DEPARTAMENTO
DE INFORMÁTICA
UNIVERSIDAD DE ALMERÍA

Individual assignment 3

Memory (and disk) investigation with Sysinternals: VMMap and RAMMap.

prof: José Antonio Álvarez Bermejo*
joseantonio@ual.es

Devices and Interfaces
Escuela Superior de Ingeniería

Deadline: *June, 1st to 5th, 2023*

Resumen

In topic 3, we have studied how the L3 cache (LLC) is essential for the system to boot. We temporarily convert it into RAM. We needed a space to build a stack and place data and instructions in order to activate microcontrollers within the chipset and finally initialize the memory controller. In topic 4, we have studied the importance of the chipset in controlling the use of RAM (through the memory controller, which helps the MMU obtain the actual physical address where the data is located). And this is because our memory is paged and segmented. Remember that transitioning from *real mode* to *protected mode* was crucial, necessary, and had to be done as quickly as possible. In topic 5, we have seen disk access and how the *After Life* preserves access to them; we have also seen how HDD and SSD disks differ. Memory (topic 4) and disk (topic 5) are closely linked: pagination and virtual memory. In this activity, you will learn more about your memory system as managed by Windows. To do this, we will use the **Sysinternals** tools, which you should already be familiar with, such as **ProcessExplorer**. In this activity, we will work with VMMap and RAMMap.

Keywords: SRAM, RAM, DDR, Virtual Memory, Paged Memory, HDD, SSD.

1. How Windows Manages Our Memory

Memory (RAM) and disk (HDD or SSD) are intimately linked through the concept of virtual memory (and paging).

2. Virtual Memory

Although memory management depends on the system, it is true that virtual memory is a concept that is understood and conceived in the same way in both Windows and Linux. Virtual memory is a layer of abstraction between your applications and the actual physical RAM. Virtual memory *has nothing to do with page files*, but when configuring the page file, it is interesting to note that in Windows, this configuration is referred to as *virtual memory*. Virtual memory will always be there, even if you disable the page file. Without virtual memory, a developer's job would be a disaster because they would have to literally comb through the RAM for each process to find a (sufficiently large) space to accommodate the process. Thanks to virtual memory, each process is presented with an identical address space. Each process can write to address zero, and the memory manager will take care of finding a suitable space in RAM. Figure 1 shows how the available 4GB is distributed for a process. By default, a process sees 2GB of system space and 2GB of personal space. We can increase this space to 3GB at the expense of reducing the space that **all processes** share to 1GB: **/3GB or /USERVA** are options in Windows to modify the default configuration for space allocation to processes.

Why switch to 64-bit? Because having, for example, 1GB of system space and 16GB of installed RAM would mean that almost all of the system

*If you encounter any difficulties or have any questions you can use, at any time, the e-mail, discord or aulavirtual.

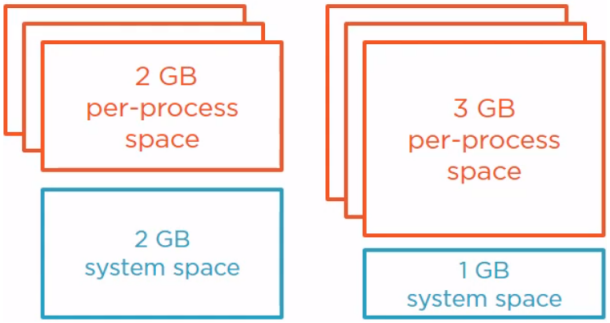


Figura 1. Available Memory

memory would be used to store page tables to address all that RAM. Therefore, it is better to have more space for the operating system (and by extension, for processes). If processes have the "large address space aware"flag enabled, they can opt for 3GB on 32-bit systems or up to 4GB on 64-bit systems. In Figure 2, it can be observed that even though 2^{64}

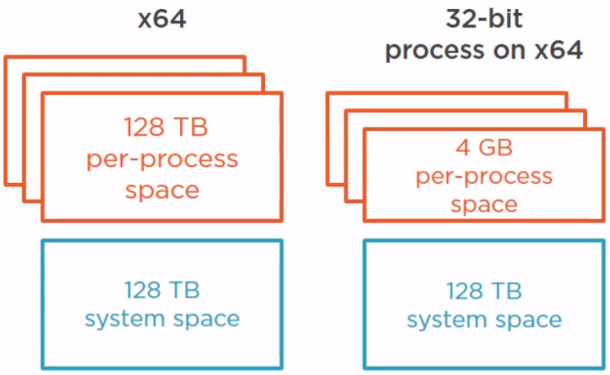


Figura 2. Memoria disponible 32 vs 64

= 17,179,869,184 GB, current architectures do not support more than 48 bits, which is 262,144 GB or 256 TB (for x64) or 50 bits, which is 1,048,576 GB or 1024 TB (for IA-64). The operating system on these architectures also plays a role in determining the available memory for your applications or devices because until Windows 8 and Windows Server 2012, only up to 44 bits could be addressed. Windows 2012 R2 fixed this issue. Therefore, it is important to know the version of the operating system you have because it can affect your available RAM. This effect can be seen in the following Figure 3. First, we can see the available free memory for the process **notepad.exe**, which is 2GB, compared to the memory consumption of the same process on a 64-bit machine with Windows 8 and Windows 8.1.

The application used to analyze this information is

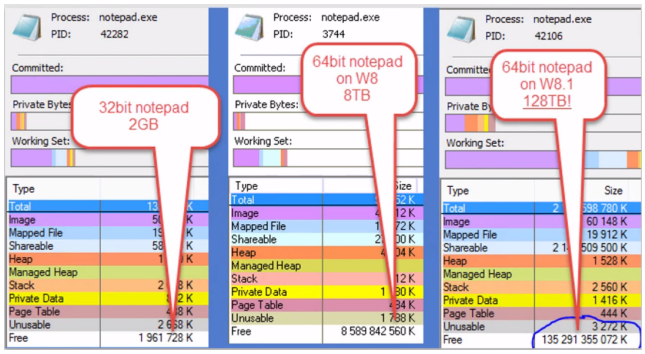


Figura 3. Different memory consumptions

called **VMMap**. Once this is understood, two questions need to be addressed (typical of operating systems):

- Do threads execute in the kernel space or the user space?
- Is memory accessible from the kernel space or the user space?

Threads can execute in both contexts, and they can even be switched from one to the other. By switching from user to kernel mode, they can access kernel memory instead of user memory.

2.1. Things to know about Windows

Windows has demand-paged memory management, which means:

- Processes request memory when they need it. Processes start with almost no assigned memory.
- There is no swapping.
- The page size is 4KB (8KB on Itanium), but larger pages can be requested (remember the concept of PAE that we saw).
- Pages have a maximum alignment of 64KB, which can lead to memory fragmentation.

Regarding virtual memory, it has two states: reserved or in-use. Additionally, the address space is divided into two parts:

- Private (process-specific space, such as the **heap**). It can be either reserved or in-use.
- Shareable (exe, dll, shared memory, etc.). Space that can be shared among multiple processes.

When you first start the notepad.exe, it has to be loaded from disk. However, once it is open, subsequent instances do not need to be loaded from disk because they already have the **shareable** pages in memory, thanks to the previous instance, and the executable has not changed. If the executable does change (e.g., due to changes in its global variables), the copy-on-write technique is used.

2.2. Common RAM Issues

The most common issue is **memory leaks**, as we saw in theory. An example of this is when a process leaks private memory. One of the consequences of this is that the process keeps requesting more and more RAM without proper control, leading to system instability.

2.3. System Commit Limit

In Windows, the System Commit Limit refers to the amount of virtual memory that the operating system is obligated to provide to processes. In fact, if we translate "System Commit"to Spanish, it would mean something like .ºbligation of the system."The resources used to fulfill this obligation of providing memory come from physical memory or the page file (paging). For example, when processes request private space, if you don't have enough RAM and don't have a page file, the application will fail to start. Another example is supporting shared memory among multiple processes. Copy-on-write pages (we saw this concept in topic 4) also require a response from the system when memory is requested. Read/write pages are used when loading information into a process, and the system's code and data, both paged and nonpaged, also contribute to the System Commit. When the limit is reached, virtual memory fails. There are two values related to the commit:

- System Commit Limit: When the operating system reaches this limit, processes start to fail, and the information (data) they hold may become corrupted.
- Current Commit Charge:

2.3.1. Changing the System Commit Limit

There are several ways to change the System Commit Limit. Firstly, you can increase the amount of RAM, or you can expand the size of the page file. The commit can grow automatically if the page file is configured to increase automatically or if the available space on the hard disk has not been exhausted. But how do you configure it optimally? This limit should be able to handle peak workload (remember the benchmark tests from topic 1). The higher the commit limit that can be served by RAM, the better.

2.4. Backups

The **backup system** for paged memory (subject to Commit) is the page file, which is a secure area for the private memory of the respective process. The **storage for paged and mapped files in memory** is the files themselves. One of the misconceptions heard in the world of Windows is that disabling the page file eliminates paging, but that's not true. Disabling the page file only eliminates paging in the page file itself. This will cause the paging of mapped files in memory to increase because now they have to handle paging for private areas as well.

2.5. Where is the page file located?

Well, it's in the system registry. After startup, the session manager creates the key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles.

- Windows supports up to 16 page files.
- On an x86 architecture, the file size can go up to 4095MB.
- On an x64 (or x86 with PAE), it can go up to 16TB.

If you want to modify the page file configuration (in 99 % of cases, the default configuration is optimal), in specific cases where, for example, the system load never changes, it could be modified to adapt the page file. Another case is when the system runs only one application for which the manufacturer recommends not using the page file. Security experts may also configure the page file to prevent program memory from being paged to a file. Low disk space on an SSD is another case where page file configuration may need to be adjusted.

2.5.1. How to measure I/O on the page file?

The performance does not depend on the size of the page file but on the speed of the page file. How do you know if you need it? In case you don't need it, should you delete the page file? The answer is **NO**. The first

thing you should do is make sure that the page file is actually being used because Windows does not use it unless it is truly necessary. For this purpose, you can use tools (from Sysinternals) such as *Resmon, Procmon, or WPT (Windows Performance Toolkit)*. If you don't want to use any of these tools, you can use Perfmon with a little trick that involves moving the page file to a separate partition where only the page file is located, and then use *perfmon* to measure access to that partition. With *ResMon*, you will have the information shown in Figure 4.

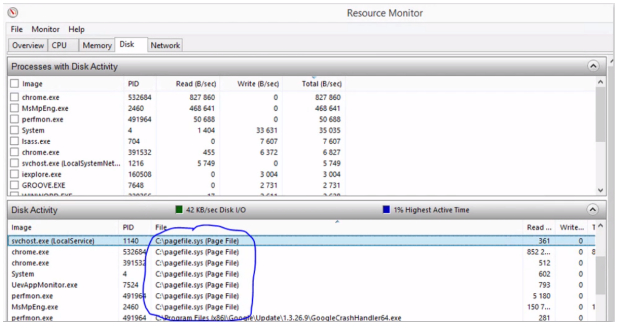


Figura 4. Resmon and the page file

If you still want to optimize, it's because:

- If you are using the page file a lot: **add more RAM**. If you consider adding a SATA SSD, keep in mind that it is 60 times slower than RAM. If you are thinking about an NVMe SSD that uses PCIe, remember that it can read up to 4GB per second, but currently they are around 2GB per second. RAM, on the other hand, can move up to 60GB per second.
- Use a fast USB (ReadyBoost).
- Add a dedicated and exclusive page file (you can have up to 16) on a private I/O channel if:
  - I/O with the page file is the bottleneck.
  - You cannot add more RAM.
  - For this, an fast and small SSD would be ideal.
  - If you have multiple page files, Windows will use the one with the shortest I/O queue.
- Place the page file **on the most used partition of the least used drive**.

Be careful when deleting the page file from C: because memory dumps cannot be performed correctly when the system fails. If you remove it to avoid this, you need to locate the registry key *dedicated memory dump file* and specify where the memory dump should be saved. In Figure 5, we can see how the system works with a page file, where the process has its virtual memory, part of which is in RAM. Regarding the private part of the process, we can see that part is in RAM and the other part is in pages of the page file.

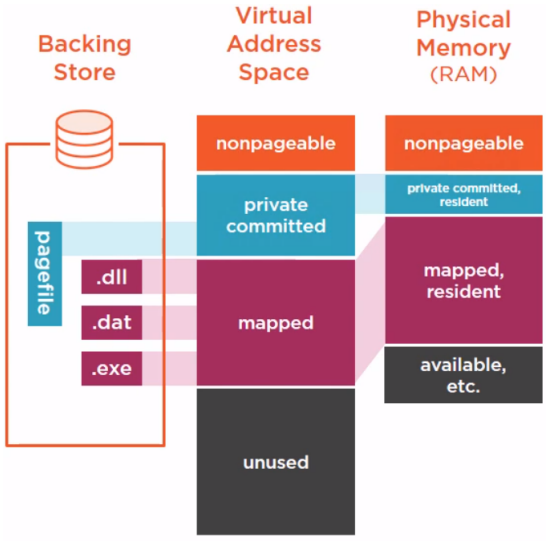


Figura 5. With page file

However, if we decide not to have a page file, the situation changes as shown in Figure 6. The process's virtual memory, unable to find pages to store private memory, is forced to keep everything in the RAM associated with the process, draining the space. Therefore, not all mapped files can reside in RAM, and we will have to go **to the disk to retrieve them**, increasing the execution time.

The purpose of the page file is to have enough memory to serve the commits (requests) we have referred to as the System Commit Charge. If this System Commit Charge reaches the System Commit Limit, processes will fail, crash, or hang. Having a page file also supports memory dumps generated in the event of a system failure, and for this, we need to be able to accommodate as much memory as the RAM plus 257 MB. It also provides a means to move less frequently used pages to disk, which is slower.

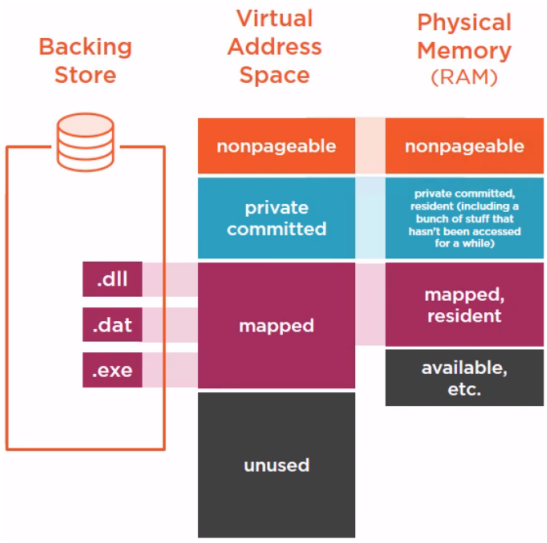


Figura 6. Without page file

Exercise 1: How much space does your kernel occupy? What about your page file? To determine the size of your system's kernel, download **notmyfault.exe** from Sysinternals (notmyfault page) and trigger a crash on your machine. Check the file size and ensure that your page file can accommodate it.

2.6. Page File and SSD

In Windows 10 with SSDs, the situation changes a bit, and it is not recommended to write to the page file due to what we have studied in Topic 5 (wear control). Starting from Windows 10, all applications have a private "store"that is stored precisely in the system process's address space (in the user area). In Figure 7, we see that when a working set is removed from RAM, it remains in a list of modified elements because it may have data that has not been saved to the page file yet. After a while, it should be written to the disk and remain in the disk cache to return to the standby list if someone needs it, thus being reused.

But starting from Windows 10, there is an additional step. In that space, the pages from the "modified list.[a]re stored compressed instead of going to disk. And even after compressing the page, if we still need to send it to the page file, it is moved to the "modified"list and then written to disk. This means that we perform fewer writes and reads on the SSD compared to not having the orange box on the right in Figure 8, which appears starting from Windows 10.

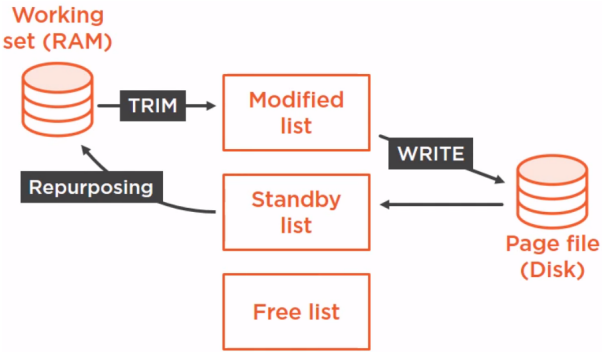


Figura 7. Antes de Windows 10

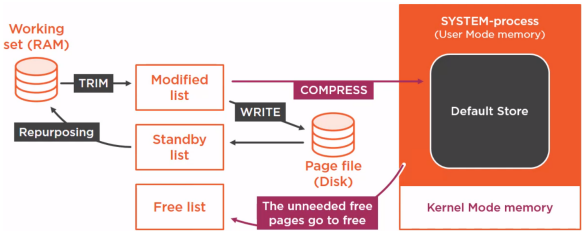


Figura 8. After Windows 10

2.7. Analyze your virtual memory

Download VMMAP from Sysinternals: link to the website

If you launch VMMAP, you can analyze the Commit values we mentioned, which represent the *amount of available virtual memory* for a specific process. Figure 9 shows how you can launch, for example, *notepad.exe*.

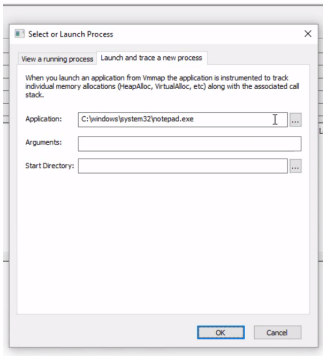


Figura 9. Launch notepad from VMMAP

Exercise 2: Determine how much addressable memory your system has (*Task Manager > Performance*), and now compare the addressable memory with the virtual memory available for the process you just launched (*notepad.exe*). You can find this information in the *FREE* section. Refer to Figure 7 and Figure 8.

As an example, on a 32-bit machine with 8 GB of RAM, you will see that the process only uses 4 GB (3.9), while on a 64-bit machine with 4 GB of RAM, the process uses 4 GB of RAM. Where does the difference lie? If I launch notepad on the 64-bit machine, I will have 128 TB of virtual memory available. On the 32-bit machine, I will have 2 GB of RAM available. Having a 64-bit system does not imply having more memory, as in both cases I will use 2 to 4 GB (if large space aware is included).

Exercise 3: Launch Excel in VMMAP. Excel is a process with *large space aware*. **How much memory has been assigned to it? Why?**

Exercise 4: Launch notepad.exe in VMMAP with 32 bits, use syswow64 which is the 32-bit subsystem To do this, you need to use the path c:\windows\syswow64\notepad.exe **How much memory has been assigned to it? Why?**

As you have seen, there is a significant difference between 32-bit and 64-bit applications, as well as between 32-bit and 64-bit operating systems. In this section, you will use the following tools:

- **processexplorer**: You will be able to see the memory consumption of each process you launch, including private memory, system memory, virtual memory, etc. (process explorer page)
- **testlimit**: You will use this tool to create a process that consumes memory (either virtual or physical, as you choose). You will use three options: **-r, -c**, and **-d**. When you launch this process, you can monitor it in process explorer and gather information about its memory usage from the task manager. (testlimit page)
- **procmon**: (process monitor page)
- **NotMyFault**: (not my fault page)
- **Resmon.exe**

Sure, I can launch either *testlimit* or *testlimit64*. Since you have a 64-bit machine, it's preferable to use *testlimit64*. Now you will see the difference between launching it with one or the other. Let's launch *testlimit*, which is the 32-bit version, to demonstrate how it works. You should have Process Explorer open at the same time to observe memory consumption. To launch *testlimit*, you can use the command line (cmd).

Reserve 1GB

testlimit -r 1024 -c 1 This command means that we are launching a program that **reserves** 1024 MB once. Pay attention to the line in Process Explorer that identifies this process. Look at the *Private Bytes* value, which represents the process's private area in physical RAM, and it indicates that it only uses 908 K. The **Working Set** (reserved space) is 4 GB (2 GB for the system and 2 GB for the process). The **Virtual size** indicates that it has 1 GB in virtual memory, while **WS Private** is only 600 K of real RAM. We have reserved 1 GB of RAM, but it's clear that we are not using that space. **Close with Ctrl+C**

Reserve all possible memory

testlimit -r Since it's a 32-bit program, it will reserve up to 2 GB. The data is identical to the previous example, except now we have seen the maximum limit that a 32-bit process can reserve (look at the **Virtual size** column). **Close with Ctrl+C**

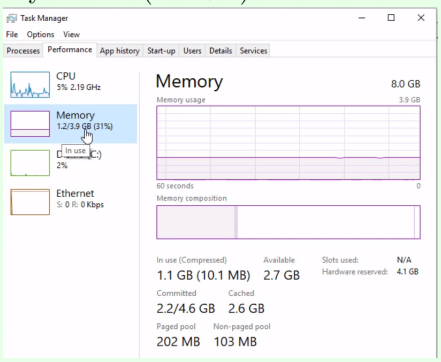
Can't close it?

Exercise 5: Why do you think you can't close the *testlimit -r* process?

Now, pay attention, instead of reserving memory, we are going to *allocate (commit) private memory.*

Allocate private memory

testlimit -m 1024 -c 1 This command requests 1 GB of memory to avoid running out (we only have two to request). Now you can see the difference in Process Explorer. In the **Private Bytes** column, you have 1 GB, yet the **WS Private** (working set private) is very small (600 KB). If you open the Task Manager for this example, you will see that it indicates we have only used 1.2 GB out of 3.9 GB available. Look at the *Committed memory* section in the Task Manager, it now shows 2.2 GB out of 4.6 GB available. What happens when you close (Ctrl+C) testlimit?



Now, the next step is to use the memory we allocated (touch it):

Touch memory

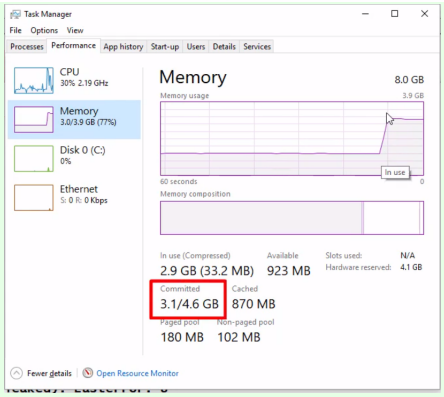
testlimit -d This command reserves memory in 1 MB increments and touches it. It indicates that it stops at 2 GB. If you check the Task Manager now, you will see two things: an increase in Committed memory and an increase in used memory. If you look at Process Explorer, you will see that the reserved 2 GB are now being used (Private Bytes and WS Private).

```
C:\Windows\system32>testlimit -d

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

Process ID: 4836

Leaking private bytes with touch (MB)...
Leaked 1977 MB of private memory (1977 MB total leaked). Lasterror: 8
```



As an additional experiment, what would happen if we open another command prompt and launch another *testlimit*? If you keep the Task Manager visible with the current values (77 % of memory used, which is 3.0 GB out of 3.9 GB), and 3.1 GB out of 4.6 GB *committed* memory. **Also, take note that it indicates you are using 3.0 GB in use (Compressed 33.4 MB)**, which is what we saw before Windows 10. You have 33.4 MB compressed. Okay, the command we are going to run is **testlimit -m** , and you will notice that the Committed memory limit has increased to 5.0 GB out of 6.1 GB available. This is due to the behavior of the pagefile, which is configured to grow automatically. Although this is not an exercise for submission, you can search for your pagefile. Go to *System Properties* > *Advanced* > *Performance Settings* > *Advanced* > *Change*. In the virtual memory window, you will see that the option .Automatically manage paging file size for all drivesïs checked. And if we want, we can make it increase further. If we open another command prompt and launch another **testlimit -m**, if we go back to the virtual memory window (where you saw the checkbox .Automatically manage paging file size for all drives"), you will see that the allocated memory size at

the bottom of that window has increased. After these three **testlimit** commands, we can see the current state of our memory: we are using 2.5 GB out of 3.9 GB.

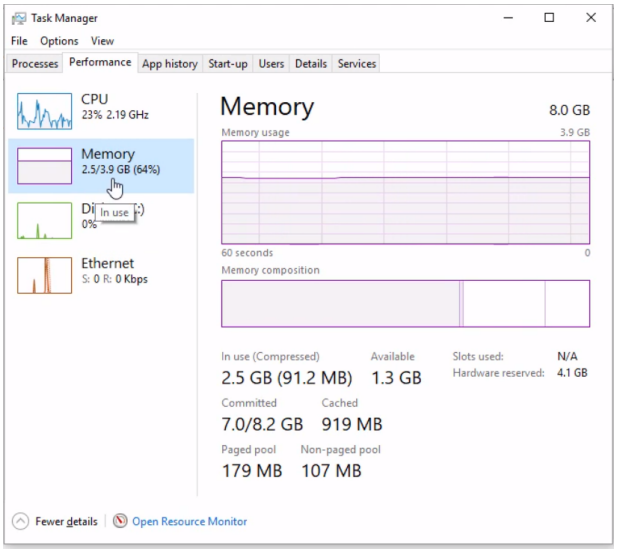


Figura 10. Current memory consumption

In this situation, we are going to put more strain on the machine. We will launch another **testlimit**, but this time with the -d flag to reserve and use memory: **testlimit -d**. In my case, you can see how we have increased the pagefile size to 8.9 GB, and we have the limit set at 10 GB. And we could continue like this until we exhaust the disk space or reach the maximum limit of the pagefile size.

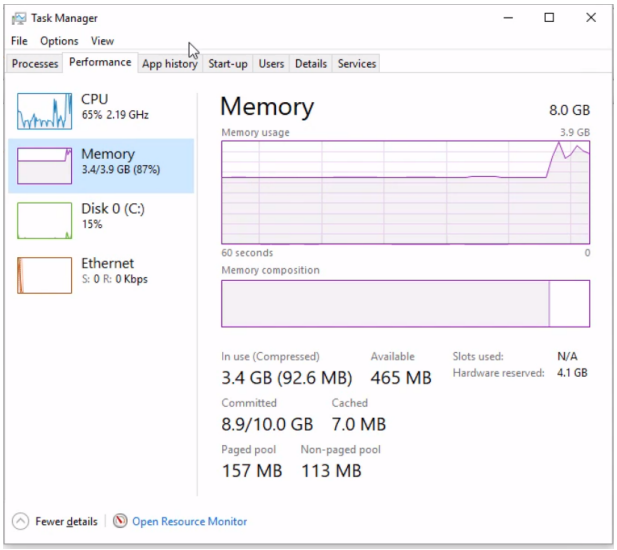


Figura 11. Forzando la máqunia

Exercise 6: Open Task Manager. Open Command Prompt and explain why we run out of memory when launching **testlimit -m 1024 -c 3**.
Now run **testlimit64 -m 1024 -c 3**. Why does it work now? Observe the *Committed* memory value in Task Manager.
Close the previous command.
Launch **testlimit -d 1024 -c 3**. What happened? Does the pagefile come into play?
Do not close this process and continue reading.

If we use the Windows Resource Monitor (resmon.exe) and open the Memory tab as shown in Figure 15, we can observe the **testlimit64** process as well as **Memory Compression**, which indicates compressed memory (as we saw at the beginning of the document). It is important to observe the green bar, which represents memory in use. We can see that while the command is running, there are hard page faults, indicating that the system needs to write data to disk. We also notice that as we occupy more memory, the system starts evicting pages from other processes to make room for our demands.

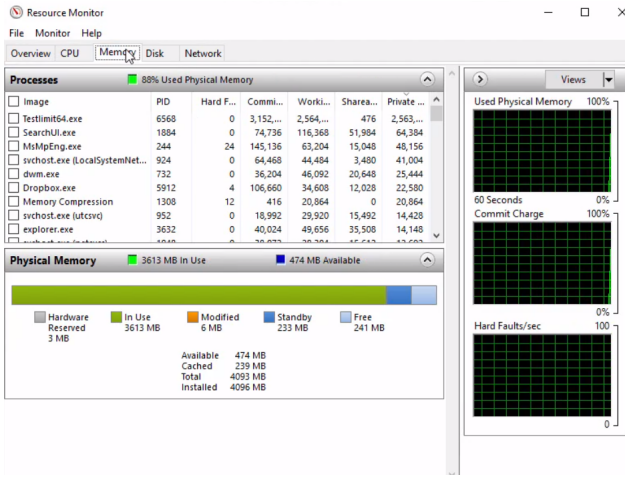
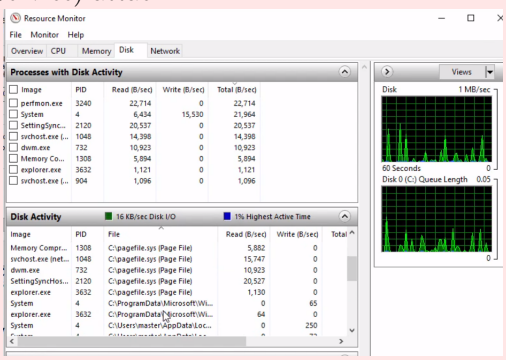


Figura 12. Resmon

Using the **NotMyFault64.exe** tool, we can generate more memory demand in the system. With this tool, we can provoke demand for paged or non-paged memory (i.e., memory that goes directly into the RAM of the process). Launch **NotMyFault64** (keep in mind that you are putting a heavy load on your memory controller, and the system may become slower), go to the **leak** tab, and set Leak/second to 20,000 KB. Then select *Leak Nonpaged* to allocate memory directly to RAM.

Exercise 7: Explain what happened after running *notmyfault64* and waiting for a few minutes.
Explain, based on the image, why all processes are writing to the pagefile.
You have experienced a self DoS (Denial of Service) attack.



As a curiosity, you can use **procmon** with filters to focus only on the activity of *pagefile.sys*. This way, when there is paging activity, you can quickly detect it, as shown in Figure 17.

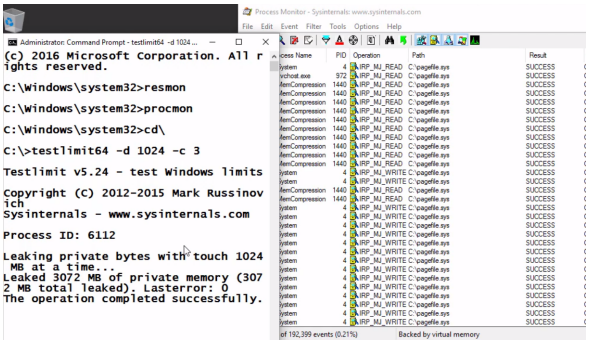


Figura 13. procmon

3. Physical Memory

In the previous section, we mentioned the concept of the *Working Set*, which refers to the set of physical pages that belong to a specific process. Initially, it is empty, but as the process starts requesting pages, they are brought from the disk to the working set located in RAM. This means that anything in the working set can be referenced with a *soft fault*, which means the page is retrieved from memory, as opposed to *hard faults*, which involve accessing the disk and require disk I/O. Pages that are very old are recycled for other processes. When a process is created, it has a minimum value and a maximum value for its working set. The minimum value controls the maximum number of locked pages (reserved for your use, and the operating system takes care of their management). The maximum value is not used for anything nowadays. If there is enough memory to hold the working set in RAM, then everything the process needs will be in RAM. If things get complicated in terms of space, then the working set will start getting trimmed by sending out pages (see Figure 3). The working set consists of two types of pages:

- Shareable: These are pages from the executable, for example, from *notepad.exe*, as they can be used by another instance of *notepad.exe* to launch itself, saving memory and improving performance.
- Private: These are pages private to the process.

Pages that have been evicted from the working set (see Figure 7 and Figure 8) are no longer considered part of it, even if they are still in cache. The concept of **Copy-On-Write**, which has been mentioned before, refers to when a process writes to a shared page with another process. In that case, the page is copied before modification and given to the other process (see Figure 18).

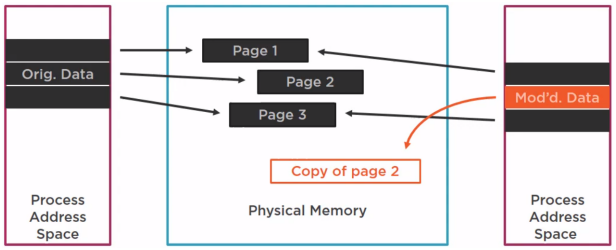


Figura 14. Copy on write

Pages that are not allocated are in one of these lists:

- Free pages list.
- Modified pages list.
- Standby pages list: with eight priority levels.
- Zeroed pages list.
- ROM pages list.
- Bad pages list.

When a page is requested, it is retrieved from the free pages list or the zeroed pages list and added to the working set. Pages that are removed from the working set go to either the modified pages list or, if

they haven't been modified (determined by the dirty bit), to the standby pages list. The pages in these lists still remember which process they belong to. These two lists (modified and standby) form the **system cache**. Pages in the modified list take time to be written back to the disk; they are only written if necessary (e.g., when the list reaches a certain size). Windows considers everything in the system cache to be potential candidates for reuse by processes.

The free pages list is used for page reads. If something needs to be brought into a process (e.g., I/O from a file), it is allocated from this list. Modified pages from a process are also returned to this list when the process terminates. These pages contain data and are not zeroed. The zeroed pages list is used when someone requests a page that should not contain any previous content. The zeroed pages list steals pages from the free pages list and formats them to zero when it has more than eight pages.

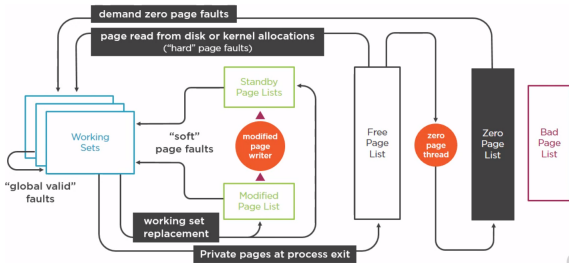


Figura 15. Trasiego de páginas

The private page space of a process does not provide information about the pages in the standby or modified lists, nor does it provide information about the page tables. However, this can be analyzed using the RAMMAP tool.

One fascinating feature of Windows (starting from Windows Vista) is **SuperFetch**, a system memory manager capable of repopulating RAM with optimal pages for each moment, not only with the appropriate priority level but also understanding when to load pages for Outlook binaries even before you try to launch Outlook. For example, SuperFetch understands that you use email from Monday to Friday and adapts the content of RAM accordingly. The priorities in the standby list are as follows:

- Priority 7: Pretrained static set by Microsoft, loaded at ***system startup***. These are pages that require ultra-fast responses, such as double-clicking, the Start menu, etc.
- Priority 6: Pages loaded by **SuperFetch**.
- Priority 5: Normal priority for user processes.
- Priority 1: Low priority.

3.1. Hidden Memory

There are elements that remain hidden in RAM and are not easy to see. These include:

- Reserved memory: Once reserved, even if you haven't used it as intended, it is added to the page table, occupying additional space.
- Shared memory: Shared memory that is only used by one process and does not appear as private memory.
- Locked pages (VirtualLock): Memory that is managed independently of the operating system.
- Memory locked by drivers: For example, Hyper-V, VMware, etc., which lock memory and make it unavailable for use.
- AWE memory: Address Windows Extensions, used, for example, by SQL Server.

3.2. Studying RAM

If you open the Resource Monitor, go to the Memory tab, and pay attention (see Figure 19), you will see the amount of memory in each of the mentioned lists. However, there is a lot of memory that does not appear here.

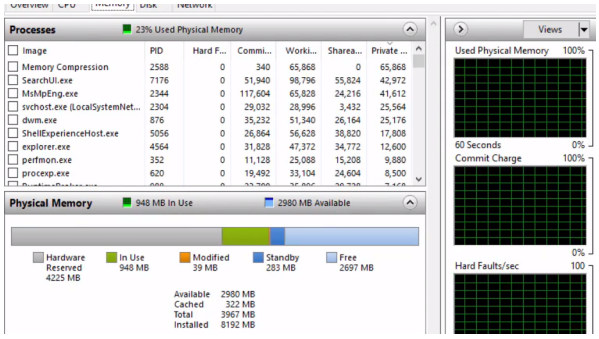


Figura 16. Resmon and memory

Figure 19 shows a snapshot of a 32-bit Windows 10, which can only use 4GB of RAM. That's why 4GB is reserved for hardware. Then, in green, you have the memory in use, which is 934MB out of the 4GB. There are 39MB in the modified pages list (memory that has not yet been moved to the pagefile or mapped files) and 284MB in the standby list or cache (which has 8 priorities). It is easy to demonstrate that a system needs more RAM by checking (in the Resource Monitor) the *Memory > Hard Faults/sec*

tab. If there are multiple processes with sustained values, it means that the processes are causing disk activity. If it's only one process, then it's probably a poorly designed application. If, as shown in Figure 20, we click on the Commit bar, we can see different memory details, such as the amount of memory in the standby list (grouped by priorities). Another important value is the kernel size (because the page table is stored there).

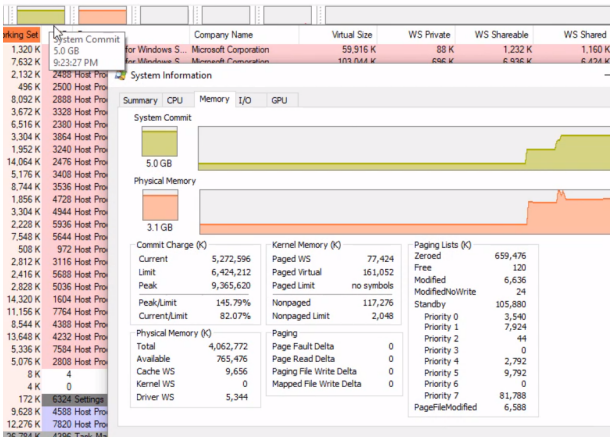


Figura 17. Memory

In this standby list, if we allocate excessive memory, we can see how priorities 0 to 5 disappear to make space for the process that claims all that memory.

Exercise 8 Launch **notmyfault64.exe** on your machine and run *Leak > 20000 KB > Leak nonpaged*. Observe what happens to the value of hard page faults in the Resource Monitor and explain why this happens.
Priority list: Observe what happens to it and explain what is happening.

Exercise 9 (Hidden Memory, Page Table) Open Process Explorer and the Command Prompt (cmd). Run **testlimit64 -r**. Go to the process details in Process Explorer. Analyze what happens to the Virtual Size space, and if you click on the memory window to view the values of that process, you will see that Nonpaged memory keeps increasing. What is happening? (hint: page table)

Exercise 10 (Hidden Memory, Shared Memory) Run **testlimit64 -s 2048 -c 1**. Look at Process Explorer. You just reserved 2GB for this process, but where are they?

Attention, if you have hibernation issues: Hidden memory in VirtualLock Run **testlimit64 -v 2048 -c 1**. Now you are acting like VirtualBox, for example, using 2GB of RAM, but not making it visible to the system. Look at Process Explorer. You just reserved 2GB for this process, and they appear in Private Bytes, Working Set, and Virtual Size. If there are processes that do not release this memory, you may have problems shutting down properly. In the Task Manager (Details tab), it shows that the process consumes 2GB, but we don't see any information about the type of memory it is.

Figure 18 shows how memory is displayed in RamMAP, and it is interesting to see the consumption of Driver Locked Memory (which is HyperV) when running two virtual machines.

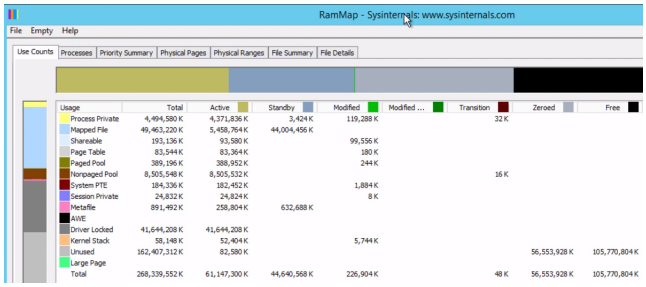


Figura 18. RAMmap

Translate the following LaTeX from Spanish to English:

3.3. Hidden Memory

There are elements that remain hidden in RAM and are not easy to see. These are:

- Reserved Memory: once reserved, even if you haven't used it as you should, it is added to the page table, occupying additional space.
- Shared Memory: shared memory used by a single process that does not appear as private memory.
- Locked Pages (VirtualLock): memory that is managed independently of the operating system.

- Memory locked by drivers: for example, Hyper-V, VMware, etc., that lock memory and make it unavailable for use.

- AWE Memory: Address Windows Extensions, used by SQL Server, for example.

3.4. Studying RAM

If you open the Resource Monitor, go to the Memory tab, and pay attention (see Figure 19), you will see the amount of memory allocated in each of the mentioned lists. However, there is still a lot of memory that does not appear here.

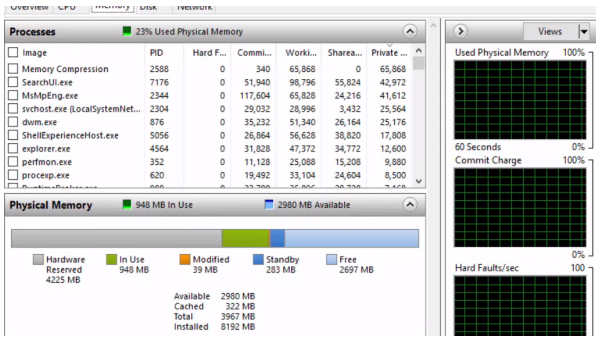


Figura 19. Resmon and Memory

Figure 19 shows a snapshot of a 32-bit Windows 10 system that can only use 4GB of RAM. That's why it shows 4GB reserved for hardware. Then, in green, you have the memory in use, which is 934 MB out of 4GB. 39MB from the modified page list (memory that has not yet been moved to the pagefile or mapped files) and 284MB from the standby list or cache (which had 8 priorities). It is easy to demonstrate that a system needs more RAM by looking at the *Memory > Hard Faults/sec* tab in the Resource Monitor. If there are several processes with values (sustained over time), it means that the processes are causing disk usage. If it's only one process, it is likely an application with many flaws. If, as shown in Figure 20, you click on the Commit bar, you can see different memory-related aspects, such as the amount of memory in the standby list (grouped by priorities). Another important value is the size of the kernel (as the page table is stored there).

In this standby list, if we allocate excess memory, we can see how priorities 0 to 5 disappear to make room for the process that claims all that memory.

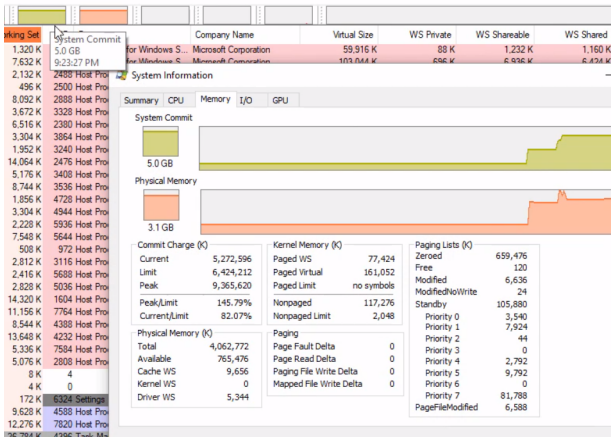


Figura 20. Memory

Exercise 8 Run **notmyfault64.exe** on your machine and launch *Leak > 20000 KB > Leak nonpaged*. Observe in the Resource Monitor what happens to the value of hard page faults and explain why this happens.
Priority list: Observe what happens to it and explain what is going on.

What happens on an SSD when a write operation begins? Before delving into this topic, it should be noted that NAND flash is organized into 4KB pages, which are further organized into blocks ranging from 64 to 1024 pages. A block is the smallest unit that can be erased. Before writing anything to a block, it needs to be erased. So, if the block is empty, the write operation is super fast, but if it's not, then the content of the block needs to be copied to the disk's RAM (remember that we learned that SSDs have internal RAM). Then the block has to be erased, the copied data in RAM needs to be updated with the new content you want to write, and the entire block needs to be written again from RAM, making the write operation slow. Writing to an empty block is 1023 times faster than doing it on a block with content. **WARNING: SSDs become slower as they start to fill up with data**.

Let's do the math: An MLC SSD with 128 pages per block would have a total of 2,048 blocks. If we write 1 block per second, it would take 23.7 days to fill up the disk. If the disk is full and we can only use one block, at that rate, we would fill up that block in a little over 15 minutes.

That's why when you buy a 1TB SSD, you ac-

tually get 1.2TB because they provide an extra 20 % of blocks to compensate. Along with wear leveling circuitry (which we discussed in topic 5), this is done to extend the lifespan of the disk.

Another achievement of SSDs is that, since they work with *pages*, they can perform similar operations to what the operating system does with pages in RAM. Internally, the disk tries to determine which pages (within the blocks) can be reused to optimize its performance (this process, called TRIMMING in RAM, is also used by SSDs, but internally, from their circuitry). The operating system is unaware of this internal SSD policy regarding page reuse.

DEFRAGMENTATION: One of you asked me about defragmentation on SSDs. You don't need to disable defragmentation in Windows because Windows recognizes that it's an SSD and instead of defragmenting, it performs TRIMMING. It doesn't defragment but rearranges the pages (this operation is usually done every 28 days).

Exercise 11. Run DiskSpeed and test how your disk performs with the tests you send it. Read here about how to use DiskSPD. An example configuration to test your disk or compare it against another disk is:

- Blocksize: 4KB
- Read/Write: 20 operations
- Out. IO: 8
- Access type: Random
- Workers: 1
- Duration: 20 secs.

Another excellent tool is DiskView from Sysinternals.

3.4.1. Disk Performance Analysis

The first tool to use is Resource Monitor. In the main tab, you can monitor disk activity. However, you can see more detailed information in the Disk tab. Normally, people pay attention to the % Active Time column (which indicates the percentage of time the disk is active), but this is not a real indicator. To identify if the disk is a bottleneck, you should look at the *Disk Queue Length* column. A normal queue should be at 1 or below 1 most of the time. If you want to take a closer look at how disks behave, you can download the tool called iometer (iometer.org). This tool is almost a standard when it comes to analyzing disk operation speed. However, I recommend using (I use it before iometer) **DiskSpeed** (you can find its source code at view Diskspd source code or download it from the Microsoft website (Microsoft website to download Diskspd). The only issue is that it is a PowerShell tool (you launch it as .\DiskSpeed.ps1).