**Boris Jurosevic**

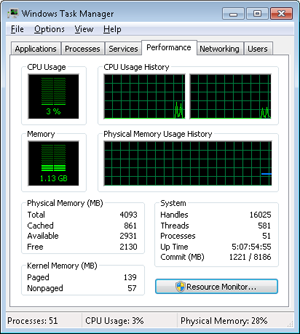
**CS 3100**

**Lab 2: Examining Memory Usage**

The Memory and Process performance counter objects provide access to most of the details about system and process memory utilization. Throughout the chapter, we’ll include references to specific performance counters that contain information related to the component being described. We’ve included relevant examples and experiments throughout the chapter. One word of caution, however: different utilities use varying and sometimes inconsistent or confusing names when displaying memory information. The following experiment illustrates this point. (We’ll explain the terms used in this example in subsequent sections.)

**EXPERIMENT: Viewing System Memory Information**

The Performance tab in the **Windows Task Manager**, shown in the following screen shot, displays basic system memory information. This information is a subset of the detailed memory information available through the performance counters. It includes data on both physical and virtual memory usage.

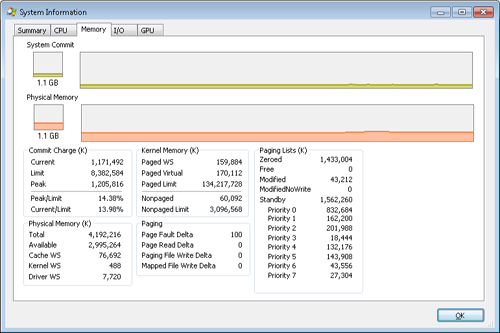


The following table shows the meaning of the memory-related values.

| **Task Manager Value** | **Definition** |
| --- | --- |
| Memory bar histogram | Bar/chart line height shows physical memory in use by Windows (not available as a performance counter). The remaining height of the graph is equal to the Available counter in the Physical Memory section, described later in the table. The total height of the graph is equal to the Total counter in that section. This represents the total RAM usable by the operating system, and does not include BIOS shadow pages, device memory, and so on. |
| Physical Memory (MB): Total | Physical memory usable by Windows |
| Physical Memory (MB): Cached | Sum of the following performance counters in the Memory object: Cache Bytes, Modified Page List Bytes, Standby Cache Core Bytes, Standby Cache Normal Priority Bytes, and Standby Cache Reserve Bytes (all in Memory object) |
| Physical Memory (MB): Available | Amount of memory that is immediately available for use by the operating system, processes, and drivers. Equal to the combined size of the standby, free, and zero page lists. |
| Physical Memory (MB): Free | Free and zero page list bytes |
| Kernel Memory (MB): Paged | Pool paged bytes. This is the total size of the pool, including both free and allocated regions |
| Kernel Memory (MB): Nonpaged | Pool nonpaged bytes. This is the total size of the pool, including both free and allocated regions |
| System: Commit (two numbers shown) | Equal to performance counters Committed Bytes and Commit Limit, respectively |

To see the specific usage of paged and nonpaged pool, use the [**Poolmon utility**](http://msdn.microsoft.com/en-us/library/windows/hardware/ff550442(v=vs.85).aspx).

The [**Process Explorer tool**](http://technet.microsoft.com/en-us/sysinternals/bb896653) can show considerably more data about physical and virtual memory. On its main screen, click View and then System Information, and then choose the Memory tab. Here is an example display from a 32-bit Windows system:



We will explain most of these additional counters in the relevant sections later in this lab.

Two other tools show extended memory information:

* [**VMMap**](http://technet.microsoft.com/en-us/sysinternals/dd535533) shows the usage of virtual memory within a process to an extremely fine level of detail.
* [**RAMMap**](http://technet.microsoft.com/en-us/sysinternals/ff700229) shows detailed physical memory usage.

These tools will be featured in experiments found later in these labs.

**EXPERIMENT: Reserved vs. Committed Pages**

The [**TestLimit utility**](http://download.sysinternals.com/files/TestLimit.zip) can be used to allocate large amounts of either reserved or private committed virtual memory, and the difference can be observed via Process Explorer. First, open two Command Prompt windows. Invoke TestLimit in one of them to create a large amount of reserved memory:

C:\temp>testlimit -r 1 -c 800

Testlimit v5.2 - test Windows limits

Copyright (C) 2012 Mark Russinovich

Sysinternals - wwww.sysinternals.com

Process ID: 1544

Reserving private bytes 1 MB at a time ...

Leaked 800 MB of reserved memory (800 MB total leaked). Lasterror: 0

The operation completed successfully.

In the other window, create a similar amount of committed memory:

C:\temp>testlimit -m 1 -c 800

Testlimit v5.2 - test Windows limits

Copyright (C) 2012 Mark Russinovich

Sysinternals - wwww.sysinternals.com

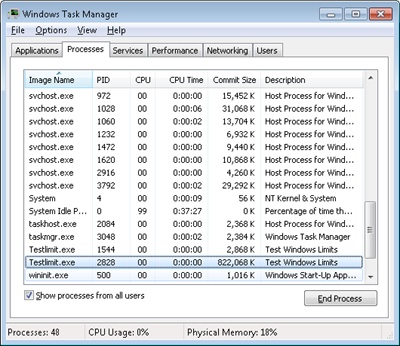
Process ID: 2828

Leaking private bytes 1 KB at a time ...

Leaked 800 MB of private memory (800 MB total leaked). Lasterror: 0

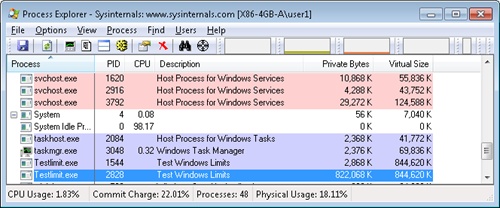
The operation completed successfully.

Now run **Task Manager**, go to the Processes tab, and use the Select Columns command on the View menu to include Memory—Commit Size in the display. Find the two instances of TestLimit in the list. They should appear something like the following figure.

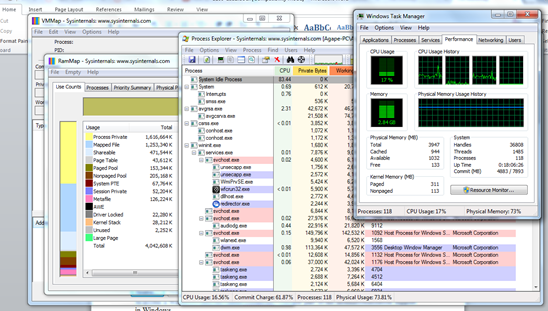


Task Manager shows the committed size, but it has no counters that will reveal the reserved memory in the other TestLimit process.

Finally, invoke **Process Explorer**. Choose View, Select Columns, select the Process Memory tab, and enable the Private Bytes and Virtual Size counters. Find the two TestLimit processes in the main display:



Notice that the virtual sizes of the two processes are identical, but only one shows a value for Private Bytes comparable to that for Virtual Size. The large difference in the other TestLimit process (process ID 1544) is due to the reserved memory. The same comparison could be made in Performance Monitor by looking at the Process | Virtual Bytes and Process | Private Bytes counters.

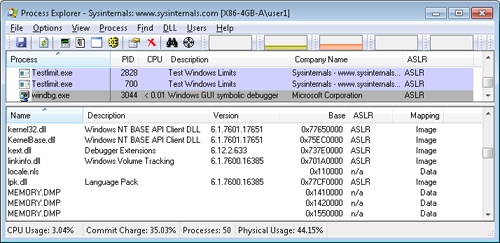


I didn’t have any difficulties doing this part of the lab. After installing it took me a while what it was all about. After running the command prompt first it did not let me do it, but I later happen to find out that command prompt was space sensitive. After doing that I went to Windows task manager and I did see right away the two processes called test limit. First I was shocked to see those two processes being on there because I thought after you have process you can’t have two at the same thing. I was surprised. Also I thought that they were same exact processes, but I was wrong which lab confirmed as well. Right after I went into a process explorer only one showed a value for private bytes. What I thought was most interesting was the difference between them which was reserved memory. Overall this part of the lab was straight to the point, and I could see it being very useful in the future when it comes to memory.

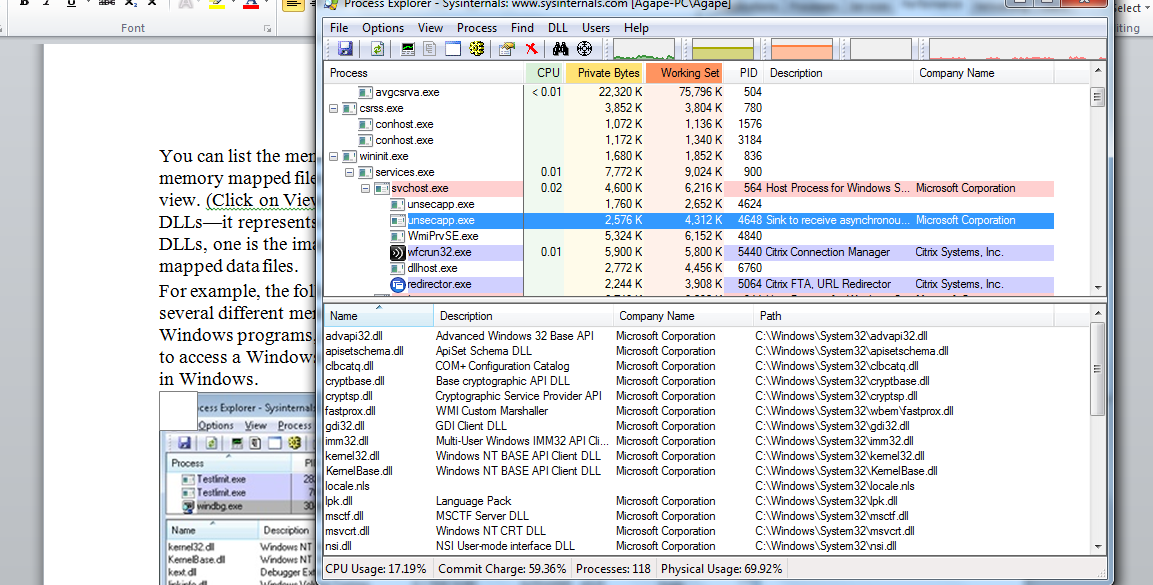
**EXPERIMENT: Viewing Memory Mapped Files**

You can list the memory mapped files in a process by using **Process Explorer**. To view the memory mapped files by using Process Explorer, configure the lower pane to show the DLL view. (Click on View, Lower Pane View, DLLs.) Note that this is more than just a list of DLLs—it represents all memory mapped files in the process address space. Some of these are DLLs, one is the image file (EXE) being run, and additional entries might represent memory mapped data files.

For example, the following display from Process Explorer shows a WinDbg process using several different memory mappings to access the memory dump file being examined. Like most Windows programs, it (or one of the Windows DLLs it is using) is also using memory mapping to access a Windows data file called Locale.nls, which is part of the internationalization support in Windows.



You can also search for memory mapped files by clicking Find, DLL. This can be useful when trying to determine which process(es) are using a DLL or a memory mapped file that you are trying to replace.



I am glad we have done this part of the lab. Going back to the last lab I was kind of playing with process explorer right after I was done with lab one, and I was actually curious what the DLL was. I saw that it wasn’t just a list, and it had to mean something. Well I was finally glad to see when I read that it was it represents all memory mapped files in the process address space. Well that right away answered my question. The reason I am saying this is because I was looking through google.com to find out what those actually mean. Overall I thought that this was something that needed to be covered and at the same time it answered my question.

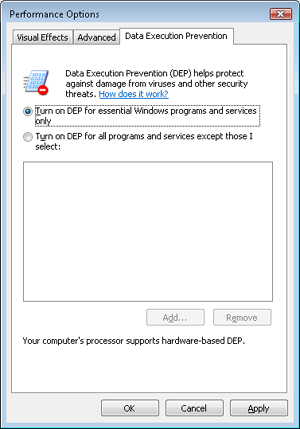
**EXPERIMENT: Looking at DEP Protection on Processes**

No execute page protection (also referred to as data execution prevention, or DEP) causes an attempt to transfer control to an instruction in a page marked as “no execute” to generate an access fault.

**Process Explorer** can show you the current DEP status for all the processes on your system, including whether the process is opted in or benefiting from permanent protection. To look at the DEP status for processes, right-click any column in the process tree, choose Select Columns, and then select DEP Status on the Process Image tab. Three values are possible:

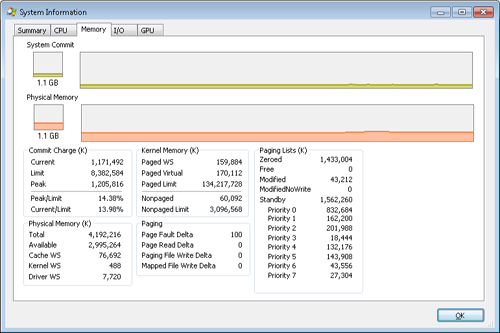
* **DEP (permanent)** This means that the process has DEP enabled because it is a “necessary Windows program or service.”
* **DEP** This means that the process opted in to DEP. This may be due to a systemwide policy to opt in all 32-bit processes, an API call such as *SetProcessDEPPolicy*, or setting the linker flag /NXCOMPAT when the image was built.
* **Nothing** If the column displays no information for this process, DEP is disabled, either because of a systemwide policy or an explicit API call or shim.

The following Process Explorer window shows an example of a system on which DEP is set to OptOut, Turn On DEP For All Programs And Services Except Those That I Select. Note that two processes running in the user’s login, a third-party sound-card manager and a USB port monitor, show simply DEP, meaning that DEP can be turned off for them via the dialog box shown in the figure below. The other processes shown are running Windows in-box programs and show DEP (Permanent), indicating that DEP cannot be disabled for them.

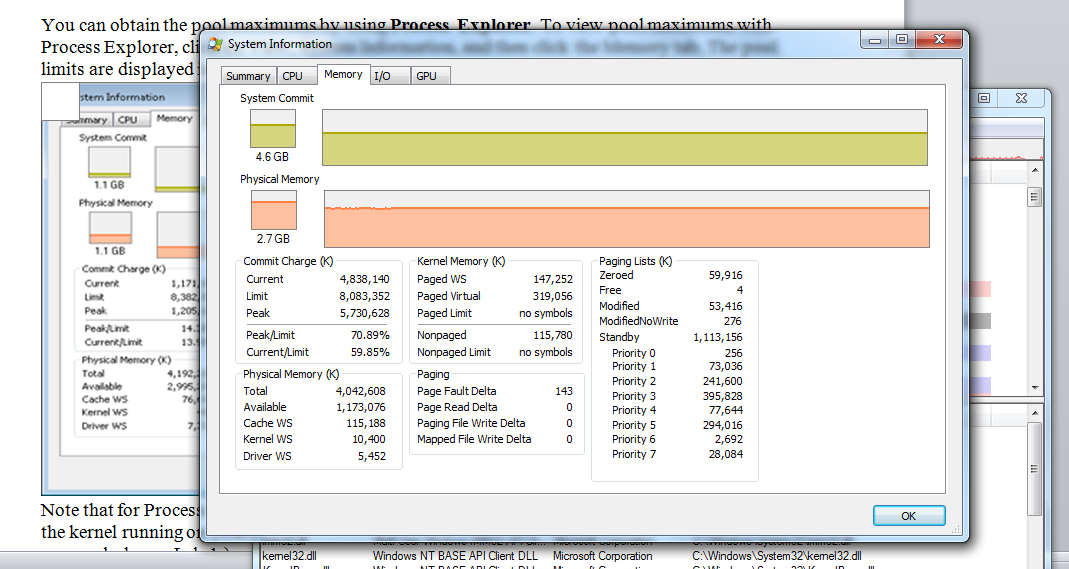


**EXPERIMENT: Determining the Maximum Pool Sizes**

You can obtain the pool maximums by using **Process Explorer**. To view pool maximums with Process Explorer, click on View, System Information, and then click the Memory tab. The pool limits are displayed in the Kernel Memory middle section, as shown here:



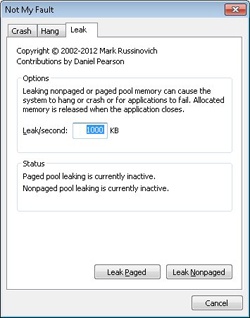
Note that for Process Explorer to retrieve this information, it must have access to the symbols for the kernel running on your system. (For a description of how to configure Process Explorer to use symbols, see Lab 1.)



This was a very different lab, I thought it was very interesting. I did not know that you could access the memory and get a full picture to see the memory and what it is all about. I thought that process explorer was very useful because it can show you the current DEP status for all the processes on your system, including whether the process is opted in or benefiting from permanent protection just like the lab said. What I thought was really interesting was the three values which are possible that shows DEP status for processes. Out of those three I thought DEP was the one I liked the most because it is a necessary Windows program or service which shows you the importance of it. I did not have any difficulties, this part of the lab was straight forward.

**EXPERIMENT: Troubleshooting a Pool Leak**

In this experiment, you will fix a real paged pool leak on your system so that you can put to use the techniques described in the previous section to track down the leak. The leak will be generated by the [Notmyfault tool](http://download.sysinternals.com/files/NotMyFault.zip) from Sysinternals. When you run Notmyfault.exe, it loads the device driver Myfault.sys and presents the following dialog box:



1. Click the Leak tab, ensure that Leak/Second is set to 1000 KB, and click the Leak Paged button. This causes Notmyfault to begin sending requests to the Myfault device driver to allocate paged pool. Notmyfault will continue sending requests until you click the Stop Paged button. Note that paged pool is not normally released even when you close a program that has caused it to occur (by interacting with a buggy device driver); the pool is permanently leaked until you reboot the system. However, to make testing easier, the Myfault device driver detects that the process was closed and frees its allocations.
2. While the pool is leaking, first open Task Manager and click on the Performance tab. You should notice Kernel Memory (MB): Paged climbing. You can also check this with Process Explorer’s System Information display. (Click View, System Information, and then the Memory tab.)
3. To determine the pool tag that is leaking, run Poolmon and press the B key to sort by the number of bytes. Press P twice so that Poolmon is showing only paged pool. You should notice the pool tag “Leak” climbing to the top of the list. (Poolmon shows changes to pool allocations by highlighting the lines that change.)
4. Now press the Stop Paged button so that you don’t exhaust paged pool on your system.
5. Using the technique described in the below\*, run [**Strings.exe**](http://technet.microsoft.com/en-us/sysinternals/bb897439.aspx) (from Sysinternals) to look for driver binaries that contain the pool tag “Leak”:

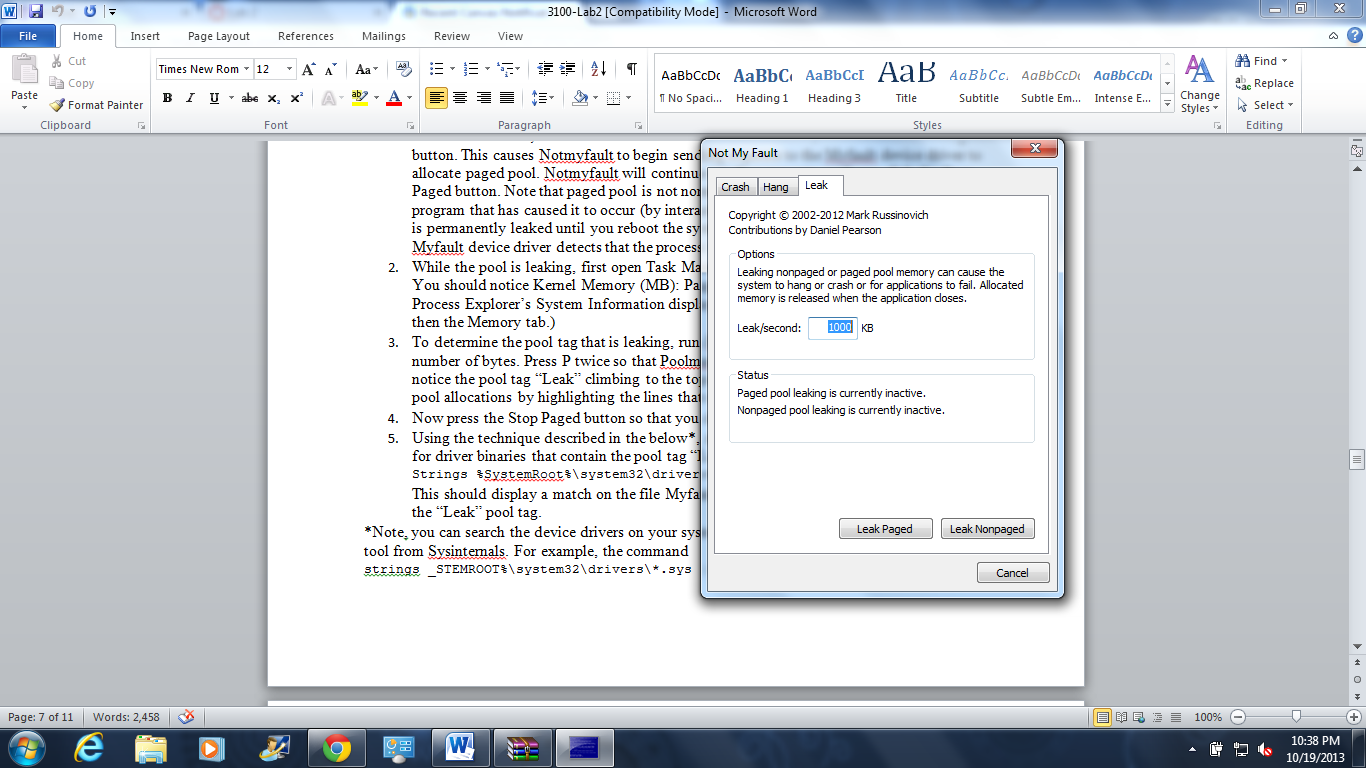
Strings %SystemRoot%\system32\drivers\\*.sys | findstr Leak

This should display a match on the file Myfault.sys, thus confirming it as the driver using the “Leak” pool tag.

\*Note, you can search the device drivers on your system for a pool tag by using the Strings.exe tool from Sysinternals. For example, the command

strings \_STEMROOT%\system32\drivers\\*.sys | findstr /i "abcd"

will display drivers that contain the string “abcd”. Note that device drivers do not necessarily have to be located in %SystemRoot%\System32\Drivers—they can be in any folder. To list the full path of all loaded drivers, open the Run dialog box from the Start menu, and then type Msinfo32. Click Software Environment, and then click System Drivers. As already noted, if a device driver has been loaded and then deleted from the system, it will not be listed here.



This was the first exercise I did for this lab because I was really curious after reading through it. I noticed that I had a problem right away because I was running my laptop as a administrator. I kept getting frustrated at first but when I switched it worked like a charm. I was excited at first but I ran into a little problem when I was trying to determine the pool tag that is leaking because I could not see kernel memory at first. I later I found out that I that I did not press B twice, instead I kept doing it once. I noticed right away a device driver has been loaded and it was listed here. I have never heard of Pool Tag before that is leaking, but I thought that it was very important when it comes to our operating system.

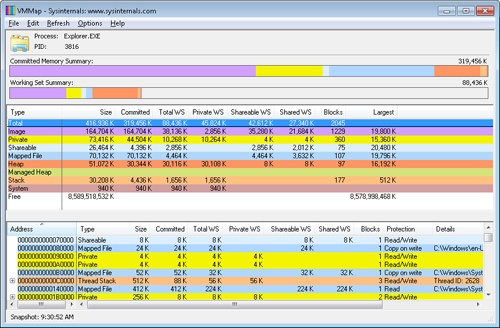
**EXPERIMENT: Analyzing User Virtual Address Space**

Just as address space in the kernel is dynamic, the user address space is also built dynamically—the addresses of the thread stacks, process heaps, and loaded images (such as DLLs and an application’s executable) are dynamically computed (if the application and its images support it) through a mechanism known as Address Space Layout Randomization, or ASLR.

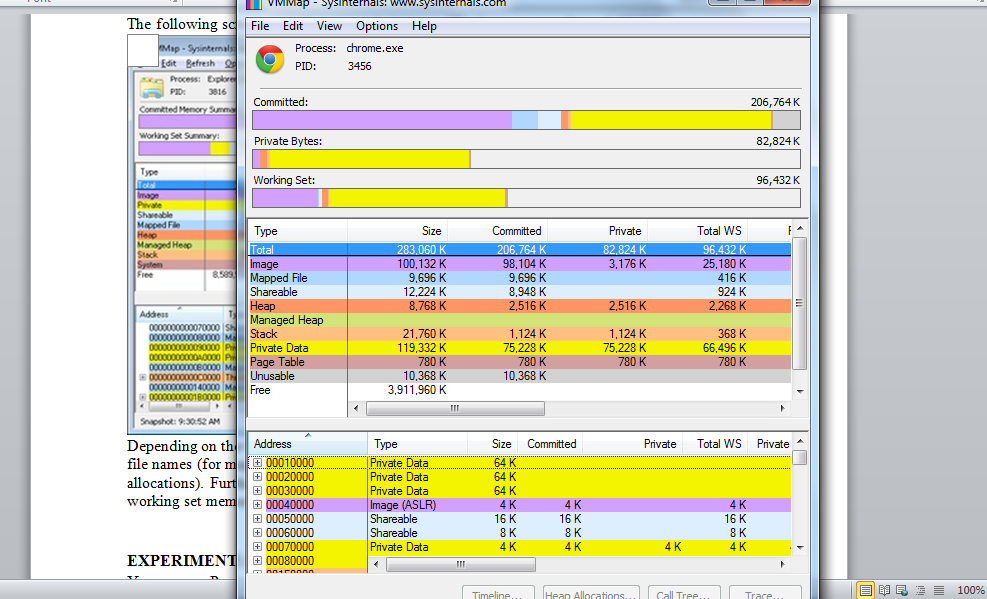
The **VMMap utility** can show you a detailed view of the virtual memory being utilized by any process on your machine, divided into categories for each type of allocation, summarized as follows:

* **Image** Displays memory allocations used to map the executable and its dependencies (such as dynamic libraries) and any other memory mapped image (portable executable format) files
* **Private** Displays memory allocations marked as private, such as internal data structures, other than the stack and heap
* **Shareable** Displays memory allocations marked as shareable, typically including shared memory (but not memory mapped files, which are either Image or Mapped File)
* **Mapped File** Displays memory allocations for memory mapped data files
* **Heap** Displays memory allocated for the heap(s) that this process owns
* **Stack** Displays memory allocated for the stack of each thread in this process
* **System** Displays kernel memory allocated for the process (such as the process object)

The following screen shot shows a typical view of Explorer as seen through VMMap.



Depending on the type of memory allocation, VMMap can show additional information, such as file names (for mapped files), heap IDs (for heap allocations), and thread IDs (for stack allocations). Furthermore, each allocation’s cost is shown both in committed memory and working set memory. The size and protection of each allocation is also displayed.



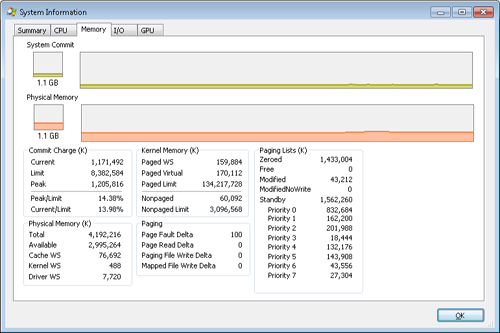
Wow! I totally loved this part of the lab. I thought it was most enjoyable and definitely most useful. As I was finishing chapter 9 and Virtual Address Space came on I was ready to this because I was interested in actually doing just to see what it is in real life picture. What I thought was straight to the point was, it can show you a detailed view of the virtual memory being utilized by any process on your machine. Also I liked how it was divided into categories for each type of allocation. Pretty clear and straight to the point on what you what to know. I liked how it also had file names. Size and protection of each allocation is something you cannot forget, because it one of the most important things we have when it comes to Virtual Address in operating system.

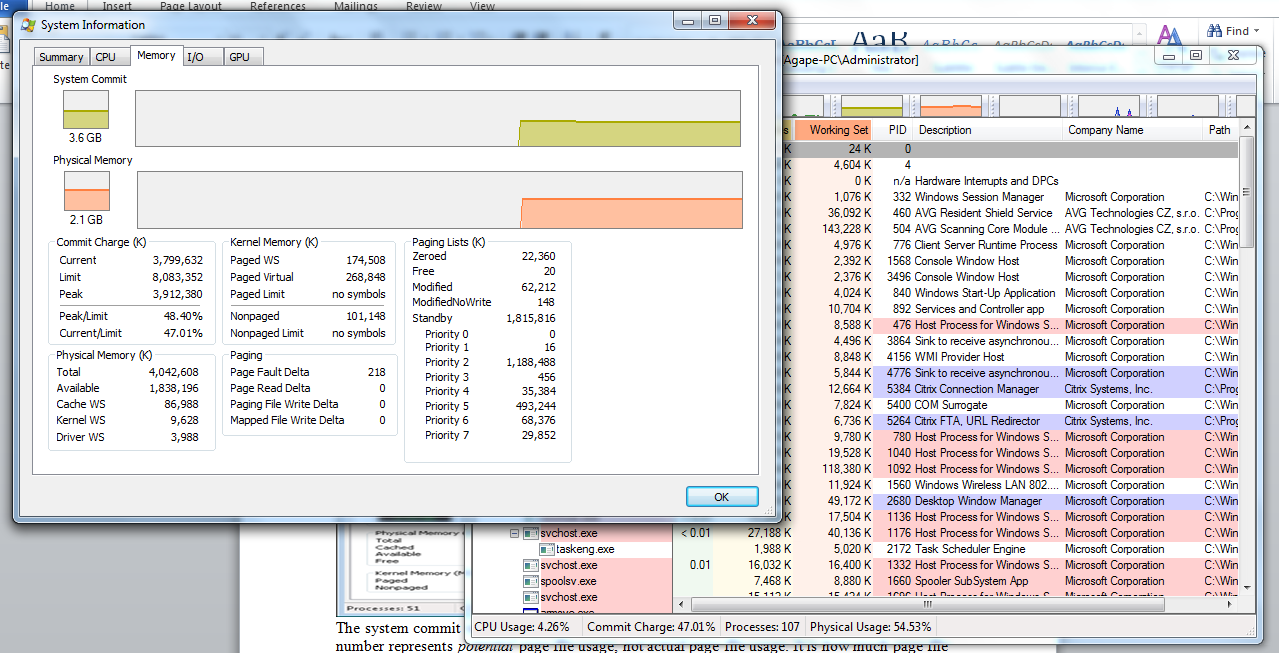
**EXPERIMENT: Looking at ASLR Protection on Processes**

You can use Process Explorer to look over your processes (and, just as important, the DLLs they load) to see if they support ASLR. Note that even if just one DLL loaded by a process does not support ASLR, it can make the process much more vulnerable to attacks.

To look at the ASLR status for processes, right-click on any column in the process tree, choose Select Columns, and then check ASLR Enabled on the Process Image tab. Notice that not all in-box Windows programs and services are running with ASLR enabled, and there is one visible example of a third-party application that does not have ASLR enabled either.

In the example, we have highlighted the Notepad.exe process. In this case, its load address is 0xFE0000. If you were to close all instances of Notepad and then start another, you would find it at a different load address. If you shut down and reboot the system and then try the experiment again, you would find that the ASLR-enabled DLLs are at different load addresses after each boot.

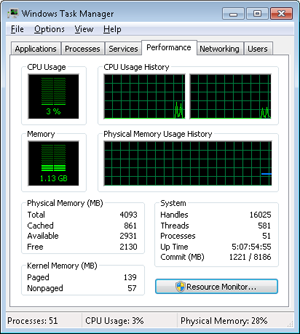




This was one of those parts I did not know what I was about at first. But reading through it was useful because it makes process much more vulnerable to attacks. Just out of curiosity I did close all instances of Notepad and then start another, I found out that it did have different load address. I was blown away! Wow! As I was doing this I actually came back later at night and restarted my laptop and there it was again, it had different load address. This was one of those things that will stick in your mind for a long time. I did not have any difficulties doing this task at all, I had more fun finding out this type of a Protection on Processes. Very useful!

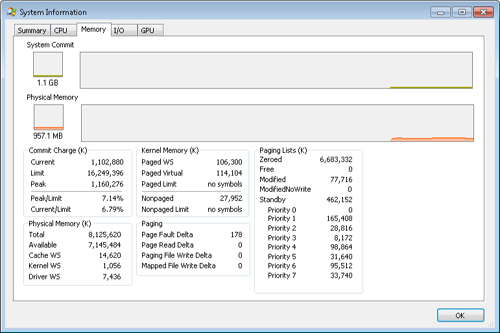
**EXPERIMENT: Viewing Page File Usage with Task Manager**

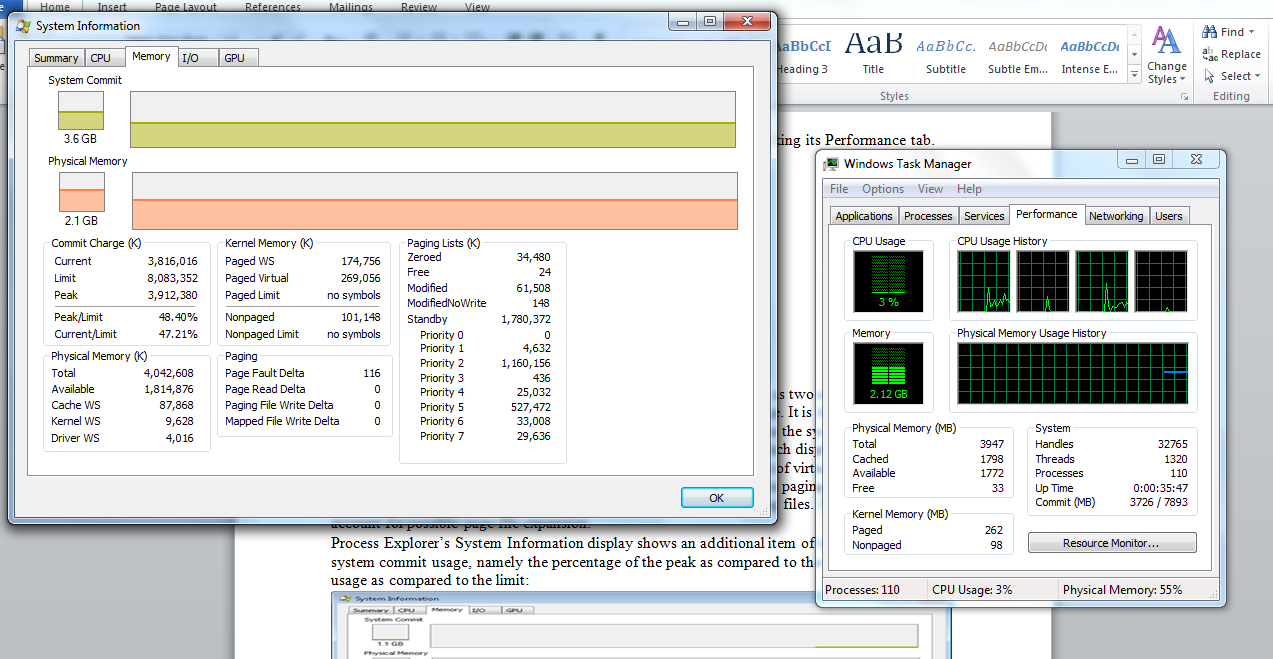
You can view committed memory usage with **Task Manager** by clicking its Performance tab. You’ll see the following counters related to page files:



The system commit total is displayed in the lower-right System area as two numbers. The first number represents *potential* page file usage, not actual page file usage. It is how much page file space would be used if all of the private committed virtual memory in the system had to be paged out all at once. The second number displayed is the *commit limit*, which displays the maximum virtual memory usage that the system can support before running out of virtual memory (it includes virtual memory backed in physical memory as well as by the paging files). The commit limit is essentially the size of RAM plus the current size of the paging files. It therefore does not account for possible page file expansion.

Process Explorer’s System Information display shows an additional item of information about system commit usage, namely the percentage of the peak as compared to the limit and the current usage as compared to the limit:





It was one of those things that I was interested as we were studying chapter 8 and 9, and all these materials had to do with what we are doing in this lab. Studying this week’s chapters, page file had my interests right away; I wanted to know more about it. Potential page and commit limit was something I thought it was interesting. I had no idea that commit limit, which displays the maximum virtual memory usage that the system can support before running out of virtual memory. Reading about it this week I knew that the commit limit is essentially the size of RAM but I was surprised as well to find out that it was the current size of the paging files as well. Working with windows task manager for a while now I think I can say that I have learned enough about it. And this was my goal actually because I never knew one thing that is part of it. Also System Explorer is something that I will keep in my laptop, and definitely try to learn more things around it. It does have my interest now after doing this lab. This goes to every single program that I have installed. I thought that this lab was more fun than the last lab, maybe because I already have some experience from this lab.

I am really glad that I did both labs. Before starting the first lab I did not know absolutely anything when it came to these operating system tasks. I am thankful that I did this lab as well because it gave me a great starting point.