54

Virtual Memory

Peter J. Denning
Naval Postgraduate School in Monterey

54.1 Introduction

Virtual memory is one of the engineering triumphs of the computer age. In the 1960s, it became a standard feature of nearly every operating system and computer chip. In the 1990s, it became also a standard feature of the Internet and the World Wide Web (WWW). It was the subject of intense controversy after its introduction in 1959; today, it is such an ordinary part of infrastructure that few people think much about it.

Early programmers had to solve a memory *overlay problem* as part of their programming work. Early computers had very small amounts of random access memory (RAM) because storage technology was so expensive. Programmers stored a master copy of their programs (and data) on a secondary storage system—then a drum, today a disk—and pulled pieces into the RAM as needed. Deciding which pieces to pull and which parts of RAM to replace was called "overlaying." It was estimated that most programmers spent half to two-thirds of their time planning overlay sequences. As a reliable method of automating, virtual memory had the potential to increase programmer productivity and reduce debugging by severalfold. Virtual memory was invented to automate solutions to the overlay problem.

Virtual memory was taken into the large commercial operating systems of the 1960s to simplify the new features of time-sharing and multiprogramming. The RAMs were considerably larger, sometimes allowing individual programs to be fully loaded. Virtual memory provided three additional benefits in these systems: it isolated users from each other, it allowed dynamic relocation of program pieces within RAM, and it provided read–write access control to individual pieces. These benefits were so for fault tolerance (Denning 1976) that virtual memory was used even when there was sufficient RAM for every program. Thus, the story of virtual memory is not simply a story of progress in automatic storage allocation; it is a story of machines helping programmers to protect information, reuse and share objects, and link software components.

The first operating system with virtual memory was the 1959 Manchester Atlas operating system (Fotheringham 1961, Kilburn et al. 1962). They called their virtual memory a "one-level store." They simulated a large memory—sufficient to hold an entire program—by moving fixed-size pages of a large file on disk in and out of a small RAM. The system appeared as a large, slower RAM to its users. The term "virtual memory" soon superseded the original name because of an analogy to optics: a virtual image in a mirror or lens is an illusion, made from traces of a real object that is actually somewhere else.

54.2 Early Virtual Memory Systems

Electronic digital computers have always had memory hierarchies consisting of at least two levels (Figure 54.1):

1. The main memory, often called RAM, is directly connected to the central processing unit (CPU). RAM is fast and can keep up with the CPU. RAM is volatile; loss of power erases its contents. RAM is expensive.
2. The secondary memory is connected to RAM by a subsystem that implements "up" and "down" moves of blocks of data. It was originally rotating drums, and today it is usually rotating or solid state disks. It is persistent, meaning that data are written as magnetic or optical patterns that do not disappear until explicitly erased. It is much cheaper than RAM.

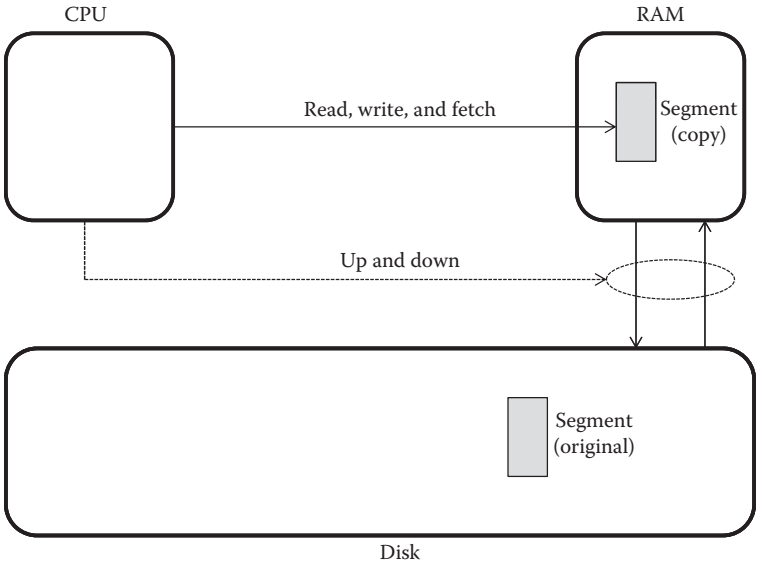

FIGURE 54.1 Two-level memory hierarchy. A processor executes a program from main memory. The entire program is held in the secondary memory (DISK), and segments of it are copied to the main memory (RAM) for processing. The overlay problem is to partition the program into segments and schedule their moves up and down the hierarchy. The manual work to construct a program's overlay sequence doubles or triples the programming time from when no overlays are needed.

Today's disk technology is about a thousand (10^3) times cheaper per bit than RAM and about a million (10^6) times slower. These differentials make it a challenge to find a good trade-off between main and secondary storage. The goal is enough RAM to run programs at close to CPU speed and sufficient permanent storage for all data.

The secondary storage system can include other technologies such as solid state disks, CD/DVD recorders and players, flash memory sticks, tape backups, and remote Internet storage (such as dropbox.com). These non-disk secondary devices are not usually part of the virtual memory system.

Data on RAM or disk are organized into blocks of contiguous bytes. The blocks are units of storage and transfer between the memory levels. Many systems use all blocks of the same fixed size, called *pages*. A few systems allow blocks of different sizes, in which case they are called *segments*.

54.2.1 Address Mapping

The original purpose of virtual memory was to solve the burdensome overlay problem by automating the movement of pages between memory levels. The 1959 Atlas computer system at the University of Manchester was the first working prototype (Fotheringham 1961, Kilburn et al. 1962). The heart of this system was a radical innovation: a distinction between *address* and memory *location*. An address was the name of a byte, and a location was the physical storage holding the byte. This distinction enabled designers to provide programmers a large address space (AS) even though the system had a smaller amount of RAM. Programmers could then design software as if all the code and data fit into a single, contiguous AS.

This distinction led the Atlas designers to three inventions: (1) They built an *address translator*, hardware that automatically converted each address generated by the CPU to its current memory location. (2) They devised *demand paging*, an interrupt mechanism triggered by the address translator that moved a missing page from secondary memory to RAM. (3) They built the first *replacement algorithm*, a procedure to detect and move the least useful pages back to secondary memory.

Despite the success of the Atlas memory system, the literature of 1961 records a spirited debate about the feasibility of automatic storage allocation in general-purpose computers. By that time, Cobol, Algol, Fortran, and Lisp had become the first widely used higher level programming languages. These languages made storage allocation harder because programs were larger, more portable, more modular, and their dynamics more dependent on their input data. Through the 1960s, there were dozens of experimental studies that sought to either affirm or deny that virtual memory systems could do a better job at storage allocation than any compiler or programmer (Denning 1970). The matter was finally laid to rest—in favor of virtual memory—by an extensive study of system performance by an IBM research team led by David Sayre (Sayre 1969).

54.2.2 Multiprogramming

Convinced that virtual memory was the right way to go, the makers of major commercial computers adopted it in the 1960s. Virtual memory was in the IBM 360/67, CDC 7600, Burroughs 5500/6500, RCA Spectra/70, and Multics for the GE 645. By the mid-1970s, the IBM 370, DEC VMS, DEC TENEX, and Unix also had virtual memory.

All these systems used multiprogramming, a mode in which several programs simultaneously reside in RAM. Multiprogramming was intended for better CPU utilization and throughput: if a program stops for input–output, the CPU can switch to another program.

Virtual memory married nicely with multiprogramming. It gave a clean way to logically partition the RAM among multiple programs, preventing them from interfering with one another. This worked because the address translator in the CPU used a distinct mapping table for each program; the mapping table pointed only to the pages of that program's AS. The address translator also recognized access codes, thus protecting read-only pages from being overwritten.

54.2.3 Thrashing

But these design choices exacted an unexpected price. The first multiprogrammed virtual memory systems succumbed to an unexpected and most unwelcome surprise: *thrashing*. Thrashing was a condition of near-total performance collapse when the multiprogramming level became too high (Denning 1968). The performance collapse worked as follows. The system would be operating well, with good throughput. Occasionally, the activation of one additional program pushed the system over an edge into a state with extremely low throughput. Engineers soon determined that in this state, every job was waiting in the disk queue for a page to be moved; and when its page was moved, every job very quickly rejoined the queue with a new request. The engineers called this state "paging to death." They did not understand the conditions that triggered it or how to design a load control that would prevent it.

The solution to thrashing lay with a new model of program behavior, called the *working set (WS) model* (Denning 1968). This model hypothesized that every program had a WS, a dynamic subset of pages that it needed in RAM for efficient execution; and that most of the time, the WS did not change much from one page reference to the next. Thrashing would be impossible as long as every program has its WS loaded in RAM. The operating system would not activate any new program whose WS could not fit into the unused part of RAM.

A series of early experiments confirmed the WS hypothesis and gave engineers insights on how to retrofit their multiprogrammed virtual memory systems to avoid thrashing. It also stimulated a long line of experiments and models seeking to understand why computations would exhibit locality, the principle behind the working-set behavior. By the late 1970s, these investigations had produced deep understanding of the principle of locality and had confirmed that working-set memory controllers would achieve throughput close to the theoretical optimum (Denning 1980).

New studies by Adrian McMenamin in 2011 for Linux reconfirmed the locality behavior of programs in modern systems and the efficacy of WS memory management (McMenamin 2011).

54.3 Cache Systems

The address mapping principle of virtual memory attracted hardware designers as well as software designers. In 1965, Maurice Wilkes proposed the *slave memory*, a small high-speed store included in the CPU to hold a small number of most recently used words from RAM (Wilkes 1965). Like virtual memory, slave memory used address translation, demand loading, and usage-based replacement. Wilkes argued that designing translation, loading, and replacement strategies is easier when the memory hierarchy is two forms of RAM: moving small blocks is efficient because of smaller speed differences. By exploiting the principle of locality, a small slave memory eliminates many data transfers between CPU and main memory, allowing the CPU to run much faster with hardly any increase of memory cost.

This idea became popular among hardware designers. IBM introduced a cache memory in 1968 as part of its 360/85 machine. IBM and others placed small caches in the CPU, enabling the CPU to run faster because it could bypass many accesses to RAM. Modern chips incorporate two levels of cache—L1 operates at register speed and simulates a large set of registers, while L2 operates at a slower speed and bypasses many references to RAM. Cache memory is now a standard principle of computer architecture (Hennessey and Patterson 1990).

The term *caching* is often used to mean that a small set of data are moved to a location close to the CPU, enabling the CPU to run fast on that set, bypassing delays in accessing the data at a larger distance away. Data and instructions cached in the CPU enable faster CPU execution. Files cached in disk controllers enable fast re-access to recently used files. Web pages cached on local servers enable faster re-display and bypass the longer access times in the Internet.

54.4 Object Systems

In the 1960s, virtual memory was seen as a method to make programming easier and more secure:

1. Programmers can write their code without having to worry about overlays, which are done automatically by the system.
2. Programmers can share memory without fear of interference from other programs.

If it ended here, this story would already have guaranteed virtual memory a place in history. But the designers of the 1960s saw even more possibilities to adapt virtual memory to make programming more productive and secure. They added two new programming objectives:

3. Programmers can share, reuse, or recompile any program module without requiring changes to any other program module.
4. Programmers can define abstract data types by specifying a package consisting of a hidden internal data structure to represent an object's state and procedures to access an object's state.

The first new objective was met by a *segmented AS*, an extension of the original linear AS. The second new objective was met by *capability-based architecture*, which later morphed into *object-oriented programming*. Both extended the basic virtual memory in important ways.

In 1965, the designers of Multics at MIT believed that programmer communities develop around interactive, time-shared computing systems, and that members of these communities want to share separately compiled program modules that can be linked together on demand (Dennis 1965, Organick 1972). Link-on-demand was a significant departure from the more common practice of using a linking loader (or makefile program) to bind component files to together into an AS.

In a computer with 32 bit addresses, an AS can contain up to 2^{32} addresses. Multics added a second dimension that selected one of 2^{32} ASs, called segments. Thus, a Multics CPU address consisted of a pair of 32 bit addresses (s, x) denoting segment s, offset x.

In Multics, a program encountering a symbolic reference to a variable X within a segment S would be interrupted by a *linkage fault*. The linkage fault handler would convert the S to a segment number s and the X to a linear offset x. After the conversion, the program would not encounter the same linkage fault again.

The Multics virtual memory demonstrated innovations in sharing, reuse, access control, and protection. Many of its innovations, however, did not find their way beyond Multics; programmers were content with one private, linear AS and a handful of open files. As will be discussed, the WWW (Berners-Lee 2000) changed this: programs and documents contain *hypertext links*: symbolic pointers to other objects that are not linked until the program references them for the first time.

In 1966, Jack Dennis and Earl Van Horn published a landmark paper that initiated a new line of computer architectures: machines that helped programmers create protected managers of classes of objects. They anticipated what is now called object-oriented programming. They were especially concerned that objects be freely reusable and shareable and, at the same time, be protected from unauthorized internal access. They proposed an extension of virtual memory that used two levels of mapping instead of one to get to an object. The first level maps an object name to a persistent global identifier, called a *capability*, for the object. The second level maps the capability to the object's location. Capabilities acted as tickets granting access to their objects. You could share an object by giving another person a copy of your capability for the object. Capability addressing offered an elegant solution to the problem of sharing and reusing modules.

Several commercial and academic capability systems were built in the 1970s: notably the Plessey 250, IBM System 38, Cambridge CAP, Intel 432, SWARD, and Hydra. These systems implemented capabilities as long addresses (for example, 64 bits), which the hardware protected from alteration (Fabry 1974, Myers 1982, Wilkes and Needham 1979). The reduced instruction set computer, coupled with programming languages with type checking, made capability-managing hardware obsolete by the mid-1980s.

But software-managed capabilities, now called *handles*, are indispensable in modern object-oriented programming systems, databases, and distributed operating systems (Chase et al. 1994). The same conceptual structure reappeared in a proposal to manage objects and intellectual property in the Internet (Kahn and Wilensky 1995). It is a powerful structure indeed.

54.5 Virtual Memory in Other Systems

Some people wonder why virtual memory, which was so popular and ultimately successful in the operating systems of the 1960s and 1970s, was not present in initial versions of several new technologies of the 1980s:

- Personal computers (PCs)
- Highly parallel supercomputers
- Distributed memory computers
- WWW

These technologies emerged from new communities that initially had no contact with the earlier generation of operating systems or rebelled against that generation. PC developers, for example, distanced themselves from mainframe companies because they wanted computing affordable to an individual home user. They did not have the memory capacity or knowhow to implement a full operating system and often believed they could do much better with extreme simplicity. After a few years, however, they rediscovered the same programming issues that motivated the early operating system designers. They started to use locality and virtual memory to improve their productivity and build faster machines.

For PCs, the pundits of the microcomputer revolution proclaimed that PCs would not succumb to the diseases of the large commercial operating systems; the PC would be simple, fast, and cheap. Bill Gates once said that no user of a PC would ever need more than 640 K of main memory. His first Microsoft operating system, DOS (1982), did not include most of the common functions, such as multiprogramming and virtual memory. Eventually, the PC makers (Apple, Microsoft, and IBM) added multiprogramming and virtual memory to their operating systems. They were able to do this because the major chip makers had not lost faith; Intel offered virtual memory and cache in its 80386 chip in 1985; Motorola did likewise in its 68020 chip. Apple offered multiprogramming in its MultiFinder and virtual memory in its System 6 operating system. Microsoft offered multiprogramming in Windows 3.1 and virtual memory in Windows 95. IBM offered multiprogramming and virtual memory in OS/2.

A similar pattern appeared in the early development of distributed-memory multicomputers beginning in the mid-1980s. These machines allowed for a large number of computers, sharing a high-speed interconnection network, to work concurrently on a single problem. Around 1985, Intel and N-Cube introduced the first hypercube machines consisting of 128 component microcomputers. Shortly thereafter, Thinking Machines produced the first commercial supercomputer of this genre, the Connection Machine, with as many as 65,536 component computer chips. These machines soon challenged the traditional supercomputer by offering the same aggregate processing speed at a much lower cost (Denning and Tichy 1990). Their designers initially eschewed virtual memory, believing that address translation and page swapping would seriously detract from the machine's performance. But they quickly encountered new programming problems in synchronizing processes on different chips and exchanging data among them. The ordinary method of passing a message to another chip involved three copy operations: first from the sender's local memory to a local buffer, then across the network to a buffer in the receiver, and finally to the receiver's local memory. With virtual memory, the same transfer takes one copy, invoked at the time of reference. Virtual memory significantly decreased communication costs in these machines. Tannenbaum (1995) describes a variety of implementation issues under the topic of distributed shared memory.

The WWW, started in 1989 by Tim Berners-Lee, sought to link together all information in the world. It accomplished this by creating a virtual AS for objects in the WWW. The name of an object is given by its *uniform resource locator* (URL). The Web protocols (such as hypertext transfer protocol [HTTP] and

domain name service [DNS]) map URLs to hosts and files in the Internet. The mapping operations are performed only when someone tries to use a URL by clicking a mouse on it. The URL differs from a standard virtual address map because it is possible that a URL points to a nonexistent object or possibly to a new object given the same name as a previously deleted object. To avoid the problem of URLs becoming invalid when the object's owner moves it to a new machine, Kahn and Wilensky proposed that objects be named by global persistent handles; handles are translated with a two-level mapping scheme first into a URL, then into the machine hosting the object (Denning and Kahn 2010, Kahn and Wilensky 1995). The handle scheme recalls the Dennis-Van-Horn capability system of the 1960s, but now with worldwide, decentralized mapping systems. The Java language allowed URLs to address programs as well as documents; when a Java interpreter encounters the URL of another Java program, it brings a copy of that program to the local machine and executes it as an applet. These technologies, now seen as essential for the Internet, vindicate the view of the Multics designers in 1965—that many large-scale computations will consist of many processes roaming a large space of shared objects.

In the Internet, some objects such as Web pages or Web sites become very popular and attract many links. People using the links can create a high demand for a site, causing a queue of backlogged requests and thus a significant delay to a user wanting a fast access to the site. Even though the Internet architecture is relatively flat—the access time ("ping time") to most sites is typically 20–30 ms—queueing at popular sites can cause very slow response times much longer than 30 ms. Caching solves this problem. A cache server can be placed in a cluster of users where it collects their recent web page requests; it can satisfy many repeat requests locally without going to the main site and its long queue. The company Akamai has become a leader in this performance-enhancing technology for the Internet.

From time to time over the past 50 years, various people have argued that virtual memory is not really necessary because advancing memory technology would soon permit us to have all the RAM we could possibly want. Each new generation of users has discovered that its ambitions for processing, memory, and sharing led it to virtual memory. It is unlikely that today's predictions of the passing of virtual memory will prove to be any more reliable than similar predictions made every year since 1960. Virtual memory accommodates essential patterns in the way people use computers to communicate and share information. It will still be used when we are all gone.

54.6 Structure of Virtual Memory

Let us now examine the structure of virtual memory systems. We will begin with the simplest form, a paging system, which was historically the first form.

Virtual memory was originally designed to solve the overlay problem in systems with two levels of memory. In 1959, the time of the first virtual memory, main memory (RAM) access times were about 10^{-6} s (1 μs) and secondary memory (drum) access times were about 10^{-2} s (10 ms), giving a speed ratio of about 10^4. A single-page fault (up-move) would force the CPU be idle for about 10^4 instructions, a stiff penalty. The designers, therefore, sought replacement policies that would minimize the number of page faults. The situation is worse today, with RAM access times around 10^{-9} s and disk around 10^{-3}, for a speed ratio of 10^6. These speed ratios do not make virtual memory very attractive for those who believe that its primary purpose is to move pages up and down the hierarchy.

All virtual memories are based on the principle of distinguishing addresses from locations, and providing a dynamic map that translates an address to its storage location. The map can be updated whenever the location changes.

Every program and its data must fit inside a virtual AS, which is a sequential set of addresses that the CPU can generate. If addresses are k bits long, the AS consists of 2^k bytes, designated 0, 1,…, 2^k–1. Thus, 16 bit addresses can span a space of 65,536 bytes. Today, 32 bit addresses are common, spanning spaces of about 4 gigabytes.

It would be too costly to build a mapping table that mapped individual bytes to their locations. With 32 bit addresses, such a table would require 4 gigabytes of memory. A system with 64 running user

programs would require around a terabyte of memory just for the 64 mapping tables. The size of mapping tables is significantly reduced by dividing AS into equal size blocks, called *pages*, and the RAM into same-size blocks, called *frames*. The table maps pages to frames. If the page size were 1024 bytes (10^{10}), the mapping table would be smaller by a factor of 1024. Such tables are feasible.

The mapping is organized as follows. A *page table* contains entries of the form, one for each page:

$$(P, U, M, A, B)$$

where

- P is a *presence bit* set to 1 when the page is in RAM
- U is a *use bit* set to 1 when the page is read or written
- M is a *modified bit* set to 1 when the page is written
- A is an *access code* indicating read or write permission
- B is the *base address* of the page in RAM

A virtual address of the CPU is encoded as

$$(i,x)$$

meaning page i, line x. This address is translated to a memory address by adding the displacement x to the page's base address B.

With the help of Figure 54.2, let us walk through the components of a paged virtual memory. The circled numbers in the figure flag four key elements. Every virtual AS has its own page table; by

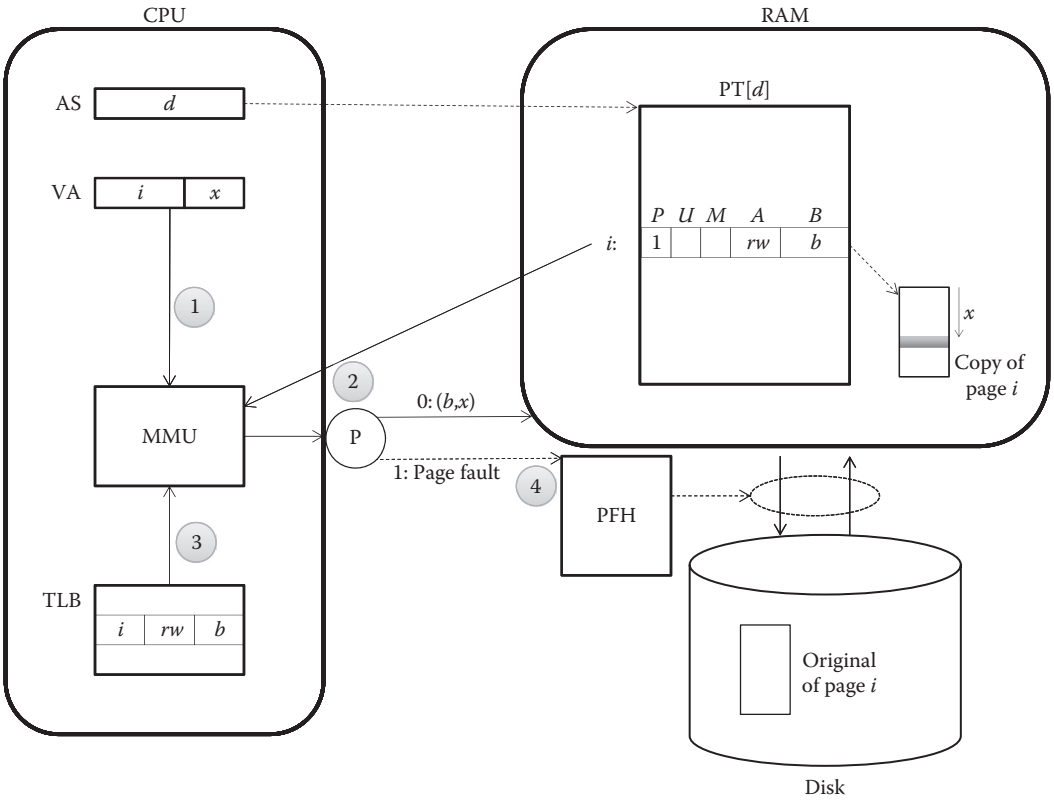

FIGURE 54.2 Structure of paged virtual memory system. The four numbered elements are discussed in the text.

default there are no shared pages. The CPU register AS points to page table the CPU uses for address mapping on behalf of the process it is currently running.

54.6.1 Element 1: Providing the Virtual Address to the Memory Mapping Unit

The memory mapping unit (MMU) is a hardware component that translates virtual addresses to memory locations. A virtual address is a linear offset into the AS. The high order bits of the virtual address in register VA are interpreted as a page number and the low order bits as a line number. An example will show how this works. Assume a page size of $2^{10} = 1024$ bytes in a system with 32-bit addresses. In this case, the virtual AS will contain $2^{22} = 4$ million pages. In a 32-bit address, the 22 high-order bits are page number and 10 low order bits are the line number in the page. Thus, address 8044 will be on page 7, line 876, because $8044 = (7)(2^{10}) + 876$.

54.6.2 Element 2: Mapping the Address

The MMU's job is to map the incoming virtual address to a memory location. The MMU looks up the frame for the current page in the page table. In our example, the page table has one entry per page for a total of 2^{22} entries. If page 7 is in frame 50, the base field B of the page table entry for page 7 will contain 50, which we can write as $PT[d,7].B = 50$. The MMU decomposes the target address 8044 into the two components (7,876), retrieves 50 from page table entry 7, and then presents the location address (50,876) to the RAM.

The P-bit of a page table entry records whether the page is present in RAM. If page 7 is not present, its P-bit will be 0. In that case, the MMU cannot map the virtual address. Instead, it halts with an error condition called *page fault*. The page fault condition triggers an interrupt, which passes control to an operating system routine called *page fault handler* (PFH). The operation of PFH will be discussed shortly at step 4.

54.6.3 Element 3: Checking the Translation Lookaside Buffer

The translation lookaside buffer (TLB) is a small set of high-speed associative registers attached to the MMU. Its purpose is to enable the MMU to bypass the page table lookup as often as possible. Without it, the virtual memory would run at half the rated RAM speed because every virtual address requires two RAM accesses.

A typical register in TLB contains the three components

$$(i,A,B)$$

where

- i is the page number (the tag in the TLB)
- A is the access code from the page table
- B is the frame number from the page table

Before accessing the page table, the MMU checks the TLB for an entry tagged as page i. If it finds such an entry, it immediately retrieves the access code A and the frame number B without having to look them up in the page table. If it does not find an entry for i, it proceeds with the normal page table lookup as described earlier and also creates a new TLB entry reflecting that access. The new entry replaces the least recently used (LRU) TLB entry.

Experience shows that with a relatively small TLB—from 32 to 128 registers—the MMU can achieve a sufficiently high hit rate in the TLB that the average slowdown for page table lookups is 1%–3% (Hennessey and Patterson 1990).

54.6.4 Element 4: Managing the RAM Contents

The operating system maintains a complete copy of a program's AS on the disk. The operating system also specifies a RAM allocation as a maximum number of pages from the AS. Thus, RAM contents are a subset of disk contents.

A page fault is an exceptional condition generated by the MMU when it encounters a missing page ($P = 0$). The operating system interrupts the running program and instead runs a special routine called PFH, which follows these steps:

1. Locates the needed page in the secondary memory
2. Uses a replacement policy to select a frame of main memory to put that page in
3. Empties that frame
4. Copies the needed page into that frame
5. Updates the page table entry to reflect these changes
6. Restarts the interrupted program, this time, allowing it to complete its reference

The replacement policy (step 2) has a significant effect on performance of a virtual memory system. Replacement policies generally try to predict which pages are most likely to be reused in the immediate future and protect them from replacement. Because recently used pages tend to be the most likely to be reused soon, the replacement policy identifies the pages with $U = 0$ as candidates for replacement. Among those, it favors one with $M = 0$ because an unmodified page does not need to be copied back to disk.

While no replacement policy can give a perfect prediction of the future, many years of experience and experiment have led to the consensus that the LRU policy is best when RAM allocations are fixed, and the WS policy is best when RAM allocations can vary (Denning 1980, McMenamin 2011).

54.6.5 Summary

Virtual memory makes address translation transparent to the programmer. Since the operating system maintains the contents of the map, it can alter the correspondence between pages and frames dynamically. A program can now be executed on a wide range of system configurations, from small to large main memories, without recompiling it.

54.7 Cache Memories

The caching principle used in the TLB was first proposed by Wilkes (1965) as a direct hardware method for speeding up memory accesses for pages already loaded in RAM (Figure 54.3). The cache contains a set of "slots," each the size of a page. A tag associated with a slot indicates which frame of RAM is copied in that slot. When the MMU of the processor generates location address (b,x)—meaning frame b line x—the cache hardware searches all of the tag registers in parallel for a match on b. If there is a match, it addresses byte x of that slot. If not, the hardware copies frame b into a slot, sets the slot's tag to b, and then addresses byte x of that slot (see Hennessey and Patterson 1990).

Note how the virtual memory mapping principle has been applied to components of the virtual memory system itself. The TLB is a cache of recent (page and frame) paths; a match bypasses the page table lookup. The slot-cache holds copies of recently used RAM frames; a match bypasses the RAM access. The TLB and cache access times are much faster than the corresponding RAM times. The virtual system will demonstrate significant accelerations with these caches, even when they are a small fraction of their maximum potential size. The costs of cache are small compared to full memory size.

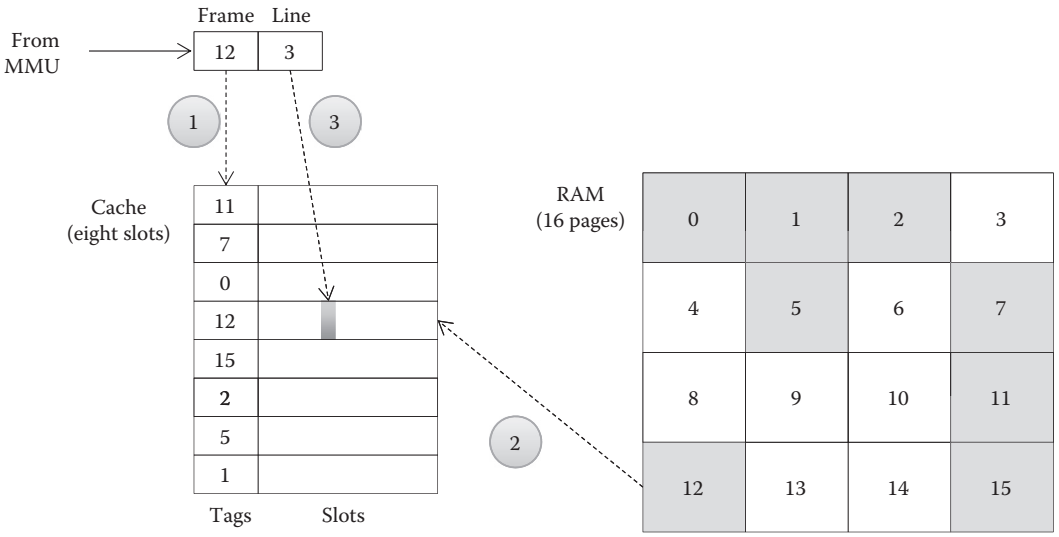

FIGURE 54.3 In a cache memory, the slots hold copies of RAM frames, and the tags indicate which frame is in a slot. The cache hardware searches the tag registers in parallel for a match on the addressed page (1). If no match is found, a page is loaded into the least recently used cache slot (2). Then, the line number selects a byte of that page (3). The search can be accelerated by dividing the tags into 2^m sets, using the m low-order bits of the frame number to select the set, and restricting the parallel search to that set (the figure is drawn for $m = 0$.) This partitions the frames equally among the sets and thereby limits the number of slots into which a given frame may be placed. In the worst case, when 2^m equals the number of cache slots, the set size is 1, and each block can be loaded into one slot only.

54.8 Multiprogramming

Multiprogramming is a mode of operation in which the RAM is shared simultaneously by multiple jobs. The set of pages a job has in RAM is called its resident set (or partition). The resident sets of all loaded jobs partition the RAM into disjoint subsets of frames.

Multiprogramming was originally introduced in the operating systems of the 1960s as a means to improve the utilization of a CPU: When a job stops to wait for an event, such as reading a file from a disk, the CPU can switch to another job ready to go. In the 1990s, multiprogramming was extended to allow each user to start and load multiple jobs in RAM. Users could switch among active programs such as word processor, spreadsheet, mail, and browser. Because virtual memory confines each process to its assigned AS, it provides an elegant and flexible way of partitioning a multiprogrammed memory.

Multiprogramming can be done with fixed or variable partitions. Fixed partitions are easier to implement but variable partitions offer much better performance. With variable partitions, the operating system can adjust the size of the partition so that the rate of page faults stays within acceptable limits. The operating system can transfer space from processes with small memory needs to processes with large memory needs. Variable partitions often improve over fixed even when the variation is random (Denning 1980). System throughput will be near optimal when the virtual memory guarantees each active process just enough space to hold its WS (Denning 1980).

54.9 Performance and the Principle of Locality

We can discuss performance of virtual memory systems in terms of decisions about loading and replacing pages. Loading refers to the actions that bring a page into main memory. Most systems load pages only on demand because they have no reliable method of predicting future page use. Because the time

to load a page is usually 6 orders of magnitude higher than the time to access a loaded page, a few bad predictive guesses are more expensive than simply waiting for page faults. However, in some cases, enough is known about future references that pages can reliably be loaded ahead of time; for example, an application doing a long, linear scan of a database.

Replacement refers to the actions of removing pages from RAM. Each removal has a cost because most pages will be recalled into RAM by future page references. Because this is an ongoing process—removal followed by recall—replacement is a major determinant of virtual memory performance.

The main metrics of system performance are *throughput* and *response time*. Throughput is measured as jobs or transactions completed per second and response time as the average number of seconds to complete a job or transaction after submission.

The main metric of memory usage is *space-time*, which is the total number of page-seconds accumulated by a job while it holds RAM. Space-time is a like "rent" paid by a job for memory usage. There is a relation between throughput and space-time. It is called the space-time law and works as follows. If C jobs complete in T seconds, the throughput is $X = C/T$. When the total memory is M bytes, the total space-time available in the system is MT, and therefore the space-time per job is $Y = MT/C = M/X$. Thus, $M = XY$. This invariant relation is fundamental: it says that minimizing space-time is the same as maximizing throughput for a given amount of main memory. For this reason, the ideal of a replacement policy is minimizing the space-time of each job.

When a job holds RAM, it circulates among servers of the system for pieces of service it needs to complete. These servers include disks, directories, network requests, printing devices, and more. Each service delay accumulates some space-time toward the job's total. The delays caused by page faults are only a subset of all the delays contributing to space-time. Still, reducing the number of page faults will reduce a job's total space-time.

The parameters of system usage outside of virtual memory are normally unaffected by paging within the virtual memory. Therefore, the minimum space-time occurs when the page faults are minimum. The ideal policy—let us call it MIN—replaces the page that will not be used again for the longest time (Belady 1966). This policy causes the intervals between page faults to be as long as possible, which minimizes total number of page faults.

MIN requires advance knowledge of the future. Because such knowledge is not usually available, replacement policies use past observations to predict future references. The predictions are necessarily imperfect.

The LRU replacement policy predicts that time until next reference to a page is same as time since last reference. Although not as good as MIN, LRU has been found to be quite robust over a range of programs, typically doing as well or better than other common policies, such as first-in-first-out (Belady 1966). LRU is often used in caches. A simple memory scan approximates LRU well: a pointer cycles through all the pages in the job's resident set, skipping over those with usage bit $U = 1$ (and resetting to $U = 0$), until it finds an unused page.

If we remove the constraints that memory size is fixed and that replacements occur only at page-fault times, we can do better than MIN. The ideal variable-space policy—let us call it VMIN—operates as follows (Prieve and Fabry 1974). After each page reference, VMIN looks ahead to the moment of the next reference to that page: if the time to that reference exceeds a threshold T, VMIN immediately replaces the page; otherwise, it retains the page in memory until that next reference. As T gets larger, VMIN retains more pages and generates fewer page faults. Thus, the parameter T trades off the amount of memory used against the amount of paging generated. The reason no other policy can do as well is that VMIN minimizes space-time at every reference: VMIN retains a page only if the cost of recovering it by page-fault at the next reference exceeds the cost of retaining it.

The WS model gives a good approximation to VMIN (Denning 1968, 1970, 1980). A process's *WS* is defined as the set of pages referenced in a window of size T references looking backwards from the current time. The WS replacement policy guarantees that every page of the WS is in RAM.

Under WS replacement, a page reference causes a page fault only if the time since prior reference to that page exceeds *T*. In other words, WS replacement generates exactly the same faults as VMIN. The difference between WS and VMIN is due solely to the extra space-time generated by WS for retaining pages for *T* seconds when the time until next reference exceeds *T*. The space-time "overshoot" is worst during a change of locality: VMIN unloads the pages of the current locality set prior to the change to the new locality set, while WS keeps the former locality set in memory for a while after the change. Many researchers looked at ways to reduce this overshoot, but none gave much benefit. The WS policy is about as close to optimal as can be found among non-lookahead replacement policies.

The WS policy works is especially well suited for multiprogramming. All jobs use a common window size *T*. The scheduler admits waiting processes to the RAM, one at a time, until the RAM space is filled with WSs. The global window size *T* can be adjusted empirically until it maximizes system throughput. System throughput may improve further by using a customized window size for each running process—but the space-time improvement is at most 5%–10% (Denning 1980).

Working-set memory management became popular because it prevented a system instability called *thrashing* (see Figure 54.4). Many early virtual memory systems attempted to extend the LRU policy to the entire memory by keeping track of the reference times of all pages in RAM. Extended this way, LRU is subject to thrashing because the normal cycles of the scheduler cause a process's pages to look old by before it gets its next time slice. Extended LRU does not have a built-in load limit like the WS policy. Thrashing can be avoided by limiting the multiprogramming level either by a fixed limit or by a working-set policy. The working-set policy generally leads to more stable performance with higher throughput.

These replacement policies all do well because of the *principle of locality*. This principle says that a computation tends to reference the same pages in the immediate future as it referenced in the immediate past.

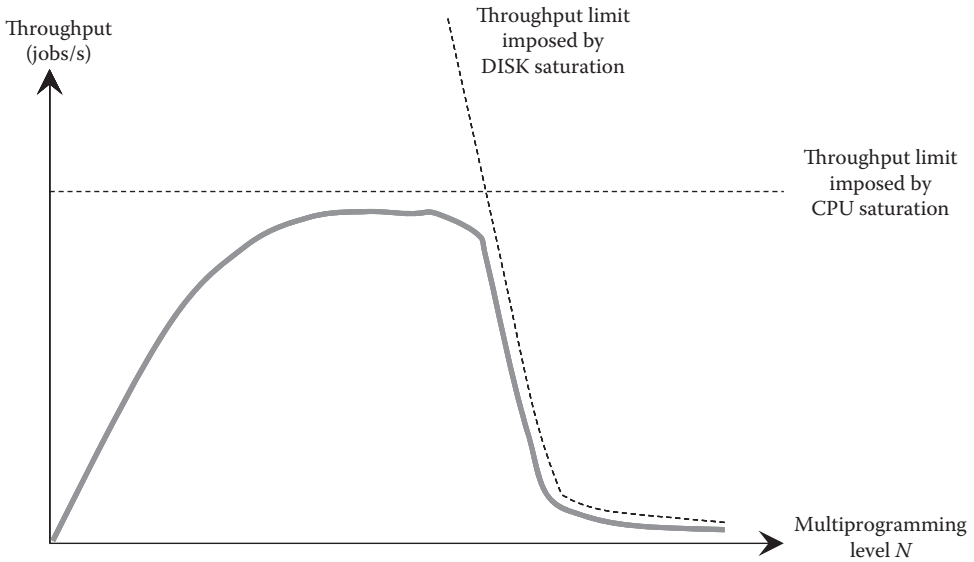

FIGURE 54.4 The system throughput is depicted as a function of the multiprogramming level *N*, which is the number of active programs in RAM. When *N* is too large, programs have too little space for their working sets. This drives the paging rates up, turns the disk into the bottleneck, and dramatically slows down the system. The slowdown is called thrashing. The WS policy is ideal; it holds *N* near the throughput peak but does not allow it to fall off the thrashing cliff.

In fact, computations tend to reference from the same subset of pages over extended intervals. The memory usage of a computation can be represented as a sequence

$$(L_1,P_1), (L_2,P_2),\ldots, (L_k,P_k),\ldots$$

where

L_i is a "locality set"

P_k is the length of time the computation uses only that set (P_k is called the phase time)

Abrupt changes in locality sets at phase transitions are the norm. This principle accounts for the success of WS policies: the backward looking window T is usually much smaller than the phase times, and thus the WS sees the current locality set most of the time. In this way, WS reveals the pages that need to stay in RAM in the immediate future (Denning 1980).

Over the years, it became clear that locality is a deep principle of computing. It accounts for the universal success of caches in every form, whether as part of a virtual memory, a Web browser, or an edge server in the Internet. By 1980, we knew that locality is a product of the way the human mind works—notably its tendencies to pay attention to the same things for a while and to divide big problems into smaller pieces for separate solution (Denning 1980).

In 2010, Moshe Vardi cited some new thinking on the question, "What is an algorithm?" (Vardi 2012). He cites a work of Yuri Gurevich, who defines an algorithm as a description of an abstract state machine, where states can be any data structure, and each operation can cause only a bounded change of state. By limiting the effect of an operation to a local set, this definition pulls the principle of locality into a fundamental definition of computation. Without locality, there is no computation. Locality has long been accepted as a fundamental principle of memory behavior, but it has never been so clearly linked to effective computation. By definition, every computation exhibits locality; some form of backward window can observe the pages of the current locality set and exempt them from replacement.

54.10 Object-Oriented Virtual Memory

Many computing environments offer abstractions and functions that require virtual addressing, but which are not easily accommodated by paging. These include

- Program objects—arrays, procedures, structures, and objects
- System objects—processes, message buffers, files, directories, and streams
- Concurrent processes (threads) with varying permissions sharing the same AS
- Modular programs
- Very large ASs containing many objects shared among many users

The designers of early virtual memories anticipated these uses with segmented and capability-based virtual memories (Dennis 1965, Dennis and Van Horn 1966, Fabry 1974). The resulting storage systems are called object-oriented virtual memories. The Internet, an example of the last bullet, was not anticipated in the 1960s, but the addressing principles discovered then contributed to solutions of its addressing problems.

The earliest form of object-oriented virtual memory was the segmented AS. It appeared as a collection of named blocks (segments) of various sizes. Each segment was a container for a program object. In the Burroughs B5000 and later series, for example, the Algol compiler created program segments containing procedures and data segments containing array rows (Organick 1973). The compiler generated virtual addresses of the form (i,x), meaning segment i, line x. The size of each segment was explicitly recorded in the mapping table, so that the mapper could reject out-of-bounds addresses x.

Multics went further than the Burroughs Algol compiler. It let the programmer define the segments. In Multics PL/I, operands had symbolic two-part names *S.X*; the operating system used a

linkage fault to invoke a routine that mapped a symbolic name to its corresponding virtual segment number *s* and line number *x* (Organick 1972).

54.10.1 Two-Level Mapping

In recognition that paging led to much simpler virtual memory structures, Multics combined segmentation with paging. It allowed each segment to be divided into pages. The offset into a segment was subdivided into a page and line number. It used a *two-level map* as follows to resolve an address (*s*,*x*):

1. The segment number *s* selects a *segment table* entry, which points to a page table for the segment. There is one segment table for each AS.
2. The offset *x* decomposes into a page number and line number, which are mapped through the page table as before. There is one page table for each segment.

The two-level mapping scheme makes it easy to share segments. Users simply share the segment's page table. Users sharing a segment are likely to get different local segment numbers for their segment tables; but all those segment table entries point to the *same* page table. When it moves a page, the operating system records the new location in the page table. Instantly, all users sharing the page are mapped correctly to the new location. There is no need for the operating system to locate all the users and update their segment mapping tables.

Inspired by Dennis and Van Horn (1966), Robert Fabry formalized the two-level mapping scheme to allow any number of users to share any number of objects in large systems (Fabry 1974). Fabry's addressing principle is summarized in Figure 54.5. The key idea is that the first level of mapping takes a segment number *i* to a system handle *h*; the second level takes a handle to a descriptor of the storage

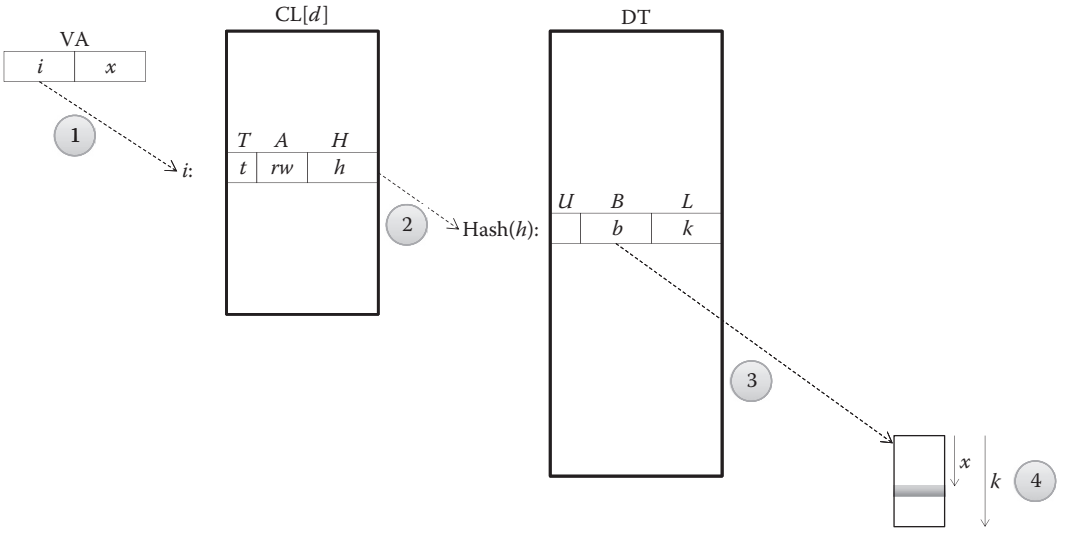

FIGURE 54.5 The two-level mapping scheme replaces a page table with two tables. The capability list CL[*d*] explicitly lists objects accessible in domain *d*. Each entry in CL consists of a tag, an access code, and a handle. The tag identifies the type of object (such as file or directory), the access code the type of access allows (such as read or write), and the handle is the unique global system name for the object. The descriptor table DT is a list of descriptors for every object in the system. This example shows a standard segment with base *b* and length *k*. A handle *h* is passed through a hash function to localize it within the DT for fast mapping. Although the full addressing sequence 1-2-3-4 requires several RAM accesses, the key elements of the path, (T,A,B,L), can be cached in the TLB for fast future lookup.

space holding the object (Fabry used the terms "capability," where we now say "handle"). All information about the location of an object is recorded in the descriptor; any changes instantly affect anyone trying to use the object.

54.10.2 Protection of Handles and Objects

In the two level mapping system, a program that holds a handle can access the object regardless of whether it has permission to do so. In other words, having a handle confers permission to access the object.

Because the mere fact of holding a handle was like having super-user privilege to the designated object, the designers of early capability addressing systems thought that handles had to be heavily protected by the hardware from alteration. They saw no other way to prevent someone from manufacturing a handle pointing to someone else's files or to prevent an erroneous program from corrupting a handle. Some early commercial systems such as Plessey 250 or IBM 360/38 used hardware protection for capabilities. The Cambridge CAP system, a research project, also used hardware protection but concluded that hardware protection led to unwieldy complexity in the operating system (Wilkes and Needham 1979). Eventually, operating system structures were discovered that hid handles from users, thus prevent tampering or alteration, without special hardware protection. For example, file and directory handles are stored in directories, where the user can invoke them by giving their symbolic names; but the handles themselves never enter the user's virtual AS.

One of the fundamental requirements of an operating system is that users cannot interfere with each other. That means, by default, they cannot see each other's ASs. The virtual memory system plays an integral role in meeting this requirement. The images of ASs are always in disjoint regions of main memory. This property of virtual memory is called *logical partitioning*.

With simple virtual memory (Figure 54.2), a processor can address only the pages listed in its page table. With extended virtual memory (Figure 54.5), a processor can address only the objects listed in its capability list. In either case, unlisted objects are inaccessible. In effect, the operating system walls each process off, giving it no chance to read or write the private objects of any other process.

The mapping mechanisms further restrict individual accesses to those permitted by the access codes in the mapping tables. Thus, a read-only page or segment cannot be modified, or a read-only directory cannot be searched.

54.10.3 Protection of Procedures

One of the biggest vulnerabilities of systems is that procedures can be called incorrectly. The normal call puts the CPU instruction pointer to a designated "entry point" of the procedure. But a buggy or malicious calling program can transfer to some other location in the procedure, causing erroneous operation or bypassing security checks at the procedure's entry. The designers of early capability systems therefore provided *protected entry*, a method to guarantee that a procedure call could only start the CPU at the authorized entry point.

It is straightforward to structure a capability system to provide a protected entry operation (Dennis and Van Horn 1966, Wulf et al. 1974). An instruction "ENTER *i*" works only if object *i* (in the caller's domain *d1*) is an enter capability. An enter capability points to the capability list of a new domain *d2*. Object "0" in every domain's capability is the domain's entry procedure. The effect of the enter instruction is to call the entry procedure of the target domain, simultaneously making its capability list the current capability list used by the CPU. The caller's domain and instruction pointer are saved on the stack and restored when the called procedure returns (Figure 54.6).

These structures have important benefits for system fault tolerance. Should a process run amok, it can damage only its own objects: a program crash does not imply a system crash. An untrusted program can be encapsulated in a domain whose capability list contains only the objects it needs to execute;

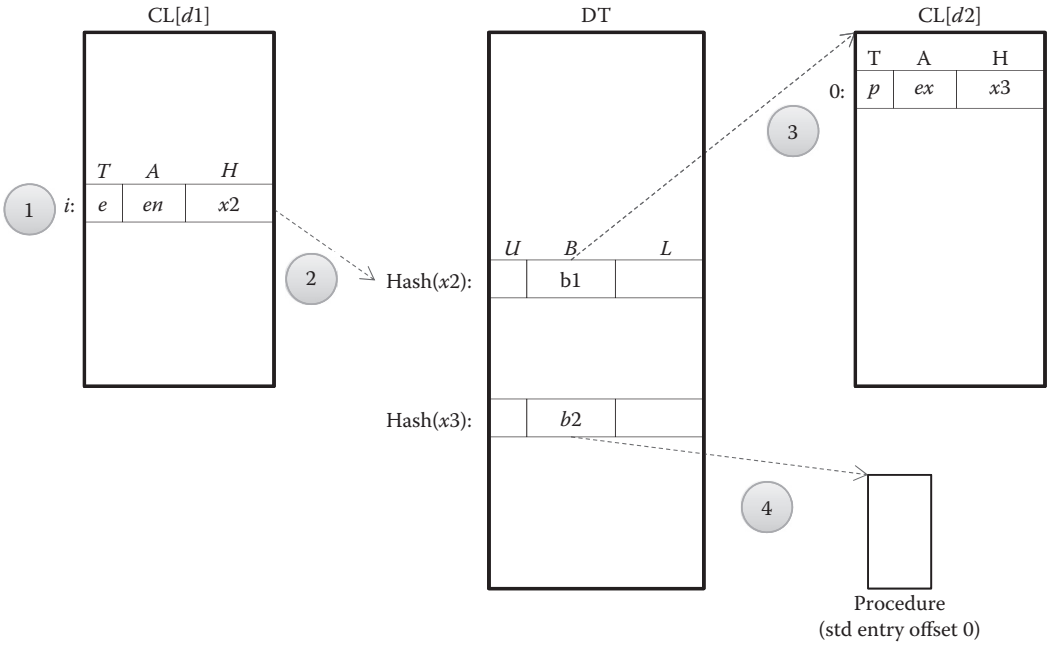

FIGURE 54.6 Protected entry into a protection domain is easily implemented in a capability system. The user in domain *d1* has a domain enter capability (tag *e*) with enter permission (*en*) that points to another domain *d2*. By convention the first entry in every domain (object "0") is a procedure (tag *p*) with execute permission (*ex*) to be automatically invoked when the domain is called. The instruction "ENTER *i*" causes a procedure call on object "0" in the new domain, and the capability list of the new domain becomes the current capability list.

any erroneous or malicious actions cannot access or damage anything outside this encapsulated domain. Systems that support this structure have found it effective against Trojan horses and other malware.

These benefits are so important that many systems use virtual memory even if they have sufficient RAM to eliminate overlaying.

54.11 Distributed Shared Memory

Starting in the mid-1980s, Sequent, Intel, Thinking Machines, N-Cube, and then later IBM, Cray, Kendall Square, and a few others introduced commercial multicomputers. These machines allowed for a large number of computers, sharing a high-speed interconnection network, to work concurrently on a single problem. Multicomputers soon began to challenge the traditional supercomputer by offering the same aggregate processing speed at a much lower cost (Denning and Tichy 1990).

In the mid 1990s, new operating systems like Beowulf provided the means to distribute tasks among ordinary computers registered as a cluster on a network. A large problem can be divided into numerous small problems for the cluster to solve, and their respective answers combined back into an answer to the original problem. Clusters provide supercomputer-grade processing power by mobilizing a large number of ordinary computers.

These new architectures introduced a host of new programming problems having to do with synchronizing the processes on the different computers and exchanging data among them. Because these systems offered no common AS among all of the component computers, their programmers could share data only by copying it between machines. That causes high message overhead.

Much of the overhead can be eliminated if the multicomputers all share an AS. Now data can be passed from one computer to another simply by passing the virtual address. The cost of moving a single pointer is much less than copying the entire object from one AS to another.

A virtual memory organized in this way is often referred to as *distributed shared memory* because the shared AS is composed from the RAMs of the individual computers (Tannenbaum 1995). Examples of such systems today include OpenSSI and MOSIX.

54.12 World Wide Web: A Global Name Space

The WWW allows an author to embed, anywhere in a document, a link to any file in the Internet. The link contains a URL, which is the symbolic name of a file. By clicking the mouse on a URL string, the user triggers the operating system to map the URL to the file and then bring a copy of that file from the remote server to the local workstation for viewing.

A URL is a string of the form *hostname/pathname*, where hostname identifies an Internet host, and pathname the name of a file in that host's directory system. Since hostnames and pathnames are unique, the URL names a single object anywhere in the Internet. Tim Berners-Lee invented URL's for the WWW. They define a scheme for naming objects that can be shared anywhere in the Web. The HTTP maps a hostname to its IP address (using the DNS) and sends a read-request message containing the pathname to that machine.

The URL system has a drawback that often causes problems: the URL names are not persistent. For example, if an owner deletes a file, any URL pointing to it will, by design, no longer function. However, if that owner creates a different file of the same name later, anyone holding the old URL will now get a copy of the new file. This means that a URL cannot be guaranteed to point to the same file as it did when a user first acquired it. Many service providers, notably publishers, want a naming system that uniquely identifies each published object and guarantees that the unique identifier will never point to another object.

The handle system (handle.net) was invented in 1995 by Robert Kahn to provide persistent names for objects. The most well known use of the handle system is with *digital object identifiers* (DOIs) that are of the form *A/B*, where *A* is a numerical string assigned by a DOI registrar to a publisher, and *B* is a unique numerical string assigned by the publisher (Denning and Kahn 2010). For example, the author's paper about WSs (Denning 1968) has DOI 10.1145/363095.363141, where "10.1145" is ACM's unique identifier assigned by the DOI Foundation, and "363095.363141" is a number chosen by ACM to distinguish that paper from every other that ACM has ever published, or ever will. ACM provides a server that translates DOI's to the current URLs of the papers. Thus, invoking the Web address "http://doi.acm.org/10.1145/363095.363141" will get a copy of the paper. ACM can change the URLs, but no matter where ACM actually stores the paper the DOI will always find it.

The handle system (including DOI's) is functionally similar to the handles described earlier in the extended, object-oriented virtual memory. The difference is that no attempt is made to hide the handles from users.

54.13 Conclusion

Virtual memory systems are used to meet one or more of these needs:

- *Automatic storage allocation*: Solving the overlay problem that arises when a program exceeds the size of the computational store available to it. Furthermore, it includes the problems of relocation and partitioning arising with multiprogramming.
- *Logical partitioning*: Each process is given access to a limited set of objects, its protection domain. The operating system enforces the rights granted in a protection domain by restricting references to the memory regions in which objects are stored. Processes cannot access objects beyond those listed in its local mapping tables.
- *Access control*: Within a protection domain, the operating system enforces further restrictions by permitting only the types of reference stated for each object (for example, read, write, or apply a function). These constraints are easily checked by the hardware in parallel with the main computation.

- *Encapsulation and protected entry*: Any untrusted software can be encapsulated in its own protection domain. Any unauthorized action will be automatically blocked by the access controls. It will be impossible to attempt access to any object outside the protection domain.
- *Modular programs*: Programmers should be able to combine separately compiled, reusable, and shareable components into programs without prior arrangements about anything other than interfaces and without having to link the components manually into an AS.
- *Object-oriented programs*: Programmers should be able to define managers of classes of objects and be assured that only the manager can access and modify the internal structures of objects (Myers 1982). Objects should be freely shareable and reusable throughout a distributed system (Chase et al. 1994, Tannenbaum 1995). (This is an extension of the modular programming objective.)
- *Data-centered programming*: Computations in the WWW tend to consist of many processes navigating through a space of shared, mobile objects. Objects are be bound to a computation on demand.
- *Parallel computations on multicomputers*: Scalable algorithms that can be configured at run time for any number of processors are essential to mastery of highly parallel computations on multicomputers. Virtual memory joins the memories of the component machines into a single AS and reduces communication costs by eliminating some of the copying inherent in message passing.

Virtual memory, once the subject of intense controversy, it is now so ordinary that few people think much about it. Its original purpose—automating solutions to the overlay problem—is today less important that it's extended purposes for sharing and access objects in large name spaces. The success of virtual memory is tied directly to the principle of locality, which is a fundamental principle of computation itself. Virtual memory is an enduring technology.

Key Terms

Access control: A means of allowing access to an object based on the type of access sought, the accessor's privileges, and the owner's wishes.

Address fault: An error that halts the mapper when it cannot locate a referenced object in main memory; it invokes an interrupt, whose handler corrects the condition by loading the missing object.

Address map: A table that associates an object (or page) number with the main memory locations containing the object.

Address space: The set of all addresses that a processor can issue while processing a program.

Bounds fault: An error that halts the mapper when it detects that the offset requested into an object exceeds the object's size; it invokes an interrupt that terminates the program.

Capability: A systemwide unique identifier for an object; the bits of a capability are protected from alteration.

Context-switch: An operation that switches the CPU from one process to another by saving all of the CPU registers for the first and replacing them with the CPU registers for the second.

CPU: Central processing unit or processor.

Data-centered view: A view of computing that emphasizes navigation of many concurrent processes within a large space of objects.

Handle: A systemwide unique identifier for an object, like a capability without the system guarantee of integrity.

Location: A memory register with its own address.

Logical partitioning: A property of virtual memory, whereby the address spaces of different jobs are mapped into disjoint regions of memory.

Main memory: The highest level of the memory hierarchy; all CPU memory references are directed to main memory; CPU can access objects only when they are loaded in main memory.

Memory hierarchy: A system of memory devices of different speeds and capacities; allows for trading off between capacity and speed, and between volatility and persistence.

Memory space: The set of all hardware addresses of memory locations in RAM available to a given address space.

Modular programming: Programs are divided into parts that can be shared, reused, and recompiled without affecting other parts of the system as long as the interfaces to modules are unchanged.

Object-oriented addressing: A form of virtual addressing in which object numbers are mapped to memory regions and internal object references are mapped to offsets within an object's memory region.

Object-oriented programming: A form of programming in which data are organized into classes of objects, each with a specific set of functions that can be applied to the objects.

Page: a fixed size unit of storage and transfer in a memory hierarchy.

Page frame: A contiguous block of memory locations used to hold a page.

Paging: A method of virtual memory in which address space and memory space are paged.

Partition: A division of memory space into disjoint subsets of pages for each address space.

PC: Personal computer.

Permissions: Access rights granted by an object's owner and represented as bits in the object's access code.

Process: An abstraction of the execution of a program, usually represented as the sequence of values of its CPU state as the program traces through its instruction sequence.

Processor-centered view: A view of computing that emphasizes the work of a processor.

Protection fault: An error condition detected by the address mapper when the type of request is not permitted by the object's access code.

Response time: The time from when a command is submitted to a computer until the computer responds with the result.

RAM: Random access memory.

RISC: Reduced instruction set computer (e.g., PowerPC, Sun SPARC, DEC Alpha, and MIPS).

Secondary memory: Lower, large capacity level of a memory hierarchy, usually a set of disks.

Segmentation: An approach to virtual memory when the mapped objects were variable-size memory regions rather than fixed-size pages.

Slave memory: A hardware cache attached to a CPU, enabling fast access to recently used pages and lowering traffic on the CPU-to-main-memory bus.

Space-time: The accumulated product of the amount of memory and the amount of time used by a process.

Thrashing: A condition of performance collapse in a multiprogramming system when the number of active programs gets too large.

Throughput: The number of jobs (or transactions) per second completed by a computer system.

TLB: Translation lookaside buffer, a cache that holds the most recently followed address paths in the mapper.

Two-level map: A two-tiered mapping scheme; the upper tier converts local object numbers into system unique handles, and the second tier converts handles to the memory regions containing the objects essential for sharing.

URL: Uniform resource locator (in the WWW).

Working set: The smallest subset of a program's pages that must be loaded into main memory to assure acceptable processing efficiency; changes dynamically.

Working-set (WS) policy: A memory allocation strategy that regulates the amount of main memory allocated to a process, so that the process is guaranteed a minimum level of processing efficiency.

World Wide Web (WWW): A set of servers in the Internet and an access protocol that permits fetching documents by following hypertext links on demand.

References

Belady, L. 1966. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal* 5(2):78–101.

Berners-Lee, T. 2000. *Weaving the Web*. Harper Business, New York.

Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D. 1994. Sharing and protection in a single-address-space operating system. *ACM TOCS* 12(4):271–307.

Denning, P. J. 1968. Thrashing: Its causes and prevention. *Proc. AFIPS FJCC* 33:915–922.

Denning, P. J. 1970. Virtual memory. *Computing Survey* 2(3):153–189.

Denning, P. J. 1976. Fault tolerant operating systems. *Computing Survey* 8(3):359–390.

Denning, P. J. 1980. Working sets past and present. *IEEE Transactions on Software Engineering* SE-6(1):64–84.

Denning, P. J. and Kahn, R. E. 2010. The long quest for universal information access. *ACM Communications* 53(12):34–36.

Denning, P. J. and Tichy, W. F. 1990. Highly parallel computation. *Science* 250:1217–1222.

Dennis, J. B. 1965. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM* 12(4):589–602.

Dennis, J. B. and Van Horn, E. 1966. Programming semantics for multiprogrammed computations. *ACM Communications* 9(3):143–155.

Fabry, R. S. 1974. Capability-based addressing. *ACM Communications* 17(7):403–412.

Fotheringham, J. 1961. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *ACM Communications* 4(10):435–436.

Hennessey, J. and Patterson, D. 1990. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, San Mateo, CA.

Kahn, R. and Wilensky, R. 1995. *A Framework for Distributed Object Services*. Technical Note 95–101, Corporation for National Research Initiatives, Reston, VA. See also www.handle.net.

Kilburn, T., Edwards, D. B. G., Lanigan, M. J., and Sumner, F. H. 1962. One-level storage system. *IRE Transactions* EC-11(2):223–235.

McMenamin, A. 2011. Applying Working Set Heuristics to the Linux Kernel, MSc Project Report, Birbeck College, University of London. Available at: http://cartesianproduct.files.wordpress.com/2011/12/main.pdf, accessed October 6, 2013.

Myers, G. J. 1982. *Advances in Computer Architecture*, 2nd edn. Wiley, New York.

Organick, E. I. 1972. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA.

Organick, E. I. 1973. *Computer System Organization: The B5700/B6700 System*. Academic Press, New York.

Prieve, B. and Fabry, R. 1974. VMIN: An optimal variable space page replacement algorithm. *ACM Communications* 19(5):295–297.

Sayre, D. 1969. Is automatic folding of programs efficient enough to displace manual? *ACM Communications* 12(12):656–660.

Tannenbaum, A. S. 1995. *Distributed Operating Systems*. Prentice-Hall, Englewood Cliffs, NJ.

Vardi, M. 2012. What is an algorithm? *ACM Communications* 55(3):5.

Wilkes, M.V. 1965. Slave memories and dynamic storage allocation. *IEEE Trans. EC* 14(April):270–271.

Wilkes, M. V. 1975. *Time Sharing Computer Systems*, 3rd edn. Elsevier, Amsterdam, North Holland.

Wilkes, M. V. and Needham, R. 1979. *The Cambridge CAP Computer and Its Operating System*. Elsevier, Amsterdam, North Holland.

Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. 1974. HYDRA: The kernel of a multiprocessor operating system. *ACM Communications* 17(6):337–345.