Design elements of the FreeBSD VM system

Matthew Dillon

dillon@apollo.backplane.com

This article was originally published in the January 2000 issue of DaemonNews (http://www.daemonnews.org/). This version of the article may include updates from Matt and other authors to reflect changes in FreeBSD's VM implementation.

The title is really just a fancy way of saying that I am going to attempt to describe the whole VM enchilada, hopefully in a way that everyone can follow. For the last year I have concentrated on a number of major kernel subsystems within FreeBSD, with the VM and Swap subsystems being the most interesting and NFS being "a necessary chore". I rewrote only small portions of the code. In the VM arena the only major rewrite I have done is to the swap subsystem. Most of my work was cleanup and maintenance, with only moderate code rewriting and no major algorithmic adjustments within the VM subsystem. The bulk of the VM subsystem's theoretical base remains unchanged and a lot of the credit for the modernization effort in the last few years belongs to John Dyson and David Greenman. Not being a historian like Kirk I will not attempt to tag all the various features with peoples names, since I will invariably get it wrong.

Table of Contents

1 Introduction

Before moving along to the actual design let's spend a little time on the necessity of maintaining and modernizing any long-living codebase. In the programming world, algorithms tend to be more important than code and it is precisely due to BSD's academic roots that a great deal of attention was paid to algorithm design from the beginning. More attention paid to the design generally leads to a clean and flexible codebase that can be fairly easily modified, extended, or replaced over time. While BSD is considered an "old" operating system by some people, those of us who work on it tend to view it more as a "mature" codebase which has various components modified, extended, or replaced with modern code. It has evolved, and FreeBSD is at the bleeding edge no matter how old some of the code might be. This is an important distinction to make and one that is unfortunately lost to many people. The biggest error a programmer can make is to not learn from history, and this is precisely the error that many other modern operating systems have made. Windows NT® is the best example of this, and the consequences have been dire. Linux also makes this mistake to some degree—enough that we BSD folk can make small jokes about it every once in a while, anyway. Linux's problem is simply one of a lack of experience and history to compare ideas against, a problem that is easily and rapidly being addressed by the Linux community in the same way it has been addressed in the BSD community—by continuous code development. The Windows NT folk, on the other hand, repeatedly make the same mistakes solved by UNIX® decades ago and then spend years fixing them. Over and over again. They have a severe case of "not designed here" and "we are always right because our marketing department says so". I have little tolerance for anyone who cannot learn from history.

Much of the apparent complexity of the FreeBSD design, especially in the VM/Swap subsystem, is a direct result of having to solve serious performance issues that occur under various conditions. These issues are not due to bad algorithmic design but instead rise from environmental factors. In any direct comparison between platforms, these issues become most apparent when system resources begin to get stressed. As I describe FreeBSD's VM/Swap subsystem the reader should always keep two points in mind. First, the most important aspect of performance design is what is known as "Optimizing the Critical Path". It is often the case that performance optimizations add a little bloat to the code in order to make the critical path perform better. Second, a solid, generalized design outperforms a heavily-optimized design over the long run. While a generalized design may end up being slower than an heavily-optimized design when they are first implemented, the generalized design tends to be easier to adapt to changing conditions and the heavily-optimized design winds up having to be thrown away. Any codebase that will survive and be maintainable for years must therefore be designed properly from the beginning even if it costs some performance. Twenty years ago people were still arguing that programming in assembly was better than programming in a high-level language because it produced code that was ten times as fast. Today, the fallibility of that argument is obvious—as are the parallels to algorithmic design and code generalization.

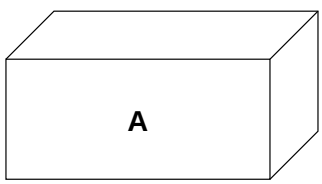
2 VM Objects

The best way to begin describing the FreeBSD VM system is to look at it from the perspective of a user-level process. Each user process sees a single, private, contiguous VM address space containing several types of memory objects. These objects have various characteristics. Program code and program data are effectively a single memory-mapped file (the binary file being run), but program code is read-only while program data is copy-on-write. Program BSS is just memory allocated and filled with zeros on demand, called demand zero page fill. Arbitrary files can be memory-mapped into the address space as well, which is how the shared library mechanism works. Such mappings can require modifications to remain private to the process making them. The fork system call adds an entirely new dimension to the VM management problem on top of the complexity already given.

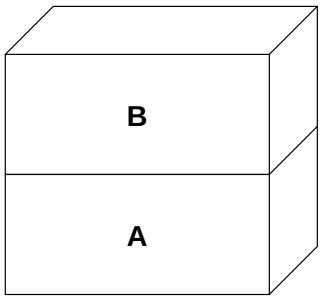
A program binary data page (which is a basic copy-on-write page) illustrates the complexity. A program binary contains a preinitialized data section which is initially mapped directly from the program file. When a program is loaded into a process's VM space, this area is initially memory-mapped and backed by the program binary itself, allowing the VM system to free/reuse the page and later load it back in from the binary. The moment a process modifies this data, however, the VM system must make a private copy of the page for that process. Since the private copy has been modified, the VM system may no longer free it, because there is no longer any way to restore it later on.

You will notice immediately that what was originally a simple file mapping has become much more complex. Data may be modified on a page-by-page basis whereas the file mapping encompasses many pages at once. The complexity further increases when a process forks. When a process forks, the result is two processes—each with their own private address spaces, including any modifications made by the original process prior to the call to `fork()`. It would be silly for the VM system to make a complete copy of the data at the time of the `fork()` because it is quite possible that at least one of the two processes will only need to read from that page from then on, allowing the original page to continue to be used. What was a private page is made copy-on-write again, since each process (parent and child) expects their own personal post-fork modifications to remain private to themselves and not effect the other.

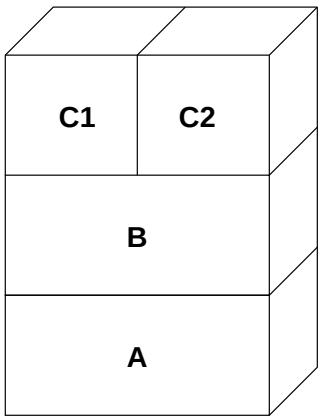
FreeBSD manages all of this with a layered VM Object model. The original binary program file winds up being the lowest VM Object layer. A copy-on-write layer is pushed on top of that to hold those pages which had to be copied from the original file. If the program modifies a data page belonging to the original file the VM system takes a fault and makes a copy of the page in the higher layer. When a process forks, additional VM Object layers are pushed on. This might make a little more sense with a fairly basic example. A `fork()` is a common operation for any *BSD system, so this example will consider a program that starts up, and forks. When the process starts, the VM system creates an object layer, let's call this A:



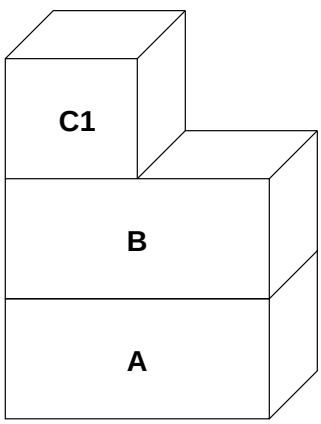
A represents the file—pages may be paged in and out of the file's physical media as necessary. Paging in from the disk is reasonable for a program, but we really do not want to page back out and overwrite the executable. The VM system therefore creates a second layer, B, that will be physically backed by swap space:



On the first write to a page after this, a new page is created in B, and its contents are initialized from A. All pages in B can be paged in or out to a swap device. When the program forks, the VM system creates two new object layers—C1 for the parent, and C2 for the child—that rest on top of B:



In this case, let's say a page in B is modified by the original parent process. The process will take a copy-on-write fault and duplicate the page in C1, leaving the original page in B untouched. Now, let's say the same page in B is modified by the child process. The process will take a copy-on-write fault and duplicate the page in C2. The original page in B is now completely hidden since both C1 and C2 have a copy and B could theoretically be destroyed if it does not represent a "real" file; however, this sort of optimization is not trivial to make because it is so fine-grained. FreeBSD does not make this optimization. Now, suppose (as is often the case) that the child process does an `exec()`. Its current address space is usually replaced by a new address space representing a new file. In this case, the C2 layer is destroyed:



In this case, the number of children of B drops to one, and all accesses to B now go through C1. This means that B

and C1 can be collapsed together. Any pages in B that also exist in C1 are deleted from B during the collapse. Thus, even though the optimization in the previous step could not be made, we can recover the dead pages when either of the processes exit or `exec()`.

This model creates a number of potential problems. The first is that you can wind up with a relatively deep stack of layered VM Objects which can cost scanning time and memory when you take a fault. Deep layering can occur when processes fork and then fork again (either parent or child). The second problem is that you can wind up with dead, inaccessible pages deep in the stack of VM Objects. In our last example if both the parent and child processes modify the same page, they both get their own private copies of the page and the original page in B is no longer accessible by anyone. That page in B can be freed.

FreeBSD solves the deep layering problem with a special optimization called the "All Shadowed Case". This case occurs if either C1 or C2 take sufficient COW faults to completely shadow all pages in B. Lets say that C1 achieves this. C1 can now bypass B entirely, so rather then have C1->B->A and C2->B->A we now have C1->A and C2->B->A. But look what also happened—now B has only one reference (C2), so we can collapse B and C2 together. The end result is that B is deleted entirely and we have C1->A and C2->A. It is often the case that B will contain a large number of pages and neither C1 nor C2 will be able to completely overshadow it. If we fork again and create a set of D layers, however, it is much more likely that one of the D layers will eventually be able to completely overshadow the much smaller dataset represented by C1 or C2. The same optimization will work at any point in the graph and the grand result of this is that even on a heavily forked machine VM Object stacks tend to not get much deeper then 4. This is true of both the parent and the children and true whether the parent is doing the forking or whether the children cascade forks.

The dead page problem still exists in the case where C1 or C2 do not completely overshadow B. Due to our other optimizations this case does not represent much of a problem and we simply allow the pages to be dead. If the system runs low on memory it will swap them out, eating a little swap, but that is it.

The advantage to the VM Object model is that `fork()` is extremely fast, since no real data copying need take place. The disadvantage is that you can build a relatively complex VM Object layering that slows page fault handling down a little, and you spend memory managing the VM Object structures. The optimizations FreeBSD makes proves to reduce the problems enough that they can be ignored, leaving no real disadvantage.

3 SWAP Layers

Private data pages are initially either copy-on-write or zero-fill pages. When a change, and therefore a copy, is made, the original backing object (usually a file) can no longer be used to save a copy of the page when the VM system needs to reuse it for other purposes. This is where SWAP comes in. SWAP is allocated to create backing store for memory that does not otherwise have it. FreeBSD allocates the swap management structure for a VM Object only when it is actually needed. However, the swap management structure has had problems historically.

Under FreeBSD 3.X the swap management structure preallocates an array that encompasses the entire object requiring swap backing store—even if only a few pages of that object are swap-backed. This creates a kernel memory fragmentation problem when large objects are mapped, or processes with large runsizes (RSS) fork. Also, in order to keep track of swap space, a "list of holes" is kept in kernel memory, and this tends to get severely fragmented as well. Since the "list of holes" is a linear list, the swap allocation and freeing performance is a non-optimal O(n)-per-page. It also requires kernel memory allocations to take place during the swap freeing process, and that creates low memory deadlock problems. The problem is further exacerbated by holes created due to the interleaving algorithm. Also, the swap block map can become fragmented fairly easily resulting in non-contiguous allocations. Kernel memory must also be allocated on the fly for additional swap management structures when a swapout occurs. It is evident that there was plenty of room for improvement.

For FreeBSD 4.X, I completely rewrote the swap subsystem. With this rewrite, swap management structures are allocated through a hash table rather than a linear array giving them a fixed allocation size and much finer granularity. Rather then using a linearly linked list to keep track of swap space reservations, it now uses a bitmap of swap blocks arranged in a radix tree structure with free-space hinting in the radix node structures. This effectively makes swap allocation and freeing an O(1) operation. The entire radix tree bitmap is also preallocated in order to avoid having to allocate kernel memory during critical low memory swapping operations. After all, the system tends to swap when it is low on memory so we should avoid allocating kernel memory at such times in order to avoid potential deadlocks. Finally, to reduce fragmentation the radix tree is capable of allocating large contiguous chunks at once, skipping over smaller fragmented chunks. I did not take the final step of having an "allocating hint pointer" that would trundle through a portion of swap as allocations were made in order to further guarantee contiguous allocations or at least locality of reference, but I ensured that such an addition could be made.

4 When to free a page

Since the VM system uses all available memory for disk caching, there are usually very few truly-free pages. The VM system depends on being able to properly choose pages which are not in use to reuse for new allocations. Selecting the optimal pages to free is possibly the single-most important function any VM system can perform because if it makes a poor selection, the VM system may be forced to unnecessarily retrieve pages from disk, seriously degrading system performance.

How much overhead are we willing to suffer in the critical path to avoid freeing the wrong page? Each wrong choice we make will cost us hundreds of thousands of CPU cycles and a noticeable stall of the affected processes, so we are willing to endure a significant amount of overhead in order to be sure that the right page is chosen. This is why FreeBSD tends to outperform other systems when memory resources become stressed.

The free page determination algorithm is built upon a history of the use of memory pages. To acquire this history, the system takes advantage of a page-used bit feature that most hardware page tables have.

In any case, the page-used bit is cleared and at some later point the VM system comes across the page again and sees that the page-used bit has been set. This indicates that the page is still being actively used. If the bit is still clear it is an indication that the page is not being actively used. By testing this bit periodically, a use history (in the form of a counter) for the physical page is developed. When the VM system later needs to free up some pages, checking this history becomes the cornerstone of determining the best candidate page to reuse.

What if the hardware has no page-used bit?

For those platforms that do not have this feature, the system actually emulates a page-used bit. It unmaps or protects a page, forcing a page fault if the page is accessed again. When the page fault is taken, the system simply marks the page as having been used and unprotects the page so that it may be used. While taking such page faults just to determine if a page is being used appears to be an expensive proposition, it is much less expensive than reusing the page for some other purpose only to find that a process needs it back and then have to go to disk.

FreeBSD makes use of several page queues to further refine the selection of pages to reuse as well as to determine when dirty pages must be flushed to their backing store. Since page tables are dynamic entities under FreeBSD, it costs virtually nothing to unmap a page from the address space of any processes using it. When a page candidate has been chosen based on the page-use counter, this is precisely what is done. The system must make a distinction between clean pages which can theoretically be freed up at any time, and dirty pages which must first be written to their backing store before being reusable. When a page candidate has been found it is moved to the inactive queue if

it is dirty, or the cache queue if it is clean. A separate algorithm based on the dirty-to-clean page ratio determines when dirty pages in the inactive queue must be flushed to disk. Once this is accomplished, the flushed pages are moved from the inactive queue to the cache queue. At this point, pages in the cache queue can still be reactivated by a VM fault at relatively low cost. However, pages in the cache queue are considered to be "immediately freeable" and will be reused in an LRU (least-recently used) fashion when the system needs to allocate new memory.

It is important to note that the FreeBSD VM system attempts to separate clean and dirty pages for the express reason of avoiding unnecessary flushes of dirty pages (which eats I/O bandwidth), nor does it move pages between the various page queues gratuitously when the memory subsystem is not being stressed. This is why you will see some systems with very low cache queue counts and high active queue counts when doing a `systat -vm` command. As the VM system becomes more stressed, it makes a greater effort to maintain the various page queues at the levels determined to be the most effective. An urban myth has circulated for years that Linux did a better job avoiding swapouts than FreeBSD, but this in fact is not true. What was actually occurring was that FreeBSD was proactively paging out unused pages in order to make room for more disk cache while Linux was keeping unused pages in core and leaving less memory available for cache and process pages. I do not know whether this is still true today.

5 Pre-Faulting and Zeroing Optimizations

Taking a VM fault is not expensive if the underlying page is already in core and can simply be mapped into the process, but it can become expensive if you take a whole lot of them on a regular basis. A good example of this is running a program such as ls(1) or ps(1) over and over again. If the program binary is mapped into memory but not mapped into the page table, then all the pages that will be accessed by the program will have to be faulted in every time the program is run. This is unnecessary when the pages in question are already in the VM Cache, so FreeBSD will attempt to pre-populate a process's page tables with those pages that are already in the VM Cache. One thing that FreeBSD does not yet do is pre-copy-on-write certain pages on exec. For example, if you run the ls(1) program while running `vmstat 1` you will notice that it always takes a certain number of page faults, even when you run it over and over again. These are zero-fill faults, not program code faults (which were pre-faulted in already). Pre-copying pages on exec or fork is an area that could use more study.

A large percentage of page faults that occur are zero-fill faults. You can usually see this by observing the `vmstat -s` output. These occur when a process accesses pages in its BSS area. The BSS area is expected to be initially zero but the VM system does not bother to allocate any memory at all until the process actually accesses it. When a fault occurs the VM system must not only allocate a new page, it must zero it as well. To optimize the zeroing operation the VM system has the ability to pre-zero pages and mark them as such, and to request pre-zeroed pages when zero-fill faults occur. The pre-zeroing occurs whenever the CPU is idle but the number of pages the system pre-zeros is limited in order to avoid blowing away the memory caches. This is an excellent example of adding complexity to the VM system in order to optimize the critical path.

6 Page Table Optimizations

The page table optimizations make up the most contentious part of the FreeBSD VM design and they have shown some strain with the advent of serious use of `mmap()`. I think this is actually a feature of most BSDs though I am not sure when it was first introduced. There are two major optimizations. The first is that hardware page tables do not contain persistent state but instead can be thrown away at any time with only a minor amount of management overhead. The second is that every active page table entry in the system has a governing `pv_entry` structure which is tied into the `vm_page` structure. FreeBSD can simply iterate through those mappings that are known to exist while Linux must check all page tables that *might* contain a specific mapping to see if it does, which can achieve O(n^2)

overhead in certain situations. It is because of this that FreeBSD tends to make better choices on which pages to reuse or swap when memory is stressed, giving it better performance under load. However, FreeBSD requires kernel tuning to accommodate large-shared-address-space situations such as those that can occur in a news system because it may run out of `pv_entry` structures.

Both Linux and FreeBSD need work in this area. FreeBSD is trying to maximize the advantage of a potentially sparse active-mapping model (not all processes need to map all pages of a shared library, for example), whereas Linux is trying to simplify its algorithms. FreeBSD generally has the performance advantage here at the cost of wasting a little extra memory, but FreeBSD breaks down in the case where a large file is massively shared across hundreds of processes. Linux, on the other hand, breaks down in the case where many processes are sparsely-mapping the same shared library and also runs non-optimally when trying to determine whether a page can be reused or not.

7 Page Coloring

We will end with the page coloring optimizations. Page coloring is a performance optimization designed to ensure that accesses to contiguous pages in virtual memory make the best use of the processor cache. In ancient times (i.e. 10+ years ago) processor caches tended to map virtual memory rather than physical memory. This led to a huge number of problems including having to clear the cache on every context switch in some cases, and problems with data aliasing in the cache. Modern processor caches map physical memory precisely to solve those problems. This means that two side-by-side pages in a processes address space may not correspond to two side-by-side pages in the cache. In fact, if you are not careful side-by-side pages in virtual memory could wind up using the same page in the processor cache—leading to cacheable data being thrown away prematurely and reducing CPU performance. This is true even with multi-way set-associative caches (though the effect is mitigated somewhat).

FreeBSD's memory allocation code implements page coloring optimizations, which means that the memory allocation code will attempt to locate free pages that are contiguous from the point of view of the cache. For example, if page 16 of physical memory is assigned to page 0 of a process's virtual memory and the cache can hold 4 pages, the page coloring code will not assign page 20 of physical memory to page 1 of a process's virtual memory. It would, instead, assign page 21 of physical memory. The page coloring code attempts to avoid assigning page 20 because this maps over the same cache memory as page 16 and would result in non-optimal caching. This code adds a significant amount of complexity to the VM memory allocation subsystem as you can well imagine, but the result is well worth the effort. Page Coloring makes VM memory as deterministic as physical memory in regards to cache performance.

8 Conclusion

Virtual memory in modern operating systems must address a number of different issues efficiently and for many different usage patterns. The modular and algorithmic approach that BSD has historically taken allows us to study and understand the current implementation as well as relatively cleanly replace large sections of the code. There have been a number of improvements to the FreeBSD VM system in the last several years, and work is ongoing.

9 Bonus QA session by Allen Briggs <briggs@ninthwonder.com>

1. What is "the interleaving algorithm" that you refer to in your listing of the ills of the FreeBSD 3.X swap arrangements?

FreeBSD uses a fixed swap interleave which defaults to 4. This means that FreeBSD reserves space for four swap

areas even if you only have one, two, or three. Since swap is interleaved the linear address space representing the "four swap areas" will be fragmented if you do not actually have four swap areas. For example, if you have two swap areas A and B FreeBSD's address space representation for that swap area will be interleaved in blocks of 16 pages:

A B C D A B C D A B C D A B C D

FreeBSD 3.X uses a "sequential list of free regions" approach to accounting for the free swap areas. The idea is that large blocks of free linear space can be represented with a single list node (`kern/subr_rlist.c`). But due to the fragmentation the sequential list winds up being insanely fragmented. In the above example, completely unused swap will have A and B shown as "free" and C and D shown as "all allocated". Each A-B sequence requires a list node to account for because C and D are holes, so the list node cannot be combined with the next A-B sequence.

Why do we interleave our swap space instead of just tack swap areas onto the end and do something fancier? Because it is a whole lot easier to allocate linear swaths of an address space and have the result automatically be interleaved across multiple disks than it is to try to put that sophistication elsewhere.

The fragmentation causes other problems. Being a linear list under 3.X, and having such a huge amount of inherent fragmentation, allocating and freeing swap winds up being an O(N) algorithm instead of an O(1) algorithm. Combined with other factors (heavy swapping) and you start getting into O(N^2) and O(N^3) levels of overhead, which is bad. The 3.X system may also need to allocate KVM during a swap operation to create a new list node which can lead to a deadlock if the system is trying to pageout pages in a low-memory situation.

Under 4.X we do not use a sequential list. Instead we use a radix tree and bitmaps of swap blocks rather than ranged list nodes. We take the hit of preallocating all the bitmaps required for the entire swap area up front but it winds up wasting less memory due to the use of a bitmap (one bit per block) instead of a linked list of nodes. The use of a radix tree instead of a sequential list gives us nearly O(1) performance no matter how fragmented the tree becomes.

2. I do not get the following:

> It is important to note that the FreeBSD VM system attempts to separate clean and dirty pages for the express reason of avoiding unnecessary flushes of dirty pages (which eats I/O bandwidth), nor does it move pages between the various page queues gratuitously when the memory subsystem is not being stressed. This is why you will see some systems with very low cache queue counts and high active queue counts when doing a `systat -vm` command.

How is the separation of clean and dirty (inactive) pages related to the situation where you see low cache queue counts and high active queue counts in `systat -vm`? Do the systat stats roll the active and dirty pages together for the active queue count?

Yes, that is confusing. The relationship is "goal" verses "reality". Our goal is to separate the pages but the reality is that if we are not in a memory crunch, we do not really have to.

What this means is that FreeBSD will not try very hard to separate out dirty pages (inactive queue) from clean pages (cache queue) when the system is not being stressed, nor will it try to deactivate pages (active queue -> inactive queue) when the system is not being stressed, even if they are not being used.

3. In the ls(1) / `vmstat 1` example, would not some of the page faults be data page faults (COW from executable file to private page)? I.e., I would expect the page faults to be some zero-fill and some program data. Or are you

implying that FreeBSD does do pre-COW for the program data?

A COW fault can be either zero-fill or program-data. The mechanism is the same either way because the backing program-data is almost certainly already in the cache. I am indeed lumping the two together. FreeBSD does not pre-COW program data or zero-fill, but it *does* pre-map pages that exist in its cache.

4. In your section on page table optimizations, can you give a little more detail about `pv_entry` and `vm_page` (or should vm_page be `vm_pmap`—as in 4.4, cf. pp. 180-181 of McKusick, Bostic, Karel, Quarterman)? Specifically, what kind of operation/reaction would require scanning the mappings?

How does Linux do in the case where FreeBSD breaks down (sharing a large file mapping over many processes)?

A `vm_page` represents an (object,index#) tuple. A `pv_entry` represents a hardware page table entry (pte). If you have five processes sharing the same physical page, and three of those processes's page tables actually map the page, that page will be represented by a single `vm_page` structure and three `pv_entry` structures.

`pv_entry` structures only represent pages mapped by the MMU (one `pv_entry` represents one pte). This means that when we need to remove all hardware references to a `vm_page` (in order to reuse the page for something else, page it out, clear it, dirty it, and so forth) we can simply scan the linked list of `pv_entry`'s associated with that `vm_page` to remove or modify the pte's from their page tables.

Under Linux there is no such linked list. In order to remove all the hardware page table mappings for a `vm_page` linux must index into every VM object that *might* have mapped the page. For example, if you have 50 processes all mapping the same shared library and want to get rid of page X in that library, you need to index into the page table for each of those 50 processes even if only 10 of them have actually mapped the page. So Linux is trading off the simplicity of its design against performance. Many VM algorithms which are O(1) or (small N) under FreeBSD wind up being O(N), O(N^2), or worse under Linux. Since the pte's representing a particular page in an object tend to be at the same offset in all the page tables they are mapped in, reducing the number of accesses into the page tables at the same pte offset will often avoid blowing away the L1 cache line for that offset, which can lead to better performance.

FreeBSD has added complexity (the `pv_entry` scheme) in order to increase performance (to limit page table accesses to *only* those pte's that need to be modified).

But FreeBSD has a scaling problem that Linux does not in that there are a limited number of `pv_entry` structures and this causes problems when you have massive sharing of data. In this case you may run out of `pv_entry` structures even though there is plenty of free memory available. This can be fixed easily enough by bumping up the number of `pv_entry` structures in the kernel config, but we really need to find a better way to do it.

In regards to the memory overhead of a page table verses the `pv_entry` scheme: Linux uses "permanent" page tables that are not throw away, but does not need a `pv_entry` for each potentially mapped pte. FreeBSD uses "throw away" page tables but adds in a `pv_entry` structure for each actually-mapped pte. I think memory utilization winds up being about the same, giving FreeBSD an algorithmic advantage with its ability to throw away page tables at will with very low overhead.

5. Finally, in the page coloring section, it might help to have a little more description of what you mean here. I did not quite follow it.

Do you know how an L1 hardware memory cache works? I will explain: Consider a machine with 16MB of main memory but only 128K of L1 cache. Generally the way this cache works is that each 128K block of main memory uses the *same* 128K of cache. If you access offset 0 in main memory and then offset 128K in main memory you can wind up throwing away the cached data you read from offset 0!

Now, I am simplifying things greatly. What I just described is what is called a “direct mapped” hardware memory cache. Most modern caches are what are called 2-way-set-associative or 4-way-set-associative caches. The set-associatively allows you to access up to N different memory regions that overlap the same cache memory without destroying the previously cached data. But only N.

So if I have a 4-way set associative cache I can access offset 0, offset 128K, 256K and offset 384K and still be able to access offset 0 again and have it come from the L1 cache. If I then access offset 512K, however, one of the four previously cached data objects will be thrown away by the cache.

It is extremely important… *extremely* important for most of a processor’s memory accesses to be able to come from the L1 cache, because the L1 cache operates at the processor frequency. The moment you have an L1 cache miss and have to go to the L2 cache or to main memory, the processor will stall and potentially sit twiddling its fingers for *hundreds* of instructions worth of time waiting for a read from main memory to complete. Main memory (the dynamic ram you stuff into a computer) is *slow*, when compared to the speed of a modern processor core.

Ok, so now onto page coloring: All modern memory caches are what are known as *physical* caches. They cache physical memory addresses, not virtual memory addresses. This allows the cache to be left alone across a process context switch, which is very important.

But in the UNIX world you are dealing with virtual address spaces, not physical address spaces. Any program you write will see the virtual address space given to it. The actual *physical* pages underlying that virtual address space are not necessarily physically contiguous! In fact, you might have two pages that are side by side in a processes address space which wind up being at offset 0 and offset 128K in *physical* memory.

A program normally assumes that two side-by-side pages will be optimally cached. That is, that you can access data objects in both pages without having them blow away each other’s cache entry. But this is only true if the physical pages underlying the virtual address space are contiguous (insofar as the cache is concerned).

This is what Page coloring does. Instead of assigning *random* physical pages to virtual addresses, which may result in non-optimal cache performance, Page coloring assigns *reasonably-contiguous* physical pages to virtual addresses. Thus programs can be written under the assumption that the characteristics of the underlying hardware cache are the same for their virtual address space as they would be if the program had been run directly in a physical address space.

Note that I say “reasonably” contiguous rather than simply “contiguous”. From the point of view of a 128K direct mapped cache, the physical address 0 is the same as the physical address 128K. So two side-by-side pages in your virtual address space may wind up being offset 128K and offset 132K in physical memory, but could also easily be offset 128K and offset 4K in physical memory and still retain the same cache performance characteristics. So page-coloring does *not* have to assign truly contiguous pages of physical memory to contiguous pages of virtual memory, it just needs to make sure it assigns contiguous pages from the point of view of cache performance and operation.