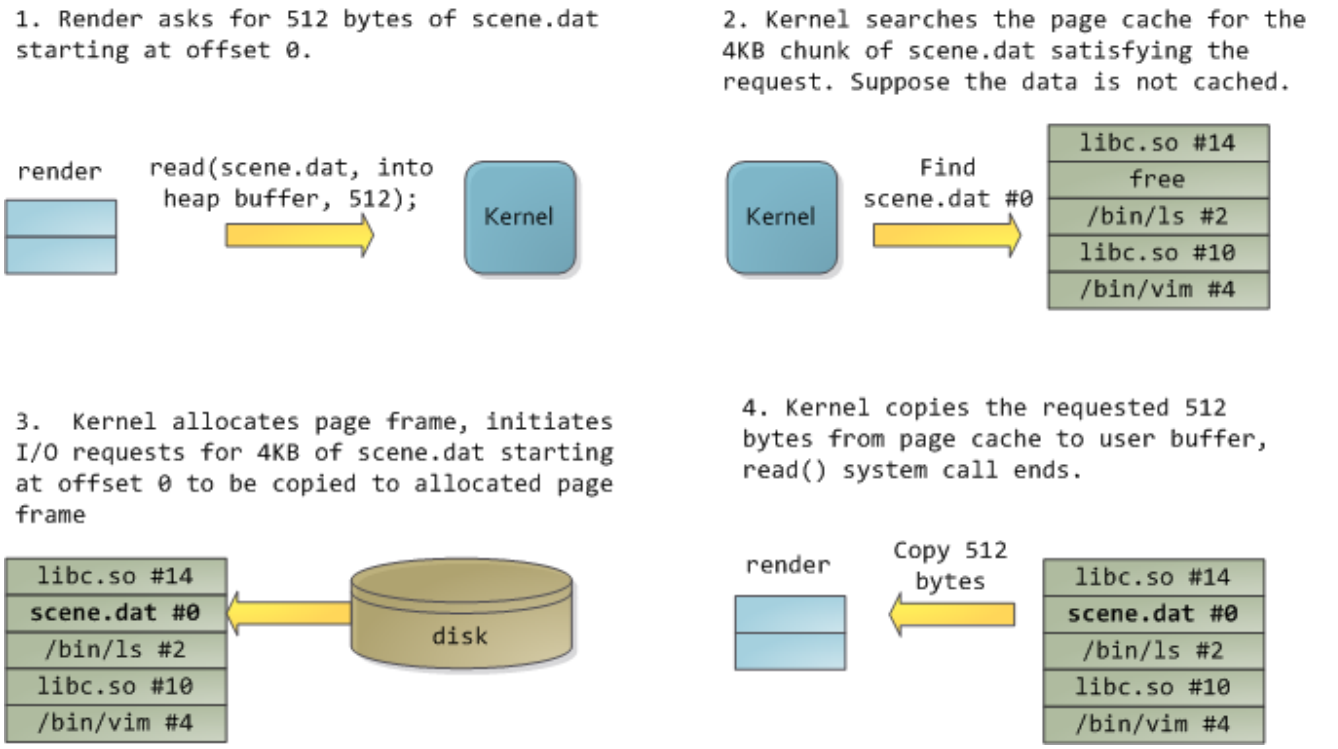
Page Cache, the Affair Between Memory and Files

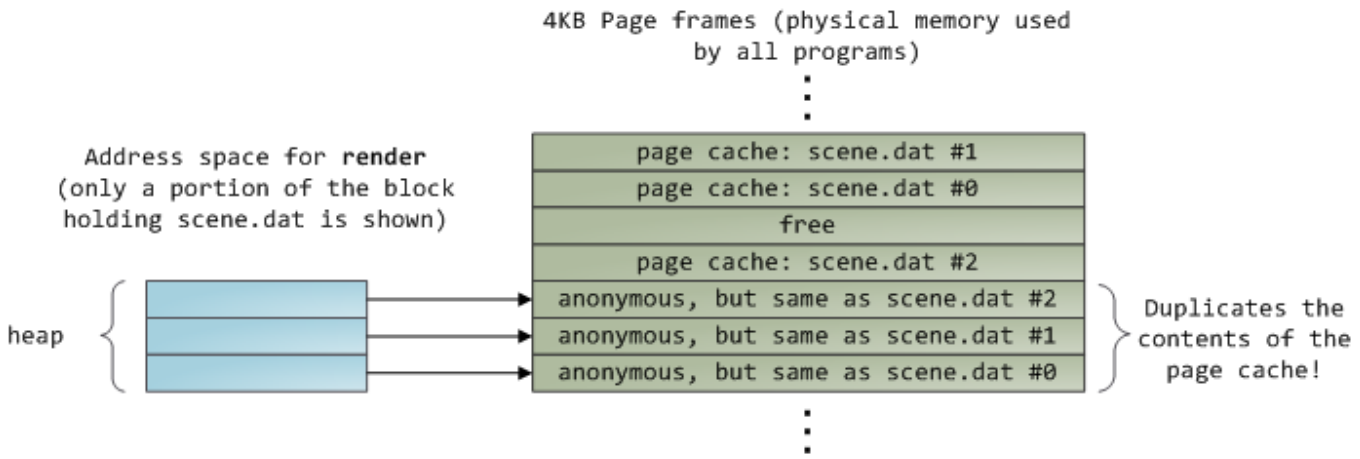
Previously we looked at how the kernel manages virtual memory for a user process, but files and I/O were left out. This post covers the important and often misunderstood relationship between files and memory and its consequences for performance.

Two serious problems must be solved by the OS when it comes to files. The first one is the mind-blowing slowness of hard drives, and disk seeks in particular, relative to memory. The second is the need to load file contents in physical memory once and *share* the contents among programs. If you use Process Explorer to poke at Windows processes, you'll see there are ~15MB worth of common DLLs loaded in every process. My Windows box right now is running 100 processes, so without sharing I'd be using up to ~1.5 GB of physical RAM *just for common DLLs*. No good. Likewise, nearly all Linux programs need ld.so and libc, plus other common libraries.

Happily, both problems can be dealt with in one shot: the **page cache**, where the kernel stores page-sized chunks of files. To illustrate the page cache, I'll conjure a Linux program named **render**, which opens file **scene.dat** and reads it 512 bytes at a time, storing the file contents into a heap-allocated block. The first read goes like this:

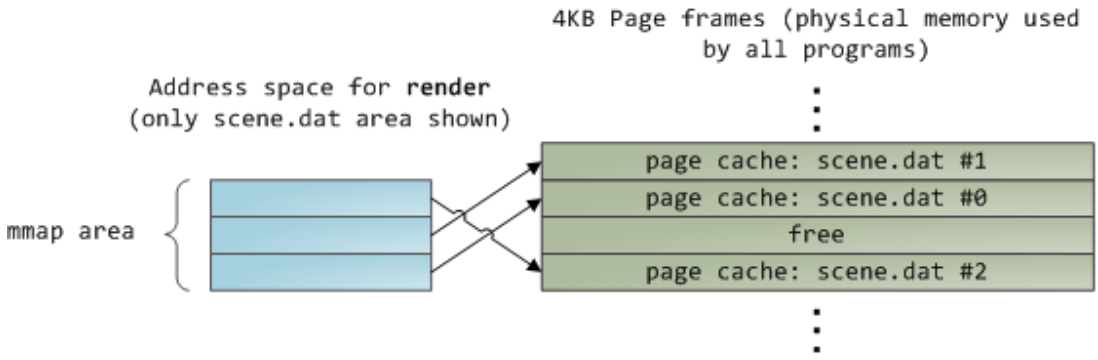


After 12KB have been read, `render`'s heap and the relevant page frames look thus:



This looks innocent enough, but there's a lot going on. First, even though this program uses regular `read` calls, three 4KB page frames are now in the page cache storing part of `scene.dat`. People are sometimes surprised by this, but **all regular file I/O happens through the page cache**. In x86 Linux, the kernel thinks of a file as a sequence of 4KB chunks. If you read a single byte from a file, the whole 4KB chunk containing the byte you asked for is read from disk and placed into the page cache. This makes sense because sustained disk throughput is pretty good and programs normally read more than just a few bytes from a file region. The page cache knows the position of each 4KB chunk within the file, depicted above as #0, #1, etc. Windows uses 256KB **views** analogous to pages in the Linux page cache.

Sadly, in a regular file read the kernel must copy the contents of the page cache into a user buffer, which not only takes cpu time and hurts the cpu caches, but also **wastes physical memory with duplicate data**. As per the diagram above, the `scene.dat` contents are stored twice, and each instance of the program would store the contents an additional time. We've mitigated the disk latency problem but failed miserably at everything else. **Memory-mapped files** are the way out of this madness:



When you use file mapping, the kernel maps your program's virtual pages directly onto the page cache. This can deliver a significant performance boost: Windows System Programming reports run time improvements of 30% and up relative to regular file reads, while similar figures are reported for Linux and Solaris in Advanced Programming in the Unix Environment. You might also save large amounts of physical memory, depending on the nature of your application.

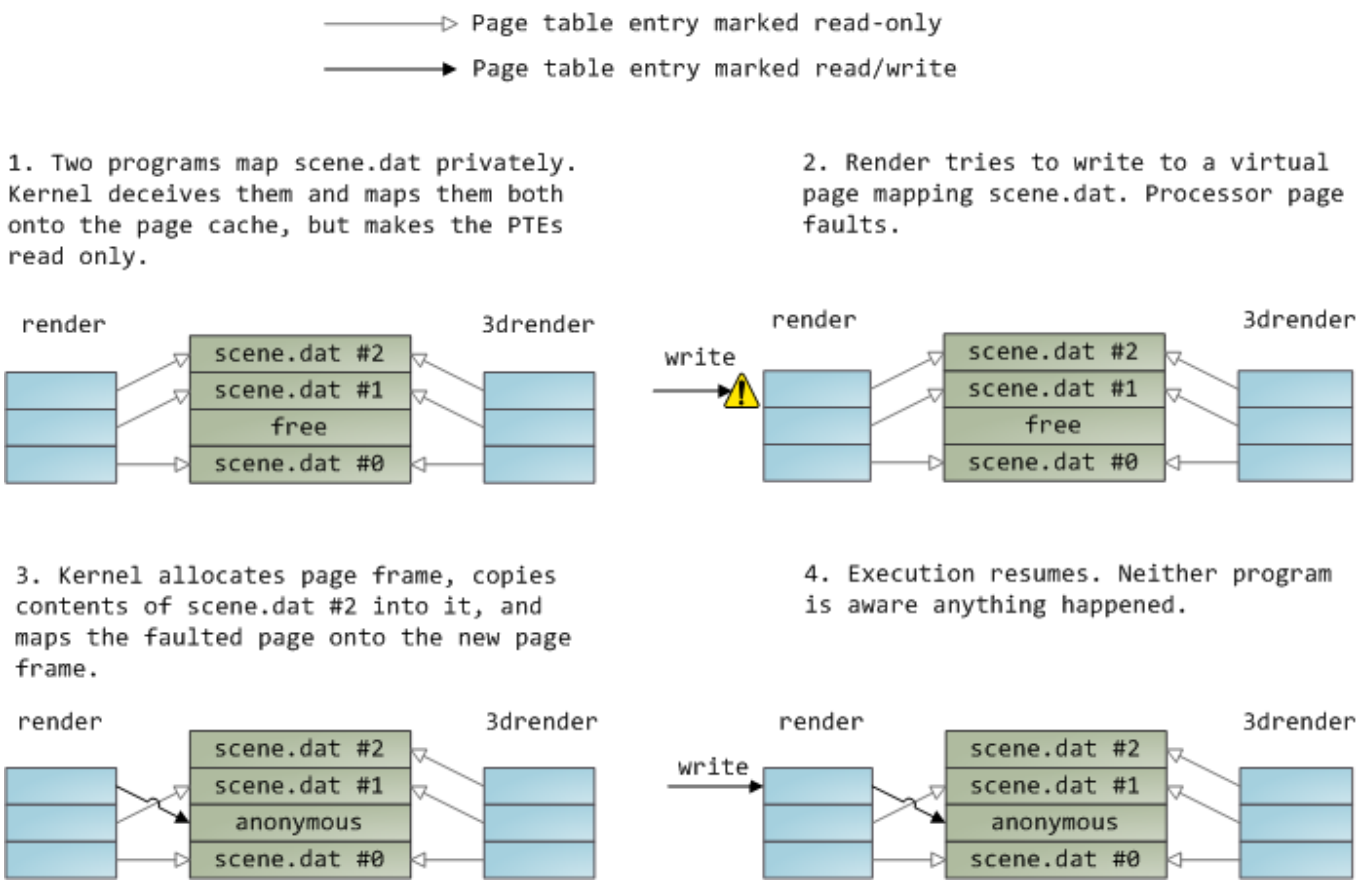
As always with performance, measurement is everything, but memory mapping earns its keep in a programmer's toolbox. The API is pretty nice too, it allows you to access a file as bytes in memory and does not require your soul and code readability in exchange for its benefits. Mind your address space and experiment with mmap in Unix-like systems, CreateFileMapping in Windows, or the many wrappers available in high level languages. When you map a file its contents are not brought into memory all at once, but rather on demand via page faults. The fault handler maps your virtual pages onto the page cache after

obtaining a page frame with the needed file contents. This involves disk I/O if the contents weren't cached to begin with.

Now for a pop quiz. Imagine that the last instance of our `render` program exits. Would the pages storing *scene.dat* in the page cache be freed immediately? People often think so, but that would be a bad idea. When you think about it, it is very common for us to create a file in one program, exit, then use the file in a second program. The page cache must handle that case. When you think *more* about it, why should the kernel *ever* get rid of page cache contents? Remember that disk is 5 orders of magnitude slower than RAM, hence a page cache hit is a huge win. So long as there's enough free physical memory, the cache should be kept full. It is therefore *not* dependent on a particular process, but rather it's a system-wide resource. If you run `render` a week from now and `scene.dat` is still cached, bonus! This is why the kernel cache size climbs steadily until it hits a ceiling. It's not because the OS is garbage and hogs your RAM, it's actually good behavior because in a way free physical memory is a waste. Better use as much of the stuff for caching as possible.

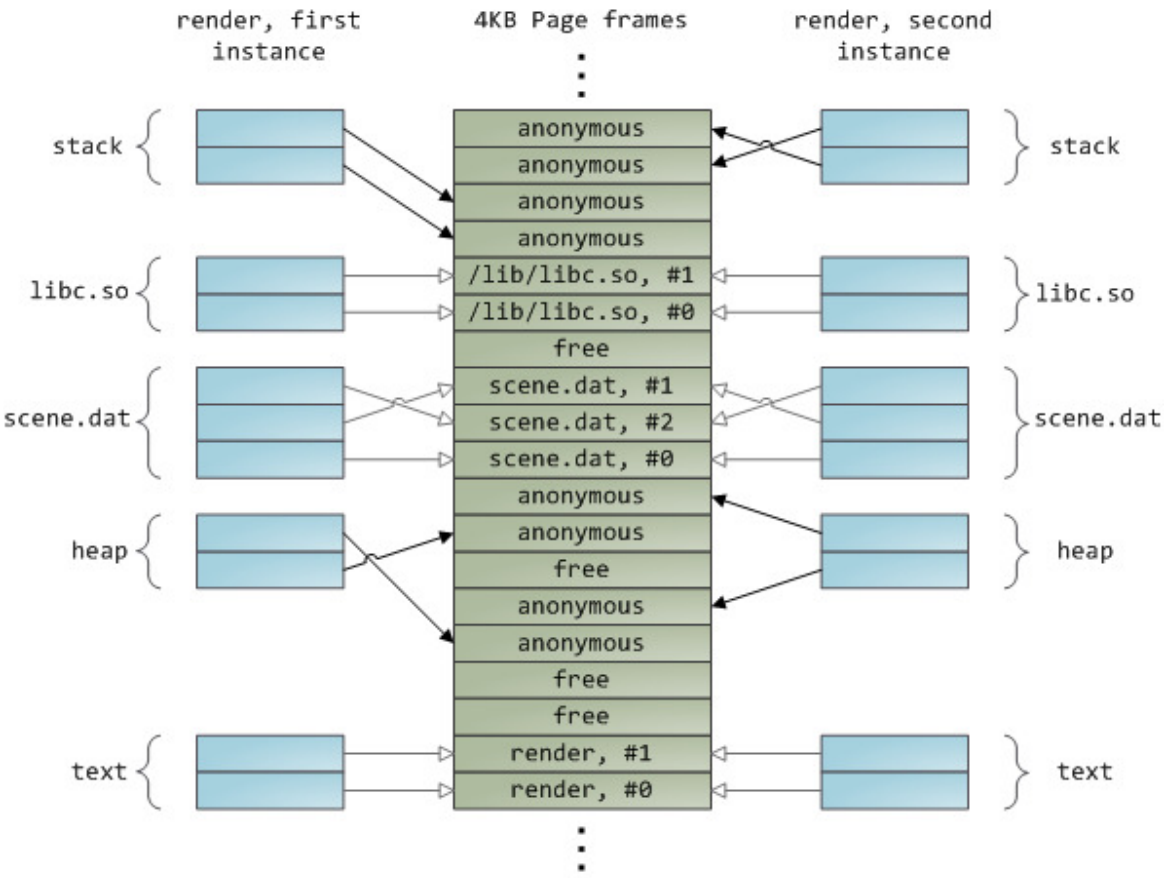
Due to the page cache architecture, when a program calls write() bytes are simply copied to the page cache and the page is marked dirty. Disk I/O normally does **not** happen immediately, thus your program doesn't block waiting for the disk. On the downside, if the computer crashes your writes will never make it, hence critical files like database transaction logs must be fsync()ed (though one must still worry about drive controller caches, oy!). Reads, on the other hand, normally block your program until the data is available. Kernels employ eager loading to mitigate this problem, an example of which is **read ahead** where the kernel preloads a few pages into the page cache in anticipation of your reads. You can help the kernel tune its eager loading behavior by providing hints on whether you plan to read a file sequentially or randomly (see madvise(), readahead(), Windows cache hints). Linux does read-ahead for memory-mapped files, but I'm not sure about Windows. Finally, it's possible to bypass the page cache using O_DIRECT in Linux or NO_BUFFERING in Windows, something database software often does.

A file mapping may be **private** or **shared**. This refers only to **updates** made to the contents in memory: in a private mapping the updates are not committed to disk or made visible to other processes, whereas in a shared mapping they are. Kernels use the **copy on write** mechanism, enabled by page table entries, to implement private mappings. In the example below, both `render` and another program called `render3d` (am I creative or what?) have mapped `scene.dat` privately. `Render` then writes to its virtual memory area that maps the file:



The read-only page table entries shown above do *not* mean the mapping is read only, they're merely a kernel trick to share physical memory until the last possible moment. You can see how 'private' is a bit of a misnomer until you remember it only applies to updates. A consequence of this design is that a virtual page that maps a file privately sees changes done to the file by other programs *as long as the page has only been read from*. Once copy-on-write is done, changes by others are no longer seen. This behavior is not guaranteed by the kernel, but it's what you get in x86 and makes sense from an API perspective. By contrast, a shared mapping is simply mapped onto the page cache and that's it. Updates are visible to other processes and end up in the disk. Finally, if the mapping above were read-only, page faults would trigger a segmentation fault instead of copy on write.

Dynamically loaded libraries are brought into your program's address space via file mapping. There's nothing magical about it, it's the same private file mapping available to you via regular APIs. Below is an example showing part of the address spaces from two running instances of the file-mapping `render` program, along with physical memory, to tie together many of the concepts we've seen.



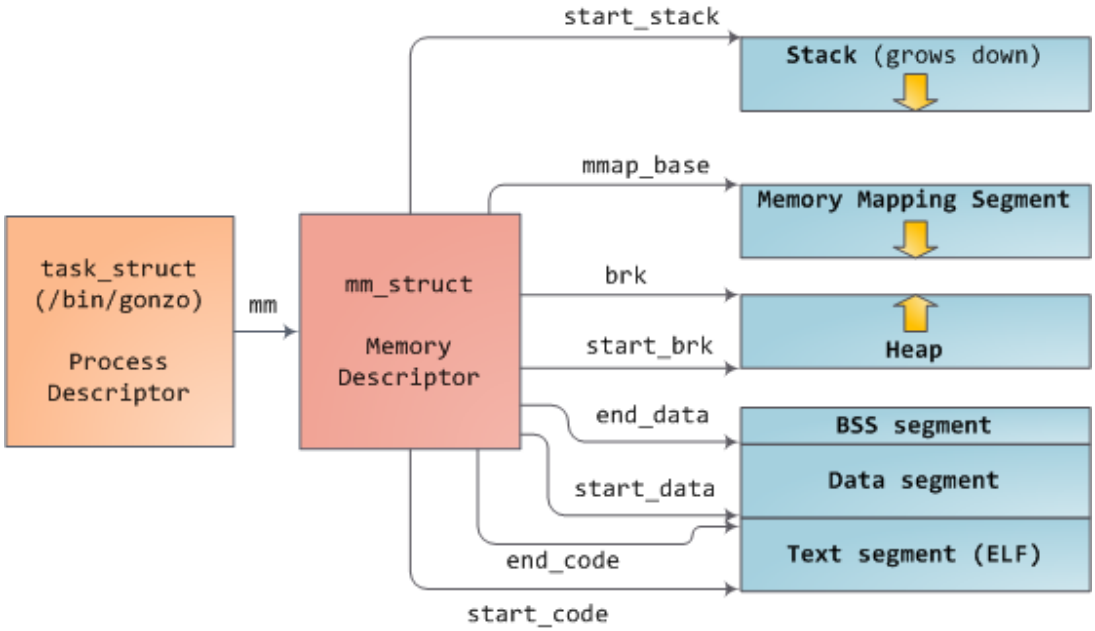
This concludes our 3-part series on memory fundamentals. I hope the series was useful and provided you with a good mental model of these OS topics. Next week there's one more post on memory usage figures, and then it's time for a change of air. Maybe some Web 2.0 gossip or something.

February 10, 2009 | Filed Under Internals, Linux, Software Illustrated

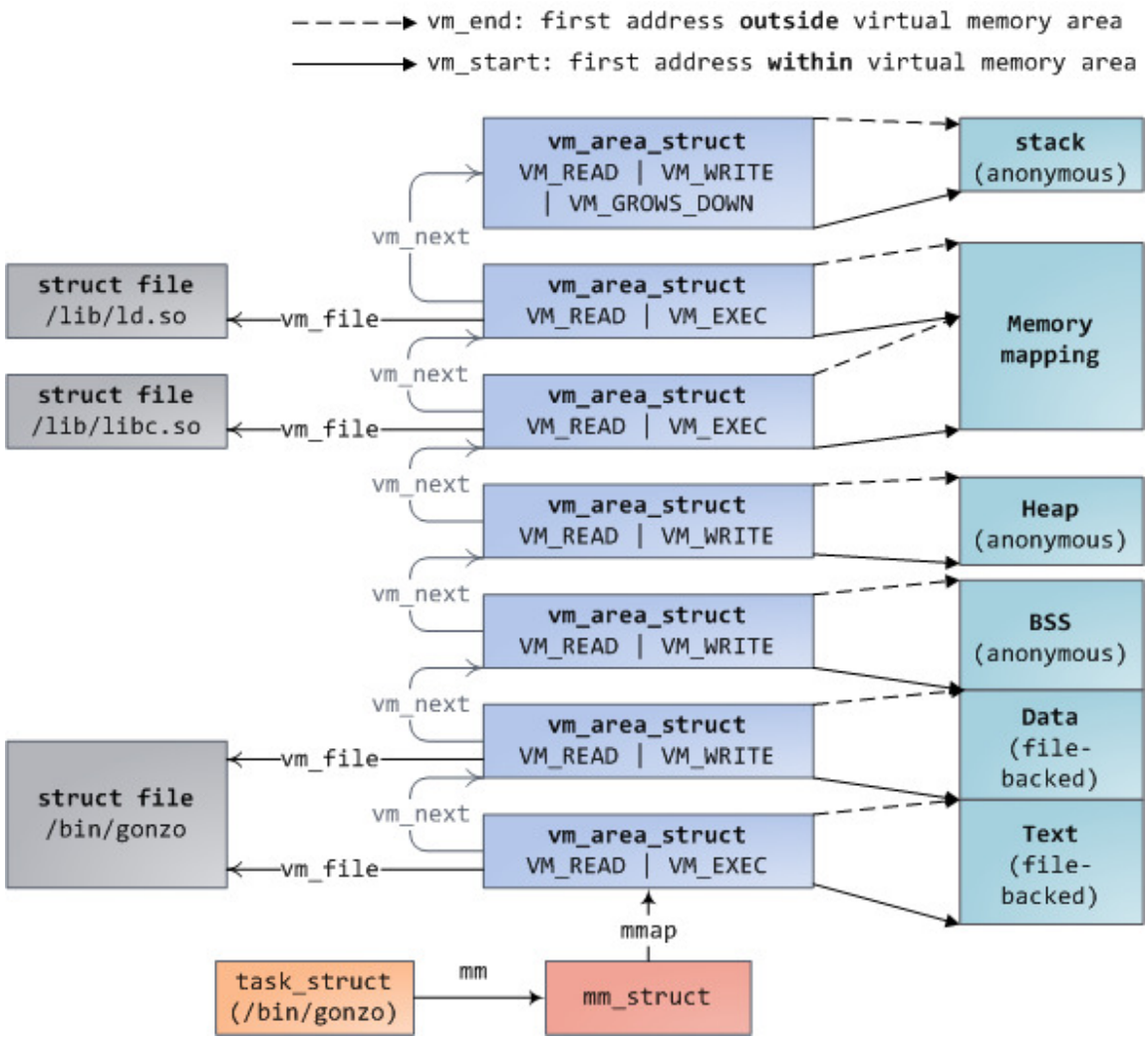
60 Comments

How The Kernel Manages Your Memory

After examining the virtual address layout of a process, we turn to the kernel and its mechanisms for managing user memory. Here is gonzo again:



Linux processes are implemented in the kernel as instances of task_struct, the process descriptor. The mm field in task_struct points to the **memory descriptor**, mm_struct, which is an executive summary of a program's memory. It stores the start and end of memory segments as shown above, the number of physical memory pages used by the process (**rss** stands for Resident Set Size), the amount of virtual address space used, and other tidbits. Within the memory descriptor we also find the two work horses for managing program memory: the set of **virtual memory areas** and the **page tables**. Gonzo's memory areas are shown below:

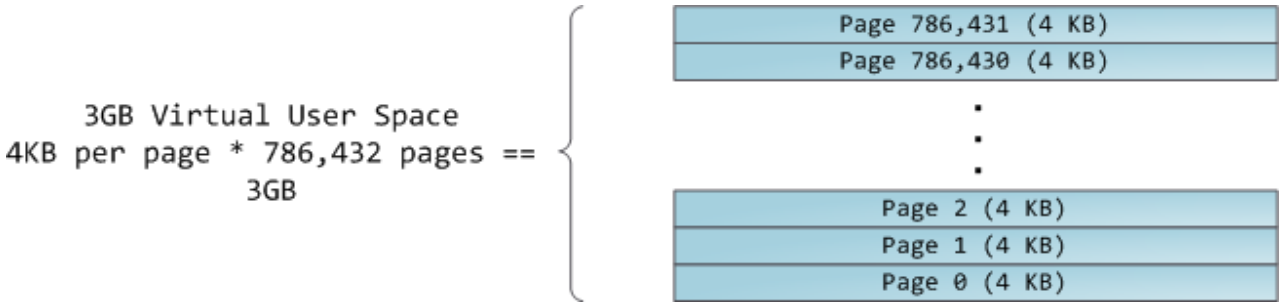


Each virtual memory area (VMA) is a contiguous range of virtual addresses; these areas never overlap. An instance of vm_area_struct fully describes a memory area, including its start and end addresses, flags to determine access rights and behaviors, and the vm_file field to specify which file is being mapped by the area, if any. A VMA that does not map a file is **anonymous**. Each memory segment above (*e.g.*, heap, stack) corresponds to a single VMA, with the exception of the memory mapping segment. This is not a requirement, though it is usual in x86 machines. VMAs do not care which segment they are in.

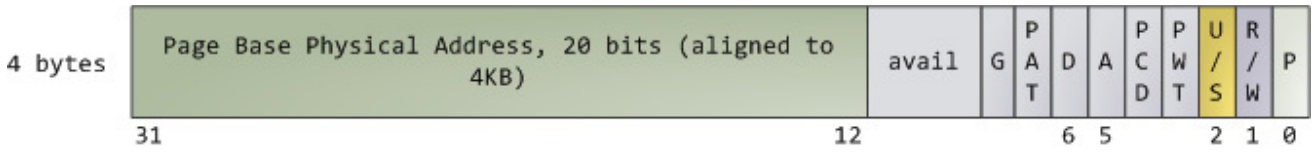
A program's VMAs are stored in its memory descriptor both as a linked list in the mmap field, ordered by starting virtual address, and as a red-black tree rooted at the mm_rb field. The red-black tree allows the kernel to search quickly for the memory area covering a given virtual address. When you read file `/proc/pid_of_process/maps`, the kernel is simply going through the linked list of VMAs for the process and printing each one.

In Windows, the EPROCESS block is roughly a mix of task_struct and mm_struct. The Windows analog to a VMA is the Virtual Address Descriptor, or VAD; they are stored in an AVL tree. You know what the funniest thing about Windows and Linux is? It's the little differences.

The 4GB virtual address space is divided into **pages**. x86 processors in 32-bit mode support page sizes of 4KB, 2MB, and 4MB. Both Linux and Windows map the user portion of the virtual address space using 4KB pages. Bytes 0-4095 fall in page 0, bytes 4096-8191 fall in page 1, and so on. The size of a VMA *must be a multiple of page size*. Here's 3GB of user space in 4KB pages:



The processor consults **page tables** to translate a virtual address into a physical memory address. Each process has its own set of page tables; whenever a process switch occurs, page tables for user space are switched as well. Linux stores a pointer to a process' page tables in the pgd field of the memory descriptor. To each virtual page there corresponds one **page table entry** (PTE) in the page tables, which in regular x86 paging is a simple 4-byte record shown below:

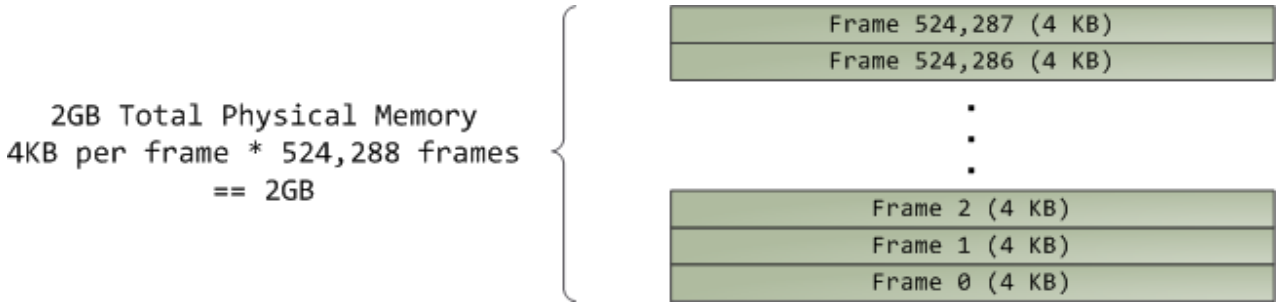


Linux has functions to read and set each flag in a PTE. Bit P tells the processor whether the virtual page is **present** in physical memory. If clear (equal to 0), accessing the page triggers a page fault. Keep in mind that when this bit is zero, **the kernel can do whatever it pleases** with the remaining fields. The R/W flag stands for read/write; if clear, the page is read-only. Flag U/S stands for user/supervisor; if clear, then the page can only be accessed by the kernel. These flags are used to implement the read-only memory and protected kernel space we saw before.

Bits D and A are for **dirty** and **accessed**. A dirty page has had a write, while an accessed page has had a write or read. Both flags are sticky: the processor only sets them, they must be cleared by the kernel. Finally, the PTE stores the starting physical address that corresponds to this page, aligned to 4KB. This naive-looking field is the source of some pain, for it limits addressable physical memory to 4 GB. The other PTE fields are for another day, as is Physical Address Extension.

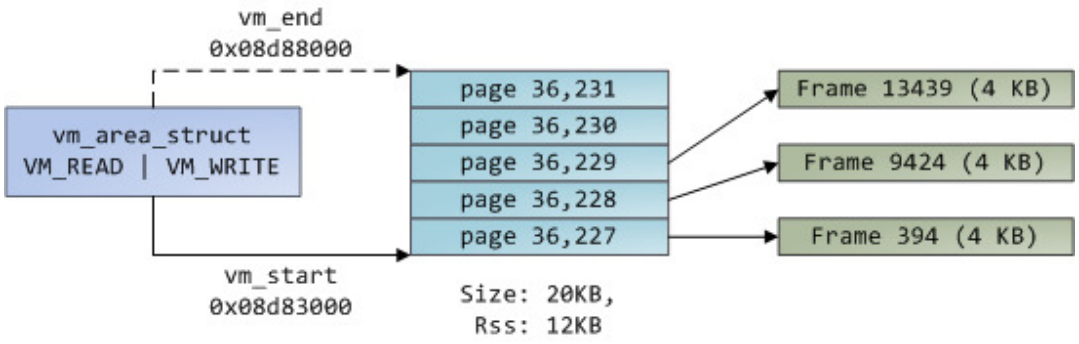
A virtual page is the unit of memory protection because all of its bytes share the U/S and R/W flags. However, the same physical memory could be mapped by different pages, possibly with different protection flags. Notice that execute permissions are nowhere to be seen in the PTE. This is why classic x86 paging allows code on the stack to be executed, making it easier to exploit stack buffer overflows (it's still possible to exploit non-executable stacks using return-to-libc and other techniques). This lack of a PTE no-execute flag illustrates a broader fact: permission flags in a VMA may or may not translate cleanly into hardware protection. The kernel does what it can, but ultimately the architecture limits what is possible.

Virtual memory doesn't store anything, it simply *maps* a program's address space onto the underlying physical memory, which is accessed by the processor as a large block called the **physical address space**. While memory operations on the bus are somewhat involved, we can ignore that here and assume that physical addresses range from zero to the top of available memory in one-byte increments. This physical address space is broken down by the kernel into **page frames**. The processor doesn't know or care about frames, yet they are crucial to the kernel because **the page frame is the unit of physical memory management.** Both Linux and Windows use 4KB page frames in 32-bit mode; here is an example of a machine with 2GB of RAM:



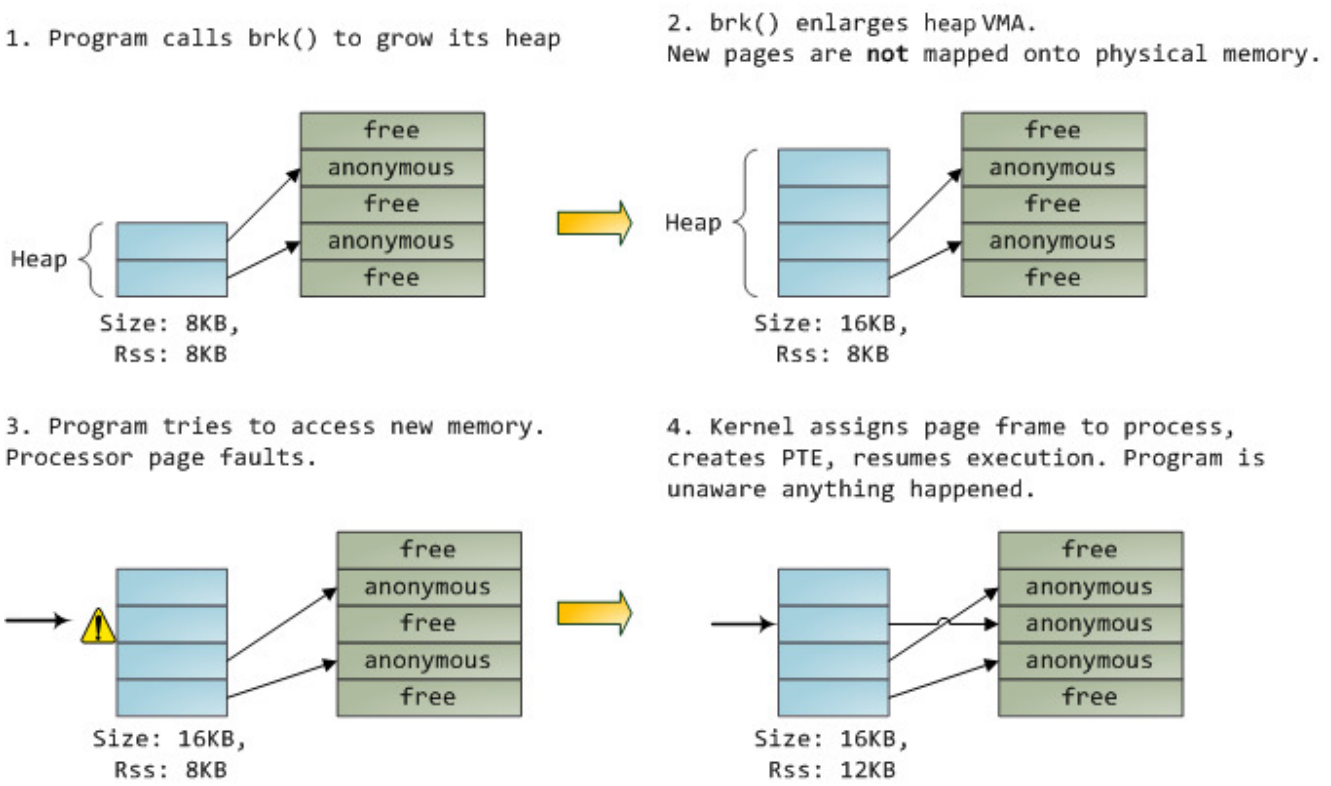
In Linux each page frame is tracked by a descriptor and several flags. Together these descriptors track the entire physical memory in the computer; the precise state of each page frame is always known. Physical memory is managed with the buddy memory allocation technique, hence a page frame is **free** if it's available for allocation via the buddy system. An allocated page frame might be **anonymous**, holding program data, or it might be in the **page cache**, holding data stored in a file or block device. There are other exotic page frame uses, but leave them alone for now. Windows has an analogous Page Frame Number (PFN) database to track physical memory.

Let's put together virtual memory areas, page table entries and page frames to understand how this all works. Below is an example of a user heap:



Blue rectangles represent pages in the VMA range, while arrows represent page table entries mapping pages onto page frames. Some virtual pages lack arrows; this means their corresponding PTEs have the **Present** flag clear. This could be because the pages have never been touched or because their contents have been swapped out. In either case access to these pages will lead to page faults, even though they are within the VMA. It may seem strange for the VMA and the page tables to disagree, yet this often happens.

A VMA is like a contract between your program and the kernel. You ask for something to be done (memory allocated, a file mapped, etc.), the kernel says "sure", and it creates or updates the appropriate VMA. But *it does not* actually honor the request right away, it waits until a page fault happens to do real work. The kernel is a lazy, deceitful sack of scum; this is the fundamental principle of virtual memory. It applies in most situations, some familiar and some surprising, but the rule is that VMAs record what has been *agreed upon*, while PTEs reflect what has *actually been done* by the lazy kernel. These two data structures together manage a program's memory; both play a role in resolving page faults, freeing memory, swapping memory out, and so on. Let's take the simple case of memory allocation:



When the program asks for more memory via the brk() system call, the kernel simply updates the heap VMA and calls it good. No page frames are actually allocated at this point and the new pages are not present in physical memory. Once the program tries to access the pages, the processor page faults and do_page_fault() is called. It searches for the VMA covering the faulted virtual address using find_vma(). If found, the permissions on the VMA are also checked against the attempted access (read or write). If there's no suitable VMA, no contract covers the attempted memory access and the process is punished by Segmentation Fault.

When a VMA is found the kernel must handle the fault by looking at the PTE contents and the type of VMA. In our case, the PTE shows the page is not present. In fact, our PTE is completely blank (all zeros), which in Linux means the virtual page has never been mapped. Since this is an anonymous VMA, we have a purely RAM affair that must be handled by do_anonymous_page(), which allocates a page frame and makes a PTE to map the faulted virtual page onto the freshly allocated frame.

Things could have been different. The PTE for a swapped out page, for example, has 0 in the Present flag but is not blank. Instead, it stores the swap location holding the page contents, which must be read from disk and loaded into a page frame by do_swap_page() in what is called a major fault.

This concludes the first half of our tour through the kernel's user memory management. In the next post, we'll throw files into the mix to build a complete picture of memory fundamentals, including consequences for performance.

February 3, 2009 | Filed Under Internals, Linux, Software Illustrated

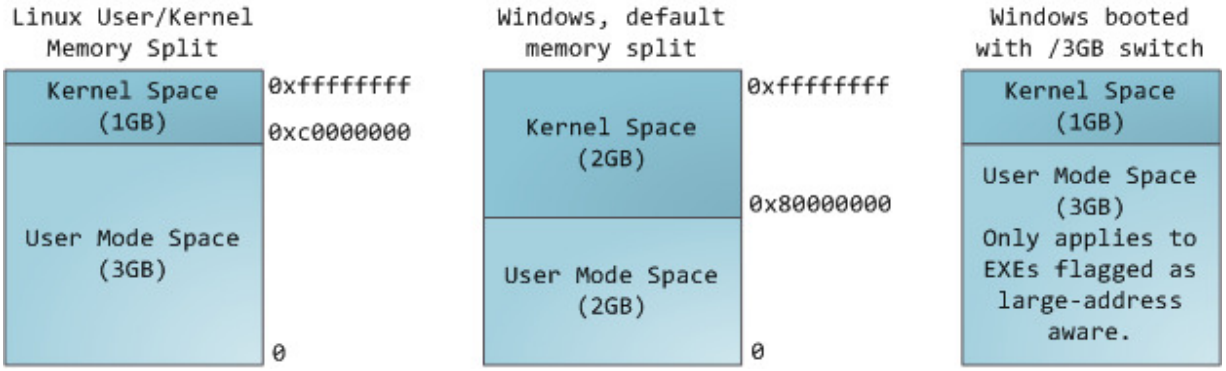
106 Comments

Anatomy of a Program in Memory

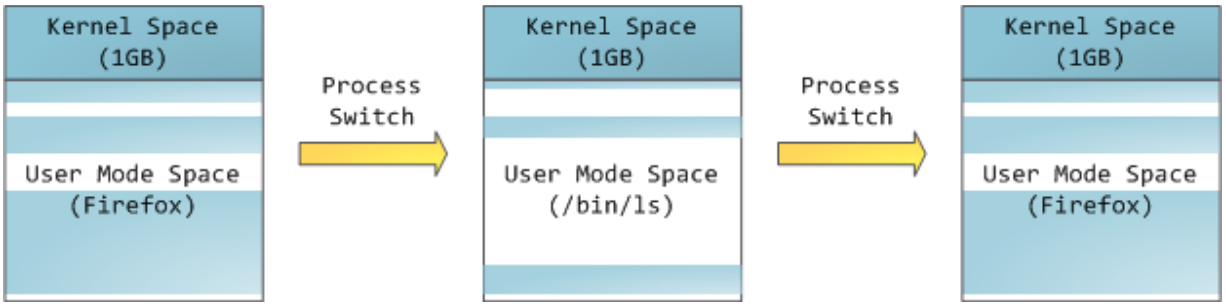
Memory management is the heart of operating systems; it is crucial for both programming and system administration. In the next few posts I'll cover memory with an eye towards practical aspects, but without

shying away from internals. While the concepts are generic, examples are mostly from Linux and Windows on 32-bit x86. This first post describes how programs are laid out in memory.

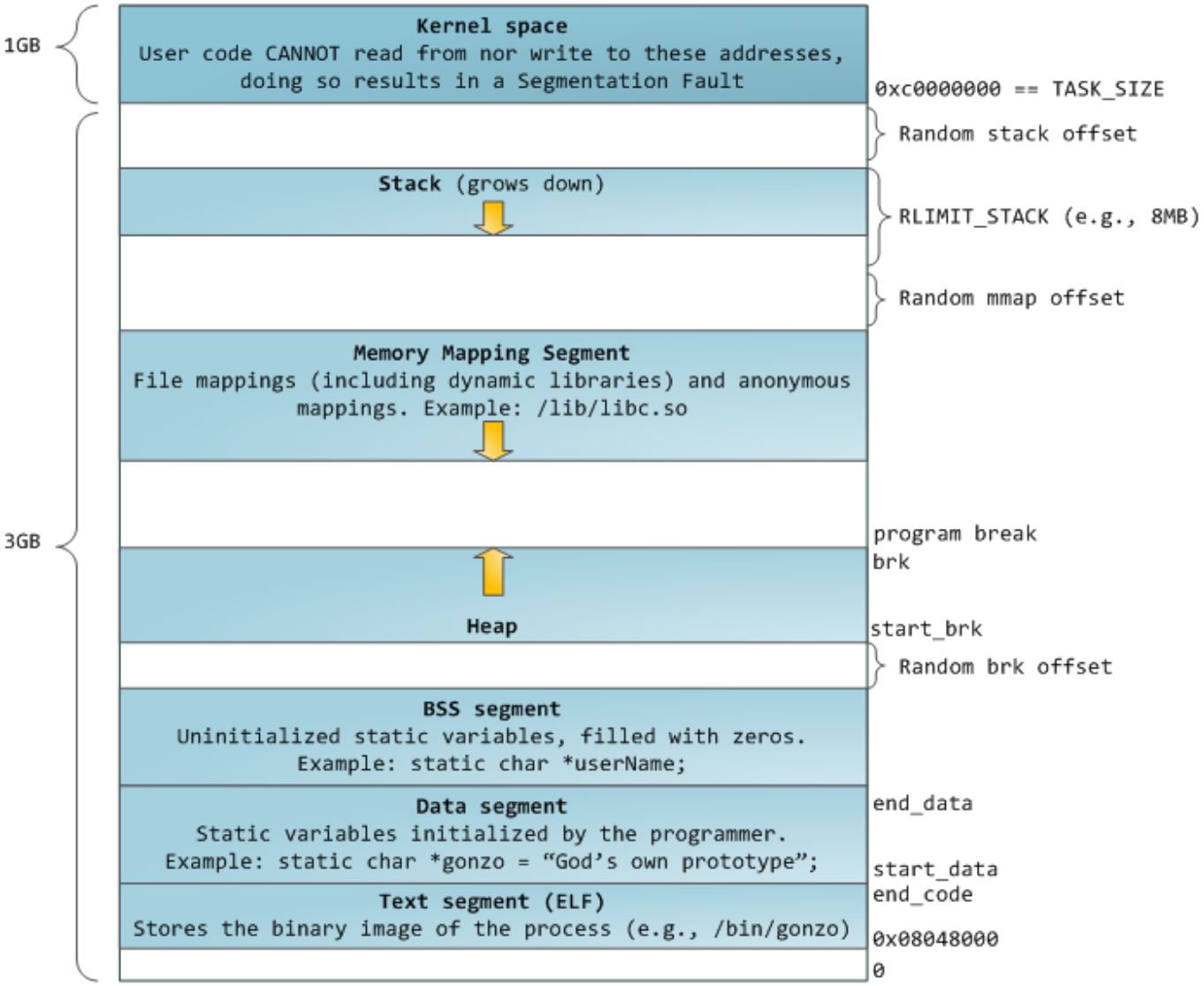
Each process in a multi-tasking OS runs in its own memory sandbox. This sandbox is the **virtual address space**, which in 32-bit mode is **always a 4GB block of memory addresses**. These virtual addresses are mapped to physical memory by **page tables**, which are maintained by the operating system kernel and consulted by the processor. Each process has its own set of page tables, but there is a catch. Once virtual addresses are enabled, they apply to *all software* running in the machine, *including the kernel itself*. Thus a portion of the virtual address space must be reserved to the kernel:



This does **not** mean the kernel uses that much physical memory, only that it has that portion of address space available to map whatever physical memory it wishes. Kernel space is flagged in the page tables as exclusive to privileged code (ring 2 or lower), hence a page fault is triggered if user-mode programs try to touch it. In Linux, kernel space is constantly present and maps the same physical memory in all processes. Kernel code and data are always addressable, ready to handle interrupts or system calls at any time. By contrast, the mapping for the user-mode portion of the address space changes whenever a process switch happens:



Blue regions represent virtual addresses that are mapped to physical memory, whereas white regions are unmapped. In the example above, Firefox has used far more of its virtual address space due to its legendary memory hunger. The distinct bands in the address space correspond to **memory segments** like the heap, stack, and so on. Keep in mind these segments are simply a range of memory addresses and *have nothing to do* with Intel-style segments. Anyway, here is the standard segment layout in a Linux process:



When computing was happy and safe and cuddly, the starting virtual addresses for the segments shown above were **exactly the same** for nearly every process in a machine. This made it easy to exploit security vulnerabilities remotely. An exploit often needs to reference absolute memory locations: an address on the stack, the address for a library function, etc. Remote attackers must choose this location blindly, counting on the fact that address spaces are all the same. When they are, people get pwned. Thus address space randomization has become popular. Linux randomizes the stack, memory mapping segment, and heap by adding offsets to their starting addresses. Unfortunately the 32-bit address space is pretty tight, leaving little room for randomization and hampering its effectiveness.

The topmost segment in the process address space is the stack, which stores local variables and function parameters in most programming languages. Calling a method or function pushes a new **stack frame** onto the stack. The stack frame is destroyed when the function returns. This simple design, possible because the data obeys strict LIFO order, means that no complex data structure is needed to track stack contents – a simple pointer to the top of the stack will do. Pushing and popping are thus very fast and deterministic. Also, the constant reuse of stack regions tends to keep active stack memory in the cpu caches, speeding up access. Each thread in a process gets its own stack.

It is possible to exhaust the area mapping the stack by pushing more data than it can fit. This triggers a page fault that is handled in Linux by expand_stack(), which in turn calls acct_stack_growth() to check whether it's appropriate to grow the stack. If the stack size is below `RLIMIT_STACK` (usually 8MB), then normally the stack grows and the program continues merrily, unaware of what just happened. This is the normal mechanism whereby stack size adjusts to demand. However, if the maximum stack size has been reached,

we have a **stack overflow** and the program receives a Segmentation Fault. While the mapped stack area expands to meet demand, it does not shrink back when the stack gets smaller. Like the federal budget, it only expands.

Dynamic stack growth is the only situation in which access to an unmapped memory region, shown in white above, might be valid. Any other access to unmapped memory triggers a page fault that results in a Segmentation Fault. Some mapped areas are read-only, hence write attempts to these areas also lead to segfaults.

Below the stack, we have the memory mapping segment. Here the kernel maps contents of files directly to memory. Any application can ask for such a mapping via the Linux mmap() system call (implementation) or CreateFileMapping() / MapViewOfFile() in Windows. Memory mapping is a convenient and high-performance way to do file I/O, so it is used for loading dynamic libraries. It is also possible to create an **anonymous memory mapping** that does not correspond to any files, being used instead for program data. In Linux, if you request a large block of memory via malloc(), the C library will create such an anonymous mapping instead of using heap memory. 'Large' means larger than `MMAP_THRESHOLD` bytes, 128 kB by default and adjustable via mallopt().

Speaking of the heap, it comes next in our plunge into address space. The heap provides runtime memory allocation, like the stack, meant for data that must outlive the function doing the allocation, unlike the stack. Most languages provide heap management to programs. Satisfying memory requests is thus a joint affair between the language runtime and the kernel. In C, the interface to heap allocation is malloc() and friends, whereas in a garbage-collected language like C# the interface is the `new` keyword.

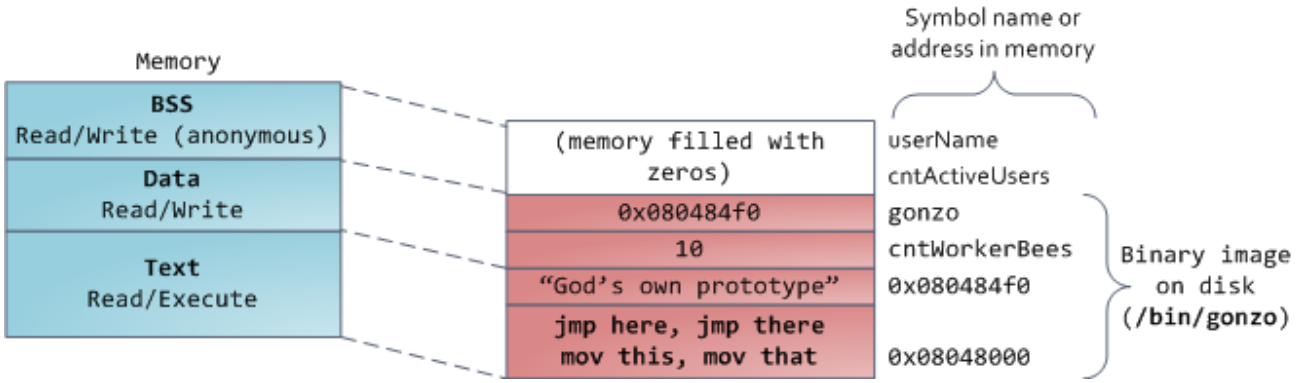
If there is enough space in the heap to satisfy a memory request, it can be handled by the language runtime without kernel involvement. Otherwise the heap is enlarged via the brk() system call (implementation) to make room for the requested block. Heap management is complex, requiring sophisticated algorithms that strive for speed and efficient memory usage in the face of our programs' chaotic allocation patterns. The time needed to service a heap request can vary substantially. Real-time systems have special-purpose allocators to deal with this problem. Heaps also become *fragmented*, shown below:

Finally, we get to the lowest segments of memory: BSS, data, and program text. Both BSS and data store contents for static (global) variables in C. The difference is that BSS stores the contents of *uninitialized* static variables, whose values are not set by the programmer in source code. The BSS memory area is anonymous: it does not map any file. If you say `static int cntActiveUsers`, the contents of `cntActiveUsers` live in the BSS.

The data segment, on the other hand, holds the contents for static variables initialized in source code. This memory area **is not anonymous**. It maps the part of the program's binary image that contains the initial static values given in source code. So if you say `static int cntWorkerBees = 10`, the contents of cntWorkerBees live in the data segment and start out as 10. Even though the data segment maps a file, it is a **private memory mapping**, which means that updates to memory are not reflected in the underlying file. This must be the case, otherwise assignments to global variables would change your on-disk binary image. Inconceivable!

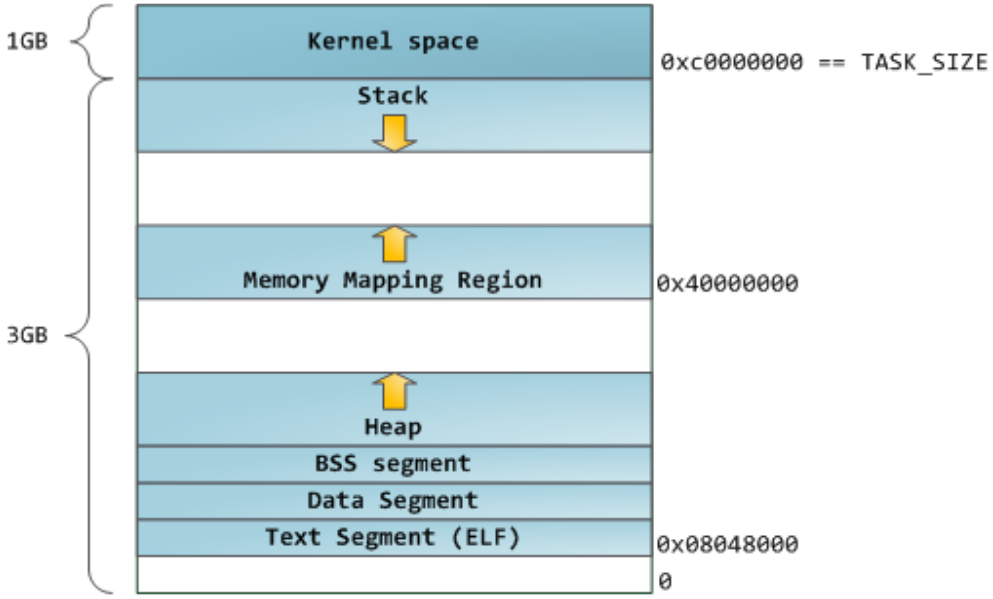
The data example in the diagram is trickier because it uses a pointer. In that case, the *contents* of pointer `gonzo` – a 4-byte memory address – live in the data segment. The actual string it points to does not, however. The string lives in the **text** segment, which is read-only and stores all of your code in addition to tidbits like string literals. The text segment also maps your binary file in memory, but writes to this area

earn your program a Segmentation Fault. This helps prevent pointer bugs, though not as effectively as avoiding C in the first place. Here's a diagram showing these segments and our example variables:



You can examine the memory areas in a Linux process by reading the file `/proc/pid_of_process/maps`. Keep in mind that a segment may contain many areas. For example, each memory mapped file normally has its own area in the mmap segment, and dynamic libraries have extra areas similar to BSS and data. The next post will clarify what 'area' really means. Also, sometimes people say "data segment" meaning all of data + bss + heap.

You can examine binary images using the nm and objdump commands to display symbols, their addresses, segments, and so on. Finally, the virtual address layout described above is the "flexible" layout in Linux, which has been the default for a few years. It assumes that we have a value for `RLIMIT_STACK`. When that's not the case, Linux reverts back to the "classic" layout shown below:



That's it for virtual address space layout. The next post discusses how the kernel keeps track of these memory areas. Coming up we'll look at memory mapping, how file reading and writing ties into all this and what memory usage figures mean.

January 27, 2009 | Filed Under Internals, Linux, Software Illustrated

149 Comments

The Divided House of GPL

Back in 2000 The Onion made fun of Libertarians and published this nugget:

Joking aside, a powerful idea attracts a wide range of people. Copyleft is such an idea. It turns copyright on its head by using authorship rights to *enforce* the public's ability to distribute, modify and use the copyrighted work, rather than to *curb* it as is normally the case. Several copyleft licenses exist, the most prominent in software being the GPL, first released by Richard Stallman in 1989.

There are two main factions supporting the GPL: the pragmatic camp of Linus Torvalds and the ideological camp of Richard Stallman. The Linus camp sees copyleft as the enabler of a superior way to produce software, in which distributed and open development takes place because people are encouraged and protected by the license. The individual programmer is assured that their contributions must always remain a public good and cannot be coopted for private gain. Others may profit from the software, sell it, or support it, but the source code must be available, modifiable and distributable. This is a powerful motivator, the same force that makes people help the Wikipedia but not for-profit outfits.

For large-scale development involving multiple corporations, copyleft solves a type of free rider problem by ensuring that all participants must give back to the common pool of development. This protects investments and tends to boost returns, and a brief look at the Linux Kernel Mailing List shows that major tech companies are happy to play along. I bet you can do some game theory and prove some results for cooperation under GPL.

To the Linus camp the GPL is a means to foster this ecosystem, the end being better software. There are no moral imperatives or political reasons behind the whole thing, which surprises some people. Proprietary software is "alchemy" while open source is science. Here's Linus:

In my book, what matters is what you do – whether you want to sell things is your personal choice, but even more importantly it is not a moral negative or positive. I'm a big believer in open source as creating good stuff, but I don't think it's a moral issue. It's engineering.

So I think open source tends to become technically better over time (but it does take time), but I don't think it's a moral imperative. I do open source because it's fun, and because I think it makes sense in the long run.

And here's more:

Just to explain the fundamental issue: To me, the GPL really boils down to "I give out code, I want you to do the same." The thing that makes me not want to use the GPLv3 in its current form is that it really tries to move more toward the "software freedom" goals. For example, the GPLv2 in no way limits your use of the software. If you're a mad scientist, you can use GPLv2'd software for your evil plans to take over the world ("Sharks with lasers on their heads!!"), and the GPLv2 just says that you have to give source code back. And that's OK by me. I like sharks with lasers. I just want the mad scientists of the world to pay me back in kind. I made source code available to them, they have to make their changes to it available to me. After that, they can fry me with their shark-mounted lasers all they want. This is where the GPLv3 diverges. It limits how you can use the software.

The Stallman camp, however, sees GPL-licensed software as the *end* itself. They claim that software should be free on moral grounds, citing several reasons. Hence it matters not whether the software or the process are superior. One must use free software regardless because it is the right thing to do, while proprietary software is inherently immoral. Here's Stallman:

Supporters of open source (which I am not) promote a "development model" in which users participate in development, claiming that this typically makes software "better" — and when they say "better", they mean that only in a technical sense. By using the term that way, implicitly, they say that only practical convenience matters — not your freedom.

I don't say they are wrong, but they are missing the point. If you neglect the values of freedom and social solidarity, and appreciate only powerful reliable software, you are making a terrible mistake.

The fact that Torvalds says "open source" instead of "free software" shows where he is coming from. I wrote the GNU GPL to defend freedom for all users of all versions of a program. I developed version 3 to do that job better and protect against new threats. Torvalds says he rejects this goal; that's probably why he doesn't appreciate GPL version 3. I respect his right to express his views, even though I think they are foolish. However, if you don't want to lose your freedom, you had better not follow him.

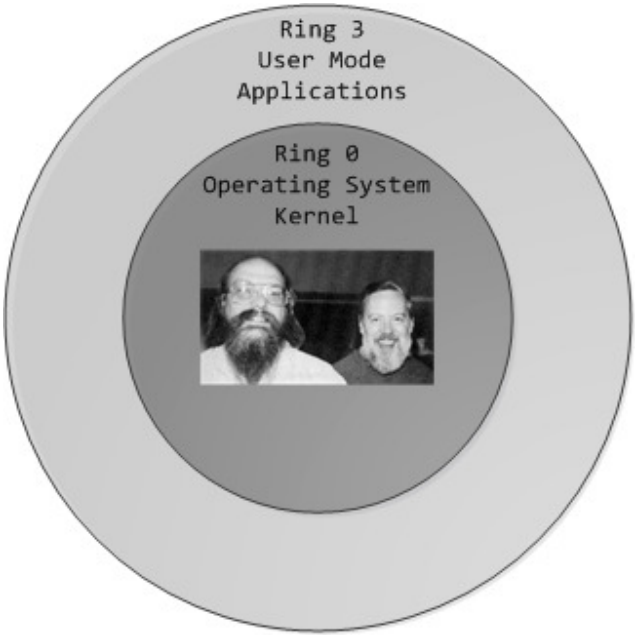
Discussions of copyleft often blur these two camps. For much development this is irrelevant – the license stands on its own irrespective of people's motivations. But this schism explains periodical battles like the GPLv3 controversy, the endless flames when a proprietary source control tool was used for the kernel, and the GNU/Linux naming controversy. The distinction is also important when thinking about free/open source software and what to make of it.

October 16, 2008 | Filed Under Culture, Linux

15 Comments

CPU Rings, Privilege, and Protection

You probably know intuitively that applications have limited powers in Intel x86 computers and that only operating system code can perform certain tasks, but do you know how this really works? This post takes a look at x86 **privilege levels**, the mechanism whereby the OS and CPU conspire to restrict what user-mode programs can do. There are four privilege levels, numbered 0 (most privileged) to 3 (least privileged), and three main resources being protected: memory, I/O ports, and the ability to execute certain machine instructions. At any given time, an x86 CPU is running in a specific privilege level, which determines what code can and cannot do. These privilege levels are often described as protection rings, with the innermost ring corresponding to highest privilege. Most modern x86 kernels use only two privilege levels, 0 and 3:



x86 Protection Rings

About 15 machine instructions, out of dozens, are restricted by the CPU to ring zero. Many others have limitations on their operands. These instructions can subvert the protection mechanism or otherwise foment chaos if allowed in user mode, so they are reserved to the kernel. An attempt to run them outside of ring zero causes a general-protection exception, like when a program uses invalid memory addresses. Likewise, access to memory and I/O ports is restricted based on privilege level. But before we look at protection mechanisms, let's see *exactly* how the CPU keeps track of the current privilege level, which involves the segment selectors from the previous post. Here they are:

Segment Selectors – Data and Code

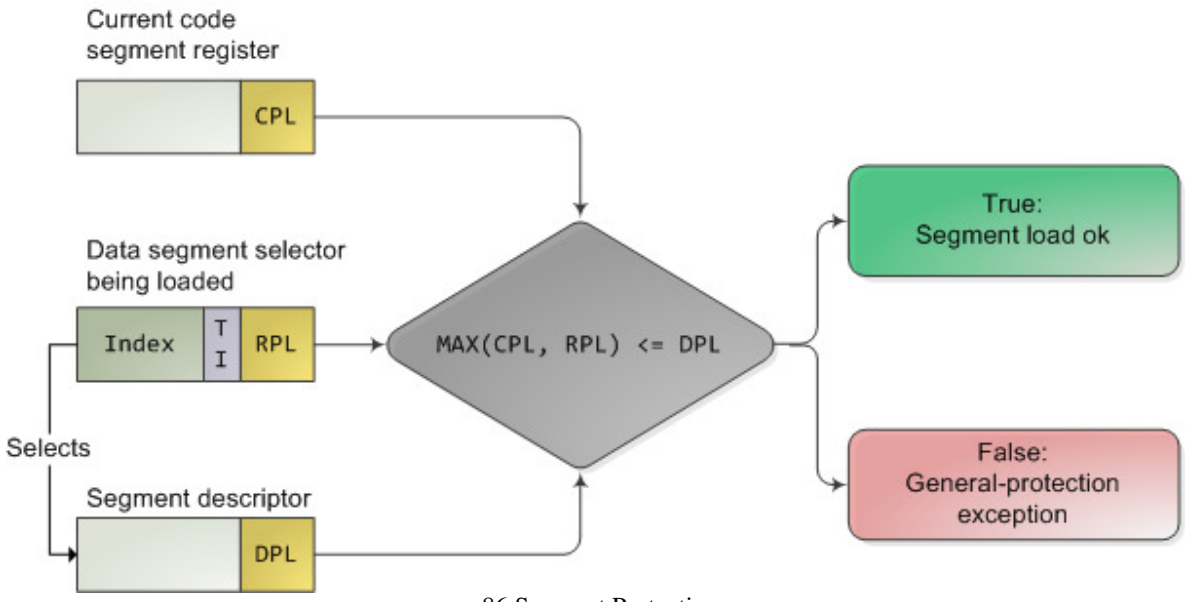
The full contents of data segment selectors are loaded directly by code into various segment registers such as ss (stack segment register) and ds (data segment register). This includes the contents of the Requested Privilege Level (RPL) field, whose meaning we tackle in a bit. The code segment register (cs) is, however, magical. First, its contents cannot be set directly by load instructions such as mov, but rather only by instructions that alter the flow of program execution, like call. Second, and importantly for us, instead of an RPL field that can be set by code, cs has a **Current Privilege Level** (CPL) field maintained by the CPU itself. This 2-bit CPL field in the code segment register **is always equal to** the CPU's current privilege level. The Intel docs wobble a little on this fact, and sometimes online documents confuse the issue, but that's the hard and fast rule. At any time, no matter what's going on in the CPU, a look at the CPL in cs will tell you the privilege level code is running with.

Keep in mind that the **CPU privilege level has nothing to do with operating system users**. Whether you're root, Administrator, guest, or a regular user, *it does not matter*. **All user code runs in ring 3** and **all kernel code runs in ring 0**, regardless of the OS user on whose behalf the code operates. Sometimes certain

kernel tasks can be pushed to user mode, for example user-mode device drivers in Windows Vista, but these are just special processes doing a job for the kernel and can usually be killed without major consequences.

Due to restricted access to memory and I/O ports, user mode can do almost *nothing* to the outside world without calling on the kernel. It can't open files, send network packets, print to the screen, or allocate memory. User processes run in a severely limited sandbox set up by the gods of ring zero. That's why it's *impossible*, by design, for a process to leak memory beyond its existence or leave open files after it exits. All of the data structures that control such things – memory, open files, etc – cannot be touched directly by user code; once a process finishes, the sandbox is torn down by the kernel. That's why our servers can have 600 days of uptime – as long as the hardware and the kernel don't crap out, stuff can run for ever. This is also why Windows 95 / 98 crashed so much: it's not because "M$ sucks" but because important data structures were left accessible to user mode for compatibility reasons. It was probably a good trade-off at the time, albeit at high cost.

The CPU protects memory at two crucial points: when a segment selector is loaded and when a page of memory is accessed with a linear address. Protection thus mirrors memory address translation where both segmentation and paging are involved. When a data segment selector is being loaded, the check below takes place:



x86 Segment Protection

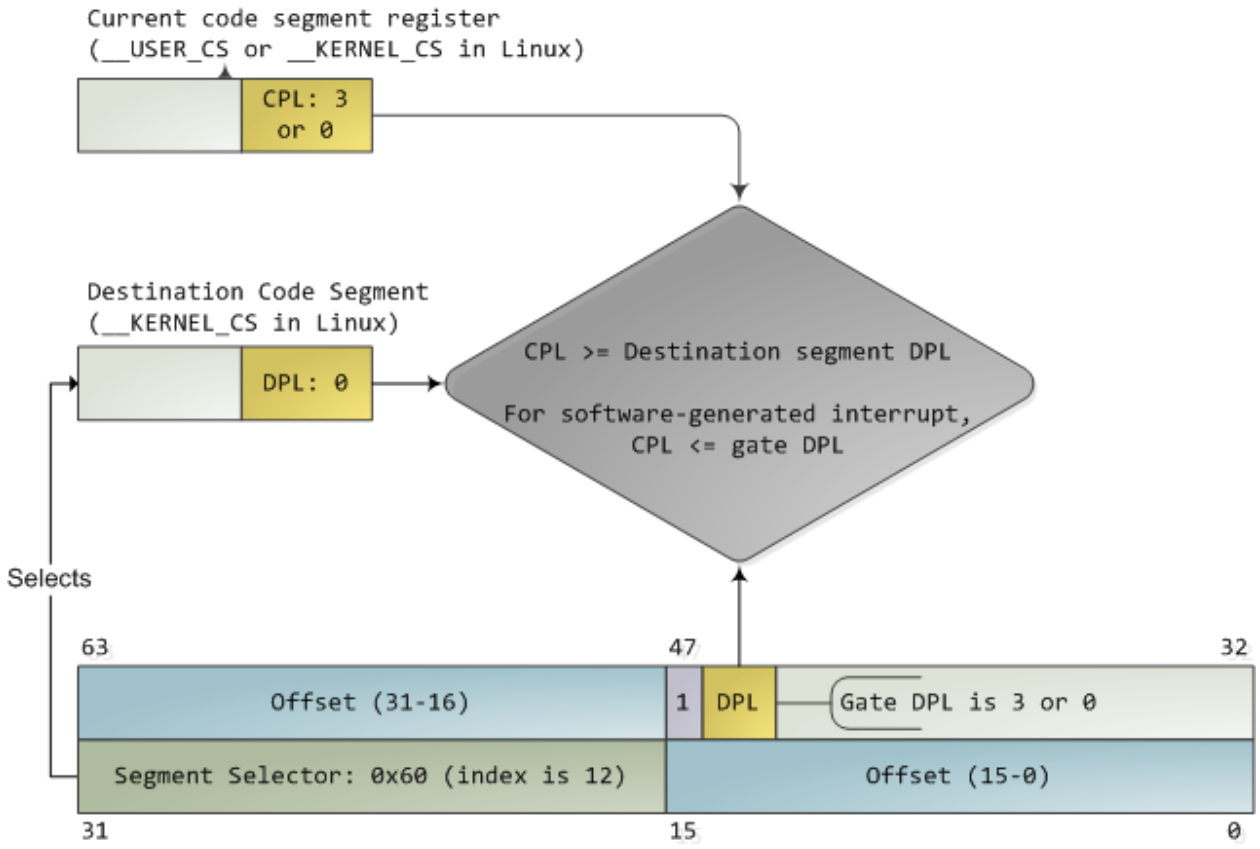
Since a higher number means less privilege, MAX() above picks the least privileged of CPL and RPL, and compares it to the descriptor privilege level (DPL). If the DPL is higher or equal, then access is allowed. The idea behind RPL is to allow kernel code to load a segment using lowered privilege. For example, you could use an RPL of 3 to ensure that a given operation uses segments accessible to user-mode. The exception is for the stack segment register ss, for which the three of CPL, RPL, and DPL must match exactly.

In truth, segment protection scarcely matters because modern kernels use a flat address space where the user-mode segments can reach the entire linear address space. Useful memory protection is done in the paging unit when a linear address is converted into a physical address. Each memory page is a block of bytes described by a **page table entry** containing two fields related to protection: a supervisor flag and a read/write flag. The supervisor flag is the primary x86 memory protection mechanism used by kernels. When it is on, the page cannot be accessed from ring 3. While the read/write flag isn't as important for enforcing privilege, it's still useful. When a process is loaded, pages storing binary images (code) are marked as read only, thereby catching some pointer errors if a program attempts to write to these pages. This flag is also used to implement copy on write when a process is forked in Unix. Upon forking, the

parent's pages are marked read only and shared with the forked child. If either process attempts to write to the page, the processor triggers a fault and the kernel knows to duplicate the page and mark it read/write for the writing process.

Finally, we need a way for the CPU to switch between privilege levels. If ring 3 code could transfer control to arbitrary spots in the kernel, it would be easy to subvert the operating system by jumping into the wrong (right?) places. A controlled transfer is necessary. This is accomplished via **gate descriptors** and via the **sysenter** instruction. A gate descriptor is a segment descriptor of type system, and comes in four sub-types: call-gate descriptor, interrupt-gate descriptor, trap-gate descriptor, and task-gate descriptor. Call gates provide a kernel entry point that can be used with ordinary call and jmp instructions, but they aren't used much so I'll ignore them. Task gates aren't so hot either (in Linux, they are only used in double faults, which are caused by either kernel or hardware problems).

That leaves two juicier ones: interrupt and trap gates, which are used to handle hardware interrupts (*e.g.*, keyboard, timer, disks) and exceptions (*e.g.*, page faults, divide by zero). I'll refer to both as an "interrupt". These gate descriptors are stored in the **Interrupt Descriptor Table** (IDT). Each interrupt is assigned a number between 0 and 255 called a **vector**, which the processor uses as an index into the IDT when figuring out which gate descriptor to use when handling the interrupt. Interrupt and trap gates are nearly identical. Their format is shown below along with the privilege checks enforced when an interrupt happens. I filled in some values for the Linux kernel to make things concrete.



Interrupt Descriptor with Privilege Check

Both the DPL and the segment selector in the gate regulate access, while segment selector plus offset together nail down an entry point for the interrupt handler code. Kernels normally use the segment selector for the kernel code segment in these gate descriptors. An interrupt can **never** transfer control from a more-privileged to a less-privileged ring. Privilege must either stay the same (when the kernel itself is interrupted) or be elevated (when user-mode code is interrupted). In either case, the resulting CPL will be equal to to the DPL of the destination code segment; if the CPL changes, a stack switch also occurs. If an interrupt is

triggered by code via an instruction like **int n**, one more check takes place: the gate DPL must be at the same or lower privilege as the CPL. This prevents user code from triggering random interrupts. If these checks fail – you guessed it – a general-protection exception happens. All Linux interrupt handlers end up running in ring zero.

During initialization, the Linux kernel first sets up an IDT in setup_idt() that ignores all interrupts. It then uses functions in include/asm-x86/desc.h to flesh out common IDT entries in arch/x86/kernel/traps_32.c. In Linux, a gate descriptor with "system" in its name is accessible from user mode and its set function uses a DPL of 3. A "system gate" is an Intel trap gate accessible to user mode. Otherwise, the terminology matches up. Hardware interrupt gates are not set here however, but instead in the appropriate drivers.

Three gates are accessible to user mode: vectors 3 and 4 are used for debugging and checking for numeric overflows, respectively. Then a system gate is set up for the SYSCALL_VECTOR, which is 0×80 for the x86 architecture. This was *the mechanism* for a process to transfer control to the kernel, to make a *system call*, and back in the day I applied for an "int 0×80″ vanity license plate 😀. Starting with the Pentium Pro, the **sysenter** instruction was introduced as a faster way to make system calls. It relies on special-purpose CPU registers that store the code segment, entry point, and other tidbits for the kernel system call handler. When sysenter is executed the CPU does no privilege checking, going immediately into CPL 0 and loading new values into the registers for code and stack (cs, eip, ss, and esp). Only ring zero can load the sysenter setup registers, which is done in enable_sep_cpu().

Finally, when it's time to return to ring 3, the kernel issues an **iret** or **sysexit** instruction to return from interrupts and system calls, respectively, thus leaving ring 0 and resuming execution of user code with a CPL of 3. Vim tells me I'm approaching 1,900 words, so I/O port protection is for another day. This concludes our tour of x86 rings and protection. Thanks for reading!

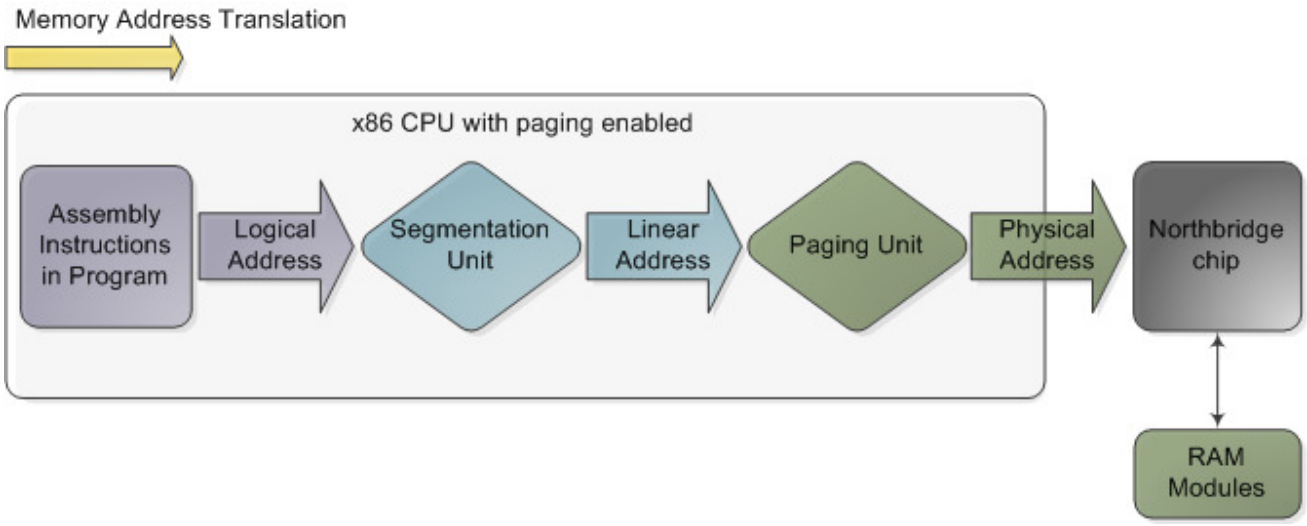
August 20, 2008 | Filed Under Internals, Linux, Security, Software Illustrated

66 Comments

Memory Translation and Segmentation

This post is the first in a series about memory and protection in Intel-compatible (x86) computers, going further down the path of how kernels work. As in the boot series, I'll link to Linux kernel sources but give Windows examples as well (sorry, I'm ignorant about the BSDs and the Mac, but most of the discussion applies). Let me know what I screw up.

In the chipsets that power Intel motherboards, memory is accessed by the CPU via the front side bus, which connects it to the northbridge chip. The memory addresses exchanged in the front side bus are **physical memory addresses**, raw numbers from zero to the top of the available physical memory. These numbers are mapped to physical RAM sticks by the northbridge. Physical addresses are concrete and final – no translation, no paging, no privilege checks – you put them on the bus and that's that. Within the CPU, however, programs use **logical memory addresses**, which must be translated into physical addresses before memory access can take place. Conceptually address translation looks like this:



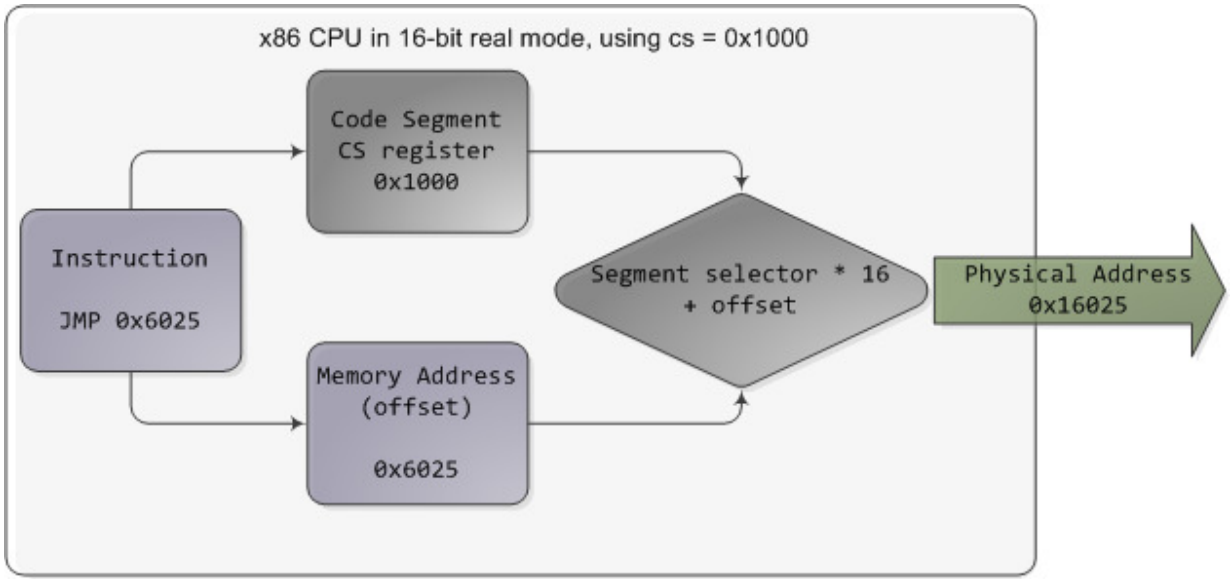
Memory address translation in x86 CPUs with paging enabled

This is **not** a physical diagram, only a depiction of the address translation process, specifically for when the CPU has paging enabled. If you turn off paging, the output from the segmentation unit is already a physical address; in 16-bit real mode that is always the case. Translation starts when the CPU executes an instruction that refers to a memory address. The first step is translating that logic address into a **linear address**. But why go through this step instead of having software use linear (or physical) addresses directly? For roughly the same reason humans have an appendix whose primary function is getting infected. It's a wrinkle of evolution. To really make sense of x86 segmentation we need to go back to 1978.

The original 8086 had 16-bit registers and its instructions used mostly 8-bit or 16-bit operands. This allowed code to work with 2^{16} bytes, or 64K of memory, yet Intel engineers were keen on letting the CPU use more memory without expanding the size of registers and instructions. So they introduced *segment registers* as a means to tell the CPU *which* 64K chunk of memory a program's instructions were going to work on. It was a reasonable solution: first you load a segment register, effectively saying "here, I want to work on the memory chunk starting at X"; afterwards, 16-bit memory addresses used by your code are interpreted as offsets into your chunk, or segment. There were four segment registers: one for the stack (ss), one for program code (cs), and two for data (ds, es). Most programs were small enough back then to fit their whole stack, code, and data each in a 64K segment, so segmentation was often transparent.

Nowadays segmentation is still present and is always enabled in x86 processors. Each instruction that touches memory implicitly uses a segment register. For example, a jump instruction uses the code segment register (cs) whereas a stack push instruction uses the stack segment register (ss). In most cases you can explicitly override the segment register used by an instruction. Segment registers store 16-bit **segment selectors**; they can be loaded directly with instructions like MOV. The sole exception is cs, which can only be changed by instructions that affect the flow of execution, like CALL or JMP. Though segmentation is always on, it works differently in real mode versus protected mode.

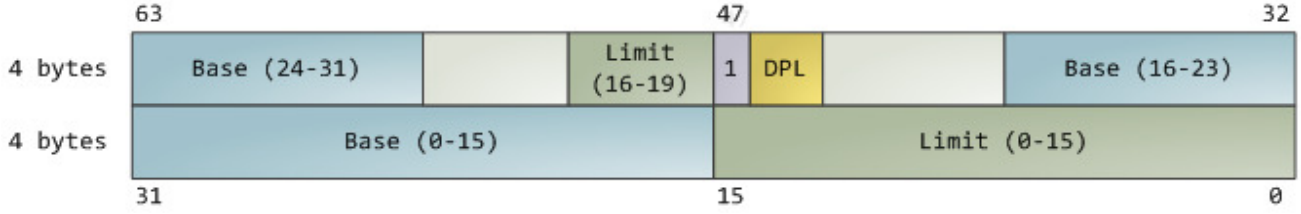
In real mode, such as during early boot, the segment selector is a 16-bit number specifying the physical memory address for the start of a segment. This number must somehow be scaled, otherwise it would also be limited to 64K, defeating the purpose of segmentation. For example, the CPU could use the segment selector as the 16 most significant bits of the physical memory address (by shifting it 16 bits to the left, which is equivalent to multiplying by 2^{16}). This simple rule would enable segments to address 4 gigs of memory in 64K chunks, but it would increase chip packaging costs by requiring more physical address pins in the processor. So Intel made the decision to multiply the segment selector by only 2^4 (or 16), which in a single stroke confined memory to about 1MB and unduly complicated translation. Here's an example showing a jump instruction where cs contains 0×1000:



Real mode segmentation

Real mode segment starts range from 0 all the way to 0xFFFF0 (16 bytes short of 1 MB) in 16-byte increments. To these values you add a 16-bit offset (the logical address) between 0 and 0xFFFF. It follows that there are multiple segment/offset combinations pointing to the same memory location, and physical addresses fall above 1MB if your segment is high enough (see the infamous A20 line). Also, when writing C code in real mode a far pointer is a pointer that contains both the segment selector *and* the logical address, which allows it to address 1MB of memory. Far indeed. As programs started getting bigger and outgrowing 64K segments, segmentation and its strange ways complicated development for the x86 platform. This may all sound quaintly odd now but it has driven programmers into the wretched depths of madness.

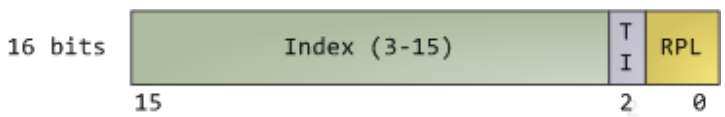
In 32-bit protected mode, a segment selector is no longer a raw number, but instead it contains an index into a table of **segment descriptors**. The table is simply an array containing 8-byte records, where each record describes one segment and looks thus:



Segment descriptor

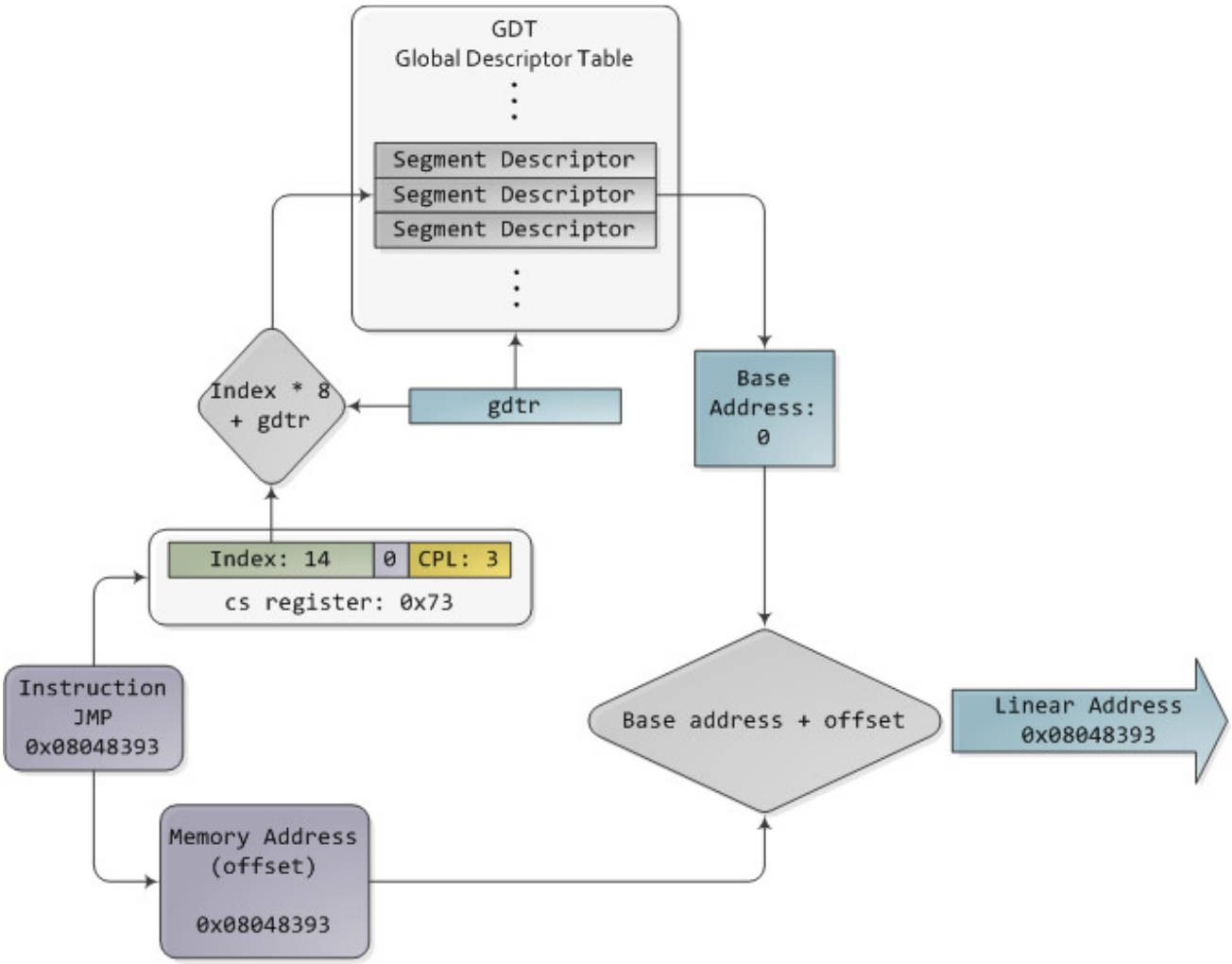
There are three types of segments: code, data, and system. For brevity, only the common features in the descriptor are shown here. The **base address** is a 32-bit linear address pointing to the beginning of the segment, while the **limit** specifies how big the segment is. Adding the base address to a logical memory address yields a linear address. DPL is the descriptor privilege level; it is a number from 0 (most privileged, kernel mode) to 3 (least privileged, user mode) that controls access to the segment.

These segment descriptors are stored in two tables: the **Global Descriptor Table** (GDT) and the **Local Descriptor Table** (LDT). Each CPU (or core) in a computer contains a register called **gdtr** which stores the linear memory address of the first byte in the GDT. To choose a segment, you must load a segment register with a **segment selector** in the following format:



Segment Selector

The TI bit is 0 for the GDT and 1 for the LDT, while the index specifies the desired segment selector within the table. We'll deal with RPL, Requested Privilege Level, later on. Now, come to think of it, when the CPU is in 32-bit mode registers and instructions can address the entire linear address space *anyway*, so there's really no need to give them a push with a base address or other shenanigan. So why not set the base address to zero and let logical addresses coincide with linear addresses? Intel docs call this "flat model" and it's exactly what modern x86 kernels do (they use the basic flat model, specifically). Basic flat model is equivalent to disabling segmentation when it comes to translating memory addresses. So in all its glory, here's the jump example running in 32-bit protected mode, with real-world values for a Linux user-mode app:



Protected Mode Segmentation

The contents of a segment descriptor are cached once they are accessed, so there's no need to actually read the GDT in subsequent accesses, which would kill performance. Each segment register has a hidden part to store the cached descriptor that corresponds to its segment selector. For more details, including more info on the LDT, see chapter 3 of the Intel System Programming Guide Volume 3a. Volumes 2a and 2b, which cover every x86 instruction, also shed light on the various types of x86 addressing operands – 16-bit, 16-bit with segment selector (which can be used by far pointers), 32-bit, etc.

In Linux, only 3 segment descriptors are used during boot. They are defined with the GDT_ENTRY macro and stored in the boot_gdt array. Two of the segments are flat, addressing the entire 32-bit space: a code segment loaded into cs and a data segment loaded into the other segment registers. The third segment is a system segment called the Task State Segment. After boot, each CPU has its own copy of the GDT. They are all nearly identical, but a few entries change depending on the running process. You can see the layout of the Linux GDT in segment.h and its instantiation is here. There are four primary GDT entries: two flat

ones for code and data in kernel mode, and another two for user mode. When looking at the Linux GDT, notice the holes inserted on purpose to align data with CPU cache lines – an artifact of the von Neumann bottleneck that has become a plague. Finally, the classic "Segmentation fault" Unix error message is *not* due to x86-style segments, but rather invalid memory addresses normally detected by the paging unit – alas, topic for an upcoming post.

Intel deftly worked around their original segmentation kludge, offering a flexible way for us to choose whether to segment or go flat. Since coinciding logical and linear addresses are simpler to handle, they became standard, such that 64-bit mode now enforces a flat linear address space. But even in flat mode segments are still crucial for x86 protection, the mechanism that defends the kernel from user-mode processes and every process from each other. It's a dog eat dog world out there! In the next post, we'll take a peek at protection levels and how segments implement them.

Thanks to Nate Lawson for a correction in this post.

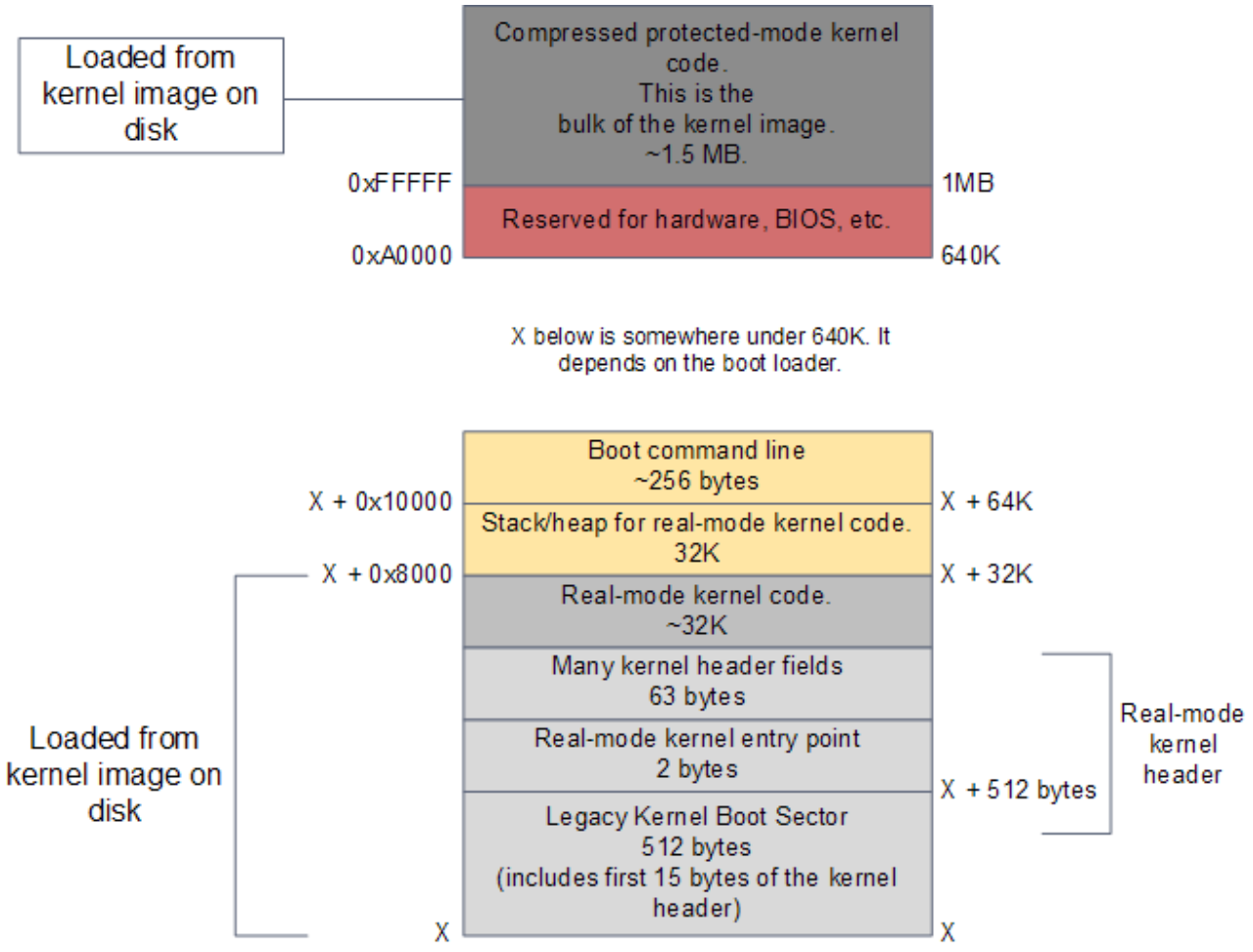
August 12, 2008 | Filed Under Internals, Linux, Software Illustrated

50 Comments

The Kernel Boot Process

The previous post explained how computers boot up right up to the point where the boot loader, after stuffing the kernel image into memory, is about to jump into the kernel entry point. This last post about booting takes a look at the guts of the kernel to see how an operating system starts life. Since I have an empirical bent I'll link heavily to the sources for Linux kernel 2.6.25.6 at the Linux Cross Reference. The sources are very readable if you are familiar with C-like syntax; even if you miss some details you can get the gist of what's happening. The main obstacle is the lack of context around some of the code, such as when or why it runs or the underlying features of the machine. I hope to provide a bit of that context. Due to brevity (hah!) a lot of fun stuff – like interrupts and memory – gets only a nod for now. The post ends with the highlights for the Windows boot.

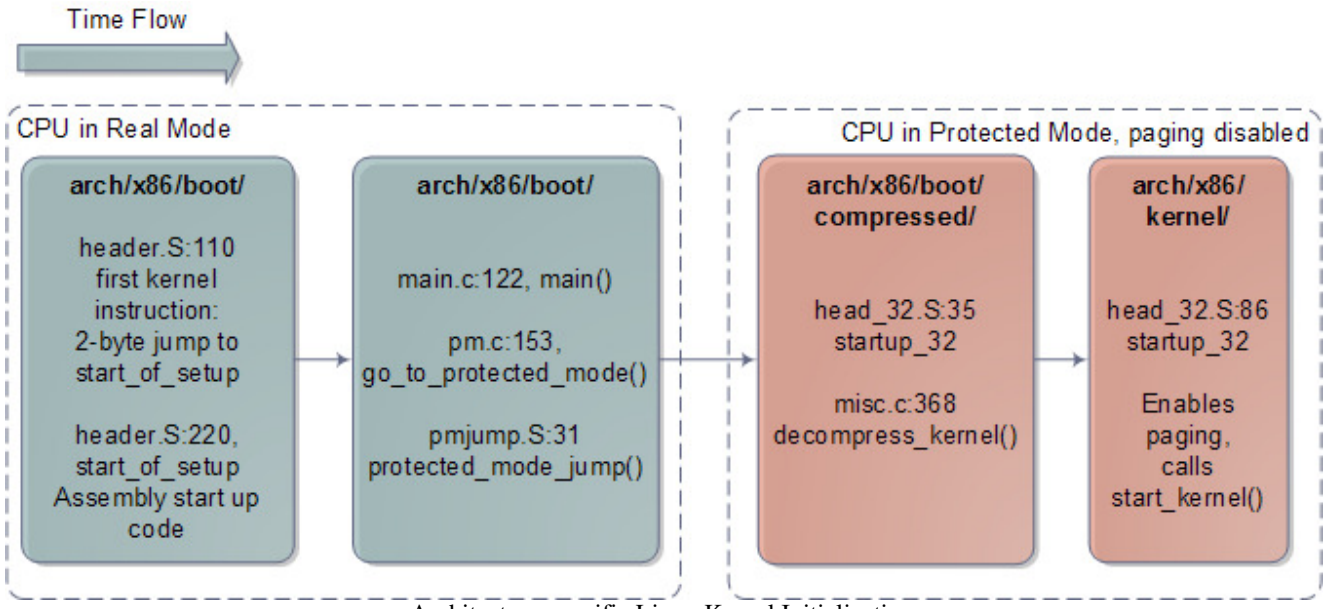
At this point in the Intel x86 boot story the processor is running in real-mode, is able to address 1 MB of memory, and RAM looks like this for a modern Linux system:



RAM contents after boot loader is done

The kernel image has been loaded to memory by the boot loader using the BIOS disk I/O services. This image is an exact copy of the file in your hard drive that contains the kernel, e.g. **/boot/vmlinuz-2.6.22-14-server**. The image is split into two pieces: a small part containing the real-mode kernel code is loaded below the 640K barrier; the bulk of the kernel, which runs in protected mode, is loaded after the first megabyte of memory.

The action starts in the real-mode kernel header pictured above. This region of memory is used to implement the Linux boot protocol between the boot loader and the kernel. Some of the values there are read by the boot loader while doing its work. These include amenities such as a human-readable string containing the kernel version, but also crucial information like the size of the real-mode kernel piece. The boot loader also *writes* values to this region, such as the memory address for the command-line parameters given by the user in the boot menu. Once the boot loader is finished it has filled in all of the parameters required by the kernel header. It's then time to jump into the kernel entry point. The diagram below shows the code sequence for the kernel initialization, along with source directories, files, and line numbers:



Architecture-specific Linux Kernel Initialization

The early kernel start-up for the Intel architecture is in file arch/x86/boot/header.S. It's in assembly language, which is rare for the kernel at large but common for boot code. The start of this file actually contains boot sector code, a left over from the days when Linux could work without a boot loader. Nowadays this boot sector, if executed, only prints a "bugger_off_msg" to the user and reboots. Modern boot loaders ignore this legacy code. After the boot sector code we have the first 15 bytes of the real-mode kernel header; these two pieces together add up to 512 bytes, the size of a typical disk sector on Intel hardware.

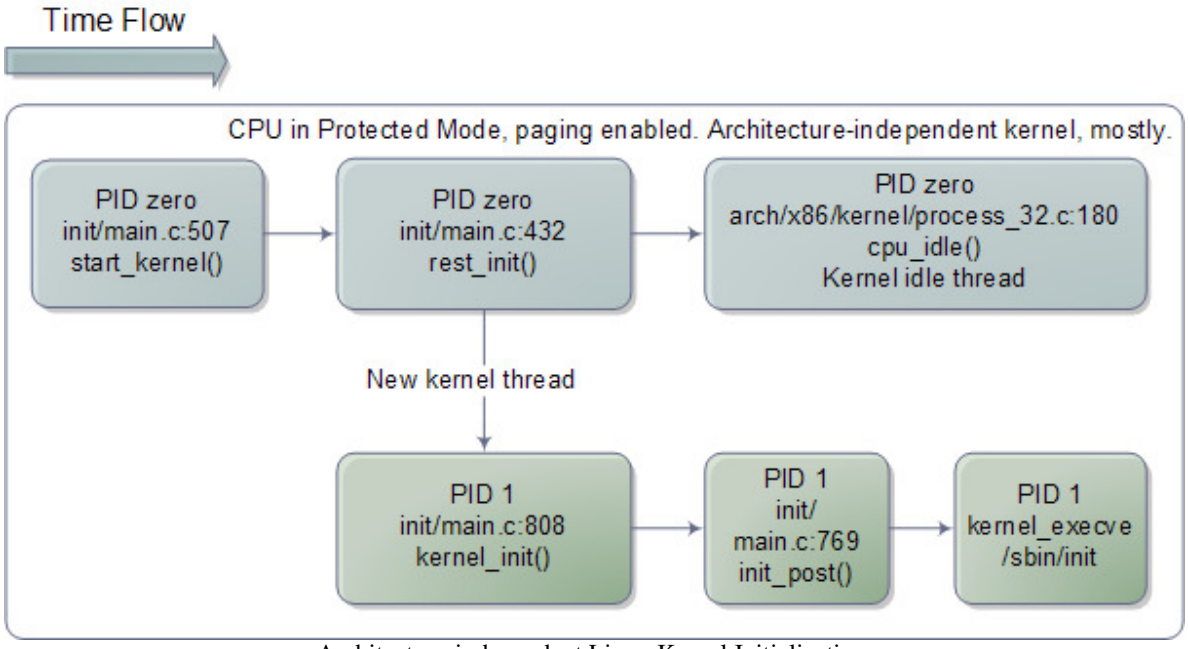
After these 512 bytes, at offset 0×200, we find the very first instruction that runs as part of the Linux kernel: the real-mode entry point. It's in header.S:110 and it is a 2-byte jump written directly in machine code as 0x3aeb. You can verify this by running hexdump on your kernel image and seeing the bytes at that offset – just a sanity check to make sure it's not all a dream. The boot loader jumps into this location when it is finished, which in turn jumps to header.S:229 where we have a regular assembly routine called start_of_setup. This short routine sets up a stack, zeroes the bss segment (the area that contains static variables, so they start with zero values) for the real-mode kernel and then jumps to good old C code at arch/x86/boot/main.c:122.

main() does some house keeping like detecting memory layout, setting a video mode, etc. It then calls go_to_protected_mode(). Before the CPU can be set to protected mode, however, a few tasks must be done. There are two main issues: interrupts and memory. In real-mode the interrupt vector table for the processor is always at memory address 0, whereas in protected mode the location of the interrupt vector table is stored in a CPU register called IDTR. Meanwhile, the translation of logical memory addresses (the ones programs manipulate) to linear memory addresses (a raw number from 0 to the top of the memory) is different between real-mode and protected mode. Protected mode requires a register called GDTR to be loaded with the address of a Global Descriptor Table for memory. So go_to_protected_mode() calls setup_idt() and setup_gdt() to install a temporary interrupt descriptor table and global descriptor table.

We're now ready for the plunge into protected mode, which is done by protected_mode_jump, another assembly routine. This routine enables protected mode by setting the PE bit in the CR0 CPU register. At this point we're running with paging **disabled**; paging is an optional feature of the processor, even in protected mode, and there's no need for it yet. What's important is that we're no longer confined to the 640K barrier and can now address up to 4GB of RAM. The routine then calls the 32-bit kernel entry point, which is startup_32 for compressed kernels. This routine does some basic register initializations and calls decompress_kernel(), a C function to do the actual decompression.

decompress_kernel() prints the familiar "Decompressing Linux…" message. Decompression happens in-place and once it's finished the uncompressed kernel image has overwritten the compressed one pictured in the first diagram. Hence the uncompressed contents also start at 1MB. decompress_kernel() then prints "done." and the comforting "Booting the kernel." By "Booting" it means a jump to the final entry point in this whole story, given to Linus by God himself atop Mountain Halti, which is the protected-mode kernel entry point at the start of the second megabyte of RAM (0×100000). That sacred location contains a routine called, uh, startup_32. But *this* one is in a different directory, you see.

The second incarnation of startup_32 is also an assembly routine, but it contains 32-bit mode initializations. It clears the bss segment for the protected-mode kernel (which is the *true* kernel that will now run until the machine reboots or shuts down), sets up the final global descriptor table for memory, builds page tables so that paging can be turned on, enables paging, initializes a stack, creates the final interrupt descriptor table, and finally jumps to to the architecture-independent kernel start-up, start_kernel(). The diagram below shows the code flow for the last leg of the boot:



Architecture-independent Linux Kernel Initialization

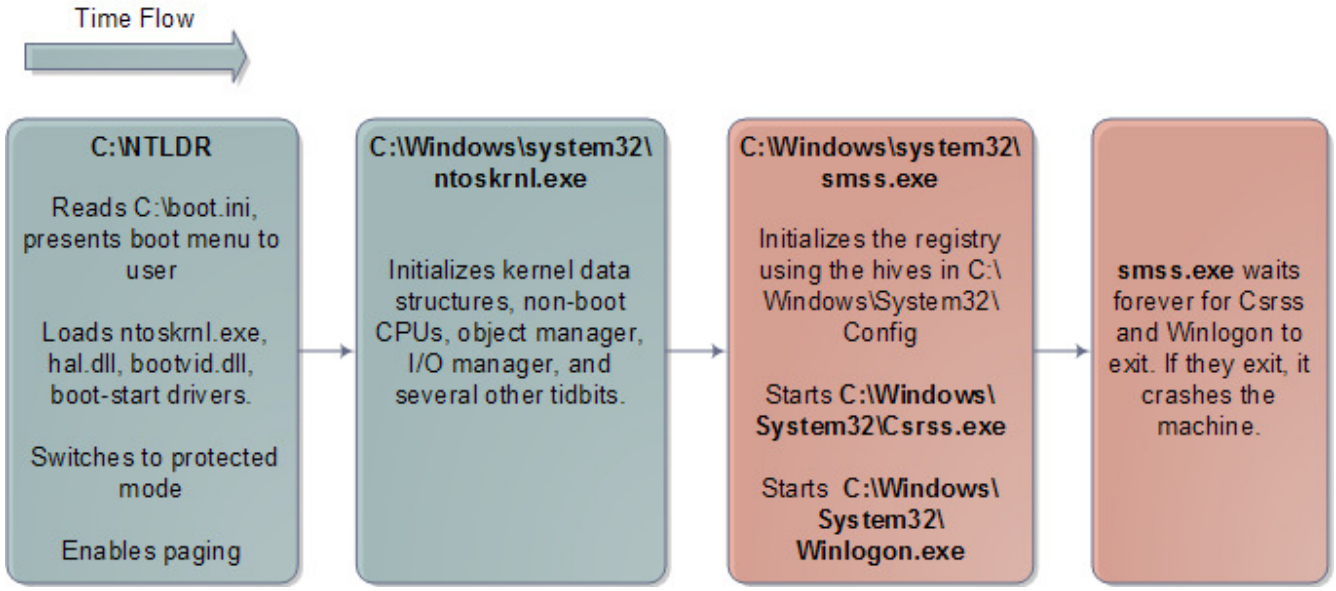
start_kernel() looks more like typical kernel code, which is nearly all C and machine independent. The function is a long list of calls to initializations of the various kernel subsystems and data structures. These include the scheduler, memory zones, time keeping, and so on. start_kernel() then calls rest_init(), at which point things are almost all working. rest_init() creates a kernel thread passing another function, kernel_init(), as the entry point. rest_init() then calls schedule() to kickstart task scheduling and goes to sleep by calling cpu_idle(), which is the idle thread for the Linux kernel. cpu_idle() runs forever and so does process zero, which hosts it. Whenever there is work to do – a runnable process – process zero gets booted out of the CPU, only to return when no runnable processes are available.

But here's the kicker for us. This idle loop is the end of the long thread we followed since boot, it's the final descendent of the very first *jump* executed by the processor after power up. All of this mess, from reset vector to BIOS to MBR to boot loader to real-mode kernel to protected-mode kernel, all of it leads right here, jump by jump by jump it ends in the idle loop for the boot processor, cpu_idle(). Which is really kind of cool. However, this can't be the whole story otherwise the computer would do no work.

At this point, the kernel thread started previously is ready to kick in, displacing process 0 and its idle thread. And so it does, at which point kernel_init() starts running since it was given as the thread entry point. kernel_init() is responsible for initializing the remaining CPUs in the system, which have been halted since boot. All of the code we've seen so far has been executed in a single CPU, called the boot processor. As the

other CPUs, called application processors, are started they come up in real-mode and must run through several initializations as well. Many of the code paths are common, as you can see in the code for startup_32, but there are slight forks taken by the late-coming application processors. Finally, kernel_init() calls init_post(), which tries to execute a user-mode process in the following order: /sbin/init, /etc/init, /bin/init, and /bin/sh. If all fail, the kernel will panic. Luckily init is usually there, and starts running as PID 1. It checks its configuration file to figure out which processes to launch, which might include X11 Windows, programs for logging in on the console, network daemons, and so on. Thus ends the boot process as yet another Linux box starts running somewhere. May your uptime be long and untroubled.

The process for Windows is similar in many ways, given the common architecture. Many of the same problems are faced and similar initializations must be done. When it comes to boot one of the biggest differences is that Windows packs all of the real-mode kernel code, and some of the initial protected mode code, into the boot loader itself (C:\NTLDR). So instead of having two regions in the same kernel image, Windows uses different binary images. Plus Linux completely separates boot loader and kernel; in a way this automatically falls out of the open source process. The diagram below shows the main bits for the Windows kernel:



Windows Kernel Initialization

The Windows user-mode start-up is naturally very different. There's no /sbin/init, but rather Csrss.exe and Winlogon.exe. Winlogon spawns **Services.exe**, which starts all of the Windows Services, and Lsass.exe, the local security authentication subsystem. The classic Windows login dialog runs in the context of Winlogon.

This is the end of this boot series. Thanks everyone for reading and for feedback. I'm sorry some things got superficial treatment; I've gotta start somewhere and only so much fits into blog-sized bites. But nothing like a day after the next; my plan is to do regular "Software Illustrated" posts like this series along with other topics. Meanwhile, here are some resources:

- The best, most important resource, is source code for real kernels, either Linux or one of the BSDs.
- Intel publishes excellent Software Developer's Manuals, which you can download for free.
- Understanding the Linux Kernel is a good book and walks through a lot of the Linux Kernel sources. It's getting outdated and it's dry, but I'd still recommend it to anyone who wants to grok the kernel. Linux Device Drivers is more fun, teaches well, but is limited in scope. Finally, Patrick Moroney suggested Linux Kernel Development by Robert Love in the comments for this post. I've heard other positive reviews for that book, so it sounds worth checking out.

- For Windows, the best reference by far is Windows Internals by David Solomon and Mark Russinovich, the latter of Sysinternals fame. This is a great book, well-written and thorough. The main downside is the lack of source code.

[Update: In a comment below, Nix covered a lot of ground on the initial root file system that I glossed over. Thanks to Marius Barbu for catching a mistake where I wrote "CR3" instead of GDTR]

June 23, 2008 | Filed Under Internals, Linux, Software Illustrated

95 Comments

Why Brazil Loves Linux

[Disclosure: I am a dual American/Brazilian citizen. I've used Linux since 1996 and Microsoft products since 1990. I like both platforms.]

Brazil often makes Linux-related headlines, the latest being the adoption of KDE in Brazilian public schools. It's clear that Brazil is enamored with Linux, but why? This is an important question for Microsoft since emerging markets are key to sales growth. Microsoft's Annual Report 2007 reported that "impressive growth included India, China, and Brazil which all delivered revenue growth that topped 40 percent", which is much faster than growth in developed countries. These markets are also friendly towards Linux and pose significant challenges for Microsoft. This post is my take on the reasons for Brazil's fondness of Linux. I speak for Brazil since I was born and raised there, but I think much of this applies to the other BRIC countries and emerging markets in general.

The first and obvious argument is economic: free as in beer is a big deal in Brazil's economy. The table below contrasts the economics of license costs in the US and in Brazil:

	US	Brazil
Gross National Income (GNI) per capita	$44,710	$4,710
Cost of Windows Vista Business	$186	$364
Cost of MS Office 2007 Standard	$289	$587
Cost of Business Licenses as % of GNI per capita	1.06%	20.1%
Cost of Windows Vista Home Basic	$116	$252
Cost of Office Home/Student	$109	$117
Cost of Home Licenses as % of GNI per capita	0.5%	7.8%

All figures in US dollars. An exchange rate of USD$1 = R$1.70 was used to compute the cost of licenses in Brazil.

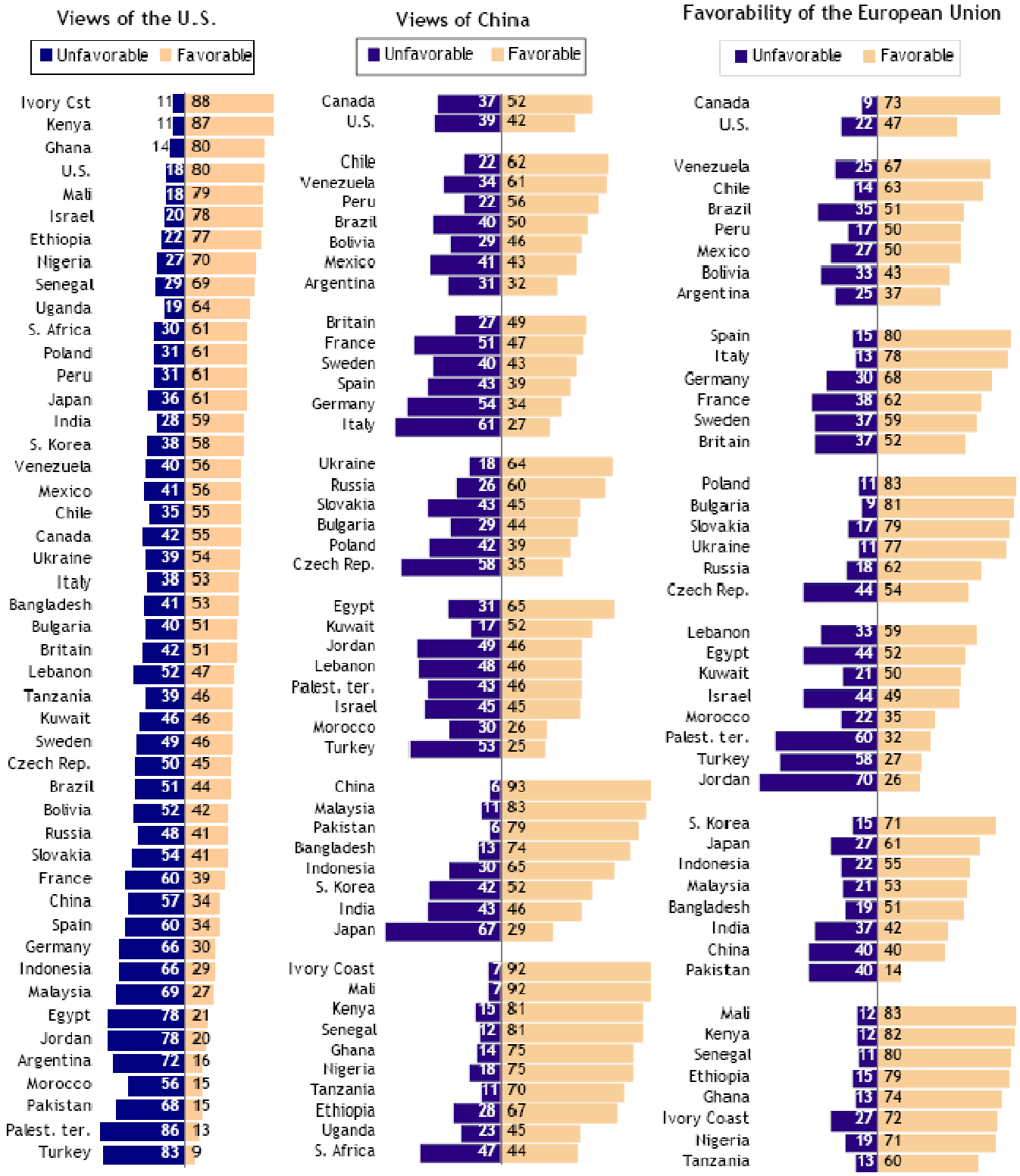
You might be surprised to learn that Microsoft licenses are **nearly twice as expensive in Brazil** in absolute terms. I imagine Microsoft charges about the same and Brazil's brutal tax burden makes up the rest (the taxes are built into the price). But the interesting result is the relative price of licenses in each society, captured as % of GNI per capita. As a proportion of national incomes, business licenses are **nineteen times more expensive to Brazilian society** and home licenses are **fifteen times more expensive**. While GNI per capita is not a perfect figure, it reflects the incomes people make, how much they spend to live, and how much they pay in taxes. It *is* a crucial number when it comes to public policy; it's not hard to understand why rational policies must dodge licensing costs when possible. If there's any hope of widespread computer access, then surely we can't expect people to spend 7.8% of their annual income on Microsoft software licenses alone. The burden on small businesses is also prohibitive. This order-of-magnitude difference is a fundamental problem that can't be solved by piecemeal license giveaways. Suppose Microsoft gave out Windows and Office wholesale to *all* schools. Then what happens if those kids need a computer at home or

in their parents' business? License costs are simply out of whack with respect to most of society. Using Linux in public schools, rarely attended by richer kids, seems inescapable.

Notice that I didn't use Windows Vista Starter Edition in the figures. This is because I find the limitation of *three simultaneous programs* absurd. It's hard to believe Microsoft put in such an abominable restriction; it's one thing to quietly omit features, it's quite another to slap people on the face with "Sorry, no, only 3 programs! Click OK to continue." Even the limited hardware supported by Vista Starter can easily run multiple programs, so that's no excuse. I imagine a kid trying to learn programming in such a machine, trying to run a few tools plus a test application, and being told to bugger off. How is this bridging the digital divide? Besides, there are limitations on buying Vista Starter – a family receiving a donated computer, for example, cannot buy a retail version of it. And to cap it off, if they went ahead and bought OEM, the dollar figure for Vista Starter + Office Home comes to 5% of GNI per capita, still an order of magnitude above the US figure.

Looking at these numbers, you might wonder how Microsoft sales could grow 40% in Brazil last year – I mean, do they even have *computers* there? It turns out that Brazil has both the 10th largest economy in the world and the 11th worst distribution of income. There are wealthy households, businesses, and government departments to whom license prices matter far less. For example, after the dollar plunge the *cost* in dollars of a programmer in Brazil is close to that of one in the US, provided the employer is paying all taxes (the norm for mid-size and large businesses). These wealthier pockets comprise a sizable market whose landscape is more similar to the US: labor costs dwarf license costs, MS Office is a near-monopoly, and inertia is in Microsoft's favor. Since this market is in Brazil's economy the licensing costs still consume relatively more purchasing power, but Microsoft can definitely compete in these niches. Except there's more to the story than economics.

Many cultural issues work against Microsoft to mobilize Brazilians in favor of the Penguin. I'll hit up the three I deem biggest: 1) utter disregard for copyright, 2) strong anti-Microsoft feelings, and 3) Linux alpha geek monopoly. A thorny reality aggravates the first two issues: anti-American sentiment. It's worth looking at this sentiment for context. The Pew Research Center runs the Pew Global Attitudes Project to track global public opinion on a variety of issues. Their latest report shows continued decline in US image, which has plummeted around the globe in the past 5 years. Here's the data:

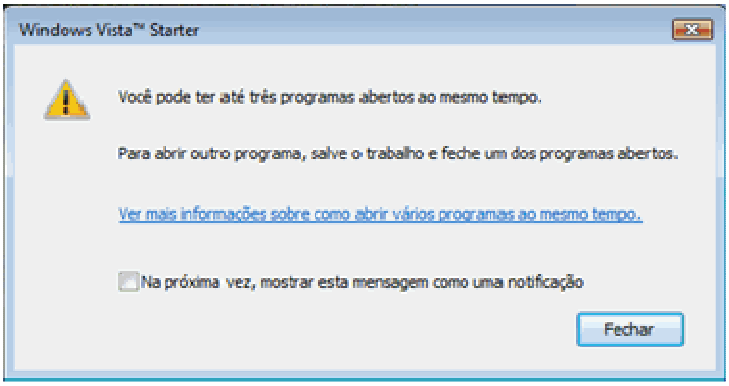


I was shocked to see these results at first. Things might not be as bleak as the numbers suggest though. Some of the backlash is not structural, but rather directed at the current US Administration. It may well subdue come January 2009. Yet it's important for American companies to factor this in when thinking about markets abroad. Nothing new there, except for how bad things got. Anti-American sentiment is particularly strong in 3 of the BRIC: Brazil, Russia, and China. (In Brazil this is only a political/ideological thing, one-on-one people are still as friendly as ever towards Americans and everyone else.) Keep these numbers in mind when thinking about the factors below, starting with disregard for copyright.

When I was growing up in Brazil, paying for software licenses was about as natural as a third arm growing out of your back. Whenever you needed software, you'd dial up a friendly pirate and buy a "collection" for, uh, $30. That included, my friends, *instant home delivery*: the guy would drive to your house and deliver the collection. If you were programming at night, he might even bring you a pizza. The best pirates had good access to the warez scene and could find anything in case you had a custom request. In the collection you'd find cracked versions of *several major pieces of software from various manufacturers*. How convenient. Borland Delphi? Check. Visual Studio? Check. Windows NT 4.0 Server, Workstation? Check. Linux? Check (saves you the download). It was like MSDN for the whole computer industry! The piracy happened *regardless* of income levels – people "buying" the software were by no means poor (otherwise they wouldn't have a computer in the first place, at that point in time). *Many* could easily afford licenses, yet felt absolutely no qualms about piracy. The *whole culture* disregarded copyrights deeply. To most people, the pirate was doing honest work: downloading all this stuff, burning it, delivering it. An honorable job.

Things have changed since then, but not much. Copyright enforcement is more serious; piracy within mid-size or large businesses is rare. There is more copyright awareness (or indoctrination, depending on whom you ask). Home users still pirate anything they can though. This is not restricted to software either: visit any campus in Brazil and you'll see rampant photocopying of text books. Street vendors sell DVDs filled with MP3s, movies, you name it. I'm sure Hollywood execs have nightmares where they're roaming the streets of Brazil. The culture still expects free distribution and the environment is very hostile to proprietary software licenses.

Before Windows Genuine Advantage, Microsoft's strategy was to ignore the pirates: sell to the corporations, let everyone else copy it. Now regular people are growing third arms and paying for Windows licenses. Fair enough: middle-class fat cats should not be ripping off your software. The trouble is that nowadays not all cats are fat: years of sound macroeconomic policy have allowed lower-middle-class people to buy computers. This is a change from when I was there. There are now many folks who, though not poor, definitely have a very hard time paying for licenses. They either go to Linux or go unpatched. The need to buy software would effectively keep these families out of computing: they do back flips just to get the hardware itself, in the hopes of giving their children a better shot. And richer people resent paying for software, however messed up that is. Every customer cut off by Windows Genuine Advantage is a possible conversion to Linux or at the very least a little more pressure towards migration. If people had to pay for Office too, there'd be gnashing of teeth. I'm not suggesting Microsoft is responsible for fixing severe income inequality or supporting middle-class free loaders; that's just the nut they need to crack with a creative revenue model because Linux fits like a glove. On to the next issue.



Vista Starter caps you at 3 programs. This is not how you win friends and influence people.

Brazil imported the anti-Microsoft stance common in American geeks, but on top of the usual arguments Microsoft is *foreign*. This adds fuel to the flame. To the Brazilian Microsoft hater, not only there is an "evil monopoly", but its profits are repatriated and its jobs are elsewhere. Practices like the 3-program limitation on Vista Starter further erode good will (Brazilians call it the "castrated Windows" among other colorful names). Add a dash of anti-American sentiment and you've got some serious resistance. This fiery mood has a strong influence, from the teenager hanging out in #hackers on Brasnet to IT departments to the

federal government. Even in a rational self-interest analysis, one might rightly point out that if free/open source software (FOSS) were to wipe out Windows, negative effects on Brazil's economy are likely minimal. The wealth, jobs, and opportunity created by Microsoft aren't in Brazil (productivity gains might be, but that's a whole different argument). The trade offs of a potential Linux/Google take over are different when there's no national off-the-shelf software industry, plus Google's revenue model works beautifully in a developing country. This mix of ideological and rational arguments torpedoes Microsoft's support.

The third cultural issue working for Linux is more subtle. In the US people talk about Microsoft losing the alpha geeks, but in Brazil FOSS has already reached a near-monopoly on them. Again, the standard reasons apply but are augmented by the local realities. Before FOSS, interesting software work was *very rare* in Brazil and the chance to shape widely used products practically *did not exist*. Imagine a place where 80% of programmers build boring, low-powered line-of-business applications working in conditions exactly opposite of Peopleware. That's the US. Make it 99% and you have Brazil. In the US we have a wildly dynamic economy full of start-ups and interesting companies soaking up talent fast, but not so in Brazil. David Solomon, co-author of Windows Internals, was working for DEC at 16. But what if there is no one building a kernel in a 3,000-mile radius? Emigration was the most realistic possibility for interesting work. A 16-year-old would have been out of luck.

The FOSS revolution plus the Internet changed all this. Now people in Brazil can actually develop interesting and widely used programs. We've got kernel hackers like Marcelo Tosatti, who maintained the 2.4 Linux kernel series, and Arnaldo Carvalho de Melo, who co-founded the Conectiva distribution. There are RedHat employees, Debian contributors, committers on various projects, and so on. Lua, the programming language, comes from Brazil. There's a practical advantage in being able to, say, tune a distribution for a particular purpose (*e.g.*, the distribution being delivered to public schools). But beyond that it's inspiring to finally be able to work with talented people in cool projects and have a chance to *participate*, rather than be handed down a proprietary product built abroad over which you have zero control. People are excited about and grateful for this. By the time you mix up these elements nearly all talented CS students and alpha geeks are well into the Linux camp. Unlike the US, the dynamic economy isn't there to add some fragmentation. When these people go on to make technology choices in government or industry, guess what they'll pick?

So that's it. I think these are the main factors in Brazil's love affair with Linux: economics, disregard for copyright, anti-Microsoft sentiment, and massive alpha geek support. These factors feed off each other, all pushing towards Linux. Millions of kids using KDE would impact the work force eventually. If Microsoft is overzealous in their anti-piracy efforts, it might precipitate faster changes in this delicate market. Meanwhile, Google Docs and Open Office are catching on. There are tactical moves Microsoft could make to counter Linux momentum, like a more sustainable licensing model for homes and small businesses (maybe their announced annuity licensing?), better native branding, and perhaps some native development. But Google has done all three already and is very well-liked in Brazil despite the anti-US feelings. My Brazilian friends, even a pragmatic IT manager who plays poker with Microsoft Brazil employees, seem to operate on the assumption that an eventual Linux take over (with some combination of Google/Google Docs/Open Office) is just a matter of time. What holds it back is that all the factors discussed here can spark things up, but until desktop Linux is ready to catch on fire you get much hype and little change. The wood does seem drier and drier, so we'll see. What do you think?

May 1, 2008 | Filed Under Linux

68 Comments

Build a quad-core, 8-gig server for $900

Most enthusiastic programmers start out as over-engineers. It's a side effect of caring deeply about our craft. With experience we learn that simplicity is king and less is more, but it takes effort. That's why I love a

quote by Ferdinand Porsche, who designed the original VW Beetle and started the Porsche company, that says:

The perfect race car crosses the finish line in first place and then falls to pieces.

After I first heard it from a ~~car fanatic~~ good friend, the quote went straight to my wall. If I start gold plating code or requirements, there's Dr. Porsche to whip my ass back into pragmatism. But no other time is more trying than when I'm building a computer. *I just need* a RAID 10 array, a quad-core Core 2 Extreme, and 16 gigs of 2-cas-latency RAM! Of course, in a disciplined frame of mind such largesse is simply gross over-engineering. And it's more fun and challenging to seek an optimal performance/money ratio than to buy your way out of thinking.

Or maybe that's just what I tell myself when I only have $1,000 bucks to spend. Either way, multi-core CPUs made powerful computers far more affordable. You can build a fine quad-core, 8-gig server within that budget. In my case I wanted a VMware rig to power this website. I talk to other people who are interested in setting up a VMware lab to learn different technologies, yet feel discouraged due to server prices. But building your own server can save 50% off the price for a similar Dell product. When I priced a PowerEdge tower comparable to my custom build, it came out at $1,900 for the Dell versus $930 for custom. That's actually a good deal for a business. But for home users, building our own is often the best (or only) option. So here is my parts list, along with some tips:

Build parts

[**Update**: The server has now been running for about 3 months with continuous uptime (a single reboot). No problems that I know of. However this parts list is about 3 months old and could be improved upon. Take a look at the comments section as people made good suggestions. Here are some ideas: 1) AMD CPUs might be a good way to go, freeing up some money to spend on perhaps more disks, 2) a cheaper motherboard (hopefully with integrated video) would be fine, provided it's a good brand, 3) stick with 16MB-cache hard disks. Just make sure to check the NewEgg reviews: see what people are saying, the overall rating, and the percentage of one-egg reviews (ie, people who hated it). Then read some of the one-egg reviews to find out what the problems are. If they're complaints about Dead On Arrival (DOA) products, then you should be ok since *every* product has those; but if something more sinister is going on, stay out. If you're going to run

Linux, Google quickly for the motherboard name + "Linux" to see what people are saying. Don't trust editor reviews: always check out what the people say. Vox populi, vox dei.]

Component	Part	Price
Motherboard	MSI P6N SLI Platinum LGA 775 NVIDIA nForce 650i SLI ATX Intel Motherboard – Retail This is a top-quality motherboard, but **make sure you upgrade the BIOS** to the newest version. They ship with a buggy BIOS which *crashes the machine* after the BIOS post when you install four 2-gig RAM modules. This happens for many RAM brands and took 2 hours to track down, the biggest time sink in this build. After upgrading the BIOS all is well. I left the machine running memtest86 for a day and a half with no problems. The motherboard works well in Linux, but there are some quirks with stock distro kernels due to the newish chipset. As usual Ubuntu had the best support, but HPET is not working. I disabled it in the BIOS for now to prevent "lost some interrupts" log messages. lm-sensors only retrieves CPU core temps and nothing else (in CentOS, lm-sensors didn't work at all). I'll look into both of these issues later, but they're minor annoyances: overall everything works well; HPET is useless anyway.	**$139.99**
CPU	Intel Core 2 Quad Q6600 Kentsfield 2.4GHz LGA 775 Quad-Core Processor Model BX80562Q6600 I don't normally overclock. Most workloads aren't CPU-constrained and overclocking often adds cost and headache to little practical advantage. By not overclocking we can use the stock heatsink that comes with the processor, saving some money. My idle core temperatures are at ~30oC; they peaked at ~60oC after a day of Prime95 (one instance per core) with no errors. These are fine temperatures, no need for a 3rd-party heatsink.	**$254.99**
Case	Antec Performance One P180 Silver cold rolled steel ATX Mid Tower Computer Case – Retail This is a glorious case, but I only bought it because it was on sale for $89.99. It's pleasurable to work with a good case, but really, you only work with it for one hour while you build the computer. Again, if you're not overclocking and hence not worried about whether a degree Celsius will corrupt your file system, you can safely buy in the $50-$80 range. Stick to cases *without* power supplies though, since bundled power supply units are usually crap.	**$129.99**
Power Supply	Antec earthwatts EA380 ATX12V v2.0 380W Power Supply – Retail	**$59.99**

Component	Part	Price
	People go overboard on power supply wattages. There's no reason to. All you get for idle wattage is a higher electrical bill and a warmer planet. This is a good-quality, energy-efficient (80+ certified) power supply with plenty of juice for our server. Perfect. Adjust accordingly if you have power-hungry video cards or peripherals to be fed. **Always buy a good power supply unit.** Power supply problems are disastrous: 1) they can fail and bring your server down, 2) they destabilize and crash the computer, and 3) they fry all your other components. Be economical on the wattage but not on quality: stick to good brands and check the Newegg reviews.	
RAM	G.SKILL 4GB(2 x 2GB) 240-Pin DDR2 SDRAM DDR2 800 (PC2 6400) Dual Channel Kit Desktop Memory Model F2-6400CL5D-4GBPQ – Retail Great reviews on newegg (5 eggs, 2% 1-egg rate), passed multiple passes of memtest86 for me. It's a winner.	$79.99 * 2 = **$159.98**
Video Card	EVGA 256-P2-N297-LX GeForce 6200LE TC 512MB (256MB on board) 64-bit GDDR2 PCI Express x16 Video Card – Retail I look for two things in video cards: fanless cooling and low power consumption. Fans move, therefore they make noise and fail. You can avoid both with a heatsink-only card. They also tend to draw less power. This one runs cool, supports 2 monitors, comes with an S-cable and DVI adapter, and costs 7 lattes.	**$27.99**
Hard drives	Western Digital Caviar SE WD3200AAJS 320GB 7200 RPM 8MB Cache SATA 3.0Gb/s Hard Drive – OEM Disk I/O is the most common bottleneck I find when troubleshooting server performance. It's also the subsystem most affected by virtualization. In fact, CPU and RAM work basically at the same speed in the host and guest. But disk suffers for many reasons. So we go with 2 hard drives. The perfect car breaks apart after the race, but it finishes in first place. This drive is a solid 5-egger (hahah) with only 2% 1-egg ratings. I trust WD as much as any other. Unfortunately, this was **not** the drive I bought, but it's what I would buy now. I bought the SAMSUNG SpinPoint T Series HD501LJ 500GB 7200 RPM 16MB Cache SATA 3.0Gb/s Hard Drive – OEM for $105. It's a 5-egg drive too but it has 8% 1-egg ratings, several of which are complaining about failure after a few months. That's much worse than a drive being dead on	2 * 72.99 = **$145.98**

Component	Part	Price
	arrival. Now I live in phear of my hard disks going bust on me. Yay for Linux software raid.	
TOTAL		**$918.91**

This hard drive drawer is a nice touch in the Antec P180. The white rubber things help damp hard drive noise.

I never buy CD/DVD drives or floppy drives. I have a USB Lite-On DVD reader/burner and a USB Sony floppy disk. I've used them for years on countless computers, installed all the OSes, flashed the BIOSes, never had a problem. It's a great solution. You can save money on every computer, plus no worries about powering a DVD burner off your power supply.

For a home server, especially one running VMware, I recommend a simple Uninterruptible Power Supply to prevent file system corruption. VMware is particularly sensitive (I have a detailed entry coming up on this). Since this type of corruption can cost hours and I only have a few hobby hours a week, I'm more than willing to pay for an UPS. I really like this APC unit. If you buy one, make sure you set everything up so that the computer **really shuts down if power is lost**, otherwise you're just running a $100 LED display.

Here's what the full build looks like

The computer runs silently. In fact it's two feet away from me right now, case open, and I can hardly hear a thing. Assembly time for this box was about one hour. I wasted another two hours figuring out why it crashed with 8 gigs of RAM. Fixing the problem was a matter of minutes: I just flashed the motherboard BIOS and it was cured. And that was the easy part of getting the server up and running. I've since been doing some experiments on file system and virtual machine performance, which takes a lot more time than plugging connectors together. I should have some results over the weekend.

Update on 2008-12-23: Doug Holton has posted a new parts list for a quad core server, now down to $500 with 1 TB of storage.