23

Complete Virtual Memory Systems

Before we end our study of virtualizing memory, let us take a closer look at how entire virtual memory systems are put together. We've seen key elements of such systems, including numerous page-table designs, interactions with the TLB (sometimes, even handled by the OS itself), and strategies for deciding which pages to keep in memory and which to kick out. However, there are many other features that comprise a complete virtual memory system, including numerous features for performance, functionality, and security. And thus, our crux:

THE CRUX: HOW TO BUILD A COMPLETE VM SYSTEM
What features are needed to realize a complete virtual memory system? How do they improve performance, increase security, or otherwise improve the system?

We'll do this by covering two systems. The first is one of the earliest examples of a "modern" virtual memory manager, that found in the **VAX/VMS** operating system [LL82], as developed in the 1970's and early 1980's; a surprising number of techniques and approaches from this system survive to this day, and thus it it is well worth studying. Some ideas, even those that are 50 years old, are still worth knowing, a thought that is well known to those in most other fields (e.g., Physics), but has to be stated in technology-driven disciplines (e.g., Computer Science).

The second is that of **Linux**, for reasons that should be obvious. Linux is a widely used system, and runs effectively on systems as small and underpowered as phones to the most scalable multicore systems found in modern datacenters. Thus, its VM system must be flexible enough to run successfully in all of those scenarios. We will discuss each system to illustrate how concepts brought forth in earlier chapters come together in a complete memory manager.

23.1 VAX/VMS Virtual Memory

The VAX-11 minicomputer architecture was introduced in the late 1970's by **Digital Equipment Corporation** (**DEC**). DEC was a massive player in the computer industry during the era of the mini-computer; unfortunately, a series of bad decisions and the advent of the PC slowly (but surely) led to their demise [C03]. The architecture was realized in a number of implementations, including the VAX-11/780 and the less powerful VAX-11/750.

The OS for the system was known as VAX/VMS (or just plain VMS), one of whose primary architects was Dave Cutler, who later led the effort to develop Microsoft's Windows NT [C93]. VMS had the general problem that it would be run on a broad range of machines, including very inexpensive VAXen (yes, that is the proper plural) to extremely high-end and powerful machines in the same architecture family. Thus, the OS had to have mechanisms and policies that worked (and worked well) across this huge range of systems.

As an additional issue, VMS is an excellent example of software innovations used to hide some of the inherent flaws of the architecture. Although the OS often relies on the hardware to build efficient abstractions and illusions, sometimes the hardware designers don't quite get everything right; in the VAX hardware, we'll see a few examples of this, and what the VMS operating system does to build an effective, working system despite these hardware flaws.

Memory Management Hardware

The VAX-11 provided a 32-bit virtual address space per process, divided into 512-byte pages. Thus, a virtual address consisted of a 23-bit VPN and a 9-bit offset. Further, the upper two bits of the VPN were used to differentiate which segment the page resided within; thus, the system was a hybrid of paging and segmentation, as we saw previously.

The lower-half of the address space was known as "process space" and is unique to each process. In the first half of process space (known as `P0`), the user program is found, as well as a heap which grows downward. In the second half of process space (`P1`), we find the stack, which grows upwards. The upper-half of the address space is known as system space (`S`), although only half of it is used. Protected OS code and data reside here, and the OS is in this way shared across processes.

One major concern of the VMS designers was the incredibly small size of pages in the VAX hardware (512 bytes). This size, chosen for historical reasons, has the fundamental problem of making simple linear page tables excessively large. Thus, one of the first goals of the VMS designers was to ensure that VMS would not overwhelm memory with page tables.

The system reduced the pressure page tables place on memory in two ways. First, by segmenting the user address space into two, the VAX-11 provides a page table for each of these regions (`P0` and `P1`) per process;

ASIDE: THE CURSE OF GENERALITY
Operating systems often have a problem known as **the curse of generality**, where they are tasked with general support for a broad class of applications and systems. The fundamental result of the curse is that the OS is not likely to support any one installation very well. In the case of VMS, the curse was very real, as the VAX-11 architecture was realized in a number of different implementations. It is no less real today, where Linux is expected to run well on your phone, a TV set-top box, a laptop computer, desktop computer, and a high-end server running thousands of processes in a cloud-based datacenter.

thus, no page-table space is needed for the unused portion of the address space between the stack and the heap. The base and bounds registers are used as you would expect; a base register holds the address of the page table for that segment, and the bounds holds its size (i.e., number of page-table entries).

Second, the OS reduces memory pressure even further by placing user page tables (for `P0` and `P1`, thus two per process) in kernel virtual memory. Thus, when allocating or growing a page table, the kernel allocates space out of its own virtual memory, in segment `S`. If memory comes under severe pressure, the kernel can swap pages of these page tables out to disk, thus making physical memory available for other uses.

Putting page tables in kernel virtual memory means that address translation is even further complicated. For example, to translate a virtual address in `P0` or `P1`, the hardware has to first try to look up the page-table entry for that page in its page table (the `P0` or `P1` page table for that process); in doing so, however, the hardware may first have to consult the system page table (which lives in physical memory); with that translation complete, the hardware can learn the address of the page of the page table, and then finally learn the address of the desired memory access. All of this, fortunately, is made faster by the VAX's hardware-managed TLBs, which usually (hopefully) circumvent this laborious lookup.

A Real Address Space

One neat aspect of studying VMS is that we can see how a real address space is constructed (Figure 23.1. Thus far, we have assumed a simple address space of just user code, user data, and user heap, but as we can see above, a real address space is notably more complex.

For example, the code segment never begins at page 0. This page, instead, is marked inaccessible, in order to provide some support for detecting **null-pointer** accesses. Thus, one concern when designing an address space is support for debugging, which the inaccessible zero page provides here in some form.

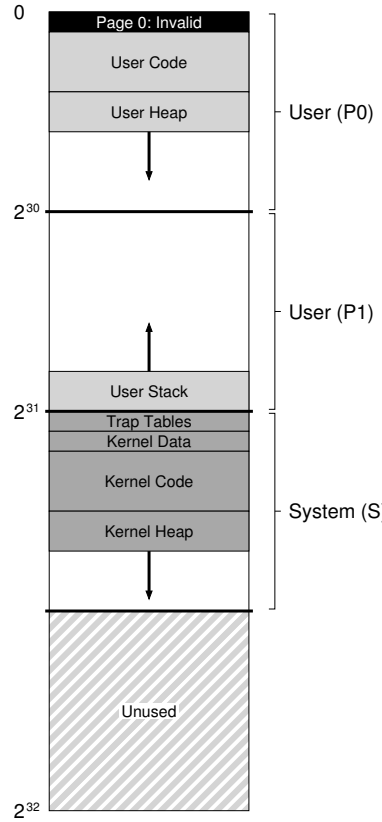


Figure 23.1: **The VAX/VMS Address Space**

Perhaps more importantly, the kernel virtual address space (i.e., its data structures and code) is a part of each user address space. On a context switch, the OS changes the `P0` and `P1` registers to point to the appropriate page tables of the soon-to-be-run process; however, it does not change the `S` base and bound registers, and as a result the "same" kernel structures are mapped into each user address space.

The kernel is mapped into each address space for a number of reasons. This construction makes life easier for the kernel; when, for example, the OS is handed a pointer from a user program (e.g., on a `write()` system call), it is easy to copy data from that pointer to its own structures. The OS is naturally written and compiled, without worry of where the data it is accessing comes from. If in contrast the kernel were located entirely in physical memory, it would be quite hard to do things like swap pages of the page table to disk; if the kernel were given its own address space,

ASIDE: WHY NULL POINTER ACCESSES CAUSE SEG FAULTS
You should now have a good understanding of exactly what happens on a null-pointer dereference. A process generates a virtual address of 0, by doing something like this:

```
int *p = NULL; // set p = 0
*p = 10;       // try to store 10 to virtual addr 0
```

The hardware tries to look up the VPN (also 0 here) in the TLB, and suffers a TLB miss. The page table is consulted, and the entry for VPN 0 is found to be marked invalid. Thus, we have an invalid access, which transfers control to the OS, which likely terminates the process (on UNIX systems, processes are sent a signal which allows them to react to such a fault; if uncaught, however, the process is killed).

moving data between user applications and the kernel would again be complicated and painful. With this construction (now used widely), the kernel appears almost as a library to applications, albeit a protected one.

One last point about this address space relates to protection. Clearly, the OS does not want user applications reading or writing OS data or code. Thus, the hardware must support different protection levels for pages to enable this. The VAX did so by specifying, in protection bits in the page table, what privilege level the CPU must be at in order to access a particular page. Thus, system data and code are set to a higher level of protection than user data and code; an attempted access to such information from user code will generate a trap into the OS, and (you guessed it) the likely termination of the offending process.

Page Replacement

The page table entry (PTE) in VAX contains the following bits: a valid bit, a protection field (4 bits), a modify (or dirty) bit, a field reserved for OS use (5 bits), and finally a physical frame number (PFN) to store the location of the page in physical memory. The astute reader might note: no **reference bit**! Thus, the VMS replacement algorithm must make do without hardware support for determining which pages are active.

The developers were also concerned about **memory hogs**, programs that use a lot of memory and make it hard for other programs to run. Most of the policies we have looked at thus far are susceptible to such hogging; for example, LRU is a *global* policy that doesn't share memory fairly among processes.

To address these two problems, the developers came up with the **segmented FIFO** replacement policy [RL81]. The idea is simple: each process has a maximum number of pages it can keep in memory, known as its **resident set size** (**RSS**). Each of these pages is kept on a FIFO list; when a

ASIDE: EMULATING REFERENCE BITS

As it turns out, you don't need a hardware reference bit in order to get some notion of which pages are in use in a system. In fact, in the early 1980's, Babaoglu and Joy showed that protection bits on the VAX can be used to emulate reference bits [BJ81]. The basic idea: if you want to gain some understanding of which pages are actively being used in a system, mark all of the pages in the page table as inaccessible (but keep around the information as to which pages are really accessible by the process, perhaps in the "reserved OS field" portion of the page table entry). When a process accesses a page, it will generate a trap into the OS; the OS will then check if the page really should be accessible, and if so, revert the page to its normal protections (e.g., read-only, or read-write). At the time of a replacement, the OS can check which pages remain marked inaccessible, and thus get an idea of which pages have not been recently used.

The key to this "emulation" of reference bits is reducing overhead while still obtaining a good idea of page usage. The OS must not be too aggressive in marking pages inaccessible, or overhead would be too high. The OS also must not be too passive in such marking, or all pages will end up referenced; the OS will again have no good idea which page to evict.

process exceeds its RSS, the "first-in" page is evicted. FIFO clearly does not need any support from the hardware, and is thus easy to implement.

Of course, pure FIFO does not perform particularly well, as we saw earlier. To improve FIFO's performance, VMS introduced two **second-chance lists** where pages are placed before getting evicted from memory, specifically a global *clean-page free list* and *dirty-page list*. When a process P exceeds its RSS, a page is removed from its per-process FIFO; if clean (not modified), it is placed on the end of the clean-page list; if dirty (modified), it is placed on the end of the dirty-page list.

If another process Q needs a free page, it takes the first free page off of the global clean list. However, if the original process P faults on that page *before* it is reclaimed, P reclaims it from the free (or dirty) list, thus avoiding a costly disk access. The bigger these global second-chance lists are, the closer the segmented FIFO algorithm performs to LRU [RL81].

Another optimization used in VMS also helps overcome the small page size in VMS. Specifically, with such small pages, disk I/O during swapping could be highly inefficient, as disks do better with large transfers. To make swapping I/O more efficient, VMS adds a number of optimizations, but most important is **clustering**. With clustering, VMS groups large batches of pages together from the global dirty list, and writes them to disk in one fell swoop (thus making them clean). Clustering is used in most modern systems, as the freedom to place pages anywhere within swap space lets the OS group pages, perform fewer and bigger writes, and thus improve performance.

Other Neat Tricks

VMS had two other now-standard tricks: demand zeroing and copy-on-write. We now describe these **lazy** optimizations. One form of laziness in VMS (and most modern systems) is **demand zeroing** of pages. To understand this better, let's consider the example of adding a page to your address space, say in your heap. In a naive implementation, the OS responds to a request to add a page to your heap by finding a page in physical memory, zeroing it (required for security; otherwise you'd be able to see what was on the page from when some other process used it!), and then mapping it into your address space (i.e., setting up the page table to refer to that physical page as desired). But the naive implementation can be costly, particularly if the page does not get used by the process.

With demand zeroing, the OS instead does very little work when the page is added to your address space; it puts an entry in the page table that marks the page inaccessible. If the process then reads or writes the page, a trap into the OS takes place. When handling the trap, the OS notices (usually through some bits marked in the "reserved for OS" portion of the page table entry) that this is actually a demand-zero page; at this point, the OS does the needed work of finding a physical page, zeroing it, and mapping it into the process's address space. If the process never accesses the page, all such work is avoided, and thus the virtue of demand zeroing.

Another cool optimization found in VMS (and again, in virtually every modern OS) is **copy-on-write** (**COW** for short). The idea, which goes at least back to the TENEX operating system [BB+72], is simple: when the OS needs to copy a page from one address space to another, instead of copying it, it can map it into the target address space and mark it read-only in both address spaces. If both address spaces only read the page, no further action is taken, and thus the OS has realized a fast copy without actually moving any data.

If, however, one of the address spaces does indeed try to write to the page, it will trap into the OS. The OS will then notice that the page is a COW page, and thus (lazily) allocate a new page, fill it with the data, and map this new page into the address space of the faulting process. The process then continues and now has its own private copy of the page.

COW is useful for a number of reasons. Certainly any sort of shared library can be mapped copy-on-write into the address spaces of many processes, saving valuable memory space. In UNIX systems, COW is even more critical, due to the semantics of `fork()` and `exec()`. As you might recall, `fork()` creates an exact copy of the address space of the caller; with a large address space, making such a copy is slow and data intensive. Even worse, most of the address space is immediately over-written by a subsequent call to `exec()`, which overlays the calling process's address space with that of the soon-to-be-exec'd program. By instead performing a copy-on-write `fork()`, the OS avoids much of the needless copying and thus retains the correct semantics while improving performance.

TIP: BE LAZY

Being lazy can be a virtue in both life as well as in operating systems. Laziness can put off work until later, which is beneficial within an OS for a number of reasons. First, putting off work might reduce the latency of the current operation, thus improving responsiveness; for example, operating systems often report that writes to a file succeeded immediately, and only write them to disk later in the background. Second, and more importantly, laziness sometimes obviates the need to do the work at all; for example, delaying a write until the file is deleted removes the need to do the write at all. Laziness is also good in life: for example, by putting off your OS project, you may find that the project specification bugs are worked out by your fellow classmates; however, the class project is unlikely to get canceled, so being too lazy may be problematic, leading to a late project, bad grade, and a sad professor. Don't make professors sad!

23.2 The Linux Virtual Memory System

We'll now discuss some of the more interesting aspects of the Linux VM system. Linux development has been driven forward by real engineers solving real problems encountered in production, and thus a large number of features have slowly been incorporated into what is now a fully functional, feature-filled virtual memory system.

While we won't be able to discuss *every* aspect of Linux VM, we'll touch on the most important ones, especially where it has gone beyond what is found in classic VM systems such as VAX/VMS. We'll also try to highlight commonalities between Linux and older systems.

For this discussion, we'll focus on Linux for Intel x86. While Linux can and does run on many different processor architectures, Linux on x86 is its most dominant and important deployment, and thus the focus of our attention.

The Linux Address Space

Much like other modern operating systems, and also like VAX/VMS, a Linux virtual address space[1] consists of a user portion (where user program code, stack, heap, and other parts reside) and a kernel portion (where kernel code, stacks, heap, and other parts reside). Like those other systems, upon a context switch, the user portion of the currently-running address space changes; the kernel portion is the same across processes. Like those other systems, a program running in user mode cannot access kernel virtual pages; only by trapping into the kernel and transitioning to privileged mode can such memory be accessed.

[1] Until recent changes, due to security threats, that is. Read the subsections below about Linux security for details on this modification.

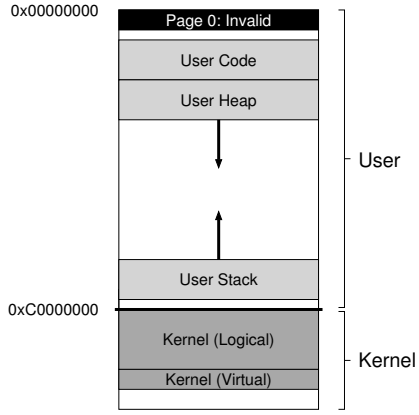


Figure 23.2: **The Linux Address Space**

In classic 32-bit Linux (i.e., Linux with a 32-bit virtual address space), the split between user and kernel portions of the address space takes place at address `0xC0000000`, or three-quarters of the way through the address space. Thus, virtual addresses `0` through `0xBFFFFFFF` are user virtual addresses; the remaining virtual addresses (`0xC0000000` through `0xFFFFFFFF`) are in the kernel's virtual address space. 64-bit Linux has a similar split but at slightly different points. Figure 23.2 shows a depiction of a typical (simplified) address space.

One slightly interesting aspect of Linux is that it contains two types of kernel virtual addresses. The first are known as **kernel logical addresses** [O16]. This is what you would consider the normal virtual address space of the kernel; to get more memory of this type, kernel code merely needs to call **`kmalloc`**. Most kernel data structures live here, such as page tables, per-process kernel stacks, and so forth. Unlike most other memory in the system, kernel logical memory *cannot* be swapped to disk.

The most interesting aspect of kernel logical addresses is their connection to physical memory. Specifically, there is a direct mapping between kernel logical addresses and the first portion of physical memory. Thus, kernel logical address `0xC0000000` translates to physical address `0x00000000`, `0xC0000FFF` to `0x00000FFF`, and so forth. This direct mapping has two implications. The first is that it is simple to translate back and forth between kernel logical addresses and physical addresses; as a result, these addresses are often treated as if they are indeed physical. The second is that if a chunk of memory is contiguous in kernel logical address space, it is also contiguous in physical memory. This makes memory allocated in this part of the kernel's address space suitable for operations which need contiguous physical memory to work correctly, such as I/O transfers to and from devices via **directory memory access** (**DMA**) (something we'll learn about in the third part of this book).

The other type of kernel address is a **kernel virtual address**. To get memory of this type, kernel code calls a different allocator, **`vmalloc`**, which returns a pointer to a virtually contiguous region of the desired size. Unlike kernel logical memory, kernel virtual memory is usually not contiguous; each kernel virtual page may map to non-contiguous physical pages (and is thus not suitable for DMA). However, such memory is easier to allocate as a result, and thus used for large buffers where finding a contiguous large chunk of physical memory would be challenging.

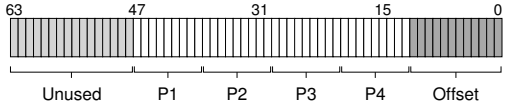
In 32-bit Linux, one other reason for the existence of kernel virtual addresses is that they enable the kernel to address more than (roughly) 1 GB of memory. Years ago, machines had much less memory than this, and enabling access to more than 1 GB was not an issue. However, technology progressed, and soon there was a need to enable the kernel to use larger amounts of memory. Kernel virtual addresses, and their disconnection from a strict one-to-one mapping to physical memory, make this possible. However, with the move to 64-bit Linux, the need is less urgent, because the kernel is not confined to only the last 1 GB of the virtual address space.

Page Table Structure

Because we are focused on Linux for x86, our discussion will center on the type of page-table structure provided by x86, as it determines what Linux can and cannot do. As mentioned before, x86 provides a hardware-managed, multi-level page table structure, with one page table per process; the OS simply sets up mappings in its memory, points a privileged register at the start of the page directory, and the hardware handles the rest. The OS gets involved, as expected, at process creation, deletion, and upon context switches, making sure in each case that the correct page table is being used by the hardware MMU to perform translations.

Probably the biggest change in recent years is the move from 32-bit x86 to 64-bit x86, as briefly mentioned above. As seen in the VAX/VMS system, 32-bit address spaces have been around for a long time, and as technology changed, they were finally starting to become a real limit for programs. Virtual memory makes it easy to program systems, but with modern systems containing many GB of memory, 32 bits were no longer enough to refer to each of them. Thus, the next leap became necessary.

Moving to a 64-bit address affects page table structure in x86 in the expected manner. Because x86 uses a multi-level page table, current 64-bit systems use a four-level table. The full 64-bit nature of the virtual address space is not yet in use, however, rather only the bottom 48 bits. Thus, a virtual address can be viewed as follows:



As you can see in the picture, the top 16 bits of a virtual address are unused (and thus play no role in translation), the bottom 12 bits (due to the 4-KB page size) are used as the offset (and hence just used directly, and not translated), leaving the middle 36 bits of virtual address to take part in the translation. The P1 portion of the address is used to index into the topmost page directory, and the translation proceeds from there, one level at a time, until the actual page of the page table is indexed by P4, yielding the desired page table entry.

As system memories grow even larger, more parts of this voluminous address space will become enabled, leading to five-level and eventually six-level page-table tree structures. Imagine that: a simple page table lookup requiring six levels of translation, just to figure out where in memory a certain piece of data resides.

Large Page Support

Intel x86 allows for the use of multiple page sizes, not just the standard 4-KB page. Specifically, recent designs support 2-MB and even 1-GB pages in hardware. Thus, over time, Linux has evolved to allow applications to utilize these **huge pages** (as they are called in the world of Linux).

Using huge pages, as hinted at earlier, leads to numerous benefits. As seen in VAX/VMS, doing so reduces the number of mappings that are needed in the page table; the larger the pages, the fewer the mappings. However, fewer page-table entries is not the driving force behind huge pages; rather, it's better TLB behavior and related performance gains.

When a process actively uses a large amount of memory, it quickly fills up the TLB with translations. If those translations are for 4-KB pages, only a small amount of total memory can be accessed without inducing TLB misses. The result, for modern "big memory" workloads running on machines with many GBs of memory, is a noticeable performance cost; recent research shows that some applications spend 10% of their cycles servicing TLB misses [B+13].

Huge pages allow a process to access a large tract of memory without TLB misses, by using fewer slots in the TLB, and thus is the main advantage. However, there are other benefits to huge pages: there is a shorter TLB-miss path, meaning that when a TLB miss does occur, it is serviced more quickly. In addition, allocation can be quite fast (in certain scenarios), a small but sometimes important benefit.

One interesting aspect of Linux support for huge pages is how it was done incrementally. At first, Linux developers knew such support was only important for a few applications, such as large databases with stringent performance demands. Thus, the decision was made to allow applications to explicitly request memory allocations with large pages (either through the `mmap()` or `shmget()` calls). In this way, most applications would be unaffected (and continue to use only 4-KB pages; a few demanding applications would have to be changed to use these interfaces, but for them it would be worth the pain.

TIP: CONSIDER INCREMENTALISM

Many times in life, you are encouraged to be a revolutionary. "Think big!", they say. "Change the world!", they scream. And you can see why it is appealing; in some cases, big changes are needed, and thus pushing hard for them makes a lot of sense. And, if you try it this way, at least they might stop yelling at you.

However, in many cases, a slower, more incremental approach might be the right thing to do. The Linux huge page example in this chapter is an example of engineering incrementalism; instead of taking the stance of a fundamentalist and insisting large pages were the way of the future, developers took the measured approach of first introducing specialized support for it, learning more about its upsides and downsides, and, only when there was real reason for it, adding more generic support for all applications.

Incrementalism, while sometimes scorned, often leads to slow, thoughtful, and sensible progress. When building systems, such an approach might just be the thing you need. Indeed, this may be true in life as well.

More recently, as the need for better TLB behavior is more common among many applications, Linux developers have added **transparent** huge page support. When this feature is enabled, the operating system automatically looks for opportunities to allocate huge pages (usually 2 MB, but on some systems, 1 GB) without requiring application modification.

Huge pages are not without their costs. The biggest potential cost is **internal fragmentation**, i.e., a page that is large but sparsely used. This form of waste can fill memory with large but little used pages. Swapping, if enabled, also does not work well with huge pages, sometimes greatly amplifying the amount of I/O a system does. Overhead of allocation can also be bad (in some other cases). Overall, one thing is clear: the 4-KB page size which served systems so well for so many years is not the universal solution it once was; growing memory sizes demand that we consider large pages and other solutions as part of a necessary evolution of VM systems. Linux's slow adoption of this hardware-based technology is evidence of the coming change.

The Page Cache

To reduce costs of accessing persistent storage (the focus of the third part of this book), most systems use aggressive **caching** subsystems to keep popular data items in memory. Linux, in this regard, is no different than traditional operating systems.

The Linux **page cache** is unified, keeping pages in memory from three primary sources: **memory-mapped files**, file data and metadata from devices (usually accessed by directing `read()` and `write()` calls to the file system), and heap and stack pages that comprise each process (sometimes called **anonymous memory**, because there is no named file underneath of

ASIDE: THE UBIQUITY OF MEMORY-MAPPING

Memory mapping predates Linux by some years, and is used in many places within Linux and other modern systems. The idea is simple: by calling `mmap()` on an already opened file descriptor, a process is returned a pointer to the beginning of a region of virtual memory where the contents of the file seem to be located. By then using that pointer, a process can access any part of the file with a simple pointer dereference.

Accesses to parts of a memory-mapped file that have not yet been brought into memory trigger **page faults**, at which point the OS will page in the relevant data and make it accessible by updating the page table of the process accordingly (i.e., **demand paging**).

Every regular Linux process uses memory-mapped files, even the code in `main()` does not call `mmap()` directly, because of how Linux loads code from the executable and shared library code into memory. Below is the (highly abbreviated) output of the `pmap` command line tool, which shows what different mapping comprise the virtual address space of a running program (the shell, in this example, `tcsh`). The output shows four columns: the virtual address of the mapping, its size, the protection bits of the region, and the source of the mapping:

```
0000000000400000    372K r-x-- tcsh
00000000019d5000   1780K rw---   [anon ]
00007f4e7cf06000   1792K r-x-- libc-2.23.so
00007f4e7d2d0000     36K r-x-- libcrypt-2.23.so
00007f4e7d508000    148K r-x-- libtinfo.so.5.9
00007f4e7d731000    152K r-x-- ld-2.23.so
00007f4e7d932000     16K rw---   [stack ]
```

As you can see from this output, the code from the `tcsh` binary, as well as code from `libc`, `libcrypt`, `libtinfo`, and code from the dynamic linker itself (`ld.so`) are all mapped into the address space. Also present are two anonymous regions, the heap (the second entry, labeled `anon`) and the stack (labeled `stack`). Memory-mapped files provide a straightforward and efficient way for the OS to construct a modern address space.

it, but rather swap space). These entities are kept in a **page cache hash table**, allowing for quick lookup when said data is needed.

The page cache tracks if entries are **clean** (read but not updated) or **dirty** (a.k.a., **modified**). Dirty data is periodically written to the backing store (i.e., to a specific file for file data, or to swap space for anonymous regions) by background threads (called `pdflush`), thus ensuring that modified data eventually is written back to persistent storage. This background activity either takes place after a certain time period or if too many pages are considered dirty (both configurable parameters).

In some cases, a system runs low on memory, and Linux has to decide

which pages to kick out of memory to free up space. To do so, Linux uses a modified form of **2Q** replacement [JS94], which we describe here.

The basic idea is simple: standard LRU replacement is effective, but can be subverted by certain common access patterns. For example, if a process repeatedly accesses a large file (especially one that is nearly the size of memory, or larger), LRU will kick every other file out of memory. Even worse: retaining portions of this file in memory isn't useful, as they are never re-referenced before getting kicked out of memory.

The Linux version of the 2Q replacement algorithm solves this problem by keeping two lists, and dividing memory between them. When accessed for the first time, a page is placed on one queue (called `A1` in the original paper, but the **inactive list** in Linux); when it is re-referenced, the page is promoted to the other queue (called `Aq` in the original, but the **active list** in Linux). When replacement needs to take place, the candidate for replacement is taken from the inactive list. Linux also periodically moves pages from the bottom of the active list to the inactive list, keeping the active list to about two-thirds of the total page cache size [G04].

Linux would ideally manage these lists in perfect LRU order, but, as discussed in earlier chapters, doing so is costly. Thus, as with many OSes, an approximation of LRU (similar to **clock** replacement) is used.

This 2Q approach generally behaves quite a bit like LRU, but notably handles the case where a cyclic large-file access occurs by confining the pages of that cyclic access to the inactive list. Because said pages are never re-referenced before getting kicked out of memory, they do not flush out other useful pages found in the active list.

Security And Buffer Overflows

Probably the biggest difference between modern VM systems (Linux, Solaris, or one of the BSD variants) and ancient ones (VAX/VMS) is the emphasis on security in the modern era. Protection has always been a serious concern for operating systems, but with machines more interconnected than ever, it is no surprise that developers have implemented a variety of defensive countermeasures to halt those wily hackers from gaining control of systems.

One major threat is found in **buffer overflow** attacks[2], which can be used against normal user programs and even the kernel itself. The idea of these attacks is to find a bug in the target system which lets the attacker inject arbitrary data into the target's address space. Such vulnerabilities sometime arise because the developer assumes (erroneously) that an input will not be overly long, and thus (trustingly) copies the input into a buffer; because the input is in fact too long, it overflows the buffer, thus overwriting memory of the target. Code as innocent as the below can be the source of the problem:

[2]See `https://en.wikipedia.org/wiki/Buffer_overflow` for some details and links about this topic, including a reference to the famous article by the security hacker Elias Levy, also known as "Aleph One".

```
int some_function(char *input) {
  char dest_buffer[100];
  strcpy(dest_buffer, input); // oops, unbounded copy!
}
```

In many cases, such an overflow is not catastrophic, e.g., bad input innocently given to a user program or even the OS will probably cause it to crash, but no worse. However, malicious programmers can carefully craft the input that overflows the buffer so as to inject their own code into the targeted system, essentially allowing them to take it over and do their own bidding. If successful upon a network-connected user program, attackers can run arbitrary computations or even rent out cycles on the compromised system; if successful upon the operating system itself, the attack can access even more resources, and is a form of what is called **privilege escalation** (i.e., user code gaining kernel access rights). If you can't guess, these are all Bad Things.

The first and most simple defense against buffer overflow is to prevent execution of any code found within certain regions of an address space (e.g., within the stack). The **NX bit** (for No-eXecute), introduced by AMD into their version of x86 (a similar XD bit is now available on Intel's), is one such defense; it just prevents execution from any page which has this bit set in its corresponding page table entry. The approach prevents code, injected by an attacker into the target's stack, from being executed, and thus mitigates the problem.

However, clever attackers are ... clever, and even when injected code cannot be added explicitly by the attacker, arbitrary code sequences can be executed by malicious code. The idea is known, in its most general form, as a **return-oriented programming** (**ROP**) [S07], and really it is quite brilliant. The observation behind ROP is that there are lots of bits of code (**gadgets**, in ROP terminology) within any program's address space, especially C programs that link with the voluminous C library. Thus, an attacker can overwrite the stack such that the return address in the currently executing function points to a desired malicious instruction (or series of instructions), followed by a return instruction. By stringing together a large number of gadgets (i.e., ensuring each return jumps to the next gadget), the attacker can execute arbitrary code. Amazing!

To defend against ROP (including its earlier form, the **return-to-libc attack** [S+04]), Linux (and other systems) add another defense, known as **address space layout randomization** (**ASLR**). Instead of placing code, stack, and the heap at fixed locations within the virtual address space, the OS randomizes their placement, thus making it quite challenging to craft the intricate code sequence required to implement this class of attacks. Most attacks on vulnerable user programs will thus cause crashes, but not be able to gain control of the running program.

Interestingly, you can observe this randomness in practice rather easily. Here's a piece of code that demonstrates it on a modern Linux system:

```
int main(int argc, char *argv[]) {
    int stack = 0;
    printf("%p\n", &stack);
    return 0;
}
```

This code just prints out the (virtual) address of a variable on the stack. In older non-ASLR systems, this value would be the same each time. But, as you can see below, the value changes with each run:

```
prompt> ./random
0x7ffd3e55d2b4
prompt> ./random
0x7ffe1033b8f4
prompt> ./random
0x7ffe45522e94
```

ASLR is such a useful defense for user-level programs that it has also been incorporated into the kernel, in a feature unimaginatively called **kernel address space layout randomization** (**KASLR**). However, it turns out the kernel may have even bigger problems to handle, as we discuss next.

Other Security Problems: Meltdown And Spectre

As we write these words (August, 2018), the world of systems security has been turned upside down by two new and related attacks. The first is called **Meltdown**, and the second **Spectre**. They were discovered at about the same time by four different groups of researchers/engineers, and have led to deep questioning of the fundamental protections offered by computer hardware and the OS above. See `meltdownattack.com` and `spectreattack.com` for papers describing each attack in detail. Spectre is considered the more problematic of the two.

The general weakness exploited in each of these attacks is that the CPUs found in modern systems perform all sorts of crazy behind-the-scenes tricks to improve performance. One class of technique that lies at the core of the problem is called **speculative execution**, in which the CPU guesses which instructions will soon be executed in the future, and starts executing them ahead of time. If the guesses are correct, the program runs faster; if not, the CPU undoes their effects on architectural state (e.g., registers) tries again, this time going down the right path.

The problem with speculation is that it tends to leave traces of its execution in various parts of the system, such as processor caches, branch predictors, etc. And thus the problem: as the authors of the attacks show, such state can make vulnerable the contents of memory, even memory that we thought was protected by the MMU.

One avenue to increasing kernel protection was thus to remove as much of the kernel address space from each user process and instead have

a separate kernel page table for most kernel data (called **kernel page-table isolation**, or **KPTI**) [G+17]. Thus, instead of mapping the kernel's code and data structures into each process, only the barest minimum is kept therein; when switching into the kernel, then, a switch to the kernel page table is now needed. Doing so improves security and avoids some attack vectors, but at a cost: performance. Switching page tables is costly. Ah, the costs of security: convenience *and* performance.

Unfortunately, KPTI doesn't solve all of the security problems laid out above, just some of them. And simple solutions, such as turning off speculation, would make little sense, because systems would run thousands of times slower. Thus, it is an interesting time to be alive, if systems security is your thing.

To truly understand these attacks, you'll (likely) have to learn a lot more first. Begin by understanding modern computer architecture, as found in advanced books on the topic, focusing on speculation and all the mechanisms needed to implement it. Definitely read about the Meltdown and Spectre attacks, at the websites mentioned above; they actually also include a useful primer on speculation, so perhaps are not a bad place to start. And study the operating system for further vulnerabilities. Who knows what problems remain?

23.3 Summary

You have now seen a top-to-bottom review of two virtual memory systems. Hopefully, most of the details were easy to follow, as you should have already had a good understanding of the basic mechanisms and policies. More detail on VAX/VMS is available in the excellent (and short) paper by Levy and Lipman [LL82]. We encourage you to read it, as it is a great way to see what the source material behind these chapters is like.

You have also learned a bit about Linux. While a large and complex system, it inherits many good ideas from the past, many of which we have not had room to discuss in detail. For example, Linux performs lazy copy-on-write copying of pages upon `fork()`, thus lowering overheads by avoiding unnecessary copying. Linux also demand zeroes pages (using memory-mapping of the `/dev/zero` device), and has a background swap daemon (**`swapd`**) that swaps pages to disk to reduce memory pressure. Indeed, the VM is filled with good ideas taken from the past, and also includes many of its own innovations.

To learn more, check out these reasonable (but, alas, outdated) books [BC05,G04]. We encourage you to read them on your own, as we can only provide the merest drop from what is an ocean of complexity. But, you've got to start somewhere. What is any ocean, but a multitude of drops? [M04]

References

[B+13] "Efficient Virtual Memory for Big Memory Servers" by A. Basu, J. Gandhi, J. Chang, M. D. Hill, M. M. Swift. ISCA '13, June 2013, Tel-Aviv, Israel. *A recent work showing that TLBs matter, consuming 10% of cycles for large-memory workloads. The solution: one massive segment to hold large data sets. We go backward, so that we can go forward!*

[BB+72] "TENEX, A Paged Time Sharing System for the PDP-10" by D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, R. S. Tomlinson. CACM, Volume 15, March 1972. *An early time-sharing OS where a number of good ideas came from. Copy-on-write was just one of those; also an inspiration for other aspects of modern systems, including process management, virtual memory, and file systems.*

[BJ81] "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits" by O. Babaoglu, W. N. Joy. SOSP '81, Pacific Grove, California, December 1981. *How to exploit existing protection machinery to emulate reference bits, from a group at Berkeley working on their own version of* UNIX*: the* ***Berkeley Systems Distribution*** *(****BSD****). The group was influential in the development of virtual memory, file systems, and networking.*

[BC05] "Understanding the Linux Kernel" by D. P. Bovet, M. Cesati. O'Reilly Media, November 2005. *One of the many books you can find on Linux, which are out of date, but still worthwhile.*

[C03] "The Innovator's Dilemma" by Clayton M. Christenson. Harper Paperbacks, January 2003. *A fantastic book about the disk-drive industry and how new innovations disrupt existing ones. A good read for business majors and computer scientists alike. Provides insight on how large and successful companies completely fail.*

[C93] "Inside Windows NT" by H. Custer, D. Solomon. Microsoft Press, 1993. *The book about Windows NT that explains the system top to bottom, in more detail than you might like. But seriously, a pretty good book.*

[G04] "Understanding the Linux Virtual Memory Manager" by M. Gorman. Prentice Hall, 2004. *An in-depth look at Linux VM, but alas a little out of date.*

[G+17] "KASLR is Dead: Long Live KASLR" by D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, S. Mangard. Engineering Secure Software and Systems, 2017. Available: `https://gruss.cc/files/kaiser.pdf` *Excellent info on KASLR, KPTI, and beyond.*

[JS94] "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm" by T. Johnson, D. Shasha. VLDB '94, Santiago, Chile. *A simple but effective approach to building page replacement.*

[LL82] "Virtual Memory Management in the VAX/VMS Operating System" by H. Levy, P. Lipman. IEEE Computer, Volume 15:3, March 1982. *Read the original source of most of this material. Particularly important if you wish to go to graduate school, where all you do is read papers, work, read some more papers, work more, eventually write a paper, and then work some more.*

[M04] "Cloud Atlas" by D. Mitchell. Random House, 2004. *It's hard to pick a favorite book. There are too many! Each is great in its own unique way. But it'd be hard for these authors not to pick "Cloud Atlas", a fantastic, sprawling epic about the human condition, from where the the last quote of this chapter is lifted. If you are smart – and we think you are – you should stop reading obscure commentary in the references and instead read "Cloud Atlas"; you'll thank us later.*

[O16] "Virtual Memory and Linux" by A. Ott. Embedded Linux Conference, April 2016. https://events.static.linuxfound.org/sites/events/files/slides/elc_2016_mem.pdf . *A useful set of slides which gives an overview of the Linux VM.*

[RL81] "Segmented FIFO Page Replacement" by R. Turner, H. Levy. SIGMETRICS '81, Las Vegas, Nevada, September 1981. *A short paper that shows for some workloads, segmented FIFO can approach the performance of LRU.*

[S07] "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by H. Shacham. CCS '07, October 2007. *A generalization of return-to-libc. Dr. Beth Garner said in Basic Instinct, "She's crazy! She's brilliant!" We might say the same about ROP.*

[S+04] "On the Effectiveness of Address-space Randomization" by H. Shacham, M. Page, B. Pfaff, E. J. Goh, N. Modadugu, D. Boneh. CCS '04, October 2004. *A description of the return-to-libc attack and its limits. Start reading, but be wary: the rabbit hole of systems security is deep...*