Paging And Virtual Memory

Bindu Singh
Dept of Computer Department
Vadodara Institute of Engineering

Abstract- *Mention the abstract for the article. An abstract is a brief summary of a research article, thesis, review, conference proceeding or any in-depth analysis of a particular subject or discipline, and is often used to help the reader quickly ascertain the paper's purpose. When used, an abstract always appears at the beginning of a manuscript, acting as the point-of-entry for any given scientific paper or patent application.*

I. INTRODUCTION

Virtual memory is a memory management capability of an OS that uses hardware and software to allow a computer to compensate for physical memory shortages by temporarily transferring data from random access memory (RAM) to disk storage. The basic idea behind virtual memory is that each program has its own address space, which is broken up into chunks called pages. Each page is a contiguous range of addresses. These pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time to run the program. When the program references a part of its address space that is in physical. Memory, the hardware performs the necessary mapping on the fly. When the program references a part of its address space that is not in physical memory, the operating system is alerted to go get the missing piece and re-execute the instruction that failed. With virtual memory, instead of having separate relocation for just the text and data segments, the entire address space can be mapped onto physical memory in fairly small units. We will show how virtual memory is implemented below. Virtual memory works just fine in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for pieces of itself to be read in, the CPU can be given to another process.

II. PAGING

It is sometimes said that the operating system takes one of two approaches when solving most any space-management problem. The first approach is to chop things up into variable-sized pieces, as we saw with segmentation in virtual memory. Unfortunately, this solution has inherent difficulties. In particular, when dividing a space into different-size chunks, the space itself can become fragmented, and thus allocation becomes more challenging over time. Thus, it may be worth considering the second approach: to chop up space into fixed-sized pieces. Instead of splitting up a process's address space into some number of variable-sized logical segments (e.g., code, heap, stack), we divide it into fixed-sized units, each of which we call a page. Correspondingly, we view physical memory as an array of fixed-sized slots called page frames; each of these frames can contain a single virtual-memory page. To help make this approach more clear, let's illustrate it with a simple example. Figure 18.1 (page 2) presents an example of a tiny address space, only 64 bytes total in size, with four 16-byte pages (virtual pages 0, 1, 2, and 3). Real address spaces are much bigger, of course, commonly 32 bits and thus 4-GB of address space, or even 64 bits1 ; in the book, we'll often use tiny examples to make them easier to digest.

0-16		Page 0
16-32		Page 1
32-48		Page 2
48-64		Page 3

Physical memory, as shown in second figure consists of a number of fixed-sized slots, in this case eight page frames (making for a 128-byte physical memory, also ridiculously small). As you can see in the diagram, the pages of the virtual address space have been placed at different locations throughout physical memory; the diagram also shows the OS using some of physical memory for itself. Paging, as we will see, has a number of advantages over our previous approaches. Probably the most important improvement will be flexibility: with a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space; we won't, for example, make assumptions about the direction the heap and stack grow and how they are used. Another advantage is the simplicity of free-space management that paging affords. For example, when the OS wishes to place our tiny 64-byte address space into our eight-page physical memory, it simply finds four free pages; perhaps the OS keeps a free list of all free pages for this, and just grabs the first four free pages off of this list. In the example, the OS has placed virtual page 0 of the address space (AS) in physical frame 3, virtual page 1 of the AS in physical frame 7, page 2 in frame 5, and page 3 in frame 2. Page frames 1, 4, and 6 are currently free. To record where each virtual page of the address space is placed in

physical memory, the operating system usually keeps a per-process data structure known as a page table. The major role of the page table is to store address translations for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides. For our simple example (Figure 18.2, page 2), the page table would thus have the following four entries: (Virtual Page 0 → Physical Frame 3), (VP 1 → PF 7), (VP 2 → PF 5), and (VP 3 → PF 2). It is important to remember that this page table is a per-process data structure (most page table structures we discuss are per-process structures; an exception we'll touch on is the inverted page table). If another process were to run in our example above, the OS would have to manage a different page table for it, as its virtual pages obviously map to different physical pages (modulo any sharing going on).

0-16	Reserved for os	page frame 0
16-32	Unused	page frame 1
32-48	Page 3 of AS	page frame 2
48-64	Page 0 of AS	page frame 3
64-80	Unused	page frame 4
80-96	Page 2 o fAS	page frame 5
96-112	Unused	page frame 6
112-128	Page 1 of AS	page frame 7

This approach works the best in guidance of fellow researchers. In this the authors continuously receives or asks inputs from their fellows. It enriches the information pool of your paper with expert comments or up gradations. And the researcher feels confident about their work and takes a jump to start the paper writing.

III. PAGE TABLES

Page tables can get terribly large, much bigger than the small segment table or base/bounds pair we have discussed previously. For example, imagine a typical 32-bit address space, with 4KB pages. This virtual address splits into a 20-bit VPN and 12-bit offset (recall that 10 bits would be needed for a 1KB page size, and just add two more to get to 4KB). A 20-bit VPN implies that there are 2 20 translations that the OS would have to manage for each process (that's roughly a million); assuming we need 4 bytes per page table entry (PTE) to hold the physical translation plus any other useful stuff, we get an immense 4MB of memory needed for each page table! That is pretty large. Now imagine there are 100 processes running: this means the OS would need 400MB of memory just for all those address translations! Even in the modern era, where machines have gigabytes of memory, it seems a little crazy to use a large chunk of it just for translations, no? And we won't even think about how big such a page table would be for a 64-bit address space; that would be too gruesome and perhaps scare you off entirely. Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process. Instead, we store the page table for each process in memory somewhere. Let's assume for now that the page tables live in physical memory that the OS manages; later we'll see that much of OS memory itself can be virtualized, and thus page tables can be stored in OS virtual memory (and even swapped to disk), but that is too confusing right now, so we'll ignore it. What's Actually In The Page Table? Let's talk a little about page table organization. The page table is just a data structure that is used to map virtual addresses (or really, virtual page numbers) to physical addresses (physical frame numbers). Thus, any data structure could work. The simplest form is called a linear page table, which is just an array. The OS indexes the array by the virtual page number (VPN), and looks up the page-table entry (PTE) at that index in order to find the desired physical frame number (PFN). For now, we will assume this simple linear structure; in later chapters, we will make use of more advanced data structures to help solve some problems with paging. As for the contents of each PTE, we have a number of different bits in there worth understanding at some level. A valid bit is common to indicate whether the particular translation is valid; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked invalid, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process. Thus, the valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory. We also might have protection bits, indicating whether the page could be read from, written to, or executed from. Again, accessing a page in a way not allowed by these bits will generate a trap to the OS. There are a couple of other bits that are important but we won't talk about much for now. A present bit indicates whether this page is in physical memory or on disk (i.e., it has been swapped out). We will understand this machinery further when we study how to swap parts of the address space to disk to support address spaces that are larger than physical memory; swapping allows the OS to free up physical memory by moving rarely-used pages to disk. A dirty bit is also common, indicating whether the page has been modified since it was brought into memory. A reference bit (a.k.a. accessed bit) is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory; such knowledge is critical during page replacement, a topic we will study in great detail in subsequent chapters. Figure 18.5 shows an example page table entry from the x86 architecture [I09]. It contains a present bit (P); a read/write bit (R/W)

which determines if writes are allowed to this page; a user/supervisor bit (U/S) which determines if user-mode processes can access the page; a few bits (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages; an accessed bit (A) and a dirty bit (D)

IV. CONCLUSION

Paging is a process to translate virtual address to physical address. Virtual memory is the memory used by process whenever there is a shortfall in the memory in the physical memory.

REFERENCES

[1] pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf
[2] www.uobabylon.edu.iq/download/M.S%202013-2014/Operating_System_Concepts,_8th_Edition%5BA4%5D.pdf