3

MEMORY MANAGEMENT

Main memory (RAM) is an important resource that must be very carefully managed. While the average home computer nowadays has 10,000 times more memory than the IBM 7094, the largest computer in the world in the early 1960s, programs are getting bigger faster than memories. To paraphrase Parkinson's Law, "Programs expand to fill the memory available to hold them." In this chapter we will study how operating systems create abstractions from memory and how they manage them.

What every programmer would like is a private, infinitely large, infinitely fast memory that is also nonvolatile, that is, does not lose its contents when the electric power is switched off. While we are at it, why not make it inexpensive, too? Unfortunately, technology does not provide such memories at present. Maybe you will discover how to do it.

What is the second choice? Over the years, people discovered the concept of a **memory hierarchy**, in which computers have a few megabytes of very fast, expensive, volatile cache memory, a few gigabytes of medium-speed, medium-priced, volatile main memory, and a few terabytes of slow, cheap, nonvolatile magnetic or solid-state disk storage, not to mention removable storage, such as DVDs and USB sticks. It is the job of the operating system to abstract this hierarchy into a useful model and then manage the abstraction.

The part of the operating system that manages (part of) the memory hierarchy is called the **memory manager**. Its job is to efficiently manage memory: keep track of which parts of memory are in use, allocate memory to processes when they need it, and deallocate it when they are done.

In this chapter we will investigate several different memory management models, ranging from very simple to highly sophisticated. Since managing the lowest level of cache memory is normally done by the hardware, the focus of this chapter will be on the programmer's model of main memory and how it can be managed. The abstractions for, and the management of, permanent storage—the disk—are the subject of the next chapter. We will first look at the simplest possible schemes and then gradually progress to more and more elaborate ones.

3.1 NO MEMORY ABSTRACTION

The simplest memory abstraction is to have no abstraction at all. Early mainframe computers (before 1960), early minicomputers (before 1970), and early personal computers (before 1980) had no memory abstraction. Every program simply saw the physical memory. When a program executed an instruction like

```
MOV REGISTER1,1000
```

the computer just moved the contents of physical memory location 1000 to *REGISTER1*. Thus, the model of memory presented to the programmer was simply physical memory, a set of addresses from 0 to some maximum, each address corresponding to a cell containing some number of bits, commonly eight.

Under these conditions, it was not possible to have two running programs in memory at the same time. If the first program wrote a new value to, say, location 2000, this would erase whatever value the second program was storing there. Nothing would work and both programs would crash almost immediately.

Even with the model of memory being just physical memory, several options are possible. Three variations are shown in Fig. 3-1. The operating system may be at the bottom of memory in RAM (Random Access Memory), as shown in Fig. 3-1(a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Fig. 3-1(b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below, as shown in Fig. 3-1(c). The first model was formerly used on mainframes and minicomputers but is rarely used any more. The second model is used on some handheld computers and embedded systems. The third model was used by early personal computers (e.g., running MS-DOS), where the portion of the system in the ROM is called the **BIOS** (Basic Input Output System). Models (a) and (c) have the disadvantage that a bug in the user program can wipe out the operating system, possibly with disastrous results.

When the system is organized in this way, generally only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a user new command. When the operating system receives the command, it loads a new program into memory, overwriting the first one.

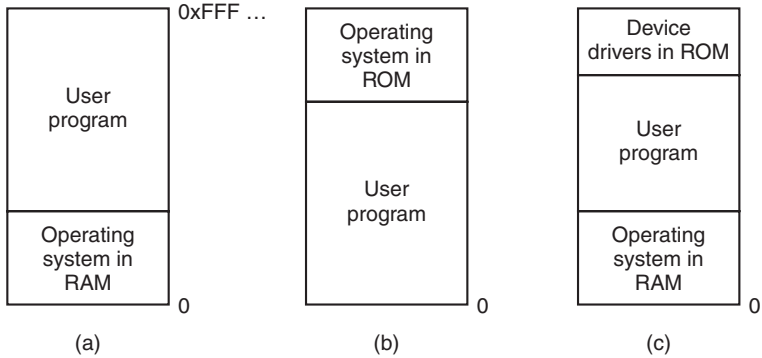

Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

One way to get some parallelism in a system with no memory abstraction is to program with multiple threads. Since all threads in a process are supposed to see the same memory image, the fact that they are forced to is not a problem. While this idea works, it is of limited use since what people often want is *unrelated* programs to be running at the same time, something the threads abstraction does not provide. Furthermore, any system that is so primitive as to provide no memory abstraction is unlikely to provide a threads abstraction.

Running Multiple Programs Without a Memory Abstraction

However, even with no memory abstraction, it is possible to run multiple programs at the same time. What the operating system has to do is save the entire contents of memory to a disk file, then bring in and run the next program. As long as there is only one program at a time in memory, there are no conflicts. This concept (swapping) will be discussed below.

With the addition of some special hardware, it is possible to run multiple programs concurrently, even without swapping. The early models of the IBM 360 solved the problem as follows. Memory was divided into 2-KB blocks and each was assigned a 4-bit protection key held in special registers inside the CPU. A machine with a 1-MB memory needed only 512 of these 4-bit registers for a total of 256 bytes of key storage. The PSW (Program Status Word) also contained a 4-bit key. The 360 hardware trapped any attempt by a running process to access memory with a protection code different from the PSW key. Since only the operating system could change the protection keys, user processes were prevented from interfering with one another and with the operating system itself.

Nevertheless, this solution had a major drawback, depicted in Fig. 3-2. Here we have two programs, each 16 KB in size, as shown in Fig. 3-2(a) and (b). The former is shaded to indicate that it has a different memory key than the latter. The

first program starts out by jumping to address 24, which contains a MOV instruction. The second program starts out by jumping to address 28, which contains a CMP instruction. The instructions that are not relevant to this discussion are not shown. When the two programs are loaded consecutively in memory starting at address 0, we have the situation of Fig. 3-2(c). For this example, we assume the operating system is in high memory and thus not shown.

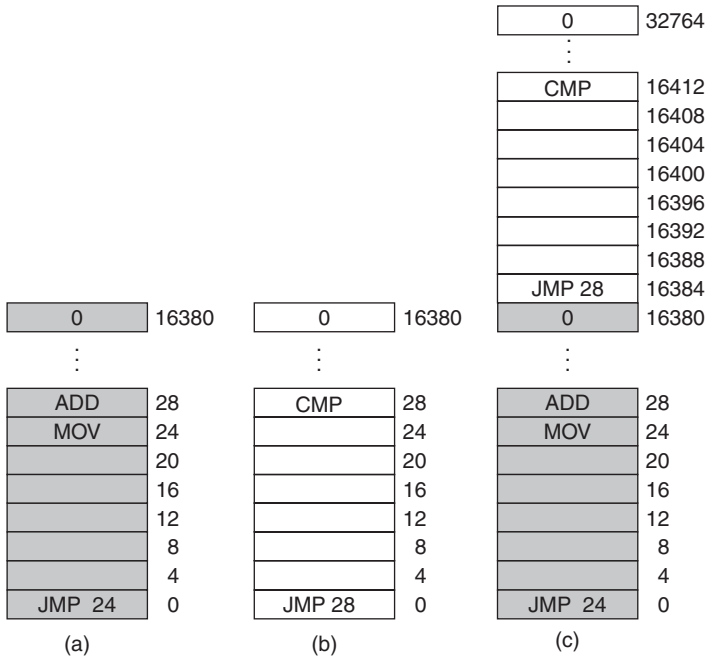

Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

After the programs are loaded, they can be run. Since they have different memory keys, neither one can damage the other. But the problem is of a different nature. When the first program starts, it executes the JMP 24 instruction, which jumps to the instruction, as expected. This program functions normally.

However, after the first program has run long enough, the operating system may decide to run the second program, which has been loaded above the first one, at address 16,384. The first instruction executed is JMP 28, which jumps to the ADD instruction in the first program, instead of the CMP instruction it is supposed to jump to. The program will most likely crash in well under 1 sec.

The core problem here is that the two programs both reference absolute physical memory. That is not what we want at all. What we want is that each program

can reference a private set of addresses local to it. We will show how this can be acomplished shortly. What the IBM 360 did as a stop-gap solution was modify the second program on the fly as it loaded it into memory using a technique known as **static relocation**. It worked like this. When a program was loaded at address 16,384, the constant 16,384 was added to every program address during the load process (so "JMP 28" became "JMP 16,412", etc.).While this mechanism works if done right, it is not a very general solution and slows down loading. Furthermore, it requires extra information in all executable programs to indicate which words contain (relocatable) addresses and which do not. After all, the "28" in Fig. 3-2(b) has to be relocated but an instruction like

```
MOV REGISTER1,28
```

which moves the number 28 to *REGISTER1* must not be relocated. The loader needs some way to tell what is an address and what is a constant.

Finally, as we pointed out in Chap. 1, history tends to repeat itself in the computer world. While direct addressing of physical memory is but a distant memory on mainframes, minicomputers, desktop computers, notebooks, and smartphones, the lack of a memory abstraction is still common in embedded and smart card systems. Devices such as radios, washing machines, and microwave ovens are all full of software (in ROM) these days, and in most cases the software addresses absolute memory. This works because all the programs are known in advance and users are not free to run their own software on their toaster.

While high-end embedded systems (such as smartphones) have elaborate operating systems, simpler ones do not. In some cases, there is an operating system, but it is just a library that is linked with the application program and provides system calls for performing I/O and other common tasks. The **e-Cos** operating system is a common example of an operating system as library.

3.2 A MEMORY ABSTRACTION: ADDRESS SPACES

All in all, exposing physical memory to processes has several major drawbacks. First, if user programs can address every byte of memory, they can easily trash the operating system, intentionally or by accident, bringing the system to a grinding halt (unless there is special hardware like the IBM 360's lock-and-key scheme). This problem exists even if only one user program (application) is running. Second, with this model, it is difficult to have multiple programs running at once (taking turns, if there is only one CPU). On personal computers, it is common to have several programs open at once (a word processor, an email program, a Web browser), one of them having the current focus, but the others being reactivated at the click of a mouse. Since this situation is difficult to achieve when there is no abstraction from physical memory, something had to be done.

3.2.1 The Notion of an Address Space

Two problems have to be solved to allow multiple applications to be in memory at the same time without interfering with each other: protection and relocation. We looked at a primitive solution to the former used on the IBM 360: label chunks of memory with a protection key and compare the key of the executing process to that of every memory word fetched. However, this approach by itself does not solve the latter problem, although it can be solved by relocating programs as they are loaded, but this is a slow and complicated solution.

A better solution is to invent a new abstraction for memory: the address space. Just as the process concept creates a kind of abstract CPU to run programs, the address space creates a kind of abstract memory for programs to live in. An **address space** is the set of addresses that a process can use to address memory. Each process has its own address space, independent of those belonging to other processes (except in some special circumstances where processes want to share their address spaces).

The concept of an address space is very general and occurs in many contexts. Consider telephone numbers. In the United States and many other countries, a local telephone number is usually a 7-digit number. The address space for telephone numbers thus runs from 0,000,000 to 9,999,999, although some numbers, such as those beginning with 000 are not used. With the growth of smartphones, modems, and fax machines, this space is becoming too small, in which case more digits have to be used. The address space for I/O ports on the x86 runs from 0 to 16383. IPv4 addresses are 32-bit numbers, so their address space runs from 0 to $2^{32} - 1$ (again, with some reserved numbers).

Address spaces do not have to be numeric. The set of *.com* Internet domains is also an address space. This address space consists of all the strings of length 2 to 63 characters that can be made using letters, numbers, and hyphens, followed by *.com*. By now you should get the idea. It is fairly simple.

Somewhat harder is how to give each program its own address space, so address 28 in one program means a different physical location than address 28 in another program. Below we will discuss a simple way that used to be common but has fallen into disuse due to the ability to put much more complicated (and better) schemes on modern CPU chips.

Base and Limit Registers

This simple solution uses a particularly simple version of **dynamic relocation**. What it does is map each process' address space onto a different part of physical memory in a simple way. The classical solution, which was used on machines ranging from the CDC 6600 (the world's first supercomputer) to the Intel 8088 (the heart of the original IBM PC), is to equip each CPU with two special hardware registers, usually called the **base** and **limit** registers. When these registers are used,

programs are loaded into consecutive memory locations wherever there is room and without relocation during loading, as shown in Fig. 3-2(c). When a process is run, the base register is loaded with the physical address where its program begins in memory and the limit register is loaded with the length of the program. In Fig. 3-2(c), the base and limit values that would be loaded into these hardware registers when the first program is run are 0 and 16,384, respectively. The values used when the second program is run are 16,384 and 32,768, respectively. If a third 16-KB program were loaded directly above the second one and run, the base and limit registers would be 32,768 and 16,384.

Every time a process references memory, either to fetch an instruction or read or write a data word, the CPU hardware automatically adds the base value to the address generated by the process before sending the address out on the memory bus. Simultaneously, it checks whether the address offered is equal to or greater than the value in the limit register, in which case a fault is generated and the access is aborted. Thus, in the case of the first instruction of the second program in Fig. 3-2(c), the process executes a

```
JMP 28
```

instruction, but the hardware treats it as though it were

```
JMP 16412
```

so it lands on the CMP instruction as expected. The settings of the base and limit registers during the execution of the second program of Fig. 3-2(c) are shown in Fig. 3-3.

Using base and limit registers is an easy way to give each process its own private address space because every memory address generated automatically has the base-register contents added to it before being sent to memory. In many implementations, the base and limit registers are protected in such a way that only the operating system can modify them. This was the case on the CDC 6600, but not on the Intel 8088, which did not even have the limit register. It did have multiple base registers, allowing program text and data, for example, to be independently relocated, but offered no protection from out-of-range memory references.

A disadvantage of relocation using base and limit registers is the need to perform an addition and a comparison on every memory reference. Comparisons can be done fast, but additions are slow due to carry-propagation time unless special addition circuits are used.

3.2.2 Swapping

If the physical memory of the computer is large enough to hold all the processes, the schemes described so far will more or less do. But in practice, the total amount of RAM needed by all the processes is often much more than can fit in memory. On a typical Windows, OS X, or Linux system, something like 50–100

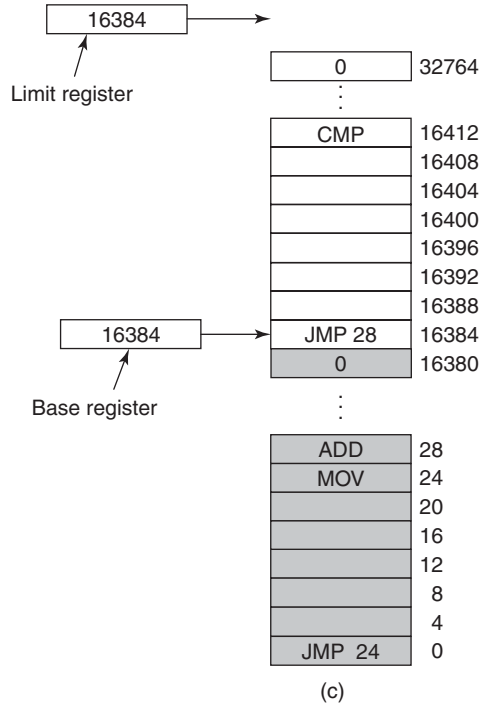

Figure 3-3. Base and limit registers can be used to give each process a separate address space.

processes or more may be started up as soon as the computer is booted. For example, when a Windows application is installed, it often issues commands so that on subsequent system boots, a process will be started that does nothing except check for updates to the application. Such a process can easily occupy 5–10 MB of memory. Other background processes check for incoming mail, incoming network connections, and many other things. And all this is before the first user program is started. Serious user application programs nowadays, like Photoshop, can easily require 500 MB just to boot and many gigabytes once they start processing data. Consequently, keeping all processes in memory all the time requires a huge amount of memory and cannot be done if there is insufficient memory.

Two general approaches to dealing with memory overload have been developed over the years. The simplest strategy, called **swapping**, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk. Idle processes are mostly stored on disk, so they do not take up any memory when they are not running (although some of them wake up periodically to do their work, then go to sleep again). The other strategy, called **virtual memory**, allows programs to run even when they are only partially in main memory. Below we will study swapping; in Sec. 3.3 we will examine virtual memory.

The operation of a swapping system is illustrated in Fig. 3-4. Initially, only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk. In Fig. 3-4(d) *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again. Since *A* is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution. For example, base and limit registers would work fine here.

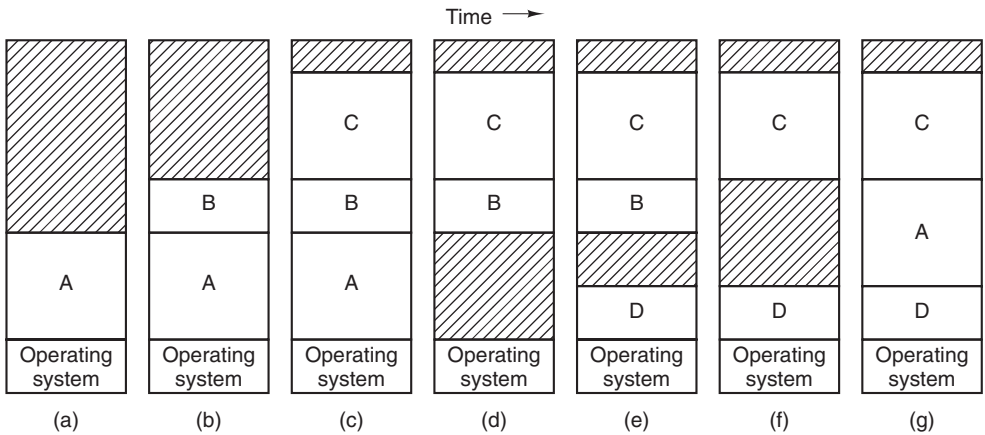

Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is known as **memory compaction**. It is usually not done because it requires a lot of CPU time. For example, on a 16-GB machine that can copy 8 bytes in 8 nsec, it would take about 16 sec to compact all of memory.

A point that is worth making concerns how much memory should be allocated for a process when it is created or swapped in. If processes are created with a fixed size that never changes, then the allocation is simple: the operating system allocates exactly what is needed, no more and no less.

If, however, processes' data segments can grow, for example, by dynamically allocating memory from a heap, as in many programming languages, a problem occurs whenever a process tries to grow. If a hole is adjacent to the process, it can be allocated and the process allowed to grow into the hole. On the other hand, if the process is adjacent to another process, the growing process will either have to be moved to a hole in memory large enough for it, or one or more processes will have to be swapped out to create a large enough hole. If a process cannot grow in memory and the swap area on the disk is full, the process will have to suspended until some space is freed up (or it can be killed).

If it is expected that most processes will grow as they run, it is probably a good idea to allocate a little extra memory whenever a process is swapped in or moved, to reduce the overhead associated with moving or swapping processes that no longer fit in their allocated memory. However, when swapping processes to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory as well. In Fig. 3-5(a) we see a memory configuration in which space for growth has been allocated to two processes.

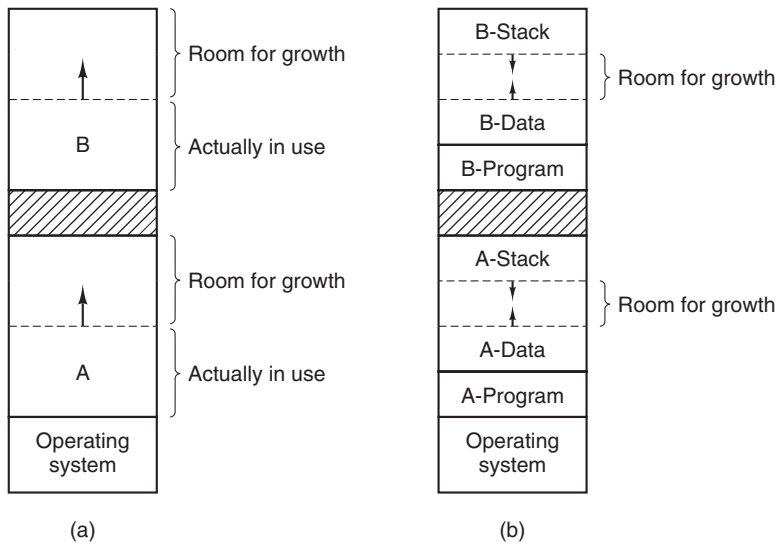

Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

If processes can have two growing segments—for example, the data segment being used as a heap for variables that are dynamically allocated and released and a stack segment for the normal local variables and return addresses—an alternative arrangement suggests itself, namely that of Fig. 3-5(b). In this figure we see that each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing upward. The memory between them can be used for either segment. If it runs out, the process will either have to be moved to a hole with sufficient space, swapped out of memory until a large enough hole can be created, or killed.

3.2.3 Managing Free Memory

When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: bitmaps and free lists. In this section and the next one we will look at these two methods. In

Chapter 10, we will look at some specific memory allocators used in Linux (like buddy and slab allocators) in more detail.

Memory Management with Bitmaps

With a bitmap, memory is divided into allocation units as small as a few words and as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 3-6 shows part of memory and the corresponding bitmap.

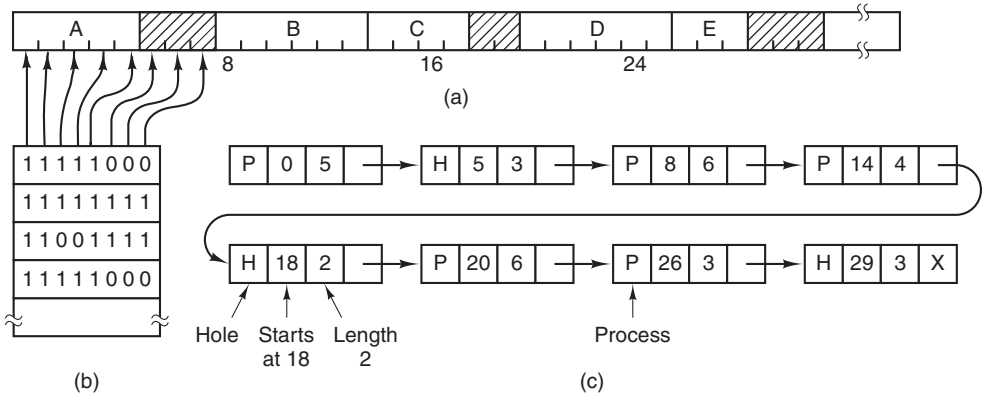

Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap. However, even with an allocation unit as small as 4 bytes, 32 bits of memory will require only 1 bit of the map. A memory of $32n$ bits will use n map bits, so the bitmap will take up only 1/32 of memory. If the allocation unit is chosen large, the bitmap will be smaller, but appreciable memory may be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit.

A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem is that when it has been decided to bring a k-unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bitmaps.

Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes. The memory of Fig. 3-6(a) is represented in Fig. 3-6(c) as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next item.

In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or bottom of memory). These may be either processes or holes, leading to the four combinations shown in Fig. 3-7. In Fig. 3-7(a) updating the list requires replacing a P by an H. In Fig. 3-7(b) and Fig. 3-7(c), two entries are coalesced into one, and the list becomes one entry shorter. In Fig. 3-7(d), three entries are merged and two items are removed from the list.

Since the process table slot for the terminating process will normally point to the list entry for the process itself, it may be more convenient to have the list as a double-linked list, rather than the single-linked list of Fig. 3-6(c). This structure makes it easier to find the previous entry and to see if a merge is possible.

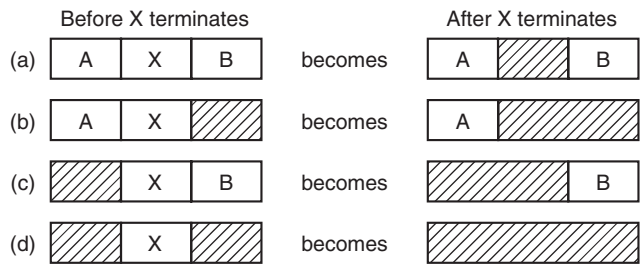

Figure 3-7. Four neighbor combinations for the terminating process, *X*.

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate. The simplest algorithm is **first fit**. The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

A minor variation of first fit is **next fit**. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does. Simulations by Bays (1977) show that next fit gives slightly worse performance than first fit.

Another well-known and widely used algorithm is **best fit**. Best fit searches the entire list, from beginning to end, and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed, to best match the request and the available holes.

As an example of first fit and best fit, consider Fig. 3-6 again. If a block of size 2 is needed, first fit will allocate the hole at 5, but best fit will allocate the hole at 18.

Best fit is slower than first fit because it must search the entire list every time it is called. Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes. First fit generates larger holes on the average.

To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about **worst fit**, that is, always take the largest available hole, so that the new hole will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

All four algorithms can be speeded up by maintaining separate lists for processes and holes. In this way, all of them devote their full energy to inspecting holes, not processes. The inevitable price that is paid for this speedup on allocation is the additional complexity and slowdown when deallocating memory, since a freed segment has to be removed from the process list and inserted into the hole list.

If distinct lists are maintained for processes and holes, the hole list may be kept sorted on size, to make best fit faster. When best fit searches a list of holes from smallest to largest, as soon as it finds a hole that fits, it knows that the hole is the smallest one that will do the job, hence the best fit. No further searching is needed, as it is with the single-list scheme. With a hole list sorted by size, first fit and best fit are equally fast, and next fit is pointless.

When the holes are kept on separate lists from the processes, a small optimization is possible. Instead of having a separate set of data structures for maintaining the hole list, as is done in Fig. 3-6(c), the information can be stored in the holes. The first word of each hole could be the hole size, and the second word a pointer to the following entry. The nodes of the list of Fig. 3-6(c), which require three words and one bit (P/H), are no longer needed.

Yet another allocation algorithm is **quick fit**, which maintains separate lists for some of the more common sizes requested. For example, it might have a table with *n* entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. Holes of, say, 21 KB, could be put either on the 20-KB list or on a special list of odd-sized holes.

With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge with them

is possible is quite expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

3.3 VIRTUAL MEMORY

While base and limit registers can be used to create the abstraction of address spaces, there is another problem that has to be solved: managing bloatware. While memory sizes are increasing rapidly, software sizes are increasing much faster. In the 1980s, many universities ran a timesharing system with dozens of (more-or-less satisfied) users running simultaneously on a 4-MB VAX. Now Microsoft recommends having at least 2 GB for 64-bit Windows 8. The trend toward multimedia puts even more demands on memory.

As a consequence of these developments, there is a need to run programs that are too large to fit in memory, and there is certainly a need to have systems that can support multiple programs running simultaneously, each of which fits in memory but all of which collectively exceed memory. Swapping is not an attractive option, since a typical SATA disk has a peak transfer rate of several hundreds of MB/sec, which means it takes seconds to swap out a 1-GB program and the same to swap in a 1-GB program.

The problem of programs larger than memory has been around since the beginning of computing, albeit in limited areas, such as science and engineering (simulating the creation of the universe or even simulating a new aircraft takes a lot of memory). A solution adopted in the 1960s was to split programs into little pieces, called **overlays**. When a program started, all that was loaded into memory was the overlay manager, which immediately loaded and ran overlay 0. When it was done, it would tell the overlay manager to load overlay 1, either above overlay 0 in memory (if there was space for it) or on top of overlay 0 (if there was no space). Some overlay systems were highly complex, allowing many overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the overlay manager.

Although the actual work of swapping overlays in and out was done by the operating system, the work of splitting the program into pieces had to be done manually by the programmer. Splitting large programs up into small, modular pieces was time consuming, boring, and error prone. Few programmers were good at this. It did not take long before someone thought of a way to turn the whole job over to the computer.

The method that was devised (Fotheringham, 1961) has come to be known as **virtual memory**. 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.

In a sense, virtual memory is a generalization of the base-and-limit-register idea. The 8088 had separate base registers (but no limit registers) for text and data. 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.

3.3.1 Paging

Most virtual memory systems use a technique called **paging**, which we will now describe. On any computer, programs reference a set of memory addresses. When a program executes an instruction like

```
MOV REG,1000
```

it does so to copy the contents of memory address 1000 to REG (or vice versa, depending on the computer). Addresses can be generated using indexing, base registers, segment registers, and other ways.

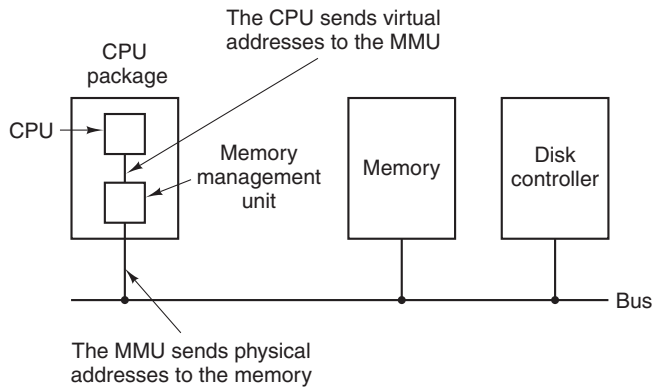

Figure 3-8. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago.

These program-generated addresses are called **virtual addresses** and form the **virtual address space**. On computers without virtual memory, the virtual address

is put directly onto the memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an **MMU** (**Memory Management Unit**) that maps the virtual addresses onto the physical memory addresses, as illustrated in Fig. 3-8.

A very simple example of how this mapping works is shown in Fig. 3-9. In this example, we have a computer that generates 16-bit addresses, from 0 up to 64K − 1. These are the virtual addresses. This computer, however, has only 32 KB of physical memory. So although 64-KB programs can be written, they cannot be loaded into memory in their entirety and run. A complete copy of a program's core image, up to 64 KB, must be present on the disk, however, so that pieces can be brought in as needed.

The virtual address space consists of fixed-size units called pages. The corresponding units in the physical memory are called **page frames**. The pages and page frames are generally the same size. In this example they are 4 KB, but page sizes from 512 bytes to a gigabyte have been used in real systems. With 64 KB of virtual address space and 32 KB of physical memory, we get 16 virtual pages and 8 page frames. Transfers between RAM and disk are always in whole pages. Many processors support multiple page sizes that can be mixed and matched as the operating system sees fit. For instance, the x86-64 architecture supports 4-KB, 2-MB, and 1-GB pages, so we could use 4-KB pages for user applications and a single 1-GB page for the kernel. We will see later why it is sometimes better to use a single large page, rather than a large number of small ones.

The notation in Fig. 3-9 is as follows. The range marked 0K–4K means that the virtual or physical addresses in that page are 0 to 4095. The range 4K–8K refers to addresses 4096 to 8191, and so on. Each page contains exactly 4096 addresses starting at a multiple of 4096 and ending one shy of a multiple of 4096.

When the program tries to access address 0, for example, using the instruction

```
MOV REG,0
```

virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

Similarly, the instruction

```
MOV REG,8192
```

is effectively transformed into

```
MOV REG,24576
```

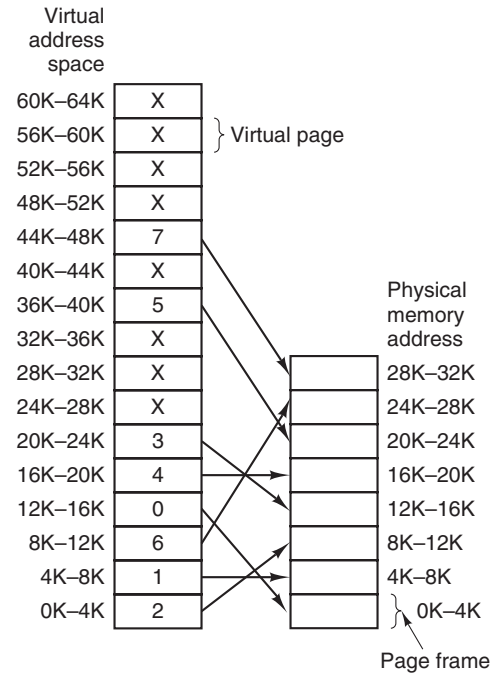

Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

because virtual address 8192 (in virtual page 2) is mapped onto 24576 (in physical page frame 6). As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

By itself, this ability to map the 16 virtual pages onto any of the eight page frames by setting the MMU's map appropriately does not solve the problem that the virtual address space is larger than the physical memory. Since we have only eight physical page frames, only eight of the virtual pages in Fig. 3-9 are mapped onto physical memory. The others, shown as a cross in the figure, are not mapped. In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory.

What happens if the program references an unmapped address, for example, by using the instruction

```
MOV REG,32780
```

which is byte 12 within virtual page 8 (starting at 32768)? The MMU notices that the page is unmapped (indicated by a cross in the figure) and causes the CPU to

trap to the operating system. This trap is called a **page fault**. The operating system picks a little-used page frame and writes its contents back to the disk (if it is not already there). It then fetches (also from the disk) the page that was just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

For example, if the operating system decided to evict page frame 1, it would load virtual page 8 at physical address 4096 and make two changes to the MMU map. First, it would mark virtual page 1's entry as unmapped, to trap any future accesses to virtual addresses between 4096 and 8191. Then it would replace the cross in virtual page 8's entry with a 1, so that when the trapped instruction is reexecuted, it will map virtual address 32780 to physical address 4108 (4096 + 12).

Now let us look inside the MMU to see how it works and why we have chosen to use a page size that is a power of 2. In Fig. 3-10 we see an example of a virtual address, 8196 (0010000000000100 in binary), being mapped using the MMU map of Fig. 3-9. The incoming 16-bit virtual address is split into a 4-bit page number and a 12-bit offset. With 4 bits for the page number, we can have 16 pages, and with 12 bits for the offset, we can address all 4096 bytes within a page.

The page number is used as an index into the **page table**, yielding the number of the page frame corresponding to that virtual page. If the *Present/absent* bit is 0, a trap to the operating system is caused. If the bit is 1, the page frame number found in the page table is copied to the high-order 3 bits of the output register, along with the 12-bit offset, which is copied unmodified from the incoming virtual address. Together they form a 15-bit physical address. The output register is then put onto the memory bus as the physical memory address.

3.3.2 Page Tables

In a simple implementation, the mapping of virtual addresses onto physical addresses can be summarized as follows: the virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits). For example, with a 16-bit address and a 4-KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095) within the selected page. However a split with 3 or 5 or some other number of bits for the page is also possible. Different splits imply different page sizes.

The virtual page number is used as an index into the page table to find the entry for that virtual page. From the page table entry, the page frame number (if any) is found. The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent to the memory.

Thus, the purpose of the page table is to map virtual pages onto page frames. Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result. Using the result of this

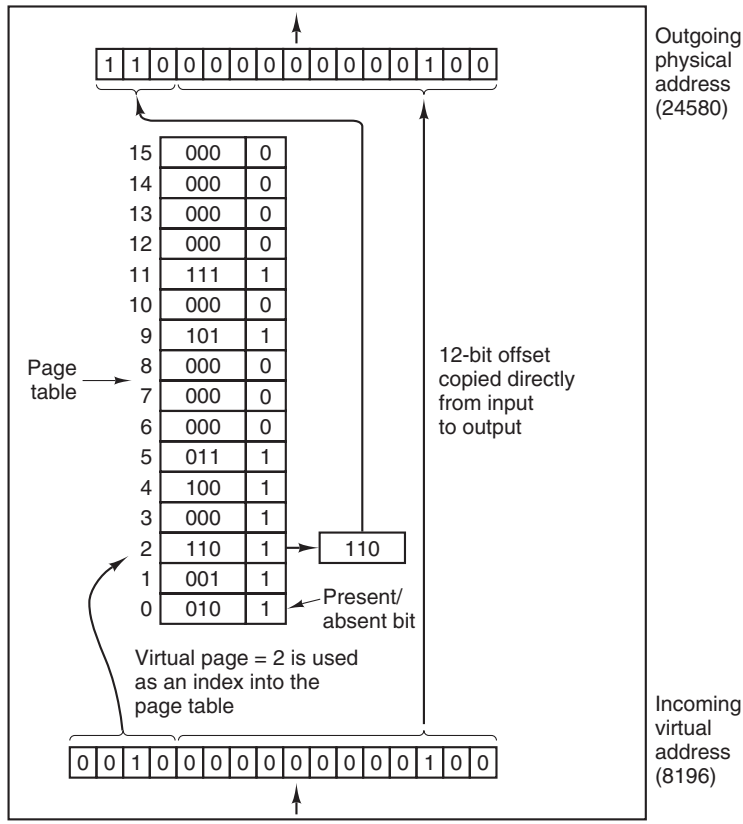

Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

function, the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

In this chapter, we worry only about virtual memory and not full virtualization. In other words: no virtual machines yet. We will see in Chap. 7 that each virtual machine requires its own virtual memory and as a result the page table organization becomes much more complicated—involving shadow or nested page tables and more. Even without such arcane configurations, paging and virtual memory are fairly sophisticated, as we shall see.

Structure of a Page Table Entry

Let us now turn from the structure of the page tables in the large, to the details of a single page table entry. The exact layout of an entry in the page table is highly machine dependent, but the kind of information present is roughly the same from machine to machine. In Fig. 3-11 we present a sample page table entry. The size

varies from computer to computer, but 32 bits is a common size. The most important field is the *Page frame number*. After all, the goal of the page mapping is to output this value. Next to it we have the *Present/absent* bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.

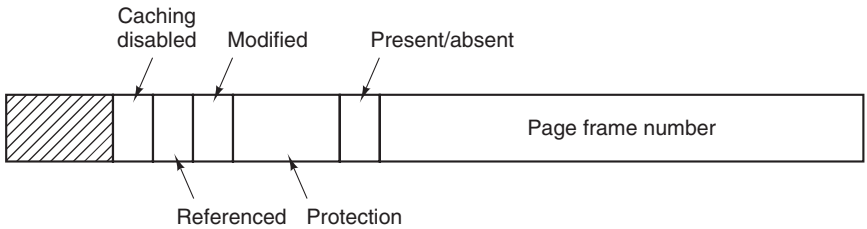

Figure 3-11. A typical page table entry.

The *Protection* bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page.

The *Modified* and *Referenced* bits keep track of page usage. When a page is written to, the hardware automatically sets the *Modified* bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified (i.e., is "dirty"), it must be written back to the disk. If it has not been modified (i.e., is "clean"), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the **dirty bit**, since it reflects the page's state.

The *Referenced* bit is set whenever a page is referenced, either for reading or for writing. Its value is used to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are far better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms that we will study later in this chapter.

Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory. If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old cached copy. With this bit, caching can be turned off. Machines that have a separate I/O space and do not use memory-mapped I/O do not need this bit.

Note that the disk address used to hold the page when it is not in memory is not part of the page table. The reason is simple. The page table holds only that information the hardware needs to translate a virtual address to a physical address.

Information the operating system needs to handle page faults is kept in software tables inside the operating system. The hardware does not need it.

Before getting into more implementation issues, it is worth pointing out again that what virtual memory fundamentally does is create a new abstraction—the address space—which is an abstraction of physical memory, just as a process is an abstraction of the physical processor (CPU). Virtual memory can be implemented by breaking the virtual address space up into pages, and mapping each one onto some page frame of physical memory or having it (temporarily) unmapped. Thus this section is bassically about an abstraction created by the operating system and how that abstraction is managed.

3.3.3 Speeding Up Paging

We have just seen the basics of virtual memory and paging. It is now time to go into more detail about possible implementations. In any paging system, two major issues must be faced:

1. The mapping from virtual address to physical address must be fast.
2. If the virtual address space is large, the page table will be large.

The first point is a consequence of the fact that the virtual-to-physical mapping must be done on every memory reference. All instructions must ultimately come from memory and many of them reference operands in memory as well. Consequently, it is necessary to make one, two, or sometimes more page table references per instruction. If an instruction execution takes, say, 1 nsec, the page table lookup must be done in under 0.2 nsec to avoid having the mapping become a major bottleneck.

The second point follows from the fact that all modern computers use virtual addresses of at least 32 bits, with 64 bits becoming the norm for desktops and laptops. With, say, a 4-KB page size, a 32-bit address space has 1 million pages, and a 64-bit address space has more than you want to contemplate. With 1 million pages in the virtual address space, the page table must have 1 million entries. And remember that each process needs its own page table (because it has its own virtual address space).

The need for large, fast page mapping is a very significant constraint on the way computers are built. The simplest design (at least conceptually) is to have a single page table consisting of an array of fast hardware registers, with one entry for each virtual page, indexed by virtual page number, as shown in Fig. 3-10. When a process is started up, the operating system loads the registers with the process' page table, taken from a copy kept in main memory. During process execution, no more memory references are needed for the page table. The advantages of this method are that it is straightforward and requires no memory references during mapping. A disadvantage is that it is unbearably expensive if the page table is

large; it is just not practical most of the time. Another one is that having to load the full page table at every context switch would completely kill performance.

At the other extreme, the page table can be entirely in main memory. All the hardware needs then is a single register that points to the start of the page table. This design allows the virtual-to-physical map to be changed at a context switch by reloading one register. Of course, it has the disadvantage of requiring one or more memory references to read page table entries during the execution of each instruction, making it very slow.

Translation Lookaside Buffers

Let us now look at widely implemented schemes for speeding up paging and for handling large virtual address spaces, starting with the former. The starting point of most optimization techniques is that the page table is in memory. Potentially, this design has an enormous impact on performance. Consider, for example, a 1-byte instruction that copies one register to another. In the absence of paging, this instruction makes only one memory reference, to fetch the instruction. With paging, at least one additional memory reference will be needed, to access the page table. Since execution speed is generally limited by the rate at which the CPU can get instructions and data out of the memory, having to make two memory references per memory reference reduces performance by half. Under these conditions, no one would use paging.

Computer designers have known about this problem for years and have come up with a solution. Their solution is based on the observation that most programs tend to make a large number of references to a small number of pages, and not the other way around. Thus only a small fraction of the page table entries are heavily read; the rest are barely used at all.

The solution that has been devised is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table. The device, called a **TLB** (**Translation Lookaside Buffer**) or sometimes an **associative memory**, is illustrated in Fig. 3-12. It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 256. Each entry contains information about one page, including the virtual page number, a bit that is set when the page is modified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located. These fields have a one-to-one correspondence with the fields in the page table, except for the virtual page number, which is not needed in the page table. Another bit indicates whether the entry is valid (i.e., in use) or not.

An example that might generate the TLB of Fig. 3-12 is a process in a loop that spans virtual pages 19, 20, and 21, so that these TLB entries have protection codes for reading and executing. The main data currently being used (say, an array being processed) are on pages 129 and 130. Page 140 contains the indices used in the array calculations. Finally, the stack is on pages 860 and 861.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB to speed up paging.

Let us now see how the TLB functions. When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel). Doing so requires special hardware, which all MMUs with TLBs have. If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB, without going to the page table. If the virtual page number is present in the TLB but the instruction is trying to write on a read-only page, a protection fault is generated.

The interesting case is what happens when the virtual page number is not in the TLB. The MMU detects the miss and does an ordinary page table lookup. It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up. Thus if that page is used again soon, the second time it will result in a TLB hit rather than a miss. When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory. The other values are already there, except the reference bit. When the TLB is loaded from the page table, all the fields are taken from memory.

Software TLB Management

Up until now, we have assumed that every machine with paged virtual memory has page tables recognized by the hardware, plus a TLB. In this design, TLB management and handling TLB faults are done entirely by the MMU hardware. Traps to the operating system occur only when a page is not in memory.

In the past, this assumption was true. However, many RISC machines, including the SPARC, MIPS, and (the now dead) HP PA, do nearly all of this page management in software. On these machines, the TLB entries are explicitly loaded by the operating system. When a TLB miss occurs, instead of the MMU going to the page tables to find and fetch the needed page reference, it just generates a TLB fault and tosses the problem into the lap of the operating system. The system must find the page, remove an entry from the TLB, enter the new one, and restart the

instruction that faulted. And, of course, all of this must be done in a handful of instructions because TLB misses occur much more frequently than page faults.

Surprisingly enough, if the TLB is moderately large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be acceptably efficient. The main gain here is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance. Software TLB management is discussed by Uhlig et al. (1994).

Various strategies were developed long ago to improve performance on machines that do TLB management in software. One approach attacks both reducing TLB misses and reducing the cost of a TLB miss when it does occur (Bala et al., 1994). To reduce TLB misses, sometimes the operating system can use its intuition to figure out which pages are likely to be used next and to preload entries for them in the TLB. For example, when a client process sends a message to a server process on the same machine, it is very likely that the server will have to run soon. Knowing this, while processing the trap to do the send, the system can also check to see where the server's code, data, and stack pages are and map them in before they get a chance to cause TLB faults.

The normal way to process a TLB miss, whether in hardware or in software, is to go to the page table and perform the indexing operations to locate the page referenced. The problem with doing this search in software is that the pages holding the page table may not be in the TLB, which will cause additional TLB faults during the processing. These faults can be reduced by maintaining a large (e.g., 4-KB) software cache of TLB entries in a fixed location whose page is always kept in the TLB. By first checking the software cache, the operating system can substantially reduce TLB misses.

When software TLB management is used, it is essential to understand the difference between different kinds of misses. A **soft miss** occurs when the page referenced is not in the TLB, but is in memory. All that is needed here is for the TLB to be updated. No disk I/O is needed. Typically a soft miss takes 10–20 machine instructions to handle and can be completed in a couple of nanoseconds. In contrast, a **hard miss** occurs when the page itself is not in memory (and of course, also not in the TLB). A disk access is required to bring in the page, which can take several milliseconds, depending on the disk being used. A hard miss is easily a million times slower than a soft miss. Looking up the mapping in the page table hierarchy is known as a **page table walk**.

Actually, it is worse that that. A miss is not just soft or hard. Some misses are slightly softer (or slightly harder) than other misses. For instance, suppose the page walk does not find the page in the process' page table and the program thus incurs a page fault. There are three possibilities. First, the page may actually be in memory, but not in this process' page table. For instance, the page may have been brought in from disk by another process. In that case, we do not need to access the disk again, but merely map the page appropriately in the page tables. This is a pretty soft miss that is known as a **minor page fault**. Second, a **major page fault**

occurs if the page needs to be brought in from disk. Third, it is possible that the program simply accessed an invalid address and no mapping needs to be added in the TLB at all. In that case, the operating system typically kills the program with a **segmentation fault**. Only in this case did the program do something wrong. All other cases are automatically fixed by the hardware and/or the operating system—at the cost of some performance.

3.3.4 Page Tables for Large Memories

TLBs can be used to speed up virtual-to-physical address translation over the original page-table-in-memory scheme. But that is not the only problem we have to tackle. Another problem is how to deal with very large virtual address spaces. Below we will discuss two ways of dealing with them.

Multilevel Page Tables

As a first approach, consider the use of a **multilevel page table**. A simple example is shown in Fig. 3-13. In Fig. 3-13(a) we have a 32-bit virtual address that is partitioned into a 10-bit *PT1* field, a 10-bit *PT2* field, and a 12-bit *Offset* field. Since offsets are 12 bits, pages are 4 KB, and there are a total of 2^{20} of them.

The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around. Suppose, for example, that a process needs 12 megabytes: the bottom 4 megabytes of memory for program text, the next 4 megabytes for data, and the top 4 megabytes for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used.

In Fig. 3-13(b) we see how the two-level page table works. On the left we see the top-level page table, with 1024 entries, corresponding to the 10-bit *PT1* field. When a virtual address is presented to the MMU, it first extracts the *PT1* field and uses this value as an index into the top-level page table. Each of these 1024 entries in the top-level page table represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 4096 bytes.

The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table. Entry 0 of the top-level page table points to the page table for the program text, entry 1 points to the page table for the data, and entry 1023 points to the page table for the stack. The other (shaded) entries are not used. The *PT2* field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

As an example, consider the 32-bit virtual address 0x00403004 (4,206,596 decimal), which is 12,292 bytes into the data. This virtual address corresponds to *PT1* = 1, *PT2* = 3, and *Offset* = 4. The MMU first uses *PT1* to index into the top-

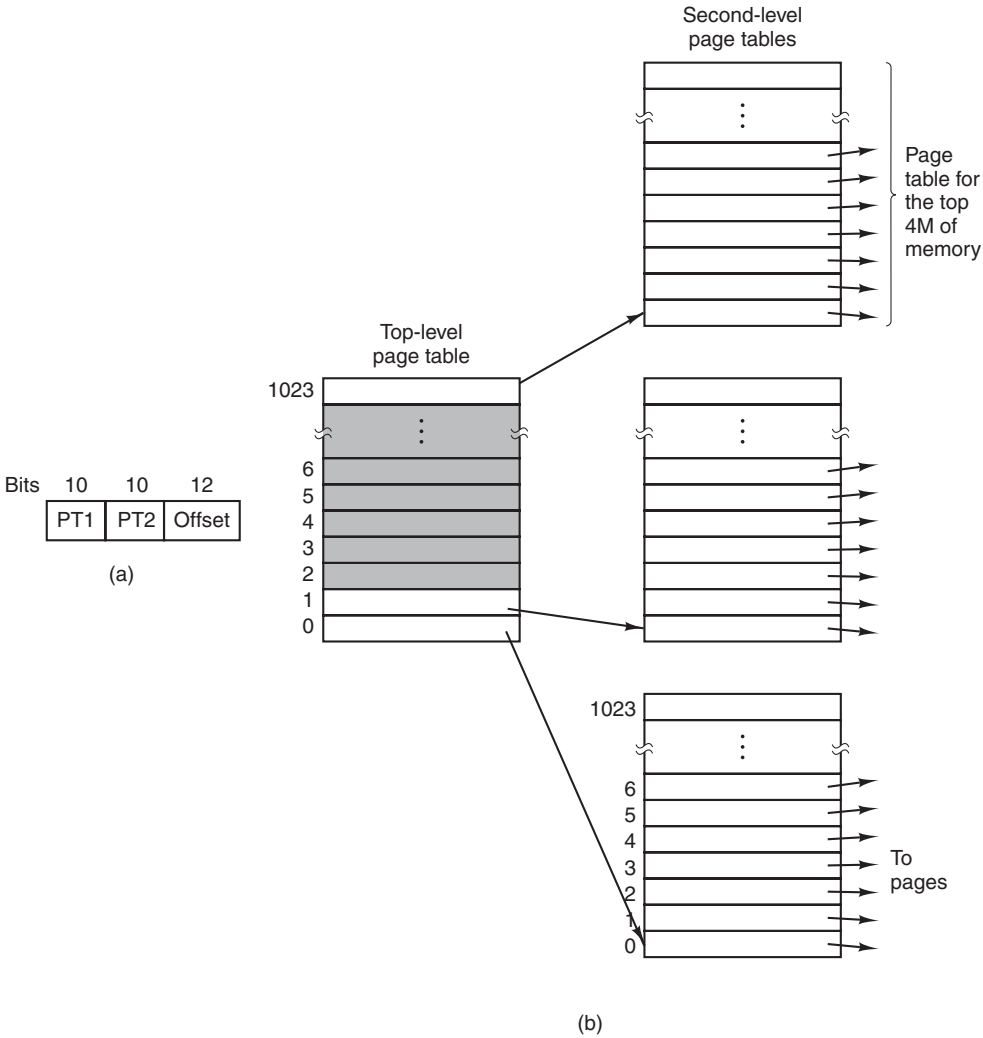

Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

level page table and obtain entry 1, which corresponds to addresses 4M to 8M – 1. It then uses *PT2* to index into the second-level page table just found and extract entry 3, which corresponds to addresses 12288 to 16383 within its 4M chunk (i.e., absolute addresses 4,206,592 to 4,210,687). This entry contains the page frame number of the page containing virtual address 0x00403004. If that page is not in memory, the *Present/absent* bit in the page table entry will have the value zero, causing a page fault. If the page is present in memory, the page frame number

taken from the second-level page table is combined with the offset (4) to construct the physical address. This address is put on the bus and sent to memory.

The interesting thing to note about Fig. 3-13 is that although the address space contains over a million pages, only four page tables are needed: the top-level table, and the second-level tables for 0 to 4M (for the program text), 4M to 8M (for the data), and the top 4M (for the stack). The *Present/absent* bits in the remaining 1021 entries of the top-level page table are set to 0, forcing a page fault if they are ever accessed. Should this occur, the operating system will notice that the process is trying to reference memory that it is not supposed to and will take appropriate action, such as sending it a signal or killing it. In this example we have chosen round numbers for the various sizes and have picked *PT1* equal to *PT2*, but in actual practice other values are also possible, of course.

The two-level page table system of Fig. 3-13 can be expanded to three, four, or more levels. Additional levels give more flexibility. For instance, Intel's 32 bit 80386 processor (launched in 1985) was able to address up to 4-GB of memory, using a two-level page table that consisted of a **page directory** whose entries pointed to page tables, which, in turn, pointed to the actual 4-KB page frames. Both the page directory and the page tables each contained 1024 entries, giving a total of $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$ addressable bytes, as desired.

Ten years later, the Pentium Pro introduced another level: the **page directory pointer table**. In addition, it extended each entry in each level of the page table hierarchy from 32 bits to 64 bits, so that it could address memory above the 4-GB boundary. As it had only 4 entries in the page directory pointer table, 512 in each page directory, and 512 in each page table, the total amount of memory it could address was still limited to a maximum of 4 GB. When proper 64-bit support was added to the x86 family (originally by AMD), the additional level *could* have been called the "page directory pointer table pointer" or something equally horri. That would have been perfectly in line with how chip makers tend to name things. Mercifully, they did not do this. The alternative they cooked up, "**page map level 4**," may not be a terribly catchy name either, but at least it is short and a bit clearer. At any rate, these processors now use all 512 entries in all tables, yielding an amount of addressable memory of $2^9 \times 2^9 \times 2^9 \times 2^9 \times 2^{12} = 2^{48}$ bytes. They could have added another level, but they probably thought that 256 TB would be sufficient for a while.

Inverted Page Tables

An alternative to ever-increasing levels in a paging hierarchy is known as **inverted page tables**. They were first used by such processors as the PowerPC, the UltraSPARC, and the Itanium (sometimes referred to as "Itanic," as it was not nearly the success Intel had hoped for). In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space. For

example, with 64-bit virtual addresses, a 4-KB page size, and 4 GB of RAM, an inverted page table requires only 1,048,576 entries. The entry keeps track of which (process, virtual page) is located in the page frame.

Although inverted page tables save lots of space, at least when the virtual address space is much larger than the physical memory, they have a serious downside: virtual-to-physical translation becomes much harder. When process *n* references virtual page *p*, the hardware can no longer find the physical page by using *p* as an index into the page table. Instead, it must search the entire inverted page table for an entry (*n*, *p*). Furthermore, this search must be done on every memory reference, not just on page faults. Searching a 256K table on every memory reference is not the way to make your machine blindingly fast.

The way out of this dilemma is to make use of the TLB. If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page tables. On a TLB miss, however, the inverted page table has to be searched in software. One feasible way to accomplish this search is to have a hash table hashed on the virtual address. All the virtual pages currently in memory that have the same hash value are chained together, as shown in Fig. 3-14. If the hash table has as many slots as the machine has physical pages, the average chain will be only one entry long, greatly speeding up the mapping. Once the page frame number has been found, the new (virtual, physical) pair is entered into the TLB.

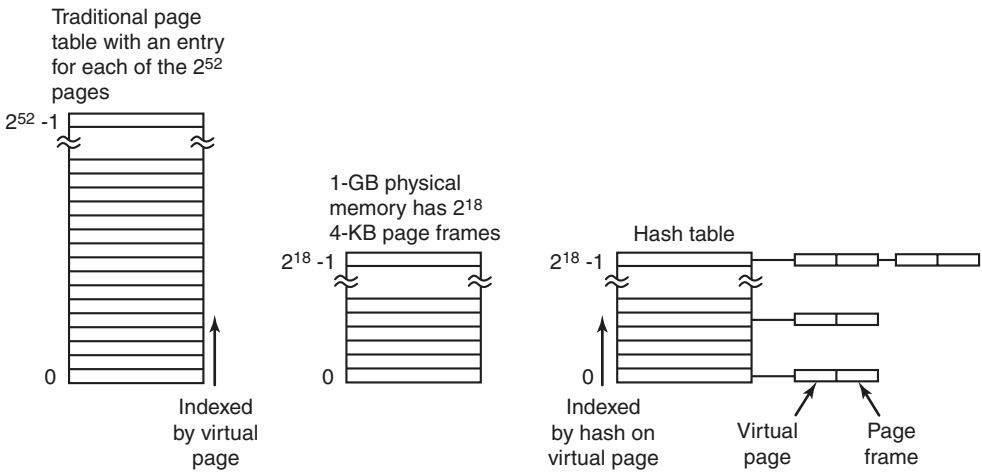

Figure 3-14. Comparison of a traditional page table with an inverted page table.

Inverted page tables are common on 64-bit machines because even with a very large page size, the number of page table entries is gigantic. For example, with 4-MB pages and 64-bit virtual addresses, 2^{42} page table entries are needed. Other approaches to handling large virtual memories can be found in Talluri et al. (1995).

3.4 PAGE REPLACEMENT ALGORITHMS

When a page fault occurs, the operating system has to choose a page to evict (remove from memory) to make room for the incoming page. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted.

While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important ones.

It is worth noting that the problem of "page replacement" occurs in other areas of computer design as well. For example, most computers have one or more memory caches consisting of recently used 32-byte or 64-byte memory blocks. When the cache is full, some block has to be chosen for removal. This problem is precisely the same as page replacement except on a shorter time scale (it has to be done in a few nanoseconds, not milliseconds as with page replacement). The reason for the shorter time scale is that cache block misses are satisfied from main memory, which has no seek time and no rotational latency.

A second example is in a Web server. The server can keep a certain number of heavily used Web pages in its memory cache. However, when the memory cache is full and a new page is referenced, a decision has to be made which Web page to evict. The considerations are similar to pages of virtual memory, except that the Web pages are never modified in the cache, so there is always a fresh copy "on disk." In a virtual memory system, pages in main memory may be either clean or dirty.

In all the page replacement algorithms to be studied below, a certain issue arises: when a page is to be evicted from memory, does it have to be one of the faulting process' own pages, or can it be a page belonging to another process? In the former case, we are effectively limiting each process to a fixed number of pages; in the latter case we are not. Both are possibilities. We will come back to this point in Sec. 3.5.1.

3.4.1 The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to actually implement. It goes like this. At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not

be referenced until 10, 100, or perhaps 1000 instructions later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced.

The optimal page replacement algorithm says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible. Computers, like people, try to put off unpleasant events for as long as they can.

The only problem with this algorithm is that it is unrealizable. At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next. (We saw a similar situation earlier with the shortest-job-first scheduling algorithm—how can the system tell which job is shortest?) Still, by running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement on the *second* run by using the page-reference information collected during the *first* run.

In this way, it is possible to compare the performance of realizable algorithms with the best possible one. If an operating system achieves a performance of, say, only 1% worse than the optimal algorithm, effort spent in looking for a better algorithm will yield at most a 1% improvement.

To avoid any possible confusion, it should be made clear that this log of page references refers only to the one program just measured and then with only one specific input. The page replacement algorithm derived from it is thus specific to that one program and input data. Although this method is useful for evaluating page replacement algorithms, it is of no use in practical systems. Below we will study algorithms that *are* useful on real systems.

3.4.2 The Not Recently Used Page Replacement Algorithm

In order to allow the operating system to collect useful page usage statistics, most computers with virtual memory have two status bits, *R* and *M*, associated with each page. *R* is set whenever the page is referenced (read or written). *M* is set when the page is written to (i.e., modified). The bits are contained in each page table entry, as shown in Fig. 3-11. It is important to realize that these bits must be updated on every memory reference, so it is essential that they be set by the hardware. Once a bit has been set to 1, it stays 1 until the operating system resets it.

If the hardware does not have these bits, they can be simulated using the operating system's page fault and clock interrupt mechanisms. When a process is started up, all of its page table entries are marked as not in memory. As soon as any page is referenced, a page fault will occur. The operating system then sets the *R* bit (in its internal tables), changes the page table entry to point to the correct page, with mode READ ONLY, and restarts the instruction. If the page is subsequently modified, another page fault will occur, allowing the operating system to set the *M* bit and change the page's mode to READ/WRITE.

The *R* and *M* bits can be used to build a simple paging algorithm as follows. When a process is started up, both page bits for all its pages are set to 0 by the operating system. Periodically (e.g., on each clock interrupt), the *R* bit is cleared, to distinguish pages that have not been referenced recently from those that have been.

When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their *R* and *M* bits:

Class 0: not referenced, not modified.
Class 1: not referenced, modified.
Class 2: referenced, not modified.
Class 3: referenced, modified.

Although class 1 pages seem, at first glance, impossible, they occur when a class 3 page has its *R* bit cleared by a clock interrupt. Clock interrupts do not clear the *M* bit because this information is needed to know whether the page has to be rewritten to disk or not. Clearing *R* but not *M* leads to a class 1 page.

The **NRU** (**Not Recently Used**) algorithm removes a page at random from the lowest-numbered nonempty class. Implicit in this algorithm is the idea that it is better to remove a modified page that has not been referenced in at least one clock tick (typically about 20 msec) than a clean page that is in heavy use. The main attraction of NRU is that it is easy to understand, moderately efficient to implement, and gives a performance that, while certainly not optimal, may be adequate.

3.4.3 The First-In, First-Out (FIFO) Page Replacement Algorithm

Another low-overhead paging algorithm is the **FIFO** (**First-In, First-Out**) algorithm. To illustrate how this works, consider a supermarket that has enough shelves to display exactly *k* different products. One day, some company introduces a new convenience food—instant, freeze-dried, organic yogurt that can be reconstituted in a microwave oven. It is an immediate success, so our finite supermarket has to get rid of one old product in order to stock it.

One possibility is to find the product that the supermarket has been stocking the longest (i.e., something it began selling 120 years ago) and get rid of it on the grounds that no one is interested any more. In effect, the supermarket maintains a linked list of all the products it currently sells in the order they were introduced. The new one goes on the back of the list; the one at the front of the list is dropped.

As a page replacement algorithm, the same idea is applicable. The operating system maintains a list of all pages currently in memory, with the most recent arrival at the tail and the least recent arrival at the head. On a page fault, the page at the head is removed and the new page added to the tail of the list. When applied to stores, FIFO might remove mustache wax, but it might also remove flour, salt, or butter. When applied to computers the same problem arises: the oldest page may still be useful. For this reason, FIFO in its pure form is rarely used.

3.4.4 The Second-Chance Page Replacement Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the *R* bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the *R* bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

The operation of this algorithm, called **second chance**, is shown in Fig. 3-15. In Fig. 3-15(a) we see pages *A* through *H* kept on a linked list and sorted by the time they arrived in memory.

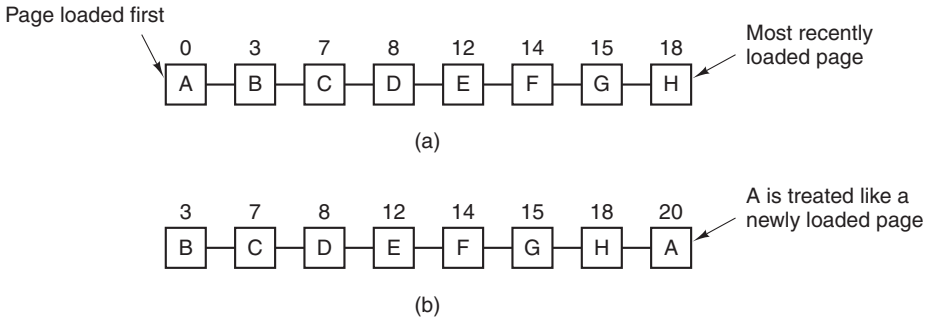

Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

Suppose that a page fault occurs at time 20. The oldest page is *A*, which arrived at time 0, when the process started. If *A* has the *R* bit cleared, it is evicted from memory, either by being written to the disk (if it is dirty), or just abandoned (if it is clean). On the other hand, if the *R* bit is set, *A* is put onto the end of the list and its "load time" is reset to the current time (20). The *R* bit is also cleared. The search for a suitable page continues with *B*.

What second chance is looking for is an old page that has not been referenced in the most recent clock interval. If all the pages have been referenced, second chance degenerates into pure FIFO. Specifically, imagine that all the pages in Fig. 3-15(a) have their *R* bits set. One by one, the operating system moves the pages to the end of the list, clearing the *R* bit each time it appends a page to the end of the list. Eventually, it comes back to page *A*, which now has its *R* bit cleared. At this point *A* is evicted. Thus the algorithm always terminates.

3.4.5 The Clock Page Replacement Algorithm

Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is constantly moving pages around on its list. A better approach is to keep all the page frames on a circular list in the form of a clock, as shown in Fig. 3-16. The hand points to the oldest page.

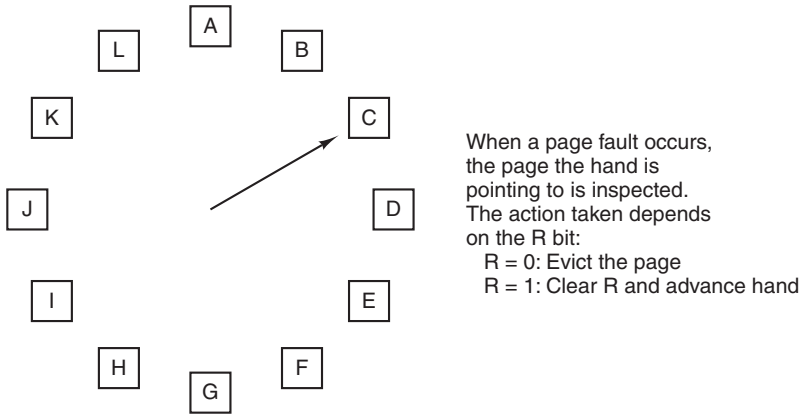

Figure 3-16. The clock page replacement algorithm.

When a page fault occurs, the page being pointed to by the hand is inspected. If its *R* bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If *R* is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with $R = 0$. Not surprisingly, this algorithm is called **clock**.

3.4.6 The Least Recently Used (LRU) Page Replacement Algorithm

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again soon. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called **LRU** (**Least Recently Used**) paging.

Although LRU is theoretically realizable, it is not cheap by a long shot. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware (assuming that such hardware could be built).

However, there are other ways to implement LRU with special hardware. Let us consider the simplest way first. This method requires equipping the hardware with a 64-bit counter, *C*, that is automatically incremented after each instruction. Furthermore, each page table entry must also have a field large enough to contain the counter. After each memory reference, the current value of *C* is stored in the

page table entry for the page just referenced. When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used.

3.4.7 Simulating LRU in Software

Although the previous LRU algorithm is (in principle) realizable, few, if any, machines have the required hardware. Instead, a solution that can be implemented in software is needed. One possibility is called the **NFU** (**Not Frequently Used**) algorithm. It requires a software counter associated with each page, initially zero. At each clock interrupt, the operating system scans all the pages in memory. For each page, the *R* bit, which is 0 or 1, is added to the counter. The counters roughly keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement.

The main problem with NFU is that it is like an elephant: it never forgets anything. For example, in a multipass compiler, pages that were heavily used during pass 1 may still have a high count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts than the pass-1 pages. Consequently, the operating system will remove useful pages instead of pages no longer in use.

Fortunately, a small modification to NFU makes it able to simulate LRU quite well. The modification has two parts. First, the counters are each shifted right 1 bit before the *R* bit is added in. Second, the *R* bit is added to the leftmost rather than the rightmost bit.

Figure 3-17 illustrates how the modified algorithm, known as **aging**, works. Suppose that after the first clock tick the *R* bits for pages 0 to 5 have the values 1, 0, 1, 0, 1, and 1, respectively (page 0 is 1, page 1 is 0, page 2 is 1, etc.). In other words, between tick 0 and tick 1, pages 0, 2, 4, and 5 were referenced, setting their *R* bits to 1, while the other ones remained 0. After the six corresponding counters have been shifted and the *R* bit inserted at the left, they have the values shown in Fig. 3-17(a). The four remaining columns show the six counters after the next four clock ticks.

When a page fault occurs, the page whose counter is the lowest is removed. It is clear that a page that has not been referenced for, say, four clock ticks will have four leading zeros in its counter and thus will have a lower value than a counter that has not been referenced for three clock ticks.

This algorithm differs from LRU in two important ways. Consider pages 3 and 5 in Fig. 3-17(e). Neither has been referenced for two clock ticks; both were referenced in the tick prior to that. According to LRU, if a page must be replaced, we should choose one of these two. The trouble is, we do not know which of them was referenced last in the interval between tick 1 and tick 2. By recording only 1 bit per time interval, we have now lost the ability to distinguish references early in the

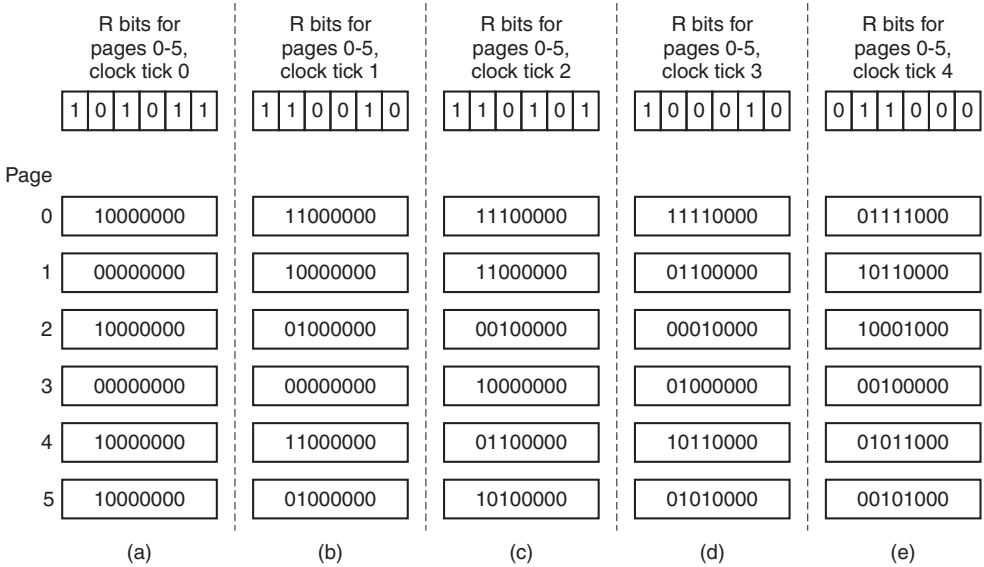

Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

clock interval from those occurring later. All we can do is remove page 3, because page 5 was also referenced two ticks earlier and page 3 was not.

The second difference between LRU and aging is that in aging the counters have a finite number of bits (8 bits in this example), which limits its past horizon. Suppose that two pages each have a counter value of 0. All we can do is pick one of them at random. In reality, it may well be that one of the pages was last referenced nine ticks ago and the other was last referenced 1000 ticks ago. We have no way of seeing that. In practice, however, 8 bits is generally enough if a clock tick is around 20 msec. If a page has not been referenced in 160 msec, it probably is not that important.

3.4.8 The Working Set Page Replacement Algorithm

In the purest form of paging, processes are started up with none of their pages in memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the operating system to bring in the page containing the first instruction. Other page faults for global variables and the stack usually follow quickly. After a while, the process has most of the pages it needs and settles down to run with relatively few page faults. This strategy is called **demand paging** because pages are loaded only on demand, not in advance.

Of course, it is easy enough to write a test program that systematically reads all the pages in a large address space, causing so many page faults that there is not

enough memory to hold them all. Fortunately, most processes do not work this way. They exhibit a **locality of reference**, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. Each pass of a multipass compiler, for example, references only a fraction of all the pages, and a different fraction at that.

The set of pages that a process is currently using is its **working set** (Denning, 1968a; Denning, 1980). If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase (e.g., the next pass of the compiler). If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly, since executing an instruction takes a few nanoseconds and reading in a page from the disk typically takes 10 msec At a rate of one or two instructions per 10 msec, it will take ages to finish. A program causing page faults every few instructions is said to be **thrashing** (Denning, 1968b).

In a multiprogramming system, processes are often moved to disk (i.e., all their pages are removed from memory) to let others have a turn at the CPU. The question arises of what to do when a process is brought back in again. Technically, nothing need be done. The process will just cause page faults until its working set has been loaded. The problem is that having numerous page faults every time a process is loaded is slow, and it also wastes considerable CPU time, since it takes the operating system a few milliseconds of CPU time to process a page fault.

Therefore, many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the **working set model** (Denning, 1970). It is designed to greatly reduce the page fault rate. Loading the pages *before* letting processes run is also called **prepaging.** Note that the working set changes over time.

It has long been known that programs rarely reference their address space uniformly, but that the references tend to cluster on a small number of pages. A memory reference may fetch an instruction or data, or it may store data. At any instant of time, t, there exists a set consisting of all the pages used by the k most recent memory references. This set, $w(k, t)$, is the working set. Because the $k = 1$ most recent references must have used all the pages used by the $k > 1$ most recent references, and possibly others, $w(k, t)$ is a monotonically nondecreasing function of k. The limit of $w(k, t)$ as k becomes large is finite because a program cannot reference more pages than its address space contains, and few programs will use every single page. Figure 3-18 depicts the size of the working set as a function of k.

The fact that most programs randomly access a small number of pages, but that this set changes slowly in time explains the initial rapid rise of the curve and then the much slower rise for large k. For example, a program that is executing a loop occupying two pages using data on four pages may reference all six pages every 1000 instructions, but the most recent reference to some other page may be a million instructions earlier, during the initialization phase. Due to this asymptotic behavior, the contents of the working set is not sensitive to the value of k chosen. To

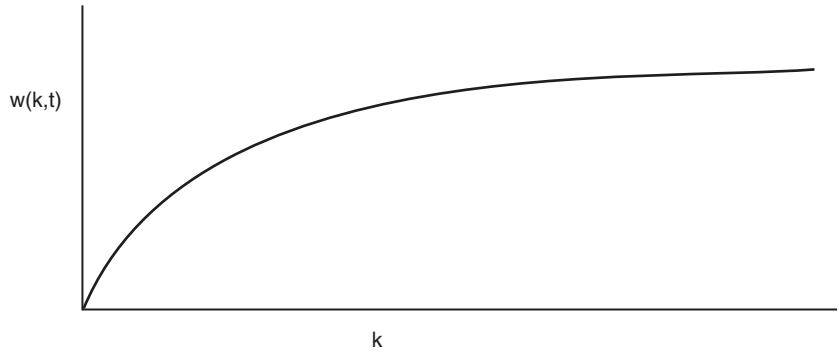

Figure 3-18. The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time t.

put it differently, there exists a wide range of k values for which the working set is unchanged. Because the working set varies slowly with time, it is possible to make a reasonable guess as to which pages will be needed when the program is restarted on the basis of its working set when it was last stopped. Prepaging consists of loading these pages before resuming the process.

To implement the working set model, it is necessary for the operating system to keep track of which pages are in the working set. Having this information also immediately leads to a possible page replacement algorithm: when a page fault occurs, find a page not in the working set and evict it. To implement such an algorithm, we need a precise way of determining which pages are in the working set. By definition, the working set is the set of pages used in the k most recent memory references (some authors use the k most recent page references, but the choice is arbitrary). To implement any working set algorithm, some value of k must be chosen in advance. Then, after every memory reference, the set of pages used by the most recent k memory references is uniquely determined.

Of course, having an operational definition of the working set does not mean that there is an efficient way to compute it during program execution. One could imagine a shift register of length k, with every memory reference shifting the register left one position and inserting the most recently referenced page number on the right. The set of all k page numbers in the shift register would be the working set. In theory, at a page fault, the contents of the shift register could be read out and sorted. Duplicate pages could then be removed. The result would be the working set. However, maintaining the shift register and processing it at a page fault would both be prohibitively expensive, so this technique is never used.

Instead, various approximations are used. One commonly used approximation is to drop the idea of counting back k memory references and use execution time instead. For example, instead of defining the working set as those pages used during the previous 10 million memory references, we can define it as the set of pages

used during the past 100 msec of execution time. In practice, such a definition is just as good and much easier to work with. Note that for each process, only its own execution time counts. Thus if a process starts running at time T and has had 40 msec of CPU time at real time T + 100 msec, for working set purposes its time is 40 msec. The amount of CPU time a process has actually used since it started is often called its **current virtual time**. With this approximation, the working set of a process is the set of pages it has referenced during the past τ seconds of virtual time.

Now let us look at a page replacement algorithm based on the working set. The basic idea is to find a page that is not in the working set and evict it. In Fig. 3-19 we see a portion of a page table for some machine. Because only pages located in memory are considered as candidates for eviction, pages that are absent from memory are ignored by this algorithm. Each entry contains (at least) two key items of information: the (approximate) time the page was last used and the *R* (Referenced) bit. An empty white rectangle symbolizes the other fields not needed for this algorithm, such as the page frame number, the protection bits, and the *M* (Modified) bit.

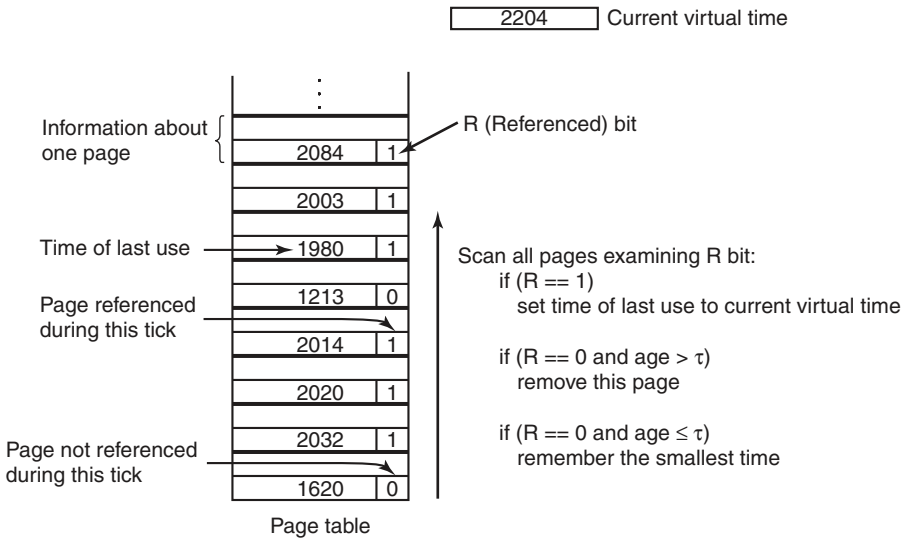

Figure 3-19. The working set algorithm.

The algorithm works as follows. The hardware is assumed to set the *R* and *M* bits, as discussed earlier. Similarly, a periodic clock interrupt is assumed to cause software to run that clears the *Referenced* bit on every clock tick. On every page fault, the page table is scanned to look for a suitable page to evict.

As each entry is processed, the *R* bit is examined. If it is 1, the current virtual time is written into the *Time of last use* field in the page table, indicating that the

page was in use at the time the fault occurred. Since the page has been referenced during the current clock tick, it is clearly in the working set and is not a candidate for removal (τ is assumed to span multiple clock ticks).

If R is 0, the page has not been referenced during the current clock tick and may be a candidate for removal. To see whether or not it should be removed, its age (the current virtual time minus its *Time of last use*) is computed and compared to τ. If the age is greater than τ, the page is no longer in the working set and the new page replaces it. The scan continues updating the remaining entries.

However, if R is 0 but the age is less than or equal to τ, the page is still in the working set. The page is temporarily spared, but the page with the greatest age (smallest value of *Time of last use*) is noted. If the entire table is scanned without finding a candidate to evict, that means that all pages are in the working set. In that case, if one or more pages with $R = 0$ were found, the one with the greatest age is evicted. In the worst case, all pages have been referenced during the current clock tick (and thus all have $R = 1$), so one is chosen at random for removal, preferably a clean page, if one exists.

3.4.9 The WSClock Page Replacement Algorithm

The basic working set algorithm is cumbersome, since the entire page table has to be scanned at each page fault until a suitable candidate is located. An improved algorithm, which is based on the clock algorithm but also uses the working set information, is called **WSClock** (Carr and Hennessey, 1981). Due to its simplicity of implementation and good performance, it is widely used in practice.

The data structure needed is a circular list of page frames, as in the clock algorithm, and as shown in Fig. 3-20(a). Initially, this list is empty. When the first page is loaded, it is added to the list. As more pages are added, they go into the list to form a ring. Each entry contains the *Time of last use* field from the basic working set algorithm, as well as the R bit (shown) and the M bit (not shown).

As with the clock algorithm, at each page fault the page pointed to by the hand is examined first. If the R bit is set to 1, the page has been used during the current tick so it is not an ideal candidate to remove. The R bit is then set to 0, the hand advanced to the next page, and the algorithm repeated for that page. The state after this sequence of events is shown in Fig. 3-20(b).

Now consider what happens if the page pointed to has $R = 0$, as shown in Fig. 3-20(c). If the age is greater than τ and the page is clean, it is not in the working set and a valid copy exists on the disk. The page frame is simply claimed and the new page put there, as shown in Fig. 3-20(d). On the other hand, if the page is dirty, it cannot be claimed immediately since no valid copy is present on disk. To avoid a process switch, the write to disk is scheduled, but the hand is advanced and the algorithm continues with the next page. After all, there might be an old, clean page further down the line that can be used immediately.

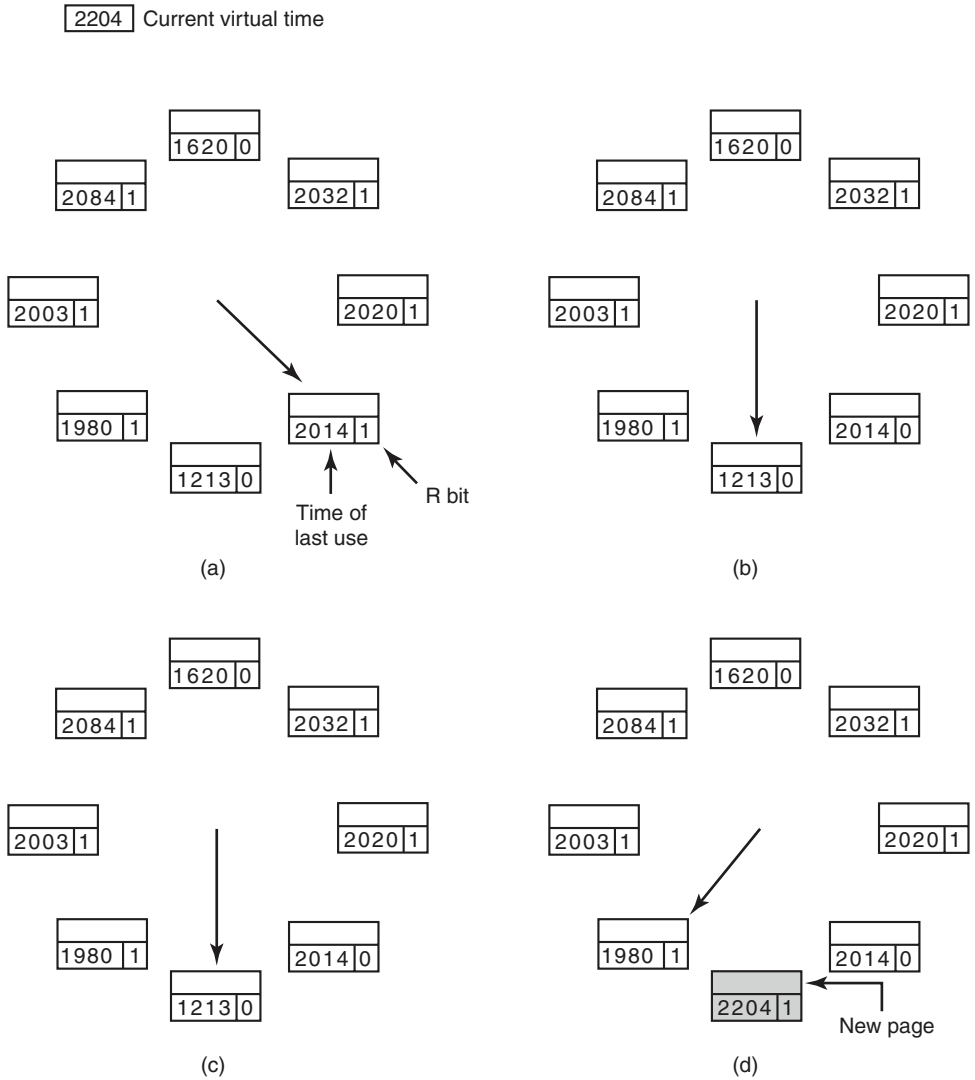

Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$. (c) and (d) give an example of $R = 0$.

In principle, all pages might be scheduled for disk I/O on one cycle around the clock. To reduce disk traffic, a limit might be set, allowing a maximum of n pages to be written back. Once this limit has been reached, no new writes would be scheduled.

What happens if the hand comes all the way around and back to its starting point? There are two cases we have to consider:

1. At least one write has been scheduled.
2. No writes have been scheduled.

In the first case, the hand just keeps moving, looking for a clean page. Since one or more writes have been scheduled, eventually some write will complete and its page will be marked as clean. The first clean page encountered is evicted. This page is not necessarily the first write scheduled because the disk driver may reorder writes in order to optimize disk performance.

In the second case, all pages are in the working set, otherwise at least one write would have been scheduled. Lacking additional information, the simplest thing to do is claim any clean page and use it. The location of a clean page could be kept track of during the sweep. If no clean pages exist, then the current page is chosen as the victim and written back to disk.

3.4.10 Summary of Page Replacement Algorithms

We have now looked at a variety of page replacement algorithms. Now we will briefly summarize them. The list of algorithms discussed is given in Fig. 3-21.

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Figure 3-21. Page replacement algorithms discussed in the text.

The optimal algorithm evicts the page that will be referenced furthest in the future. Unfortunately, there is no way to determine which page this is, so in practice this algorithm cannot be used. It is useful as a benchmark against which other algorithms can be measured, however.

The NRU algorithm divides pages into four classes depending on the state of the *R* and *M* bits. A random page from the lowest-numbered class is chosen. This algorithm is easy to implement, but it is very crude. Better ones exist.

FIFO keeps track of the order in which pages were loaded into memory by keeping them in a linked list. Removing the oldest page then becomes trivial, but that page might still be in use, so FIFO is a bad choice.

Second chance is a modification to FIFO that checks if a page is in use before removing it. If it is, the page is spared. This modification greatly improves the performance. Clock is simply a different implementation of second chance. It has the same performance properties, but takes a little less time to execute the algorithm.

LRU is an excellent algorithm, but it cannot be implemented without special hardware. If this hardware is not available, it cannot be used. NFU is a crude attempt to approximate LRU. It is not very good. However, aging is a much better approximation to LRU and can be implemented efficiently. It is a good choice.

The last two algorithms use the working set. The working set algorithm gives reasonable performance, but it is somewhat expensive to implement. WSClock is a variant that not only gives good performance but is also efficient to implement.

All in all, the two best algorithms are aging and WSClock. They are based on LRU and the working set, respectively. Both give good paging performance and can be implemented efficiently. A few other good algorithms exist, but these two are probably the most important in practice.

3.5 DESIGN ISSUES FOR PAGING SYSTEMS

In the previous sections we have explained how paging works and have given a few of the basic page replacement algorithms. But knowing the bare mechanics is not enough. To design a system and make it work well you have to know a lot more. It is like the difference between knowing how to move the rook, knight, bishop, and other pieces in chess, and being a good player. In the following sections, we will look at other issues that operating system designers must consider carefully in order to get good performance from a paging system.

3.5.1 Local versus Global Allocation Policies

In the preceding sections we have discussed several algorithms for choosing a page to replace when a fault occurs. A major issue associated with this choice (which we have carefully swept under the rug until now) is how memory should be allocated among the competing runnable processes.

Take a look at Fig. 3-22(a). In this figure, three processes, *A*, *B*, and *C*, make up the set of runnable processes. Suppose *A* gets a page fault. Should the page replacement algorithm try to find the least recently used page considering only the six pages currently allocated to *A*, or should it consider all the pages in memory? If it looks only at *A*'s pages, the page with the lowest age value is *A5*, so we get the situation of Fig. 3-22(b).

On the other hand, if the page with the lowest age value is removed without regard to whose page it is, page *B3* will be chosen and we will get the situation of Fig. 3-22(c). The algorithm of Fig. 3-22(b) is said to be a **local** page replacement

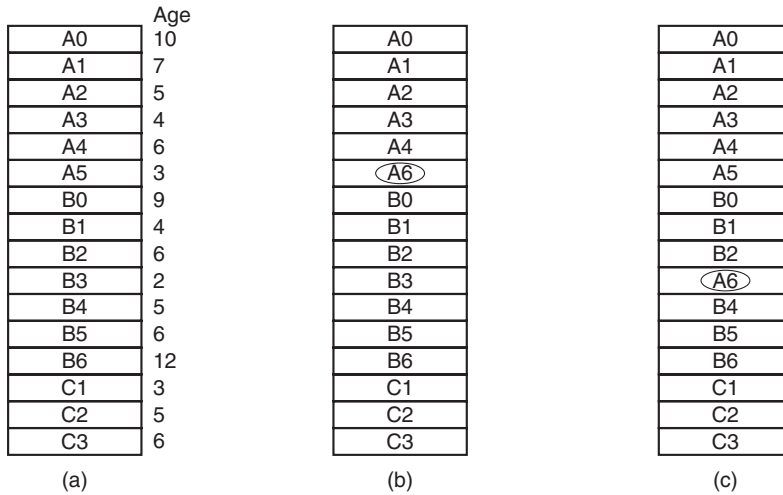

Figure 3-22. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

algorithm, whereas that of Fig. 3-22(c) is said to be a **global** algorithm. Local algorithms effectively correspond to allocating every process a fixed fraction of the memory. Global algorithms dynamically allocate page frames among the runnable processes. Thus the number of page frames assigned to each process varies in time.

In general, global algorithms work better, especially when the working set size can vary a lot over the lifetime of a process. If a local algorithm is used and the working set grows, thrashing will result, even if there are a sufficient number of free page frames. If the working set shrinks, local algorithms waste memory. If a global algorithm is used, the system must continually decide how many page frames to assign to each process. One way is to monitor the working set size as indicated by the aging bits, but this approach does not necessarily prevent thrashing. The working set may change size in milliseconds, whereas the aging bits are a very crude measure spread over a number of clock ticks.

Another approach is to have an algorithm for allocating page frames to processes. One way is to periodically determine the number of running processes and allocate each process an equal share. Thus with 12,416 available (i.e., nonoperating system) page frames and 10 processes, each process gets 1241 frames. The remaining six go into a pool to be used when page faults occur.

Although this method may seem fair, it makes little sense to give equal shares of the memory to a 10-KB process and a 300-KB process. Instead, pages can be allocated in proportion to each process' total size, with a 300-KB process getting 30 times the allotment of a 10-KB process. It is probably wise to give each process some minimum number, so that it can run no matter how small it is. On some

machines, for example, a single two-operand instruction may need as many as six pages because the instruction itself, the source operand, and the destination operand may all straddle page boundaries. With an allocation of only five pages, programs containing such instructions cannot execute at all.

If a global algorithm is used, it may be possible to start each process up with some number of pages proportional to the process' size, but the allocation has to be updated dynamically as the processes run. One way to manage the allocation is to use the **PFF** (**Page Fault Frequency**) algorithm. It tells when to increase or decrease a process' page allocation but says nothing about which page to replace on a fault. It just controls the size of the allocation set.

For a large class of page replacement algorithms, including LRU, it is known that the fault rate decreases as more pages are assigned, as we discussed above. This is the assumption behind PFF. This property is illustrated in Fig. 3-23.

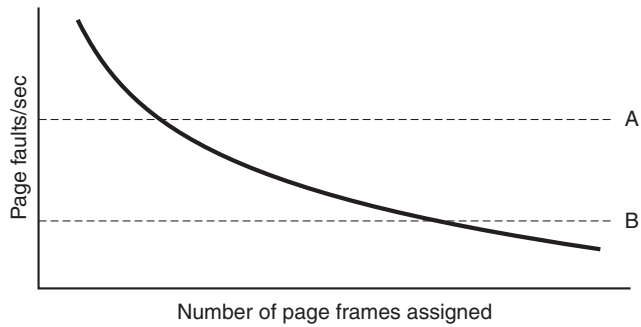

Figure 3-23. Page fault rate as a function of the number of page frames assigned.

Measuring the page fault rate is straightforward: just count the number of faults per second, possibly taking a running mean over past seconds as well. One easy way to do this is to add the number of page faults during the immediately preceding second to the current running mean and divide by two. The dashed line marked *A* corresponds to a page fault rate that is unacceptably high, so the faulting process is given more page frames to reduce the fault rate. The dashed line marked *B* corresponds to a page fault rate so low that we can assume the process has too much memory. In this case, page frames may be taken away from it. Thus, PFF tries to keep the paging rate for each process within acceptable bounds.

It is important to note that some page replacement algorithms can work with either a local replacement policy or a global one. For example, FIFO can replace the oldest page in all of memory (global algorithm) or the oldest page owned by the current process (local algorithm). Similarly, LRU or some approximation to it can replace the least recently used page in all of memory (global algorithm) or the least recently used page owned by the current process (local algorithm). The choice of local versus global is independent of the algorithm in some cases.

On the other hand, for other page replacement algorithms, only a local strategy makes sense. In particular, the working set and WSClock algorithms refer to some specific process and must be applied in that context. There really is no working set for the machine as a whole, and trying to use the union of all the working sets would lose the locality property and not work well.

3.5.2 Load Control

Even with the best page replacement algorithm and optimal global allocation of page frames to processes, it can happen that the system thrashes. In fact, whenever the combined working sets of all processes exceed the capacity of memory, thrashing can be expected. One symptom of this situation is that the PFF algorithm indicates that some processes need more memory but no processes need less memory. In this case, there is no way to give more memory to those processes needing it without hurting some other processes. The only real solution is to temporarily get rid of some processes.

A good way to reduce the number of processes competing for memory is to swap some of them to the disk and free up all the pages they are holding. For example, one process can be swapped to disk and its page frames divided up among other processes that are thrashing. If the thrashing stops, the system can run for a while this way. If it does not stop, another process has to be swapped out, and so on, until the thrashing stops. Thus even with paging, swapping may still be needed, only now swapping is used to reduce potential demand for memory, rather than to reclaim pages.

Swapping processes out to relieve the load on memory is reminiscent of two-level scheduling, in which some processes are put on disk and a short-term scheduler is used to schedule the remaining processes. Clearly, the two ideas can be combined, with just enough processes swapped out to make the page-fault rate acceptable. Periodically, some processes are brought in from disk and other ones are swapped out.

However, another factor to consider is the degree of multiprogramming. When the number of processes in main memory is too low, the CPU may be idle for substantial periods of time. This consideration argues for considering not only process size and paging rate when deciding which process to swap out, but also its characteristics, such as whether it is CPU bound or I/O bound, and what characteristics the remaining processes have.

3.5.3 Page Size

The page size is a parameter that can be chosen by the operating system. Even if the hardware has been designed with, for example, 4096-byte pages, the operating system can easily regard page pairs 0 and 1, 2 and 3, 4 and 5, and so on, as 8-KB pages by always allocating two consecutive 8192-byte page frames for them.

Determining the best page size requires balancing several competing factors. As a result, there is no overall optimum. To start with, two factors argue for a small page size. A randomly chosen text, data, or stack segment will not fill an integral number of pages. On the average, half of the final page will be empty. The extra space in that page is wasted. This wastage is called **internal fragmentation**. With n segments in memory and a page size of p bytes, $np/2$ bytes will be wasted on internal fragmentation. This reasoning argues for a small page size.

Another argument for a small page size becomes apparent if we think about a program consisting of eight sequential phases of 4 KB each. With a 32-KB page size, the program must be allocated 32 KB all the time. With a 16-KB page size, it needs only 16 KB. With a page size of 4 KB or smaller, it requires only 4 KB at any instant. In general, a large page size will cause more wasted space to be in memory than a small page size.

On the other hand, small pages mean that programs will need many pages, and thus a large page table. A 32-KB program needs only four 8-KB pages, but 64 512-byte pages. Transfers to and from the disk are generally a page at a time, with most of the time being for the seek and rotational delay, so that transferring a small page takes almost as much time as transferring a large page. It might take 64×10 msec to load 64 512-byte pages, but only 4×12 msec to load four 8-KB pages.

Also, small pages use up much valuable space in the **TLB**. Say your program uses 1 MB of memory with a working set of 64 KB. With 4-KB pages, the program would occupy at least 16 entries in the TLB. With 2-MB pages, a single TLB entry would be sufficient (in theory, it may be that you want to separate data and instructions). As TLB entries are scarce, and critical for performance, it pays to use large pages wherever possible. To balance all these trade-offs, operating systems sometimes use different page sizes for different parts of the system. For instance, large pages for the kernel and smaller ones for user processes.

On some machines, the page table must be loaded (by the operating system) into hardware registers every time the CPU switches from one process to another. On these machines, having a small page size means that the time required to load the page registers gets longer as the page size gets smaller. Furthermore, the space occupied by the page table increases as the page size decreases.

This last point can be analyzed mathematically. Let the average process size be s bytes and the page size be p bytes. Furthermore, assume that each page entry requires e bytes. The approximate number of pages needed per process is then s/p, occupying se/p bytes of page table space. The wasted memory in the last page of the process due to internal fragmentation is $p/2$. Thus, the total overhead due to the page table and the internal fragmentation loss is given by the sum of these two terms:

$$\text{overhead} = se/p + p/2$$

The first term (page table size) is large when the page size is small. The second term (internal fragmentation) is large when the page size is large. The optimum

must lie somewhere in between. By taking the first derivative with respect to p and equating it to zero, we get the equation

$$-se/p^2 + 1/2 = 0$$

From this equation we can derive a formula that gives the optimum page size (considering only memory wasted in fragmentation and page table size). The result is:

$$p = \sqrt{2se}$$

For s = 1MB and e = 8 bytes per page table entry, the optimum page size is 4 KB. Commercially available computers have used page sizes ranging from 512 bytes to 64 KB. A typical value used to be 1 KB, but nowadays 4 KB is more common.

3.5.4 Separate Instruction and Data Spaces

Most computers have a single address space that holds both programs and data, as shown in Fig. 3-24(a). If this address space is large enough, everything works fine. However, if it's too small, it forces programmers to stand on their heads to fit everything into the address space.

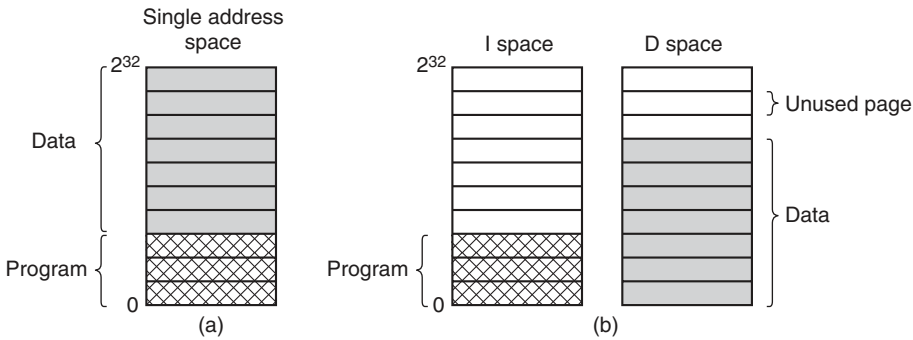

Figure 3-24. (a) One address space. (b) Separate I and D spaces.

One solution, pioneered on the (16-bit) PDP-11, is to have separate address spaces for instructions (program text) and data, called **I-space** and **D-space**, respectively, as illustrated in Fig. 3-24(b). Each address space runs from 0 to some maximum, typically $2^{16} - 1$ or $2^{32} - 1$. The linker must know when separate I- and D-spaces are being used, because when they are, the data are relocated to virtual address 0 instead of starting after the program.

In a computer with this kind of design, both address spaces can be paged, independently from one another. Each one has its own page table, with its own mapping of virtual pages to physical page frames. When the hardware wants to fetch an instruction, it knows that it must use I-space and the I-space page table. Similarly, data must go through the D-space page table. Other than this distinction, having separate I- and D-spaces does not introduce any special complications for the operating system and it does double the available address space.

While address spaces these days are large, their sizes used to be a serious problem. Even today, though, separate I- and D-spaces are still common. However, rather than for the normal address spaces, they are now used to divide the L1 cache. After all, in the L1 cache, memory is still plenty scarce.

3.5.5 Shared Pages

Another design issue is sharing. In a large multiprogramming system, it is common for several users to be running the same program at the same time. Even a single user may be running several programs that use the same library. It is clearly more efficient to share the pages, to avoid having two copies of the same page in memory at the same time. One problem is that not all pages are sharable. In particular, pages that are read-only, such as program text, can be shared, but for data pages sharing is more complicated.

If separate I- and D-spaces are supported, it is relatively straightforward to share programs by having two or more processes use the same page table for their I-space but different page tables for their D-spaces. Typically in an implementation that supports sharing in this way, page tables are data structures independent of the process table. Each process then has two pointers in its process table: one to the I-space page table and one to the D-space page table, as shown in Fig. 3-25. When the scheduler chooses a process to run, it uses these pointers to locate the appropriate page tables and sets up the MMU using them. Even without separate I- and D-spaces, processes can share programs (or sometimes, libraries), but the mechanism is more complicated.

When two or more processes share some code, a problem occurs with the shared pages. Suppose that processes *A* and *B* are both running the editor and sharing its pages. If the scheduler decides to remove *A* from memory, evicting all its pages and filling the empty page frames with some other program will cause *B* to generate a large number of page faults to bring them back in again.

Similarly, when *A* terminates, it is essential to be able to discover that the pages are still in use so that their disk space will not be freed by accident. Searching all the page tables to see if a page is shared is usually too expensive, so special data structures are needed to keep track of shared pages, especially if the unit of sharing is the individual page (or run of pages), rather than an entire page table.

Sharing data is trickier than sharing code, but it is not impossible. In particular, in UNIX, after a fork system call, the parent and child are required to share both program text and data. In a paged system, what is often done is to give each of these processes its own page table and have both of them point to the same set of pages. Thus no copying of pages is done at fork time. However, all the data pages are mapped into both processes as READ ONLY.

As long as both processes just read their data, without modifying it, this situation can continue. As soon as either process updates a memory word, the violation of the read-only protection causes a trap to the operating system. A copy is then

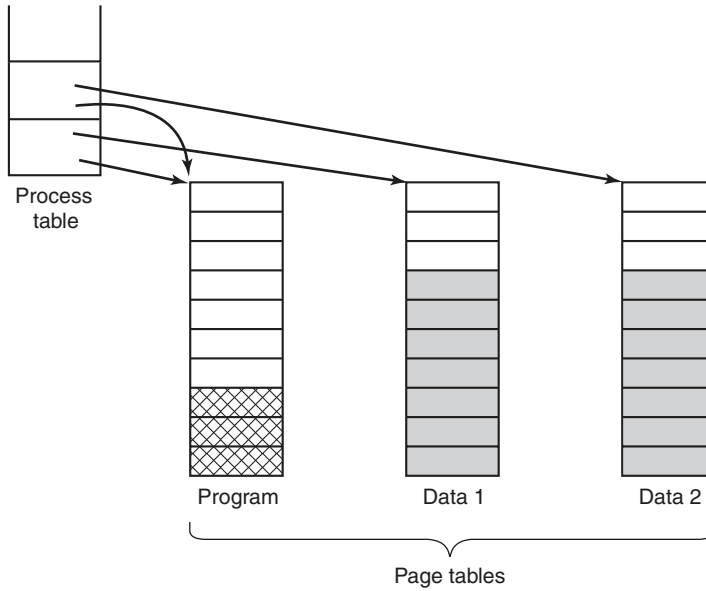

Figure 3-25. Two processes sharing the same program sharing its page tables.

made of the offending page so that each process now has its own private copy. Both copies are now set to READ/WRITE, so subsequent writes to either copy proceed without trapping. This strategy means that those pages that are never modified (including all the program pages) need not be copied. Only the data pages that are actually modified need to be copied. This approach, called **copy on write**, improves performance by reducing copying.

3.5.6 Shared Libraries

Sharing can be done at other granularities than individual pages. If a program is started up twice, most operating systems will automatically share all the text pages so that only one copy is in memory. Text pages are always read only, so there is no problem here. Depending on the operating system, each process may get its own private copy of the data pages, or they may be shared and marked read only. If any process modifies a data page, a private copy will be made for it, that is, copy on write will be applied.

In modern systems, there are many large libraries used by many processes, for example, multiple I/O and graphics libraries. Statically binding all these libraries to every executable program on the disk would make them even more bloated than they already are.

Instead, a common technique is to use **shared libraries** (which are called **DLLs** or **Dynamic Link Libraries** on Windows). To make the idea of a shared

library clear, first consider traditional linking. When a program is linked, one or more object files and possibly some libraries are named in the command to the linker, such as the UNIX command

```
ld *.o –lc –lm
```

which links all the *.o* (object) files in the current directory and then scans two libraries, */usr/lib/libc.a* and */usr/lib/libm.a*. Any functions called in the object files but not present there (e.g., *printf*) are called **undefined externals** and are sought in the libraries. If they are found, they are included in the executable binary. Any functions that they call but are not yet present also become undefined externals. For example, *printf* needs *write*, so if *write* is not already included, the linker will look for it and include it when found. When the linker is done, an executable binary file is written to the disk containing all the functions needed. Functions present in the libraries but not called are not included. When the program is loaded into memory and executed, all the functions it needs are there.

Now suppose common programs use 20–50 MB worth of graphics and user interface functions. Statically linking hundreds of programs with all these libraries would waste a tremendous amount of space on the disk as well as wasting space in RAM when they were loaded since the system would have no way of knowing it could share them. This is where shared libraries come in. When a program is linked with shared libraries (which are slightly different than static ones), instead of including the actual function called, the linker includes a small stub routine that binds to the called function at run time. Depending on the system and the configuration details, shared libraries are loaded either when the program is loaded or when functions in them are called for the first time. Of course, if another program has already loaded the shared library, there is no need to load it again—that is the whole point of it. Note that when a shared library is loaded or used, the entire library is not read into memory in a single blow. It is paged in, page by page, as needed, so functions that are not called will not be brought into RAM.

In addition to making executable files smaller and also saving space in memory, shared libraries have another important advantage: if a function in a shared library is updated to remove a bug, it is not necessary to recompile the programs that call it. The old binaries continue to work. This feature is especially important for commercial software, where the source code is not distributed to the customer. For example, if Microsoft finds and fixes a security error in some standard DLL, *Windows Update* will download the new DLL and replace the old one, and all programs that use the DLL will automatically use the new version the next time they are launched.

Shared libraries come with one little problem, however, that has to be solved, however. The problem is illustrated in Fig. 3-26. Here we see two processes sharing a library of size 20 KB (assuming each box is 4 KB). However, the library is located at a different address in each process, presumably because the programs themselves are not the same size. In process 1, the library starts at address 36K; in

process 2 it starts at 12K. Suppose that the first thing the first function in the library has to do is jump to address 16 in the library. If the library were not shared, it could be relocated on the fly as it was loaded so that the jump (in process 1) could be to virtual address 36K + 16. Note that the physical address in the RAM where the library is located does not matter since all the pages are mapped from virtual to physical addresses by the MMU hardware.

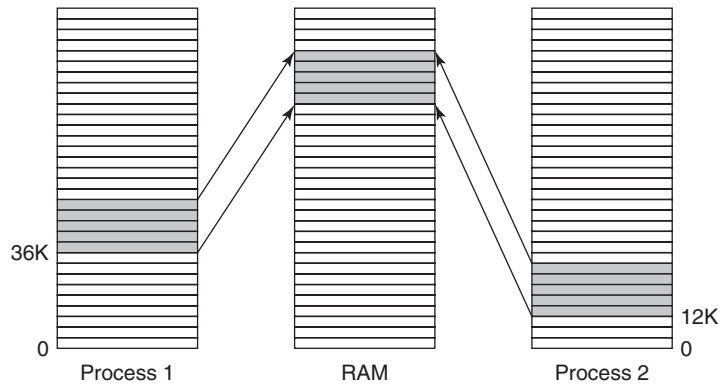

Figure 3-26. A shared library being used by two processes.

However, since the library is shared, relocation on the fly will not work. After all, when the first function is called by process 2 (at address 12K), the jump instruction has to go to 12K + 16, not 36K + 16. This is the little problem. One way to solve it is to use copy on write and create new pages for each process sharing the library, relocating them on the fly as they are created, but this scheme defeats the purpose of sharing the library, of course.

A better solution is to compile shared libraries with a special compiler flag telling the compiler not to produce any instructions that use absolute addresses. Instead only instructions using relative addresses are used. For example, there is almost always an instruction that says jump forward (or backward) by *n* bytes (as opposed to an instruction that gives a specific address to jump to). This instruction works correctly no matter where the shared library is placed in the virtual address space. By avoiding absolute addresses, the problem can be solved. Code that uses only relative offsets is called **position-independent code**.

3.5.7 Mapped Files

Shared libraries are really a special case of a more general facility called **memory-mapped files**. The idea here is that a process can issue a system call to map a file onto a portion of its virtual address space. In most implementations, no pages are brought in at the time of the mapping, but as pages are touched, they are demand paged in one page at a time, using the disk file as the backing store. When

the process exits, or explicitly unmaps the file, all the modified pages are written back to the file on disk.

Mapped files provide an alternative model for I/O. Instead, of doing reads and writes, the file can be accessed as a big character array in memory. In some situations, programmers find this model more convenient.

If two or more processes map onto the same file at the same time, they can communicate over shared memory. Writes done by one process to the shared memory are immediately visible when the other one reads from the part of its virtual address spaced mapped onto the file. This mechanism thus provides a high-bandwidth channel between processes and is often used as such (even to the extent of mapping a scratch file). Now it should be clear that if memory-mapped files are available, shared libraries can use this mechanism.

3.5.8 Cleaning Policy

Paging works best when there is an abundant supply of free page frames that can be claimed as page faults occur. If every page frame is full, and furthermore modified, before a new page can be brought in, an old page must first be written to disk. To ensure a plentiful supply of free page frames, paging systems generally have a background process, called the **paging daemon**, that sleeps most of the time but is awakened periodically to inspect the state of memory. If too few page frames are free, it begins selecting pages to evict using some page replacement algorithm. If these pages have been modified since being loaded, they are written to disk.

In any event, the previous contents of the page are remembered. In the event one of the evicted pages is needed again before its frame has been overwritten, it can be reclaimed by removing it from the pool of free page frames. Keeping a supply of page frames around yields better performance than using all of memory and then trying to find a frame at the moment it is needed. At the very least, the paging daemon ensures that all the free frames are clean, so they need not be written to disk in a big hurry when they are required.

One way to implement this cleaning policy is with a two-handed clock. The front hand is controlled by the paging daemon. When it points to a dirty page, that page is written back to disk and the front hand is advanced. When it points to a clean page, it is just advanced. The back hand is used for page replacement, as in the standard clock algorithm. Only now, the probability of the back hand hitting a clean page is increased due to the work of the paging daemon.

3.5.9 Virtual Memory Interface

Up until now, our whole discussion has assumed that virtual memory is transparent to processes and programmers, that is, all they see is a large virtual address space on a computer with a small(er) physical memory. With many systems,

that is true, but in some advanced systems, programmers have some control over the memory map and can use it in nontraditional ways to enhance program behavior. In this section, we will briefly look at a few of these.

One reason for giving programmers control over their memory map is to allow two or more processes to share the same memory. sometimes in sophisticated ways. If programmers can name regions of their memory, it may be possible for one process to give another process the name of a memory region so that process can also map it in. With two (or more) processes sharing the same pages, high bandwidth sharing becomes possible—one process writes into the shared memory and another one reads from it. A sophisticated example of such a communication channel is described by De Bruijn (2011).

Sharing of pages can also be used to implement a high-performance message-passing system. Normally, when messages are passed, the data are copied from one address space to another, at considerable cost. If processes can control their page map, a message can be passed by having the sending process unmap the page(s) containing the message, and the receiving process mapping them in. Here only the page names have to be copied, instead of all the data.

Yet another advanced memory management technique is **distributed shared memory** (Feeley et al., 1995; Li, 1986; Li and Hudak, 1989; and Zekauskas et al., 1994). The idea here is to allow multiple processes over a network to share a set of pages, possibly, but not necessarily, as a single shared linear address space. When a process references a page that is not currently mapped in, it gets a page fault. The page fault handler, which may be in the kernel or in user space, then locates the machine holding the page and sends it a message asking it to unmap the page and send it over the network. When the page arrives, it is mapped in and the faulting instruction is restarted. We will examine distributed shared memory in Chap. 8.

3.6 IMPLEMENTATION ISSUES

Implementers of virtual memory systems have to make choices among the major theoretical algorithms, such as second chance versus aging, local versus global page allocation, and demand paging versus prepaging. But they also have to be aware of a number of practical implementation issues as well. In this section we will take a look at a few of the common problems and some solutions.

3.6.1 Operating System Involvement with Paging

There are four times when the operating system has paging-related work to do: process creation time, process execution time, page fault time, and process termination time. We will now briefly examine each of these to see what has to be done.

When a new process is created in a paging system, the operating system has to determine how large the program and data will be (initially) and create a page table

for them. Space has to be allocated in memory for the page table and it has to be initialized. The page table need not be resident when the process is swapped out but has to be in memory when the process is running. In addition, space has to be allocated in the swap area on disk so that when a page is swapped out, it has somewhere to go. The swap area also has to be initialized with program text and data so that when the new process starts getting page faults, the pages can be brought in. Some systems page the program text directly from the executable file, thus saving disk space and initialization time. Finally, information about the page table and swap area on disk must be recorded in the process table.

When a process is scheduled for execution, the MMU has to be reset for the new process and the TLB flushed, to get rid of traces of the previously executing process. The new process' page table has to be made current, usually by copying it or a pointer to it to some hardware register(s). Optionally, some or all of the process' pages can be brought into memory to reduce the number of page faults initially (e.g., it is certain that the page pointed to by the program counter will be needed).

When a page fault occurs, the operating system has to read out hardware registers to determine which virtual address caused the fault. From this information, it must compute which page is needed and locate that page on disk. It must then find an available page frame in which to put the new page, evicting some old page if need be. Then it must read the needed page into the page frame. Finally, it must back up the program counter to have it point to the faulting instruction and let that instruction execute again.

When a process exits, the operating system must release its page table, its pages, and the disk space that the pages occupy when they are on disk. If some of the pages are shared with other processes, the pages in memory and on disk can be released only when the last process using them has terminated.

3.6.2 Page Fault Handling

We are finally in a position to describe in detail what happens on a page fault. The sequence of events is as follows:

1. The hardware traps to the kernel, saving the program counter on the stack. On most machines, some information about the state of the current instruction is saved in special CPU registers.
2. An assembly-code routine is started to save the general registers and other volatile information, to keep the operating system from destroying it. This routine calls the operating system as a procedure.
3. The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed. Often one of the hardware registers contains this information. If not, the operating system

must retrieve the program counter, fetch the instruction, and parse it in software to figure out what it was doing when the fault hit.

4. Once the virtual address that caused the fault is known, the system checks to see if this address is valid and the protection is consistent with the access. If not, the process is sent a signal or killed. If the address is valid and no protection fault has occurred, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to select a victim.
5. If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place, suspending the faulting process and letting another one run until the disk transfer has completed. In any event, the frame is marked as busy to prevent it from being used for another purpose.
6. As soon as the page frame is clean (either immediately or after it is written to disk), the operating system looks up the disk address where the needed page is, and schedules a disk operation to bring it in. While the page is being loaded, the faulting process is still suspended and another user process is run, if one is available.
7. When the disk interrupt indicates that the page has arrived, the page tables are updated to reflect its position, and the frame is marked as being in the normal state.
8. The faulting instruction is backed up to the state it had when it began and the program counter is reset to point to that instruction.
9. The faulting process is scheduled, and the operating system returns to the (assembly-language) routine that called it.
10. This routine reloads the registers and other state information and returns to user space to continue execution, as if no fault had occurred.

3.6.3 Instruction Backup

When a program references a page that is not in memory, the instruction causing the fault is stopped partway through and a trap to the operating system occurs. After the operating system has fetched the page needed, it must restart the instruction causing the trap. This is easier said than done.

To see the nature of this problem at its worst, consider a CPU that has instructions with two addresses, such as the Motorola 680x0, widely used in embedded systems. The instruction

```
MOV.L #6(A1),2(A0)
```

is 6 bytes, for example (see Fig. 3-27). In order to restart the instruction, the operating system must determine where the first byte of the instruction is. The value of the program counter at the time of the trap depends on which operand faulted and how the CPU's microcode has been implemented.

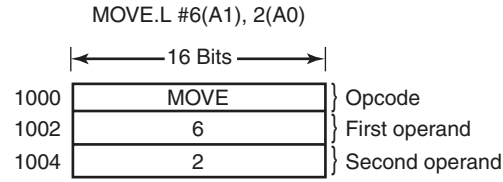

Figure 3-27. An instruction causing a page fault.

In Fig. 3-27, we have an instruction starting at address 1000 that makes three memory references: the instruction word and two offsets for the operands. Depending on which of these three memory references caused the page fault, the program counter might be 1000, 1002, or 1004 at the time of the fault. It is frequently impossible for the operating system to determine unambiguously where the instruction began. If the program counter is 1002 at the time of the fault, the operating system has no way of telling whether the word in 1002 is a memory address associated with an instruction at 1000 (e.g., the address of an operand) or an opcode.

Bad as this problem may be, it could have been worse. Some 680x0 addressing modes use autoincrementing, which means that a side effect of executing the instruction is to increment one (or more) registers. Instructions that use autoincrement mode can also fault. Depending on the details of the microcode, the increment may be done before the memory reference, in which case the operating system must decrement the register in software before restarting the instruction. Or, the autoincrement may be done after the memory reference, in which case it will not have been done at the time of the trap and must not be undone by the operating system. Autodecrement mode also exists and causes a similar problem. The precise details of whether autoincrements and autodecrements have or have not been done before the corresponding memory references may differ from instruction to instruction and from CPU model to CPU model.

Fortunately, on some machines the CPU designers provide a solution, usually in the form of a hidden internal register into which the program counter is copied just before each instruction is executed. These machines may also have a second register telling which registers have already been autoincremented or autodecremented, and by how much. Given this information, the operating system can unambiguously undo all the effects of the faulting instruction so that it can be restarted. If this information is not available, the operating system has to jump through hoops to figure out what happened and how to repair it. It is as though the hardware designers were unable to solve the problem, so they threw up their hands and told the operating system writers to deal with it. Nice guys.

3.6.4 Locking Pages in Memory

Although we have not discussed I/O much in this chapter, the fact that a computer has virtual memory does not mean that I/O is absent. Virtual memory and I/O interact in subtle ways. Consider a process that has just issued a system call to read from some file or device into a buffer within its address space. While waiting for the I/O to complete, the process is suspended and another process is allowed to run. This other process gets a page fault.

If the paging algorithm is global, there is a small, but nonzero, chance that the page containing the I/O buffer will be chosen to be removed from memory. If an I/O device is currently in the process of doing a DMA transfer to that page, removing it will cause part of the data to be written in the buffer where they belong, and part of the data to be written over the just-loaded page. One solution to this problem is to lock pages engaged in I/O in memory so that they will not be removed. Locking a page is often called **pinning** it in memory. Another solution is to do all I/O to kernel buffers and then copy the data to user pages later.

3.6.5 Backing Store

In our discussion of page replacement algorithms, we saw how a page is selected for removal. We have not said much about where on the disk it is put when it is paged out. Let us now describe some of the issues related to disk management.

The simplest algorithm for allocating page space on the disk is to have a special swap partition on the disk or, even better, on a separate disk from the file system (to balance the I/O load). Most UNIX systems work like this. This partition does not have a normal file system on it, which eliminates all the overhead of converting offsets in files to block addresses. Instead, block numbers relative to the start of the partition are used throughout.

When the system is booted, this swap partition is empty and is represented in memory as a single entry giving its origin and size. In the simplest scheme, when the first process is started, a chunk of the partition area the size of the first process is reserved and the remaining area reduced by that amount. As new processes are started, they are assigned chunks of the swap partition equal in size to their core images. As they finish, their disk space is freed. The swap partition is managed as a list of free chunks. Better algorithms will be discussed in Chap. 10.

Associated with each process is the disk address of its swap area, that is, where on the swap partition its image is kept. This information is kept in the process table. Calculating the address to write a page to becomes simple: just add the offset of the page within the virtual address space to the start of the swap area. However, before a process can start, the swap area must be initialized. One way is to copy the entire process image to the swap area, so that it can be brought *in* as needed. The other is to load the entire process in memory and let it be paged *out* as needed.

However, this simple model has a problem: processes can increase in size after starting. Although the program text is usually fixed, the data area can sometimes grow, and the stack can always grow. Consequently, it may be better to reserve separate swap areas for the text, data, and stack and allow each of these areas to consist of more than one chunk on the disk.

The other extreme is to allocate nothing in advance and allocate disk space for each page when it is swapped out and deallocate it when it is swapped back in. In this way, processes in memory do not tie up any swap space. The disadvantage is that a disk address is needed in memory to keep track of each page on disk. In other words, there must be a table per process telling for each page on disk where it is. The two alternatives are shown in Fig. 3-28.

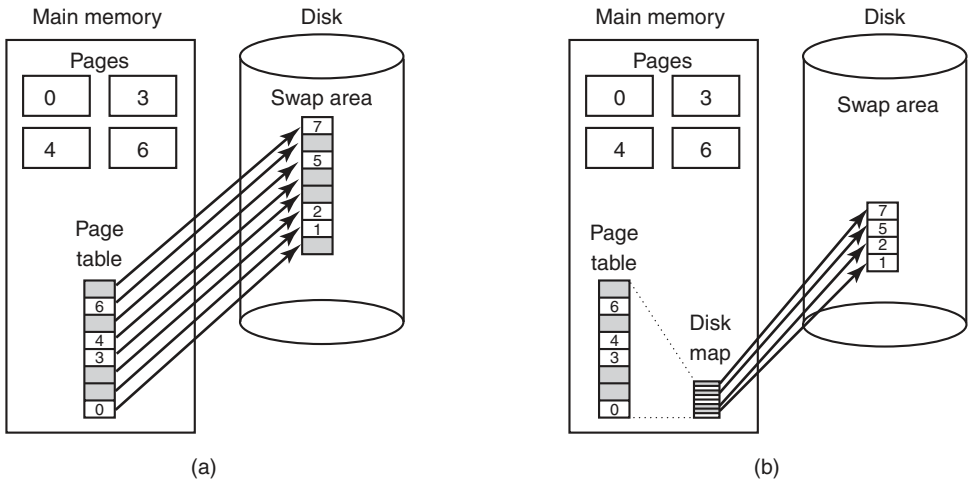

Figure 3-28. (a) Paging to a static swap area. (b) Backing up pages dynamically.

In Fig. 3-28(a), a page table with eight pages is shown. Pages 0, 3, 4, and 6 are in main memory. Pages 1, 2, 5, and 7 are on disk. The swap area on disk is as large as the process virtual address space (eight pages), with each page having a fixed location to which it is written when it is evicted from main memory. Calculating this address requires knowing only where the process' paging area begins, since pages are stored in it contiguously in order of their virtual page number. A page that is in memory always has a shadow copy on disk, but this copy may be out of date if the page has been modified since being loaded. The shaded pages in memory indicate pages not present in memory. The shaded pages on the disk are (in principle) superseded by the copies in memory, although if a memory page has to be swapped back to disk and it has not been modified since it was loaded, the (shaded) disk copy will be used.

In Fig. 3-28(b), pages do not have fixed addresses on disk. When a page is swapped out, an empty disk page is chosen on the fly and the disk map (which has

room for one disk address per virtual page) is updated accordingly. A page in memory has no copy on disk. The pages' entries in the disk map contain an invalid disk address or a bit marking them as not in use.

Having a fixed swap partition is not always possible. For example, no disk partitions may be available. In this case, one or more large, preallocated files within the normal file system can be used. Windows uses this approach. However, an optimization can be used here to reduce the amount of disk space needed. Since the program text of every process came from some (executable) file in the file system, the executable file can be used as the swap area. Better yet, since the program text is generally read only, when memory is tight and program pages have to be evicted from memory, they are just discarded and read in again from the executable file when needed. Shared libraries can also work this way.

3.6.6 Separation of Policy and Mechanism

An important tool for managing the complexity of any system is to split policy from mechanism. This principle can be applied to memory management by having most of the memory manager run as a user-level process. Such a separation was first done in Mach (Young et al., 1987) on which the discussion below is based.

A simple example of how policy and mechanism can be separated is shown in Fig. 3-29. Here the memory management system is divided into three parts:

1. A low-level MMU handler.
2. A page fault handler that is part of the kernel.
3. An external pager running in user space.

All the details of how the MMU works are encapsulated in the MMU handler, which is machine-dependent code and has to be rewritten for each new platform the operating system is ported to. The page-fault handler is machine-independent code and contains most of the mechanism for paging. The policy is largely determined by the external pager, which runs as a user process.

When a process starts up, the external pager is notified in order to set up the process' page map and allocate the necessary backing store on the disk if need be. As the process runs, it may map new objects into its address space, so the external pager is once again notified.

Once the process starts running, it may get a page fault. The fault handler figures out which virtual page is needed and sends a message to the external pager, telling it the problem. The external pager then reads the needed page in from the disk and copies it to a portion of its own address space. Then it tells the fault handler where the page is. The fault handler then unmaps the page from the external pager's address space and asks the MMU handler to put it into the user's address space at the right place. Then the user process can be restarted.

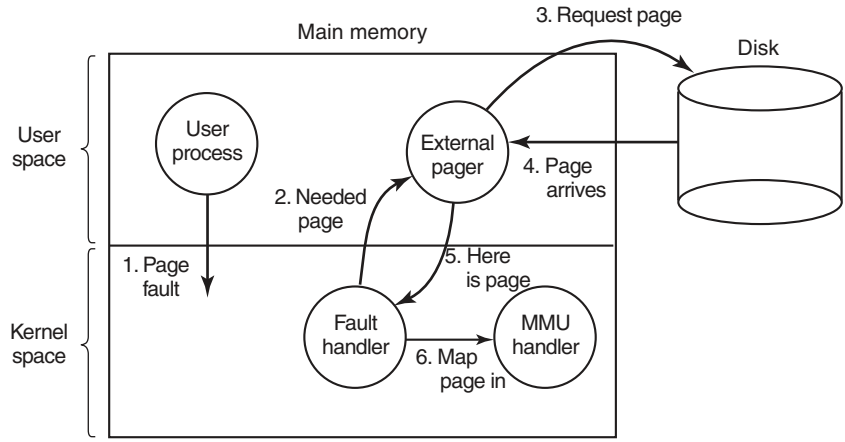

Figure 3-29. Page fault handling with an external pager.

This implementation leaves open where the page replacement algorithm is put. It would be cleanest to have it in the external pager, but there are some problems with this approach. Principal among these is that the external pager does not have access to the *R* and *M* bits of all the pages. These bits play a role in many of the paging algorithms. Thus, either some mechanism is needed to pass this information up to the external pager, or the page replacement algorithm must go in the kernel. In the latter case, the fault handler tells the external pager which page it has selected for eviction and provides the data, either by mapping it into the external pager's address space or including it in a message. Either way, the external pager writes the data to disk.

The main advantage of this implementation is more modular code and greater flexibility. The main disadvantage is the extra overhead of crossing the user-kernel boundary several times and the overhead of the various messages being sent between the pieces of the system. At the moment, the subject is highly controversial, but as computers get faster and faster, and the software gets more and more complex, in the long run sacrificing some performance for more reliable software will probably be acceptable to most implementers.

3.7 SEGMENTATION

The virtual memory discussed so far is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another. For many problems, having two or more separate virtual address spaces may be much better than having only one. For example, a compiler has many tables that are built up as compilation proceeds, possibly including

1. The source text being saved for the printed listing (on batch systems).
2. The symbol table, containing the names and attributes of variables.
3. The table containing all the integer and floating-point constants used.
4. The parse tree, containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation. In a one-dimensional memory, these five tables would have to be allocated contiguous chunks of virtual address space, as in Fig. 3-30.

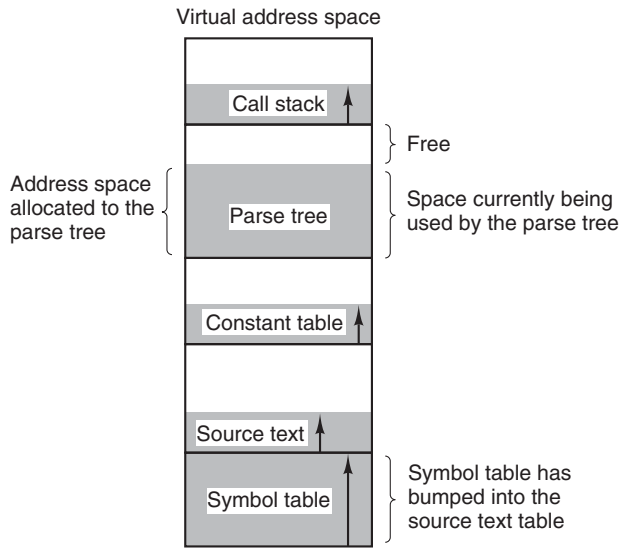

Figure 3-30. In a one-dimensional address space with growing tables, one table may bump into another.

Consider what happens if a program has a much larger than usual number of variables but a normal amount of everything else. The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables. What is needed is a way of freeing the programmer from having to manage the expanding and contracting tables, in the same way that virtual memory eliminates the worry of organizing the program into overlays.

A straightforward and quite general solution is to provide the machine with many completely independent address spaces, which are called **segments**. Each segment consists of a linear sequence of addresses, starting at 0 and going up to some maximum value. The length of each segment may be anything from 0 to the

maximum address allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack.

Because each segment constitutes a separate address space, different segments can grow or shrink independently without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up, but segments are usually very large, so this occurrence is rare. To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment. Figure 3-31 illustrates a segmented memory being used for the compiler tables discussed earlier. Five independent segments are shown here.

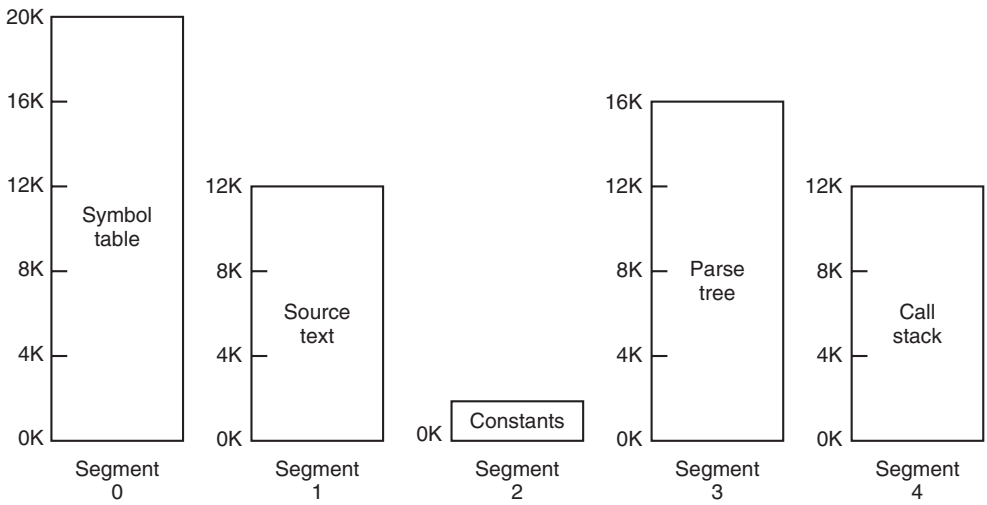

Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

We emphasize here that a segment is a logical entity, which the programmer is aware of and uses as a logical entity. A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.

A segmented memory has other advantages besides simplifying the handling of data structures that are growing or shrinking. If each procedure occupies a separate segment, with address 0 as its starting address, the linking of procedures compiled separately is greatly simplified. After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment n will use the two-part address $(n, 0)$ to address word 0 (the entry point).

If the procedure in segment *n* is subsequently modified and recompiled, no other procedures need be changed (because no starting addresses have been modified), even if the new version is larger than the old one. With a one-dimensional memory, the procedures are packed tightly right up next to each other, with no address space between them. Consequently, changing one procedure's size can affect the starting address of all the other (unrelated) procedures in the segment. This, in turn, requires modifying all procedures that call any of the moved procedures, in order to incorporate their new starting addresses. If a program contains hundreds of procedures, this process can be costly.

Segmentation also facilitates sharing procedures or data between several processes. A common example is the shared library. Modern workstations that run advanced window systems often have extremely large graphical libraries compiled into nearly every program. In a segmented system, the graphical library can be put in a segment and shared by multiple processes, eliminating the need for having it in every process' address space. While it is also possible to have shared libraries in pure paging systems, it is more complicated. In effect, these systems do it by simulating segmentation.

Since each segment forms a logical entity that programmers know about, such as a procedure, or an array, different segments can have different kinds of protection. A procedure segment can be specified as execute only, prohibiting attempts to read from or store into it. A floating-point array can be specified as read/write but not execute, and attempts to jump to it will be caught. Such protection is helpful in catching bugs. Paging and segmentation are compared in Fig. 3-32.

3.7.1 Implementation of Pure Segmentation

The implementation of segmentation differs from paging in an essential way: pages are of fixed size and segments are not. Figure 3-33(a) shows an example of physical memory initially containing five segments. Now consider what happens if segment 1 is evicted and segment 7, which is smaller, is put in its place. We arrive at the memory configuration of Fig. 3-33(b). Between segment 7 and segment 2 is an unused area—that is, a hole. Then segment 4 is replaced by segment 5, as in Fig. 3-33(c), and segment 3 is replaced by segment 6, as in Fig. 3-33(d). After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This phenomenon, called **checkerboarding** or **external fragmentation**, wastes memory in the holes. It can be dealt with by compaction, as shown in Fig. 3-33(e).

3.7.2 Segmentation with Paging: MULTICS

If the segments are large, it may be inconvenient, or even impossible, to keep them in main memory in their entirety. This leads to the idea of paging them, so that only those pages of a segment that are actually needed have to be around.

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Figure 3-32. Comparison of paging and segmentation.

Several significant systems have supported paged segments. In this section we will describe the first one: MULTICS. In the next one we will discuss a more recent one: the Intel x86 up until the x86-64.

The MULTICS operating system was one of the most influential operating systems ever, having had a major influence on topics as disparate as UNIX, the x86 memory architecture, TLBs, and cloud computing. It was started as a research project at M.I.T. and went live in 1969. The last MULTICS system was shut down in 2000, a run of 31 years. Few other operating systems have lasted more-or-less unmodified anywhere near that long. While operating systems called Windows have also have be around that long, Windows 8 has absolutely nothing in common with Windows 1.0 except the name and the fact that it was written by Microsoft. Even more to the point, the ideas developed in MULTICS are as valid and useful now as they were in 1965, when the first paper was published (Corbató and Vyssotsky, 1965). For this reason, we will now spend a little bit of time looking at the most innovative aspect of MULTICS, the virtual memory architecture. More information about MULTICS can be found at *www.multicians.org*.

MULTICS ran on the Honeywell 6000 machines and their descendants and provided each program with a virtual memory of up to 2^{18} segments, each of which

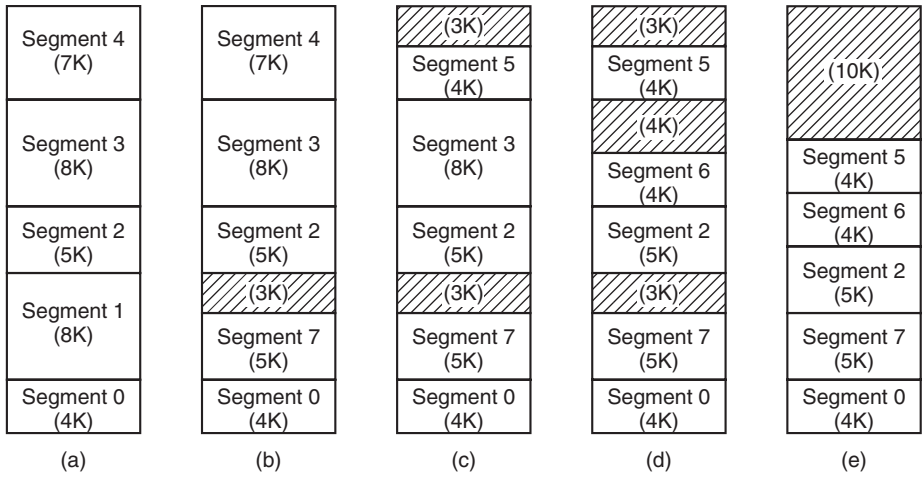

Figure 3-33. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

was up to 65,536 (36-bit) words long. To implement this, the MULTICS designers chose to treat each segment as a virtual memory and to page it, combining the advantages of paging (uniform page size and not having to keep the whole segment in memory if only part of it was being used) with the advantages of segmentation (ease of programming, modularity, protection, sharing).

Each MULTICS program had a segment table, with one descriptor per segment. Since there were potentially more than a quarter of a million entries in the table, the segment table was itself a segment and was paged. A segment descriptor contained an indication of whether the segment was in main memory or not. If any part of the segment was in memory, the segment was considered to be in memory, and its page table was in memory. If the segment was in memory, its descriptor contained an 18-bit pointer to its page table, as in Fig. 3-34(a). Because physical addresses were 24 bits and pages were aligned on 64-byte boundaries (implying that the low-order 6 bits of page addresses were 000000), only 18 bits were needed in the descriptor to store a page table address. The descriptor also contained the segment size, the protection bits, and other items. Figure 3-34(b) illustrates a segment descriptor. The address of the segment in secondary memory was not in the segment descriptor but in another table used by the segment fault handler.

Each segment was an ordinary virtual address space and was paged in the same way as the nonsegmented paged memory described earlier in this chapter. The normal page size was 1024 words (although a few small segments used by MULTICS itself were not paged or were paged in units of 64 words to save physical memory).

An address in MULTICS consisted of two parts: the segment and the address within the segment. The address within the segment was further divided into a page

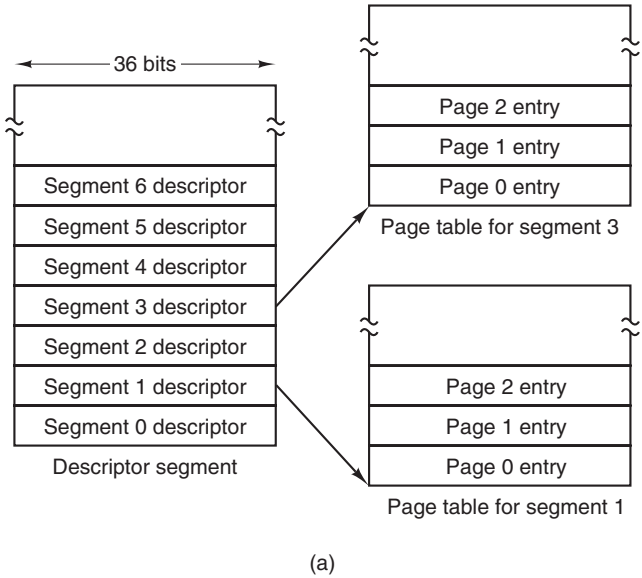

Figure 3-34. The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables. (b) A segment descriptor. The numbers are the field lengths.

number and a word within the page, as shown in Fig. 3-35. When a memory reference occurred, the following algorithm was carried out.

1. The segment number was used to find the segment descriptor.
2. A check was made to see if the segment's page table was in memory. If it was, it was located. If it was not, a segment fault occurred. If there was a protection violation, a fault (trap) occurred.

3. The page table entry for the requested virtual page was examined. If the page itself was not in memory, a page fault was triggered. If it was in memory, the main-memory address of the start of the page was extracted from the page table entry.
4. The offset was added to the page origin to give the main memory address where the word was located.
5. The read or store finally took place.

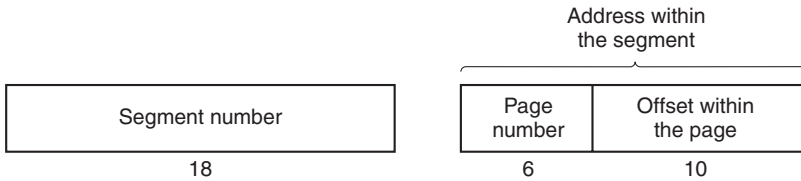

Figure 3-35. A 34-bit MULTICS virtual address.

This process is illustrated in Fig. 3-36. For simplicity, the fact that the descriptor segment was itself paged has been omitted. What really happened was that a register (the descriptor base register) was used to locate the descriptor segment's page table, which, in turn, pointed to the pages of the descriptor segment. Once the descriptor for the needed segment was been found, the addressing proceeded as shown in Fig. 3-36.

As you have no doubt guessed by now, if the preceding algorithm were actually carried out by the operating system on every instruction, programs would not run very fast. In reality, the MULTICS hardware contained a 16-word high-speed TLB that could search all its entries in parallel for a given key. This was the first system to have a TLB, something used in all modern architectures. It is illustrated in Fig. 3-37. When an address was presented to the computer, the addressing hardware first checked to see if the virtual address was in the TLB. If so, it got the page frame number directly from the TLB and formed the actual address of the referenced word without having to look in the descriptor segment or page table.

The addresses of the 16 most recently referenced pages were kept in the TLB. Programs whose working set was smaller than the TLB size came to equilibrium with the addresses of the entire working set in the TLB and therefore ran efficiently; otherwise, there were TLB faults.

3.7.3 Segmentation with Paging: The Intel x86

Up until the x86-64, the virtual memory system of the x86 resembled that of MULTICS in many ways, including the presence of both segmentation and paging. Whereas MULTICS had 256K independent segments, each up to 64K 36-bit words, the x86 has 16K independent segments, each holding up to 1 billion 32-bit

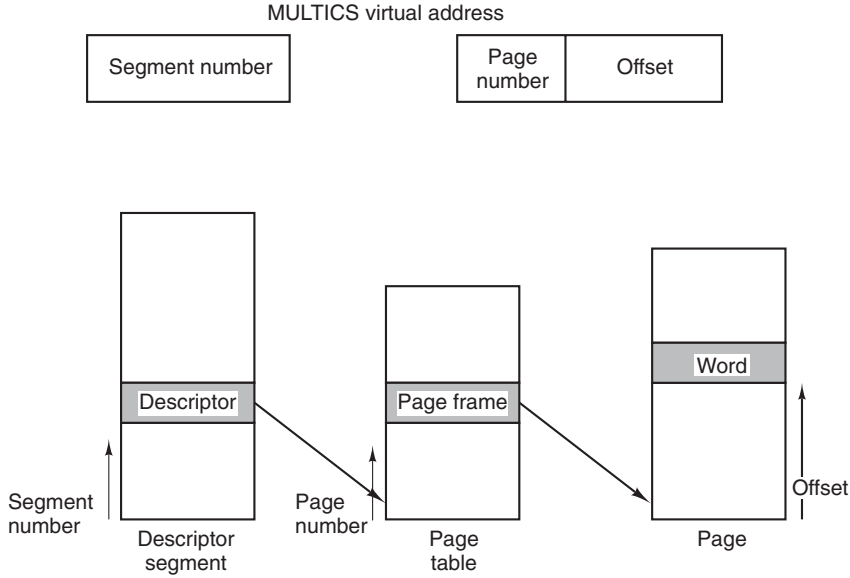

Figure 3-36. Conversion of a two-part MULTICS address into a main memory address.

Comparison field: Segment number	Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figure 3-37. A simplified version of the MULTICS TLB. The existence of two page sizes made the actual TLB more complicated.

words. Although there are fewer segments, the larger segment size is far more important, as few programs need more than 1000 segments, but many programs need large segments. As of x86-64, segmentation is considered obsolete and is no longer supported, except in legacy mode. Although some vestiges of the old segmentation

mechanisms are still available in x86-64's native mode, mostly for compatibility, they no longer serve the same role and no longer offer true segmentation. The x86-32, however, still comes equipped with the whole shebang and it is the CPU we will discuss in this section.

The heart of the x86 virtual memory consists of two tables, called the **LDT** (**Local Descriptor Table**) and the **GDT** (**Global Descriptor Table**). Each program has its own LDT, but there is a single GDT, shared by all the programs on the computer. The LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

To access a segment, an x86 program first loads a selector for that segment into one of the machine's six segment registers. During execution, the CS register holds the selector for the code segment and the DS register holds the selector for the data segment. The other segment registers are less important. Each selector is a 16-bit number, as shown in Fig. 3-38.

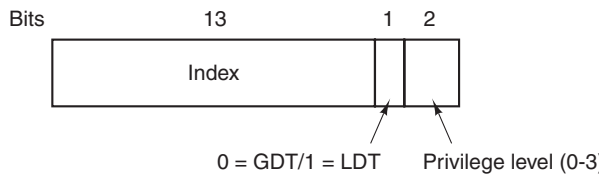

Figure 3-38. An x86 selector.

One of the selector bits tells whether the segment is local or global (i.e., whether it is in the LDT or GDT). Thirteen other bits specify the LDT or GDT entry number, so these tables are each restricted to holding 8K segment descriptors. The other 2 bits relate to protection, and will be described later. Descriptor 0 is forbidden. It may be safely loaded into a segment register to indicate that the segment register is not currently available. It causes a trap if used.

At the time a selector is loaded into a segment register, the corresponding descriptor is fetched from the LDT or GDT and stored in microprogram registers, so it can be accessed quickly. As depicted in Fig. 3-39, a descriptor consists of 8 bytes, including the segment's base address, size, and other information.

The format of the selector has been cleverly chosen to make locating the descriptor easy. First either the LDT or GDT is selected, based on selector bit 2. Then the selector is copied to an internal scratch register, and the 3 low-order bits set to 0. Finally, the address of either the LDT or GDT table is added to it, to give a direct pointer to the descriptor. For example, selector 72 refers to entry 9 in the GDT, which is located at address GDT + 72.

Let us now trace the steps by which a (selector, offset) pair is converted to a physical address. As soon as the microprogram knows which segment register is

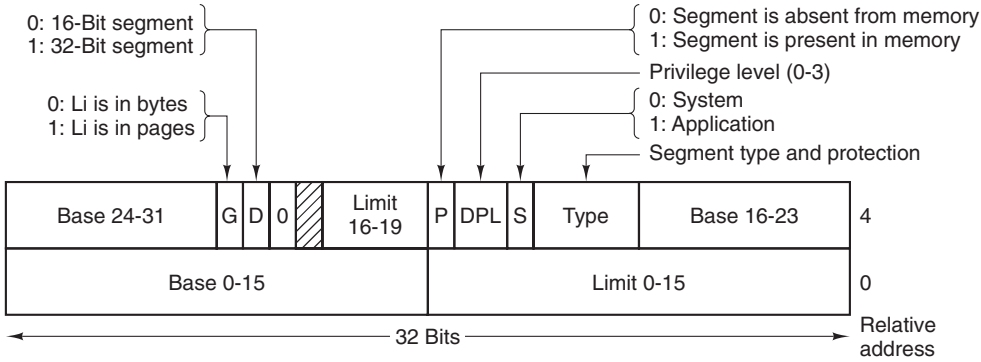

Figure 3-39. x86 code segment descriptor. Data segments differ slightly.

being used, it can find the complete descriptor corresponding to that selector in its internal registers. If the segment does not exist (selector 0), or is currently paged out, a trap occurs.

The hardware then uses the *Limit* field to check if the offset is beyond the end of the segment, in which case a trap also occurs. Logically, there should be a 32-bit field in the descriptor giving the size of the segment, but only 20 bits are available, so a different scheme is used. If the *Gbit* (Granularity) field is 0, the *Limit* field is the exact segment size, up to 1 MB. If it is 1, the *Limit* field gives the segment size in pages instead of bytes. With a page size of 4 KB, 20 bits are enough for segments up to 2^{32} bytes.

Assuming that the segment is in memory and the offset is in range, the x86 then adds the 32-bit *Base* field in the descriptor to the offset to form what is called a **linear address**, as shown in Fig. 3-40. The *Base* field is broken up into three pieces and spread all over the descriptor for compatibility with the 286, in which the *Base* is only 24 bits. In effect, the *Base* field allows each segment to start at an arbitrary place within the 32-bit linear address space.

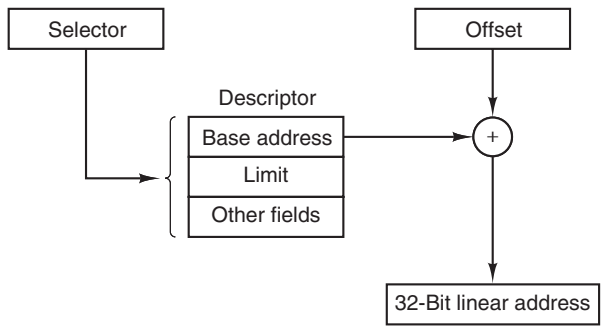

Figure 3-40. Conversion of a (selector, offset) pair to a linear address.

If paging is disabled (by a bit in a global control register), the linear address is interpreted as the physical address and sent to the memory for the read or write. Thus with paging disabled, we have a pure segmentation scheme, with each segment's base address given in its descriptor. Segments are not prevented from overlapping, probably because it would be too much trouble and take too much time to verify that they were all disjoint.

On the other hand, if paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address using page tables, pretty much as in our earlier examples. The only real complication is that with a 32-bit virtual address and a 4-KB page, a segment might contain 1 million pages, so a two-level mapping is used to reduce the page table size for small segments.

Each running program has a page directory consisting of 1024 32-bit entries. It is located at an address pointed to by a global register. Each entry in this directory points to a page table also containing 1024 32-bit entries. The page table entries point to page frames. The scheme is shown in Fig. 3-41.

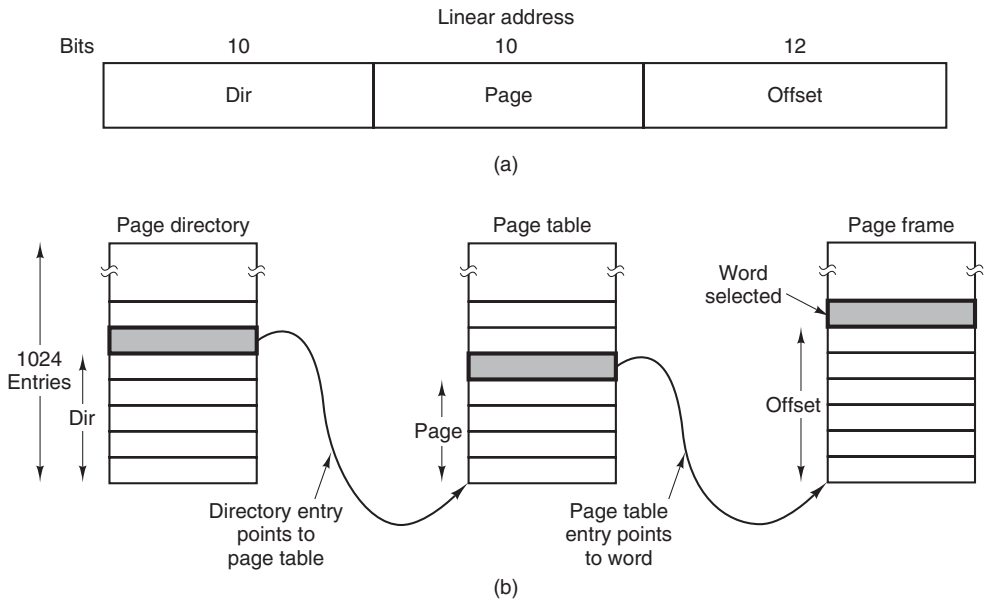

Figure 3-41. Mapping of a linear address onto a physical address.

In Fig. 3-41(a) we see a linear address divided into three fields, *Dir*, *Page*, and *Offset*. The *Dir* field is used to index into the page directory to locate a pointer to the proper page table. Then the *Page* field is used as an index into the page table to find the physical address of the page frame. Finally, *Offset* is added to the address of the page frame to get the physical address of the byte or word needed.

The page table entries are 32 bits each, 20 of which contain a page frame number. The remaining bits contain access and dirty bits, set by the hardware for the benefit of the operating system, protection bits, and other utility bits.

Each page table has entries for 1024 4-KB page frames, so a single page table handles 4 megabytes of memory. A segment shorter than 4M will have a page directory with a single entry, a pointer to its one and only page table. In this way, the overhead for short segments is only two pages, instead of the million pages that would be needed in a one-level page table.

To avoid making repeated references to memory, the x86, like MULTICS, has a small TLB that directly maps the most recently used *Dir-Page* combinations onto the physical address of the page frame. Only when the current combination is not present in the TLB is the mechanism of Fig. 3-41 actually carried out and the TLB updated. As long as TLB misses are rare, performance is good.

It is also worth noting that if some application does not need segmentation but is simply content with a single, paged, 32-bit address space, that model is possible. All the segment registers can be set up with the same selector, whose descriptor has *Base* = 0 and *Limit* set to the maximum. The instruction offset will then be the linear address, with only a single address space used—in effect, normal paging. In fact, all current operating systems for the x86 work this way. OS/2 was the only one that used the full power of the Intel MMU architecture.

So why did Intel kill what was a variant of the perfectly good MULTICS memory model that it supported for close to three decades? Probably the main reason is that neither UNIX nor Windows ever used it, even though it was quite efficient because it eliminated system calls, turning them into lightning-fast procedure calls to the relevant address within a protected operating system segment. None of the developers of any UNIX or Windows system wanted to change their memory model to something that was x86 specific because it would break portability to other platforms. Since the software was not using the feature, Intel got tired of wasting chip area to support it and removed it from the 64-bit CPUs.

All in all, one has to give credit to the x86 designers. Given the conflicting goals of implementing pure paging, pure segmentation, and paged segments, while at the same time being compatible with the 286, and doing all of this efficiently, the resulting design is surprisingly simple and clean.

3.8 RESEARCH ON MEMORY MANAGEMENT

Traditional memory management, especially paging algorithms for uniprocessor CPUs, was once a fruitful area for research, but most of that seems to have largely died off, at least for general-purpose systems, although there are some people who never say die (Moruz et al., 2012) or are focused on some application, such as online transaction processing, that has specialized requirements (Stoica and Ailamaki, 2013). Even on uniprocessors, paging to SSDs rather than to hard disks brings up new issues and requires new algorithms (Chen et al., 2012). Paging to the up-and-coming nonvolatile phase-change memories also requires rethinking

paging for performance (Lee et al., 2013), and latency reasons (Saito and Oikawa, 2012), and because they wear out if used too much (Bheda et al., 2011, 2012).

More generally, research on paging is still ongoing, but it focuses on newer kinds of systems. For example, virtual machines have rekindled interest in memory management (Bugnion et al., 2012). In the same area, the work by Jantz et al. (2013) lets applications provide guidance to the system with respect to deciding on the physical page to back a virtual page. An aspect of server consolidation in the cloud that affects paging is that the amount of physical memory available to a virtual machine can vary over time, requiring new algorithms (Peserico, 2013).

Paging in multicore systems has become a hot new area of research (Boyd-Wickizer et al., 2008, Baumann et al., 2009). One contributing factor is that multicore systems tend to have a lot of caches shared in complex ways (Lopez-Ortiz and Salinger, 2012). Closely related to this multicore work is research on paging in NUMA systems, where different pieces of memory may have different access times (Dashti et al., 2013; and Lankes et al., 2012).

Also, smartphones and tablets have become small PCs and many of them page RAM to "disk," only disk on a smartphone is flash memory. Some recent work is reported by Joo et al. (2012).

Finally, interest is memory management for real-time systems continues to be present (Kato et al., 2011).

3.9 SUMMARY

In this chapter we have examined memory management. We saw that the simplest systems do not swap or page at all. Once a program is loaded into memory, it remains there in place until it finishes. Some operating systems allow only one process at a time in memory, while others support multiprogramming. This model is still common in small, embedded real-time systems.

The next step up is swapping. When swapping is used, the system can handle more processes than it has room for in memory. Processes for which there is no room are swapped out to the disk. Free space in memory and on disk can be kept track of with a bitmap or a hole list.

Modern computers often have some form of virtual memory. In the simplest form, each process' address space is divided up into uniform-sized blocks called pages, which can be placed into any available page frame in memory. There are many page replacement algorithms; two of the better algorithms are aging and WSClock.

To make paging systems work well, choosing an algorithm is not enough; attention to such issues as determining the working set, memory allocation policy, and page size is required.

Segmentation helps in handling data structures that can change size during execution and simplifies linking and sharing. It also facilitates providing different

protection for different segments. Sometimes segmentation and paging are combined to provide a two-dimensional virtual memory. The MULTICS system and the 32-bit Intel x86 support segmentation and paging. Still, it is clear that few operating system developers care deeply about segmentation (because they are married to a different memory model). Consequently, it seems to be going out of fashion fast. Today, even the 64-bit version of the x86 no longer supports real segmentation.

PROBLEMS

1. The IBM 360 had a scheme of locking 2-KB blocks by assigning each one a 4-bit key and having the CPU compare the key on every memory reference to the 4-bit key in the PSW. Name two drawbacks of this scheme not mentioned in the text.
2. In Fig. 3-3 the base and limit registers contain the same value, 16,384. Is this just an accident, or are they always the same? It is just an accident, why are they the same in this example?
3. A swapping system eliminates holes by compaction. Assuming a random distribution of many holes and many data segments and a time to read or write a 32-bit memory word of 4 nsec, about how long does it take to compact 4 GB? For simplicity, assume that word 0 is part of a hole and that the highest word in memory contains valid data.
4. Consider a swapping system in which memory consists of the following hole sizes in memory order: 10 MB, 4 MB, 20 MB, 18 MB, 7 MB, 9 MB, 12 MB, and 15 MB. Which hole is taken for successive segment requests of

   (a) 12 MB
   (b) 10 MB
   (c) 9 MB

   for first fit? Now repeat the question for best fit, worst fit, and next fit.
5. What is the difference between a physical address and a virtual address?
6. For each of the following decimal virtual addresses, compute the virtual page number and offset for a 4-KB page and for an 8 KB page: 20000, 32768, 60000.
7. Using the page table of Fig. 3-9, give the physical address corresponding to each of the following virtual addresses:

   (a) 20
   (b) 4100
   (c) 8300
8. The Intel 8086 processor did not have an MMU or support virtual memory. Nevertheless, some companies sold systems that contained an unmodified 8086 CPU and did paging. Make an educated guess as to how they did it. (*Hint*: Think about the logical location of the MMU.)

9. What kind of hardware support is needed for a paged virtual memory to work?

10. Copy on write is an interesting idea used on server systems. Does it make any sense on a smartphone?

11. Consider the following C program:

```
int    X[N];
int step = M;      /* M is some predefined constant */
for (int i = 0; i < N; i += step) X[i] = X[i] + 1;
```

(a) If this program is run on a machine with a 4-KB page size and 64-entry TLB, what values of *M* and *N* will cause a TLB miss for every execution of the inner loop?
(b) Would your answer in part (a) be different if the loop were repeated many times? Explain.

12. The amount of disk space that must be available for page storage is related to the maximum number of processes, *n*, the number of bytes in the virtual address space, *v*, and the number of bytes of RAM, *r*. Give an expression for the worst-case disk-space requirements. How realistic is this amount?

13. If an instruction takes 1 nsec and a page fault takes an additional *n* nsec, give a formula for the effective instruction time if page faults occur every *k* instructions.

14. A machine has a 32-bit address space and an 8-KB page. The page table is entirely in hardware, with one 32-bit word per entry. When a process starts, the page table is copied to the hardware from memory, at one word every 100 nsec. If each process runs for 100 msec (including the time to load the page table), what fraction of the CPU time is devoted to loading the page tables?

15. Suppose that a machine has 48-bit virtual addresses and 32-bit physical addresses.

(a) If pages are 4 KB, how many entries are in the page table if it has only a single level? Explain.
(b) Suppose this same system has a TLB (Translation Lookaside Buffer) with 32 entries. Furthermore, suppose that a program contains instructions that fit into one page and it sequentially reads long integer elements from an array that spans thousands of pages. How effective will the TLB be for this case?

16. You are given the following data about a virtual memory system:

(a)The TLB can hold 1024 entries and can be accessed in 1 clock cycle (1 nsec).
(b) A page table entry can be found in 100 clock cycles or 100 nsec.
(c) The average page replacement time is 6 msec.

If page references are handled by the TLB 99% of the time, and only 0.01% lead to a page fault, what is the effective address-translation time?

17. Suppose that a machine has 38-bit virtual addresses and 32-bit physical addresses.

(a) What is the main advantage of a multilevel page table over a single-level one?
(b) With a two-level page table, 16-KB pages, and 4-byte entries, how many bits should be allocated for the top-level page table field and how many for the next-level page table field? Explain.

18. Section 3.3.4 states that the Pentium Pro extended each entry in the page table hierarchy to 64 bits but still could only address only 4 GB of memory. Explain how this statement can be true when page table entries have 64 bits.

19. A computer with a 32-bit address uses a two-level page table. Virtual addresses are split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset. How large are the pages and how many are there in the address space?

20. A computer has 32-bit virtual addresses and 4-KB pages. The program and data together fit in the lowest page (0–4095) The stack fits in the highest page. How many entries are needed in the page table if traditional (one-level) paging is used? How many page table entries are needed for two-level paging, with 10 bits in each part?

21. Below is an execution trace of a program fragment for a computer with 512-byte pages. The program is located at address 1020, and its stack pointer is at 8192 (the stack grows toward 0). Give the page reference string generated by this program. Each instruction occupies 4 bytes (1 word) including immediate constants. Both instruction and data references count in the reference string.

```
Load word 6144 into register 0
Push register 0 onto the stack
Call a procedure at 5120, stacking the return address
Subtract the immediate constant 16 from the stack pointer
Compare the actual parameter to the immediate constant 4
Jump if equal to 5152
```

22. A computer whose processes have 1024 pages in their address spaces keeps its page tables in memory. The overhead required for reading a word from the page table is 5 nsec. To reduce this overhead, the computer has a TLB, which holds 32 (virtual page, physical page frame) pairs, and can do a lookup in 1 nsec. What hit rate is needed to reduce the mean overhead to 2 nsec?

23. How can the associative memory device needed for a TLB be implemented in hardware, and what are the implications of such a design for expandability?

24. A machine has 48-bit virtual addresses and 32-bit physical addresses. Pages are 8 KB. How many entries are needed for a single-level linear page table?

25. A computer with an 8-KB page, a 256-KB main memory, and a 64-GB virtual address space uses an inverted page table to implement its virtual memory. How big should the hash table be to ensure a mean hash chain length of less than 1? Assume that the hash-table size is a power of two.

26. A student in a compiler design course proposes to the professor a project of writing a compiler that will produce a list of page references that can be used to implement the optimal page replacement algorithm. Is this possible? Why or why not? Is there anything that could be done to improve paging efficiency at run time?

27. Suppose that the virtual page reference stream contains repetitions of long sequences of page references followed occasionally by a random page reference. For example, the sequence: 0, 1, ... , 511, 431, 0, 1, ... , 511, 332, 0, 1, ... consists of repetitions of the sequence 0, 1, ... , 511 followed by a random reference to pages 431 and 332.

(a) Why will the standard replacement algorithms (LRU, FIFO, clock) not be effective in handling this workload for a page allocation that is less than the sequence length?
(b) If this program were allocated 500 page frames, describe a page replacement approach that would perform much better than the LRU, FIFO, or clock algorithms.

28. If FIFO page replacement is used with four page frames and eight pages, how many page faults will occur with the reference string 0172327103 if the four frames are initially empty? Now repeat this problem for LRU.

29. Consider the page sequence of Fig. 3-15(b). Suppose that the *R* bits for the pages *B* through *A* are 11011011, respectively. Which page will second chance remove?

30. A small computer on a smart card has four page frames. At the first clock tick, the *R* bits are 0111 (page 0 is 0, the rest are 1). At subsequent clock ticks, the values are 1011, 1010, 1101, 0010, 1010, 1100, and 0001. If the aging algorithm is used with an 8-bit counter, give the values of the four counters after the last tick.

31. Give a simple example of a page reference sequence where the first page selected for replacement will be different for the clock and LRU page replacement algorithms. Assume that a process is allocated 3=three frames, and the reference string contains page numbers from the set 0, 1, 2, 3.

32. In the WSClock algorithm of Fig. 3-20(c), the hand points to a page with $R = 0$. If $\tau = 400$, will this page be removed? What about if $\tau = 1000$?

33. Suppose that the WSClock page replacement algorithm uses a τ of two ticks, and the system state is the following:

Page	Time stamp	V	R	M
0	6	1	0	1
1	9	1	1	0
2	9	1	1	1
3	7	1	0	0
4	4	0	0	0

where the three flag bits *V*, R, and *M* stand for Valid, Referenced, and Modified, respectively.

(a) If a clock interrupt occurs at tick 10, show the contents of the new table entries. Explain. (You can omit entries that are unchanged.)
(b) Suppose that instead of a clock interrupt, a page fault occurs at tick 10 due to a read request to page 4. Show the contents of the new table entries. Explain. (You can omit entries that are unchanged.)

34. A student has claimed that "in the abstract, the basic page replacement algorithms (FIFO, LRU, optimal) are identical except for the attribute used for selecting the page to be replaced."

(a) What is that attribute for the FIFO algorithm? LRU algorithm? Optimal algorithm?
(b) Give the generic algorithm for these page replacement algorithms.

35. How long does it take to load a 64-KB program from a disk whose average seek time is 5 msec, whose rotation time is 5 msec, and whose tracks hold 1 MB

(a) for a 2-KB page size?
(b) for a 4-KB page size?

The pages are spread randomly around the disk and the number of cylinders is so large that the chance of two pages being on the same cylinder is negligible.

36. A computer has four page frames. The time of loading, time of last access, and the *R* and *M* bits for each page are as shown below (the times are in clock ticks):

Page	Loaded	Last ref.	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

(a) Which page will NRU replace?
(b) Which page will FIFO replace?
(c) Which page will LRU replace?
(d) Which page will second chance replace?

37. Suppose that two processes *A* and *B* share a page that is not in memory. If process *A* faults on the shared page, the page table entry for process *A* must be updated once the page is read into memory.

(a) Under what conditions should the page table update for process *B* be delayed even though the handling of process *A*'s page fault will bring the shared page into memory? Explain.
(b) What is the potential cost of delaying the page table update?

38. Consider the following two-dimensional array:

```
int  X[64][64];
```

Suppose that a system has four page frames and each frame is 128 words (an integer occupies one word). Programs that manipulate the *X* array fit into exactly one page and always occupy page 0. The data are swapped in and out of the other three frames. The *X* array is stored in row-major order (i.e., *X*[0][1] follows *X*[0][0] in memory). Which of the two code fragments shown below will generate the lowest number of page faults? Explain and compute the total number of page faults.

Fragment A

```
for (int j = 0; j < 64; j++)
  for (int i = 0; i < 64; i++) X[i][j] = 0;
```

Fragment B

```
for (int i = 0; i < 64; i++)
  for (int j = 0; j < 64; j++) X[i][j] = 0;
```

39. You have been hired by a cloud computing company that deploys thousands of servers at each of its data centers. They have recently heard that it would be worthwhile to handle a page fault at server A by reading the page from the RAM memory of some other server rather than its local disk drive.

(a) How could that be done?
(b) Under what conditions would the approach be worthwhile? Be feasible?

40. One of the first timesharing machines, the DEC PDP-1, had a (core) memory of 4K 18-bit words. It held one process at a time in its memory. When the scheduler decided to run another process, the process in memory was written to a paging drum, with 4K 18-bit words around the circumference of the drum. The drum could start writing (or reading) at any word, rather than only at word 0. Why do you suppose this drum was chosen?

41. A computer provides each process with 65,536 bytes of address space divided into pages of 4096 bytes each. A particular program has a text size of 32,768 bytes, a data size of 16,386 bytes, and a stack size of 15,870 bytes. Will this program fit in the machine's address space? Suppose that instead of 4096 bytes, the page size were 512 bytes, would it then fit? Each page must contain either text, data, or stack, not a mixture of two or three of them.

42. It has been observed that the number of instructions executed between page faults is directly proportional to the number of page frames allocated to a program. If the available memory is doubled, the mean interval between page faults is also doubled. Suppose that a normal instruction takes 1 microsec, but if a page fault occurs, it takes 2001 μsec (i.e., 2 msec) to handle the fault. If a program takes 60 sec to run, during which time it gets 15,000 page faults, how long would it take to run if twice as much memory were available?

43. A group of operating system designers for the Frugal Computer Company are thinking about ways to reduce the amount of backing store needed in their new operating system. The head guru has just suggested not bothering to save the program text in the swap area at all, but just page it in directly from the binary file whenever it is needed. Under what conditions, if any, does this idea work for the program text? Under what conditions, if any, does it work for the data?

44. A machine-language instruction to load a 32-bit word into a register contains the 32-bit address of the word to be loaded. What is the maximum number of page faults this instruction can cause?

45. Explain the difference between internal fragmentation and external fragmentation. Which one occurs in paging systems? Which one occurs in systems using pure segmentation?

46. When segmentation and paging are both being used, as in MULTICS, first the segment descriptor must be looked up, then the page descriptor. Does the TLB also work this way, with two levels of lookup?

47. We consider a program which has the two segments shown below consisting of instructions in segment 0, and read/write data in segment 1. Segment 0 has read/execute protection, and segment 1 has just read/write protection. The memory system is a demand-

paged virtual memory system with virtual addresses that have a 4-bit page number, and a 10-bit offset. The page tables and protection are as follows (all numbers in the table are in decimal):

Segment 0		Segment 1	
Read/Execute		Read/Write	
Virtual Page #	Page frame #	Virtual Page #	Page frame #
0	2	0	On Disk
1	On Disk	1	14
2	11	2	9
3	5	3	6
4	On Disk	4	On Disk
5	On Disk	5	13
6	4	6	8
7	3	7	12

For each of the following cases, either give the real (actual) memory address which results from dynamic address translation or identify the type of fault which occurs (either page or protection fault).

(a) Fetch from segment 1, page 1, offset 3
(b) Store into segment 0, page 0, offset 16
(c) Fetch from segment 1, page 4, offset 28
(d) Jump to location in segment 1, page 3, offset 32

48. Can you think of any situations where supporting virtual memory would be a bad idea, and what would be gained by not having to support virtual memory? Explain.

49. Virtual memory provides a mechanism for isolating one process from another. What memory management difficulties would be involved in allowing two operating systems to run concurrently? How might these difficulties be addressed?

50. Plot a histogram and calculate the mean and median of the sizes of executable binary files on a computer to which you have access. On a Windows system, look at all .exe and .dll files; on a UNIX system look at all executable files in */bin*, */usr/bin*, and */local/bin* that are not scripts (or use the *file* utility to find all executables). Determine the optimal page size for this computer just considering the code (not data). Consider internal fragmentation and page table size, making some reasonable assumption about the size of a page table entry. Assume that all programs are equally likely to be run and thus should be weighted equally.

51. Write a program that simulates a paging system using the aging algorithm. The number of page frames is a parameter. The sequence of page references should be read from a file. For a given input file, plot the number of page faults per 1000 memory references as a function of the number of page frames available.

52. Write a program that simulates a toy paging system that uses the WSClock algorithm. The system is a toy in that we will assume there are no write references (not very

realistic), and process termination and creation are ignored (eternal life). The inputs will be:

- The reclamation age threshhold
- The clock interrupt interval expressed as number of memory references
- A file containing the sequence of page references

(a) Describe the basic data structures and algorithms in your implementation.
(b) Show that your simulation behaves as expected for a simple (but nontrivial) input example.
(c) Plot the number of page faults and working set size per 1000 memory references.
(d) Explain what is needed to extend the program to handle a page reference stream that also includes writes.

53. Write a program that demonstrates the effect of TLB misses on the effective memory access time by measuring the per-access time it takes to stride through a large array.

(a) Explain the main concepts behind the program, and describe what you expect the output to show for some practical virtual memory architecture.
(b) Run the program on some computer and explain how well the data fit your expectations.
(c) Repeat part (b) but for an older computer with a different architecture and explain any major differences in the output.

54. Write a program that will demonstrate the difference between using a local page replacement policy and a global one for the simple case of two processes. You will need a routine that can generate a page reference string based on a statistical model. This model has N states numbered from 0 to $N - 1$ representing each of the possible page references and a probability p_i associated with each state i representing the chance that the next reference is to the same page. Otherwise, the next page reference will be one of the other pages with equal probability.

(a) Demonstrate that the page reference string-generation routine behaves properly for some small N.
(b) Compute the page fault rate for a small example in which there is one process and a fixed number of page frames. Explain why the behavior is correct.
(c) Repeat part (b) with two processes with independent page reference sequences and twice as many page frames as in part (b).
(d) Repeat part (c) but using a global policy instead of a local one. Also, contrast the per-process page fault rate with that of the local policy approach.

55. Write a program that can be used to compare the effectiveness of adding a tag field to TLB entries when control is toggled between two programs. The tag field is used to effectively label each entry with the process id. Note that a nontagged TLB can be simulated by requiring that all TLB entries have the same tag at any one time. The inputs will be:

- The number of TLB entries available
- The clock interrupt interval expressed as number of memory references
- A file containing a sequence of (process, page references) entries
- The cost to update one TLB entry

(a) Describe the basic data structures and algorithms in your implementation.
b) Show that your simulation behaves as expected for a simple (but nontrivial) input example.
(c) Plot the number of TLB updates per 1000 references.