Virtual Memory for Linux

We'll now look at some of the more intriguing features of Linux VM. Real engineers solving real problems in production drove Linux development, resulting in a fully functional, feature-rich virtual memory system.

We won't be able to cover every feature of Linux VM, but we'll focus on the most significant ones, especially where it goes beyond VAX/VMS. As well as similarities between Linux and earlier systems.

This section focuses on Linux for Intel x86. While Linux may run on a variety of processor architectures, x86 is the most common and important deployment, and hence our focus.

The Linux Address Space

A Linux virtual address space is composed of a user piece (containing user program code, stack, heap, and other components) and a kernel portion (where kernel code, stacks, heap, and other parts reside). The user component of the presently operating address space changes when a context switch occurs, while the kernel portion remains constant. A program in user mode cannot access kernel virtual pages; it must trap into the kernel and switch to privileged mode.

In traditional 32-bit Linux (with a 32-bit virtual address space), the user/kernel split occurs at 0xC0000000, or three-quarters of the way across the address space. These are the user virtual addresses, while the kernel virtual addresses (0xC0000000 through 0xFFFFFFFF) are in the user virtual address space. 64-bit Linux also splits, although at various spots. Figure 23.2 depicts a reduced address space.

Linux has two types of kernel virtual addresses. The first are kernel logical addresses. This is the kernel's usual virtual address space; to get more of it, kernel code simply calls kmalloc. Page tables, per-process kernel stacks, and other data structures sit here. Kernel logical memory cannot be switched to disk like other system memory.

The link between kernel logical addresses and physical memory is fascinating. There is a direct mapping between kernel logical addresses and the first physical memory segment. Thus, 0xC0000000 corresponds to 0x00000000, 0xC0000FFF to 0x00000FFF, and so on. This has two implications. The first is because translating between kernel logical and physical addresses is simple, therefore these addresses are generally considered as physical. And a chunk of memory that is contiguous in logical address space is contiguous in physical memory. Because the kernel's address space is contiguous, this memory can be used for activities that require contiguous physical memory, like I/O transfers to and from devices via directory memory access (DMA) (covered in the third section of this book).

A kernel virtual address is another type. To obtain this memory, the kernel uses vmalloc, which delivers a pointer to a nearly continuous region of the necessary size. Unlike kernel logical memory, kernel virtual memory is frequently not contiguous (and is thus not suitable for DMA). As a result, it is utilized for huge buffers where finding a continuous large portion of physical memory is difficult.

In 32-bit Linux, virtual addresses allow the kernel to address memory larger than 1 GB. With less memory than this, allowing access to more than 1 GB was not an issue. But as technology advanced, it became necessary to allow the kernel to consume more memory. Kernel virtual addresses and their loose link to physical memory allow this. With 64-bit Linux, the requirement for this is less essential, as the kernel is not limited to the last 1 GB of virtual address space.

Page Table Structure

The type of page-table structure given by x86 impacts what Linux can and cannot accomplish. The OS merely sets up mappings in memory, points to a privileged register at the start of the page directory, and the hardware handles the rest. The OS is engaged in process creation, deletion, and context switching, ensuring that the hardware MMU is using the correct page table for translations.

The major shift in recent years is the switch from 32-bit to 64-bit x86. A 32-bit address space has been present for a long time, and as technology improved, it became a genuine constraint for applications. With current computers containing many GB of memory, 32 bits was no longer adequate to refer to each of them. So the next jump was required.

Moving to a 64-bit address changes the page table structure in x86. The modern 64-bit systems employ a four-level page table since x86 has multiple levels. However, only the bottom 48 bits of the virtual address space are being used. A virtual address looks like this:

The top 16 bits of a virtual address are unused (and therefore not translated), the bottom 12 bits (because to the 4-KB page size) are utilized as the offset (and thus not translated), leaving the middle 36 bits to be translated. The P1 portion of the address indexes into the topmost page directory, and the translation proceeds one level at a time until P4 indexes the required page table entry.

Enabling more of this vast address space as system memory grows leads to five-level and eventually six-level page-table tree topologies. Imagine a basic page table lookup requiring six levels of translation to locate a piece of data in memory.

Large Page Support

Intel x86 supports several page sizes than the normal 4-KB page. Recent designs support hardware 2-MB and even 1-GB pages. Linux has evolved to allow apps to use these massive pages (as they are called in the world of Linux).
Using large pages has many advantages. As observed in VAX/VMS, larger pages require fewer mappings in the page table. Huge pages aren't caused by fewer page-table entries, but by better TLB behavior and concomitant performance advantages.

The TLB quickly fills up when a process actively uses a lot of memory. Without causing TLB misses, only a small amount of total memory can be accessed. The result is a considerable performance hit for modern "big memory" workloads running on machines with many GBs of memory. Huge pages allow a process to access enormous amounts of memory while requiring less TLB slots, which is the main benefit. Huge pages also have a shorter TLB-miss path, meaning that when a TLB miss occurs, it is serviced faster. Allocation can also be quite quick (in some cases), a minor but essential benefit.

The progressive nature of Linux support for big pages is intriguing. They initially thought it was only relevant to a few applications, including huge databases with high performance requirements. So it was decided to let apps directly request big page memory allocations (through mmap() or shmget()). All but a few mandatory apps would need to be altered, but the discomfort would be worth it for those that require it.

Recently, as more programs require better TLB behavior, Linux engineers implemented transparent big page support. When enabled, the operating system automatically searches for large page allocations (typically 2 MB, but on some systems 1 GB) without modifying applications.

Huge pages come at a price. The biggest possible consequence is internal fragmentation, i.e. a vast but underused page. These huge yet rarely utilized pages can clog up memory. Swapping, if enabled, also has issues with large pages, increasing the amount of I/O a system does. Allocation overhead can be undesirable (in some other cases). One thing is clear: the 4-KB page size that served systems well for many years is no longer ubiquitous; rising memory sizes necessitate greater pages and other solutions. The sluggish adoption of hardware-based technology by Linux is a sign of transition.

The Page Cache

Agressive caching schemes maintain popular data items in memory to reduce access costs to persistent storage (the third part of this book). In this way, Linux is similar to other OSs.

The Linux page cache keeps pages in memory from three sources: memory-mapped files, de- vices' file contents and metadata (accessible via read() and write() calls to the file system), and process heap and stack pages (sometimes called anonymous memory, because there is no named file underneath of it, but rather swap space). These entities are stored in a page cache hash table for rapid access.

Entries are clean (read but not updated) or dirty (written) (a.k.a., modified). This is done via background threads (called pdflush) periodically writing dirty data to the backing store (i.e., to a specific file for file data, or to swap space for anonymous areas). This background action occurs after a period of time or if too many pages are unclean (both configurable parameters).

When a system runs out of memory, Linux must pick which pages to delete. To accomplish so, Linux uses a modified form of 2Q replacement.
The concept is simple: regular LRU replacement works but is vulnerable to typical access patterns. For example, if a process regularly reads a large file (almost as large as memory), LRU will remove all other files from memory. Worse, chunks of this file are never re-referenced before being deleted from memory.

It works on Linux by keeping two lists and dividing memory between them. An initial access places a page on one queue (named A1 in the original article, but inactive in Linux); further references promote it to the other queue (called Aq in the original, but the active list in Linux). When a replacement is needed, the inactive list is used. This keeps the active list to around two-thirds of the entire page cache size.
Linux would ideally manage these lists in perfect LRU order, but this is costly. The OS uses an approximation of LRU (like clock replacement).
This 2Q technique is similar to LRU, but it handles cyclic large-file access by restricting the pages of the cyclic access to the inactive list. No additional useful pages in the active list are flushed because these pages are never re-referenced.

Security And Buffer Overflows

The modern emphasis on security is perhaps the biggest distinction between modern VM systems (Linux, Solaris, or one of the BSD variations) and old (VAX/VMS). In a world where machines are more interconnected than ever, it's no wonder that developers have developed a number of protective mechanisms to stop those wily hackers from obtaining control of systems and data.

Buffer overflow attacks can be used against standard user programs and even the kernel. These attacks aim to uncover a flaw in the target system that allows them to inject arbitrary data into its address space. When a developer estimates (erroneously) that an input would not be too long, he copies it into a buffer, which overflows, overwriting the target's memory. Code as simple as the following can cause issues:

In many circumstances, an unintentionally given faulty input to a user program or even the OS will cause it to crash, but not worse. An attacker can use the input that overflows the buffer to inject their own code into the targeted system, thus taking control and doing their bidding. Assailants can conduct arbitrary computations or even rent out cycles on the compromised system if they successfully assault a network-connected user software (i.e., user code gaining kernel access rights). Obviously, these are all Bad Things.

The first line of defense against buffer overflow is to deny access to particular regions of address space (e.g., within the stack). In AMD's x86 (Intel now includes a comparable XD bit), the NX bit inhibits execution of any page that has this bit set in its corresponding page table entry. The solution prevents an attacker from injecting code onto the target's stack, hence mitigating the problem.

However, intelligent attackers are clever, and even when injected code cannot be added intentionally, malicious code can run arbitrary code sequences. It's called return-oriented programming (ROP) and it's great. ROP observes that any program's address space contains many pieces of code (gadgets), especially C programs that link to the large C library. An attacker can so modify the stack so that the current function's return address points to a malicious instruction (or series of instructions), followed by a return instruction. An attacker can execute arbitrary code by stringing together many gadgets (ensuring each return goes to the next gadget). Amazing!

To counter ROP (and its older counterpart, the return-to-libc attack), Linux (and other systems) use address space layout randomization (ASLR) (ASLR). The OS randomly places code, stack, and heap within the virtual address space, making it difficult to create the sophisticated code sequences required to implement this type of attacks. To acquire control of the running application, most attacks against user programs induce crashes. In practice, you can see this randomness quite simply. Here's some code to show it on a current Linux system:

This code just prints the virtual address of a stack variable. Previously, this number was constant in non-ASLR systems. Like the value varies with each run:

ASLR is so helpful for user-level programs that it's been built into the kernel as kernel address space layout randomization (KASLR). But, as we'll see, the kernel may have bigger issues to deal with.

Other Security Problems: Meltdown And Spectre

Two new cyberattacks have turned the world of systems security on its head as we type (August 2018). Meltdown and Spectre are two examples. They were discovered at the same time by four independent research/engineering teams, raising serious concerns about the security of computer hardware and operating systems. See papers at meltdownattack.com and spectreattack.com for more information. Spectre is deemed the more harmful.

The common flaw exploited in these attacks is that current CPUs execute insane gimmicks to enhance speed. Speculative execution is a technique where the CPU predicts which instructions will be performed in the future and starts executing them ahead of time. It runs faster if the predictions are true; if not, the CPU undoes the effects on the architectural state (e.g., registers) and tries again, hoping for success.

The issue with speculation is that it tends to leave traces in the system, such as CPU caches and branch predictors. So the issue is that, as the attacks reveal, such a situation might render memory susceptible, even memory secured by the MMU.

To improve kernel security, isolate as much kernel address space as possible from user processes and utilize a separate kernel page table for most kernel data (called kernel page- table isolation, or KPTI). This means that instead of transferring the kernel's code and data structures into each process, only the bare minimum is preserved. This enhances security and reduces attack paths, but at a performance cost. Page table switching is costly. That is, convenience and performance.

Sadly, KPTI only addresses some of the security issues listed above. Simple remedies like stopping speculating would be useless since systems would be thousands of times slower. That is, if you are interested in systems security.

To really comprehend these attacks, you'll need to learn a lot more. Begin by learning current computer architecture, focusing on speculation and all the tools required to implement it. The aforementioned sources have information on the Meltdown and Spectre attacks, as well as a primer on conjecture. Examine the operating system for flaws. What difficulties remain?

Summary

These two virtual memory systems have now been thoroughly reviewed. Thanks to your prior knowledge of the underlying procedures and policies, most of the details should have been straightforward to grasp. Levy and Lipman's excellent (and brief) study on VAX/VMS has further detail on the subject. Reading it will give you a better understanding of the underlying material used to create these chapters.

You also learned a little Linux. In spite of its vastness and complexity, it inherited many wonderful concepts from the past, many of which we could not elaborate on. By eliminating needless copying, Linux, for example, uses lazy copy-on-write copying of pages when calling fork(). This is because Linux uses a background swap daemon (swapd) to swap pages to disk. The VM has many solid concepts from the past, as well as many original ideas.