate it at the end of a procedure. If the procedure is in the middle of the transaction path, so be it. Traditional, but it's bad design. Depending on

the architecture, allocate/de-allocate can cause OS calls with the ensuing virtual memory mapping, swap space allocation, and perhaps memory initialization/destruction. Allocating memory within the transaction path can lead to memory leaks if the programmer forgets to de-allocate.

Finally, we have forced page faults, swaps, page evicts, or thrashing. These are the worst kind of page faults. They are an indication of insufficient memory. They occur when the OS or a process needs more memory, but memory is fully allocated. Something must be forced out to make room for the requested page. This causes multiple scans of the OS memory tables to look for a free page (and there isn't one) and then look for an aged page. Then the OS must destroy it or write it to disk, and then bring in the requested page.

Finding/Fixing Thrashing

Out-of-memory page faults, otherwise known as "thrashing," have the most impact on performance. Not only does the requesting process have to wait for memory to be written out, and then the requested page brought in, but other processes are affected because some of their memory resources have been forced out to disk.

How do you find out if this is occurring? First, look for high page fault rates. If they are occurring constantly during the day, or if they spike at certain times of the day, you may have a problem. Then, look at the *Measure* CPU counter ending-free-mem. If this is zero, you probably have a problem. If it is zero most of the day you probably have a larger problem.

How do you fix thrashing? It may be as simple as not scheduling all of your batch jobs to start at the same time. Put them in a TACL macro so that they run one after the other, sequentially. Or, look to see which processes are the largest by looking at the *Measure* PROCESS counter pres-pages-end. Do a rationality check: Does it make sense that this process is this big? Is there a bug in the process?

Are all your CPUs having the problem? Maybe by shifting processes around and rebalancing you can offload the CPUs with the least free memory and stop thrashing. You could also take a look and see if you have over-allocated disk cache on the problem CPUs.

Plugging Leaks

If your system seems to slow down over time, you may have a leak. Leaks occur when transaction page faults exist, and allocates are not reversed by a corresponding de-allocate. That is, memory is being allocated for each transaction, but not being released. This will cause the process to grow, eating up memory and swap space. Leaks are fairly easy to detect by monitoring ending-free-mem over days or weeks. If you see it continue to go down over a long period of time, the next thing to do is look at the pres-pages-end counter for your larger processes over those same intervals. If those processes are growing, it's time to do some debugging.

Plain Ol' Slowness

Remember that page faults are painful. Initialization faults, while sometimes necessary, are still painful. Sometimes slowness is simply caused by too many initializations. Simple rule: Don't start a process unless you must. Unfortunately, designers from the UNIX world think nothing about starting a process (forking) when they need to do something. I've seen systems where hundreds of processes were started in a minute, each one taking a second or less to execute. Very poor design and it *will* have a performance impact on the whole system. Using one process to execute a function hundreds of times is much more efficient.

We have a similar issue in the NonStop world. TACL macros are pretty easy to create, making it too easy to start lots of unnecessary processes. For example, we frequently see TACL macros generate lots of FUP processes when only one is necessary. These macros create a FUP process for each FUP command. A better method is to start one FUP with either INLINE or INV/OUTV option and passing commands to this one FUP.

Finding Fault With Memory

Us older guys always worry about our memory; seems like someone's always finding fault with it. It's nice to turn the tables and find and fix faults with another's memory. It's a win-win.