From Memory To Disk And Back:

How Pages Of Memory Move Through The Virtual Memory State Machine

Landon Moceri

Preface:

My name is Landon Moceri, and I am a CS student who has dedicated the past year to crafting a multi-threaded memory manager for Windows machines. This program seizes control of a pool of pages (4096-byte chunks) from a computer's memory and manages them using virtual addresses, just as an operating system would. Operating in user mode, it leverages Windows' memory API to allocate and map the physical memory in the system. My state machine does all other work, including maintaining page tables, aging and trimming virtual addresses, maintaining a page file, resolving page faults, and much more. This project is meant to recreate a modern operating system created by companies such as Microsoft and Apple. I have spent the past year learning about the intricacies of how operating systems work and have tried my best to recreate the work done by these commercial operating systems. So far, I've seen great success with all of the functionalities that I've implemented.

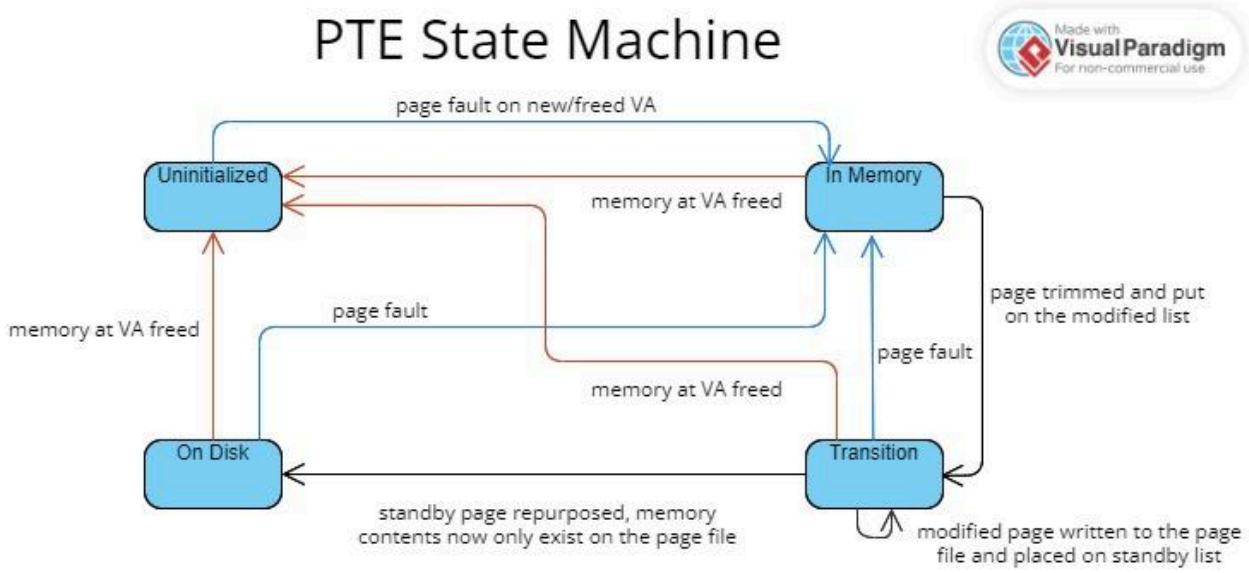
Virtual memory is a necessary component of any modern operating system (OS), and it exists so that an operating system can effectively manage physical memory, a limited resource, as virtual addresses that can have varying protections and other elaborate features in a portable manner. Some sort of security system needs to exist to safeguard a program's memory access, and most importantly, memory needs to have a stable address, which a physical address (frame

number) cannot offer. This is due to the process of memory paging, in which inactive (cold) memory is written to a "page file" on the disk, and active (hot) memory is kept in the computer's RAM. This creates a hierarchy system similar to the one that already exists between CPU cache and memory, in which a computer can utilize its memory to the maximum potential and be provided with extra space from its hard drive if needed. Moreover, it creates a digital ecosystem in which programs that exhibit "good behavior," that is - freeing memory when done with it and frequently accessing allocated memory - are rewarded with fast, physical memory, while programs with "bad behavior," such as forgetting to free memory after use or very rarely accessing it are allowed to exist without dealing harm to the system by giving them space on the page file. This data on the page file has to be loaded back into memory to be accessed, so it is possible for the physical address corresponding to a virtual address (VA) to change. The ultimate job of a virtual address is to provide this stable identifier for each page of memory.

Tree data structures known as page tables are used to accomplish this task. Page tables start with a register inside the CPU called the page table base register. This register stores the physical address of the page table base, which serves as the tree's root node. The physical page it points to is full of addresses of child nodes, and in turn, those addresses are filled with the addresses of their own child nodes. This pattern is repeated, and leaf page tables are eventually reached, which serve as the final layer before the leaf nodes. These page tables hold the addresses of Page Table Entries (PTEs). PTEs correspond with virtual addresses and store specific information about them, such as the physical address or location on the page file it points to, a valid bit to indicate whether it is an active page in memory, permission bits, and

much more. As a page is 4096 bytes and each PTE is the size of a register (8 bytes), there are 512 PTEs per page table. In my memory manager's implementation, PTEs have four states. The first of which is the "uninitialized" state. An uninitialized PTE corresponds to a VA that has either never been accessed or one that has been previously accessed and freed. These two conditions look identical to the memory manager. A PTE will be completely zeroed in this case, indicating its emptiness. Once a VA is initialized, its PTE will enter the "active" state. Its valid bit will be set to one, and its frame number entry will be filled with the page given to the VA by the memory manager. A PTE's valid bit must be set in order to be accessed by the CPU, and the CPU gives any PTE with a zero frame number back to the memory manager to resolve, which is referred to as a "Page Fault." This includes access to an uninitialized PTE. If a program attempts to access a VA associated with a non-active PTE, its PTE must be converted to the active format before the CPU can use it to generate a physical address. As memory fills up in a computer, the operating system starts to iterate through each active PTE, increasing a field in its PTE corresponding to its age. By "aging" PTEs and resetting their age when accessed, the OS is able to find the coldest memory, which it then "trims" by zeroing its valid bit and placing its page on a "modified" list. According to demand, pages from this list are written to the page file, and their state is changed to "standby." Both standby and modified pages still exist in memory, so their PTEs are put into "transition" format, indicating that their page still exists in memory. Even though standby pages exist in two different places, the memory location is still stored in the PTE so that it can be accessed quickly until the last possible moment. Both active and transition PTEs have frame numbers, while the former always has a set valid bit, and the latter's bit is always

zeroed. It is important to note that a CPU can only access an active PTE. Even though a transition format PTE corresponds to a page still in memory, the CPU will not access it due to its zeroed valid bit, which will be signaled to the memory manager. If enough stress is put on the system, standby pages will be repurposed for VAs requiring physical pages. The old page gets zeroed or overwritten and given to a new active PTE, while the old transition PTE loses its page and is converted to a “disk” state PTE. Its valid bit stays zeroed, and its frame number is replaced with a location on the page file (disk index). The VA always remains the same, and the PTE will be converted back into the active state with a new physical page copied from the page file if reaccessed. PTEs can be thought of as similar to entries in a phone book. You may switch houses (physical addresses), but you’ll always have the same identity (virtual address), and each entry in the address book (page tables) corresponds to a person. You are only taken out of the phone book if you pass away (free memory). Unlike an address book, however, a set number of VAs exist on any given system. Once their associated memory is freed, they can be used again by programs.



Note: The operating system must hand out a valid VA within the predefined virtual address space so a thread can access it. Any attempts to access a VA that is "out of bounds" of the virtual address space will result in an access violation exception.

Types of PTE Formats and Their Bit Values

Uninitialized	Active	Transition	Disk
Valid Bit: (Always Zero)	Valid Bit: (Always One)	Valid Bit: (Always Zero)	Valid Bit: (Always Zero)
Frame Number: (Always Zero)	Frame Number: (Can Be Anything Except for Zero)	Frame Number: (Can Be Anything Except for Zero)	Disk Index: (Can Be Anything)

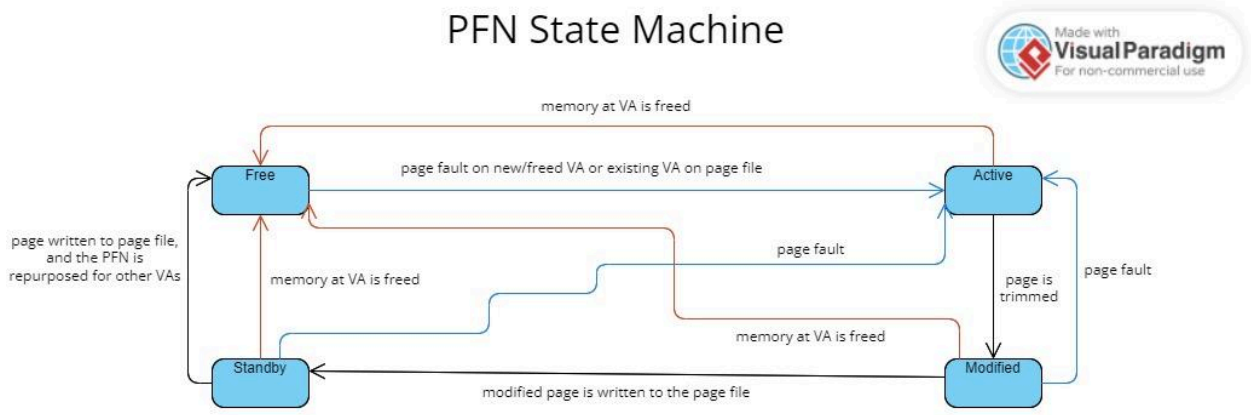
Rest of Bits: (Always Zero)	Age Bits: (Can Be Anything)	On Disk (Differentiating) Bit: (Always Zero)	On Disk (Differentiating) Bit: (Always One)
	Accessed Bit: (Can be Anything)		
How To Differentiate: The entire value will be zero	How To Differentiate: The valid bit will be one	How To Differentiate: The valid bit and On Disk bit will be zero, and the frame number will be nonzero	How To Differentiate: The On Disk bit will be one, and the valid bit will be zero

Note: The accessed bit will be described at a later point.

Note 2: The “Differentiating Bit” differentiates a Transition and Disk Format PTE, as their values can be identical, and the memory manager needs to be able to tell them apart. Additionally, a frame number of zero must not be given to PTEs; otherwise, a transition and uninitialized PTE could be identical.

Another data structure, referred to as a PFN (Page Frame Number), is needed to store data regarding physical pages. Pages need to be linked together in doubly linked lists, as pages of the same state need to be stored together and able to be removed freely. In order to link them together, PFNs need to have a forward link (flink) and backward link (blink) field. In addition, PFNs need to keep track of their state and PTE, hold individual locks for multi-threading, and store any additional information that is too large to fit in the PTE. As a PFN exists solely on the operating system's behalf, contrary to a PTE that the CPU can read, a PFN can exceed the size of a register. All PFNs start in the "free" state, where their physical pages are unowned by a VA. When a VA is faulted on for the first time, the OS gives a free page to its PTE. This transforms the PFN into an "active" state. Like a PTE, freeing the page's VA will result in the page being returned to the free state. Free pages are linked together to form a "Free Page List" from which a memory manager can pull. By the time the free list starts running out, PTEs will have been aged and then eventually trimmed by the operating system. This trimming process converts active PFNs to the "modified" state, keeping their associated data in memory while marking it as ready to be written to the page file. After they are written, their state is changed again to "standby," indicating that they are ready to be repurposed. At this point, a page's contents exist in two places: in memory and on the page file. The frame number is kept in the PTE, as it is quicker to read and write to memory than the page file, and the disk index is stored in the PFN, using it as a buffer until its PTE is fully converted to disk format. If this conversion happens, the physical page and its PFN are zeroed and given out to a new PTE, erasing any association with its old PTE. One important thing to note about standby pages is that if a page held in the page file is not

written to when made active by the memory manager, then its PFN will jump right back to the standby state, as it won't need to be written out again..Think of PFNs like plates at a restaurant. They stay with customers (VAs) as long as they are using them, and then they are washed (zeroed) and reused by the restaurant (memory manager) when another customer needs a plate. Unlike a real restaurant, however, this analogical restaurant takes plates from customers between bites of food (memory accesses), stores their food in a back room (page file), and is quick enough to return the plates when they next take a bite. The customer never knows that their plate has been used by someone else.



Now that I have laid the groundwork for understanding the virtual memory state machine, I can discuss the strategies I've used to implement it effectively. In this paper, I will be detailing how my state machine moves. Specifically, I will touch upon how memory is initially materialized, aged, trimmed, written to the page file, and read back in a page fault. As I am still

actively working on this project, the methods discussed in this paper are subject to change and are only accurate to the date this paper was published.

In regards to designing strategies, one key idea that needs to be remembered is the avoidance of deadlock. This means that different types of locks, such as PTE and PFN locks, must be acquired in the same order. If one thread acquires a PTE lock and tries to acquire a PFN lock, and the other has acquired a PFN lock and is trying for a PTE lock, then they will reach a situation where neither thread can progress forward. This causes my program to need to hold an "order of operations" in which certain locks must be accessed before others. As handling a page fault (finding the data associated with a PTE) is the core task of a memory manager, PTE locks must be put at the top of the order of operations. PFN locks must come second and linked lists third, as to add or remove a PFN from a linked list, the memory manager first has to lock it, and PFNs are accessed after PTEs and before linked lists in the logic of the page fault handler. While a system that orders these locks differently could exist, it would be more deadlock-prone, and the work it would have to do to avoid these deadlocks would be like swimming against a current.

To start, a strong aging algorithm is needed to ensure a plentiful and differentiated amount of trimming candidates in the event that more memory is needed than is currently available. While this action is associated with PFNs that relate only to physical pages in memory, it is actually done to PTEs instead. This is because multiple PTEs can point at a single PFN if memory is shared between virtual addresses. If two addresses share a PFN, one could be accessing it frequently enough to keep it at a very low age, and another could barely access it,

putting it at a high age. So, to maintain the validity of ages in the memory manager, PTEs must be aged instead of the PFN. PTEs are each given three age bits to represent eight different ages. This number is chosen because it meets a beneficial point in the tradeoff between space and the number of ages. The eight possible ages are enough to provide a meaningful difference between each group, and the three bits are a relatively small amount of data that can easily fit into a PTE. In order to achieve an ideal set of ages, each active PTE needs to be iterated through and its age bits incremented. This is a tremendous amount of work that needs to be done proactively, so naturally, it makes sense to do it on separate threads and optimize it as much as possible. In my program, one thread is responsible for aging the PTE space. My program has one lock per page table or 512 PTEs, which I also refer to as a "PTE region." This is useful while aging, as my aging function can acquire one lock per region and then iterate through all the PTEs in that region and age them, doing up to 512 times the work while still only having to acquire one lock. To speed this up, my program keeps track of a count of active pages in the region. That way, my program can prematurely stop traversing a region if I have already aged all of its active pages. However, the most beneficial change to this method is using a bitmap (array of single bits) to correspond to whether each PTE is active. This benefit is enormous, as the CPU can read 64 bits of this bitmap at a time, so in the best-case scenario, the aging thread reduces 64 PTE reads that would have been done into one single read.

While these strategies certainly help with the speed of aging, they don't do anything to manage how frequently it is done. To determine how frequently to age, my program measures

how many pages were consumed from the free or standby lists per *n* milliseconds, where *n* is the arbitrary number of milliseconds between when the aging thread will wake up. The thread uses this rate to calculate a heuristic of when the system will run out of memory. The idea is that the thread will age the PTE space once for each age between the current time and when the free/standby pages run out. The thread will adjust its heuristic accordingly if the rate speeds up or slows down. It will separate its work into one increment for each age and then further separate each increment into a fraction of each second that the thread needs to work for. This divides the time the thread works so it does not take over the CPU. If it needs to do 10 seconds of work over 100 seconds, it will space it out to do 0.1 seconds of work every second. This would prevent it from working 10 seconds straight and killing performance by taking valuable resources from other threads when they're not immediately needed. It is also important to spread aging apart as much as possible, as spacing it out destroys differentiation by leaving large durations where no aging occurs. This process will create a natural hierarchy in which a uniform distribution of ages will form so that the memory manager can pick the oldest candidates when needing to trim memory. I'm actively working on these aging changes but have only implemented parts of them since my performance traces have indicated much larger problems.

Once memory has been aged, the trimming process can pick the best possible candidates. Similarly to aging, I have decided that trimming merits its own thread because it is better to be unbounded by other actions or threads, as it is extremely important that trimming happens when needed and is not held up. Trimming is a less intensive process, although it is meant to be done

preemptively. It works by locating the oldest aged pages in the PTE space, invalidating their valid bit to protect them from being accessed in their transition state by the CPU, and moving their PFNs to the standby or modified list. This sounds like it requires another iteration of each PTE or PTE region, although the aging thread's work can be capitalized to avoid this iteration. While aging, the thread will separate its count of active pages in each region into individual counts for each age. This allows each region's count to be directly checked for pages of the desired age without looking over each PTE. To speed this up even further, a flink and blink can be added to each region, and they can be chained together into doubly linked lists of each age, with the list a region is on corresponding to its highest aged PTE. This allows the thread to instantly find a region containing pages of the highest age, cutting the O(n) time, where n corresponds to the number of PTEs in the system it will have to search, in its worst case, down to one of O(512), where 512 is the number of PTEs in a page table it will have to search in its worst case. Similarly to the aging thread, this trimming thread will use the same heuristic to determine how many pages it needs to trim to maintain the free list and then separate the count into how many need to be trimmed per second. With this foundation in place, my modified page writer should have the strongest possible candidates to write to the page file.

Writing modified pages to the page file seems easy, but there is one key concept to consider. Writing to the disk takes a long time (one Microsoft figure states that the average write takes 10ms), and this task would stall the system tremendously if not compartmentalized. Therefore, I also chose to give modified writing its own thread. As mentioned earlier, PFNs must

be accessed before page lists to avoid deadlock. The modified page writer takes in a linked list of pages and requires individual PFNs to be removed, violating the aforementioned order of operations. A more complex strategy must be employed to remove the pages from the list while holding the right locks to resolve this. First, the page list of interest is locked, which in this case would be the modified list. The head of the list is peeked at, and its PFN is captured. The thread then tries to lock the page out of order, stopping if another thread already locks it. If it has acquired the lock, then it has both locks, and the order in which it got them stops mattering. At this point, it is free to go on with both locks. If the attempt to get the lock fails, then the thread needs to relinquish the list lock and retry the same process until it works, with the idea being that the thread holding the list head's lock will have relinquished it or removed it from the list by the time the modified writing thread comes back around. After getting a batch of modified pages, the thread simply writes them to the page file and updates their PFN with the location where they were written. As the PTE state and frame number don't need to be changed and the active bit remains zeroed, the modified writer doesn't actually need to look at the PTE. After the pages have successfully been written out, the thread changes the state of the PFNs to standby, and they are ready to be consumed by other VAs.

Another change that can improve the performance of modified writing is employing the strategy of referencing. By adding a reference bit to my PFNs, my memory manager can track when another thread is "referencing" that PFN. This allows my modified writing thread to relinquish the PFN locks it holds while writing to the disk, allowing the pages to be accessed in a

period where faulting threads would previously have had to wait several milliseconds for a disk write to complete. If the data associated with that PFN is written to, the memory manager can check if the modified writer is referencing the page and invalidate its contents on the page file as it is being written to. Additionally, the memory manager must know to wait until a PFN being written is no longer being referenced so that it is not prematurely repurposed if it is freed. With this new bit, the memory manager will no longer have to wait for disk writes and can devote resources to more critical tasks. This is the change that I'm currently working on debugging. Once it's implemented, I expect a massive decrease in the runtime of my program's tests.

Different Threads In My Program

Faulting Threads	Aging Thread	Trimming Thread	Modified Writing Thread
N amount, where *N* is the amount of user threads currently faulting on a VA	Currently, one thread, Ages all pages in the VA space to create a differentiated set of ages	Currently, one thread, looks for the highest-aged pages to put on the modified list where they'll be written to the page file	Currently, one thread, Writes pages on the modified list to the page file and changes their pages in memory to the standby state

Writing to the page file also seems like a trivial task, but by examining the work that needs to be done more closely, one can realize what makes it necessary to optimize. To write to

the page file, the program must first find an open spot. As the page file can be larger than a system's RAM, the memory manager could search many gigabytes of space for one empty page. Some data structure is needed to track which pages are used, as there needs to be a quick way to verify that no VA is using a page. Otherwise, the concept of a page file will fall apart. To do this effectively, my program utilizes another bitmap. Each bit corresponds to one page on the page file and whether a PTE points to it. This is the ideal solution to map out the page file, as it minimizes its pages into the smallest possible searchable set. Once a free spot is found, its index in the bitmap is returned, equal to the index of its corresponding page in the page file to which the memory will be written. While this seems fast enough, a thread requesting a free disk index could still have to search millions of bits just to find one free space. To optimize this further, the bitmap can incorporate interlocked operations to go completely lock-free. This is important, as the page fault handler also uses this bitmap when converting Disk PTEs to active PTEs to let programs reaccess them. Erasing this lock contention resulted in a tremendous performance increase in my program. A count of total free spots can also be added so that threads do not scan over a full bitmap or try to get more spots than are available. In addition, a stack can be implemented to keep track of recently freed disk indexes, which can occur when a page is read back into memory or when memory is freed. That way, the memory manager can avoid having to traverse the bitmap unless necessary and capitalize off of previously done work. After all, a memory manager is supposed to impact a computer's resources as little as possible. This stack can also be used while traversing the bitmap if extra free spots are found. For example, if the function finds five empty spots after searching for two milliseconds, the memory manager can

save itself eight milliseconds (two per spot) by doing a tiny bit of extra work to put those four extra spots on the stack. It is crucial, though, to cap the size of this stack to minimize the program's memory footprint, as the stack would be 64 (size of the indices in bits) times larger than the page file itself. Finally, the program can keep track of its last checked index in the bitmap as it searches from one direction to the other. This can help the function not repeatedly check the same bubble of full spots over and over again. However, it is crucial to ensure that this spot is updated if a disk index is freed. If the stack of freed spots is full and the disk index is less than the one recorded as the last checked index, the newly freed index will sit inside the bubble of skipped active spots at the beginning. All of these additional strategies can also be done using interlocked operations. Implementing these strategies was the single most beneficial thing I've done in my project. Before implementing them, my performance tests showed that 55% of the program's time was spent looking for disk indices. After implementation, this number skyrocketed down to 0.03%.

One final strategy that can be used to speed up the memory manager is to speculatively load in subsequent VAs after faulting on or accessing one. The idea behind this is that a lot of the time, programmers will be linearly accessing data. In this case, speculatively faulting on subsequent PTEs will prematurely load in data that will be accessed in the following few instructions, minimizing the wait time experienced by a program. This is done the same way a regular page is loaded, except its "accessed bit" will not be set. When accessing a PTE, the CPU sets this bit, and the memory manager unsets it when the valid bit is zeroed. Additionally,

speculative PTEs are loaded in at a high age so that they will be selected as ideal trim candidates if not used. This allows the fault handler to look nearby at the PTEs surrounding its current PTE and load in more or less of these pages depending on their ages and whether they were loaded speculatively. This strategy offers tremendous benefits, as it will lessen the perceived “wait time” that a faulting thread must wait before its page can be read from the page file if the data is accessed linearly.

As you can see, managing memory quickly becomes incredibly complex as additional layers of complexity are added for efficiency. While countless additional strategies can be implemented, there is a certain point where they can take up too much memory or time and become infeasible. I would like to clarify that I am not just adding layers of complexity for the fun of it: each change is carefully deliberated and implemented for a specific problem that I’ve noticed during testing. While understanding how memory works at a deep level is incredibly useful to have learned, this project has benefitted me much more by teaching me how to write production-grade code. What I’ve really learned from this experience is how to test and improve code by using tools at my convenience instead of wildly speculating with only ideas. With the help of programs such as WinDBG and XPerf, I’ve learned to carefully analyze my code, pinpoint weak points, create strategies to address them, quantify their benefit, and fine tune them to provide the maximum possible advantage. While issues in a high school level CS project can be resolved by print statements and reading through the code, this approach does not fly with a

complex state machine and I've learned to adapt my skills accordingly in a way where I now feel I have the tools to address any problem. What this project has really taught me, is how to think.