I

The VAX/VMS Virtual Memory System

Before we end our study of virtual memory, let us take a closer look at one particularly clean and well done virtual memory manager, that found in the VAX/VMS operating system [LL82]. In this note, we will discuss the system to illustrate how some of the concepts brought forth in earlier chapters come together in a complete memory manager.

I.1 Background

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.

THE CRUX: HOW TO AVOID 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. Thus, how can an OS be built so as to run effectively on a wide 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.

I.2 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 make sure 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; 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 pro-

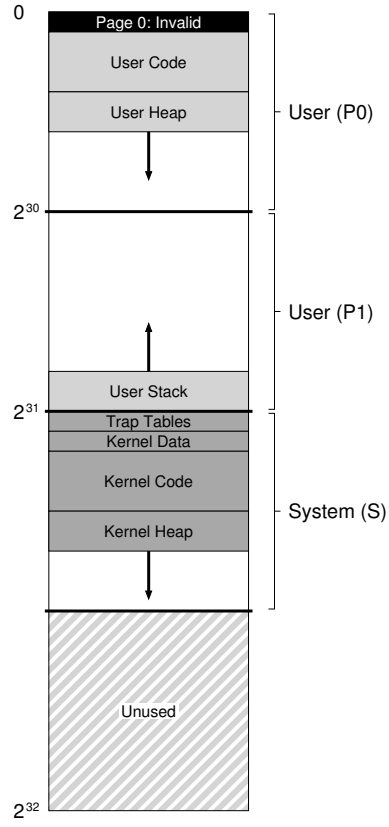

Figure I.1: **The VAX/VMS Address Space**

cess); 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.

I.3 A Real Address Space

One neat aspect of studying VMS is that we can see how a real address space is constructed (Figure I.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.

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 value 10 to virtual address 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).

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.

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, 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.

I.4 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.

Segmented FIFO

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 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].

Page Clustering

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.

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.

I.5 Other Neat VM 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 then 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 of this work is avoided, and thus the virtue of demand zeroing.

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!

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.

I.6 Summary

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

You should also learn more about the state of the art by reading about Linux and other modern systems when possible. There is a lot of source material out there, including some reasonable books [BC05]. One thing that will amaze you: how classic ideas, found in old papers such as this one on VAX/VMS, still influence how modern operating systems are built.

References

[BB+72] "TENEX, A Paged Time Sharing System for the PDP-10"
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson
Communications of the ACM, 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; inspiration for many other aspects of modern systems, including process management, virtual memory, and file systems are found herein.

[BJ81] "Converting a Swap-Based System to do Paging
in an Architecture Lacking Page-Reference Bits"
Ozalp Babaoglu and William N. Joy
SOSP '81, Pacific Grove, California, December 1981
A clever idea paper on how to exploit existing protection machinery within a machine in order to emulate reference bits. The idea came from the group at Berkeley working on their own version of UNIX*, known as the Berkeley Systems Distribution, or BSD. The group was heavily influential in the development of* UNIX*, in virtual memory, file systems, and networking.*

[BC05] "Understanding the Linux Kernel (Third Edition)"
Daniel P. Bovet and Marco Cesati
O'Reilly Media, November 2005
One of the many books you can find on Linux. They go out of date quickly, but many of the basics remain and are worth reading about.

[C03] "The Innovator's Dilemma"
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"
Helen Custer and David 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.

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"
Henry M. Levy, Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982) *Read the original source of most of this material; it is a concise and easy read. 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. But it is fun!*

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