MEMORY MANAGEMENT

Memory is an important resource that must be carefully managed. While the average home computer nowadays has fifty times as much memory as the IBM 7094, the largest computer in the world in the early 1960s, programs are getting bigger just as fast as 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 manage their memory.

Ideally, what every programmer would like is an infinitely large, fast memory that is also nonvolatile, that is, does not lose its contents when the electric power fails. While we are at it, why not also ask for it to be inexpensive, too. Unfortunately, technology does not provide such memories. Consequently, most computers have a memory hierarchy, with a small amount of very fast, expensive, volatile cache memory, some number of megabytes of medium-speed, medium-price, volatile main memory (RAM), and hundreds or thousands of megabytes of slow, cheap, nonvolatile disk storage. It is the job of the operating system to coordinate how these memories are used.

The part of the operating system that manages the memory hierarchy is called the memory manager. Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and deallocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.

In this chapter we will investigate a number of different memory management schemes, ranging from very simple to highly sophisticated. We will start at the
beginning and look first at the simplest possible memory management system and then gradually progress to more and more elaborate ones.

4.1 BASIC MEMORY MANAGEMENT

Memory management systems can be divided into two classes: those that move processes back and forth between main memory and disk during execution (swapping and paging), and those that do not. The latter are simpler, so we will study them first. Later in the chapter we will examine swapping and paging. Throughout this chapter the reader should keep in mind that swapping and paging are largely artifacts caused by the lack of sufficient main memory to hold all the programs at once. As main memory gets cheaper, the arguments in favor of one kind of memory management scheme or another may become obsolete—unless programs get bigger faster than memory gets cheaper.

4.1.1 Monoprogramming without Swapping or Paging

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system. Three variations on this theme are shown in Fig. 4-1. The operating system may be at the bottom of memory in RAM (Random Access Memory), as shown in Fig. 4-1(a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Fig. 4-1(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. 4-1(c). The latter model is used by small MS DOS systems, for example. On IBM PCs, the portion of the system in the ROM is called the BIOS (Basic Input Output System).

![Figure 4-1](image)

Figure 4-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

When the system is organized in this way, only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program into memory, executes it, and then terminates it. When the user needs yet another program, the operating system terminates the current one, copies the new one, and begins running it. We call this monoprogramming, because we can run only one program at a time.

4.1.2 Multi-programming

Although this approach is simple and efficient, it does have certain drawbacks. On a single processor, we can run only one program at a time. This means that the user will see a response to each command, but we do not. We use the CPU for only a few milliseconds at a time, then it is handed over to the next program, and so forth. In this way, we can run many programs at once.

The execution of each program is divided into small units, called tasks, into $n$ (possibly many) tasks. This is done manually we use a scheduler.

When the system is organized in this way, only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program into memory, executes it, and then terminates it. When the user needs yet another program, the operating system terminates the current one, copies the new one, and begins running it. We call this monoprogramming, because we can run only one program at a time.
requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a new command. When it receives the command, it loads a new program into memory, overwriting the first one.

### 4.1.2 Multiprogramming with Fixed Partitions

Although monoprogramming is sometimes used on small computers with simple operating systems, often it is desirable to allow multiple processes to run at once. On timesharing systems, having multiple processes in memory at once means that when one process is blocked waiting for I/O to finish, another one can use the CPU. Thus multiprogramming increases the CPU utilization. However, even on personal computers it is often useful to be able to run two or more programs at once.

The easiest way to achieve multiprogramming is simply to divide memory up into $n$ (possibly unequal) partitions. This partitioning can, for example, be done manually when the system is started up.

When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Since the partitions are fixed in this scheme, any space in a partition not used by a job is lost. In Fig. 4-2(a) we see how this system of fixed partitions and separate input queues looks.

![Figure 4-2](image)

**Figure 4-2.** (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue for a large partition is empty but the queue for a small
partition is full, as is the case for partitions 1 and 3 in Fig. 4-2(a). An alternative organization is to maintain a single queue as in Fig. 4-2(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run. Since it is undesirable to waste a large partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits. Note that the latter algorithm discriminates against small jobs as being unworthy of having a whole partition, whereas usually it is desirable to give the smallest jobs (assumed to be interactive jobs) the best service, not the worst.

One way out is to have at least one small partition around. Such a partition will allow small jobs to run without having to allocate a large partition for them.

Another approach is to have a rule stating that a job that is eligible to run may not be skipped over more than \( k \) times. Each time it is skipped over, it gets one point. When it has acquired \( k \) points, it may not be skipped again.

This system, with fixed partitions set up by the operator in the morning and not changed thereafter, was used by OS/360 on large IBM mainframes for many years. It was called MFT (Multiprogramming with a Fixed number of Tasks or OS/MFT). It is simple to understand and equally simple to implement: incoming jobs are queued until a suitable partition is available, at which time the job is loaded into that partition and run until it terminates. Nowadays, few, if any, operating systems, support this model.

Relocation and Protection

Multiprogramming introduces two essential problems that must be solved—relocation and protection. Look at Fig. 4-2. From the figure it is clear that different jobs will be run at different addresses. When a program is linked (i.e., the main program, user-written procedures, and library procedures are combined into a single address space), the linker must know at what address the program will begin in memory.

For example, suppose that the first instruction is a call to a procedure at absolute address 100 within the binary file produced by the linker. If this program is loaded in partition 1, that instruction will jump to absolute address 100, which is inside the operating system. What is needed is a call to 100K + 100. If the program is loaded into partition 2, it must be carried out as a call to 200K + 100, and so on. This problem is known as the relocation problem.

One possible solution is to actually modify the instructions as the program is loaded into memory. Programs loaded into partition 1 have 100K added to each address, programs loaded into partition 2 have 200K added to addresses, and so forth. To perform relocation during loading like this, the linker must include in the binary program a list or bit map telling which program words are addresses to be relocated and which are opcodes, constants, or other items that must not be relocated. OS/MFT worked this way. Some microcomputers also work like this.

4.2 SWAP

With OS/MFT, when a process is not effective, it can be swapped out. It stays there until needed again. Swapping has some drawbacks—more complex circuitry is needed—and swapping does not help in the situation where the machine is not big enough to run all programs in at once. Two techniques (discussed in more detail in the next chapter) on
Relocation during loading does not solve the protection problem. A malicious program can always construct a new instruction and jump to it. Because programs in this system use absolute memory addresses rather than addresses relative to a register, there is no way to stop a program from building an instruction that reads or writes any word in memory. In multiuser systems, it is undesirable to let processes read and write memory belonging to other users.

The solution that IBM chose for protecting the 360 was to divide memory into blocks of 2K bytes and assign a 4-bit protection code to each block. The PSW contained a 4-bit key. The 360 hardware trapped any attempt by a running process to access memory whose protection code differed from the PSW key. Since only the operating system could change the protection codes and key, user processes were prevented from interfering with one another and with the operating system itself.

An alternative solution to both the relocation and protection problems is to equip the machine with two special hardware registers, called the base and limit registers. When a process is scheduled, the base register is loaded with the address of the start of its partition, and the limit register is loaded with the length of the partition. Every memory address generated automatically has the base register contents added to it before being sent to memory. Thus if the base register is 100K, a CALL 100 instruction is effectively turned into a CALL 100K + 100 instruction, without the instruction itself being modified. Addresses are also checked against the limit register to make sure that they do not attempt to address memory outside the current partition. The hardware protects the base and limit registers to prevent user programs from modifying them.

The CDC 6600—the world's first supercomputer—used this scheme. The Intel 8088 CPU used for the original IBM PC used a weaker version of this scheme—base registers, but no limit registers. Starting with the 286, a better scheme was adopted.

### 4.2 SWAPPING

With a batch system, organizing memory into fixed partitions is simple and effective. Each job is loaded into a partition when it gets to the head of the queue. It stays in memory until it has finished. As long as enough jobs can be kept in memory to keep the CPU busy all the time, there is no reason to use anything more complicated.

With timesharing systems or graphically oriented personal computers, the situation is different. Sometimes there is not enough main memory to hold all the currently active processes, so excess processes must be kept on disk and brought in to run dynamically.

Two general approaches to memory management can be used, depending (in part) on the available hardware. 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. The other strategy, called virtual memory, allows programs to run even when they are only partially in main memory. Below we will studying swapping; in Sec. 4-3 we will examine virtual memory.

The operation of a swapping system is illustrated in Fig. 4-3. Initially only process A is in memory. Then processes B and C are created or swapped in from disk. In Fig. 4-3(d) A terminates or is swapped out to disk. Then D comes in and B goes out. Finally E comes in.

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

The main difference between the fixed partitions of Fig. 4-2 and the variable partitions of Fig. 4-3 is that the number, location, and size of the partitions vary dynamically in the latter as processes come and go, whereas they are fixed in the former. The flexibility of not being tied to a fixed number of partitions that may be too large or too small improves memory utilization, but it also complicates allocating and deallocating memory, as well as keeping track of it.

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 32-MB machine that can copy 16 bytes per microsecond, it takes 2 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: you allocate 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 expanded, but if it is not, the process will have to wait or be swapped out to disk, or a new hole has to be found and memory expanded to fill it.

If processes are designed with this possibility in mind, it is a good idea to allocate them more space when they are first moved, because if they are not allocated enough space to expand, they will be swapped out to disk, or their data segment will be expanded as needed. If processes have to be swapped out, either must be properly written to make efficient use of the space for processes being removed.
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 wait or 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. 4-4(a) we see a memory configuration in which space for growth has been allocated to two processes.

Figure 4-4. (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. 4-4(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, either the process will have to be moved to a hole with enough space, swapped out of memory until a large enough hole can be created, or killed.
4.2.1 Memory Management with Bit Maps

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

With a bit map, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bit map, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 4-5 shows part of memory and the corresponding bit map.

![Diagram](image)

**Figure 4-5.** (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 bit map) are free. (b) The corresponding bit map. (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 bit map. 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 32\(n\) bits will use \(n\) map bits, so the bit map will take up only 1/33 of memory. If the allocation unit is chosen large, the bit map will be smaller, but appreciable memory may be wasted in the last unit if the process size is not an exact multiple of the allocation unit.

A bit map provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bit map depends only on the size of memory and the size of the allocation unit. The main problem with it is that when it has been decided to bring a \(k\) unit process into memory, the memory manager must search the bit map to find a run of \(k\) consecutive 0 bits in the map. Searching a bit map 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 bit maps.
4.2.2 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 is either a process or a hole between two processes. The memory of Fig. 4-5(a) is represented in Fig. 4-5(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 entry.

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 of Fig. 4-6. In Fig. 4-6(a) updating the list requires replacing a P by an H. In Fig. 4-6(b) and Fig. 4-6(c), two entries are coalesced into one, and the list becomes one entry shorter. In Fig. 4-6(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. 4-5(c). This structure makes it easier to find the previous entry and to see if a merge is possible.

![Figure 4-6. Four neighbor combinations for the terminating process, X.](image)

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created or swapped in process. 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 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 algorithm is best fit. Best fit searches the entire list 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.

As an example of first fit and best fit, consider Fig. 4-5 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 hole broken off 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. 4-5(c), the holes themselves can be used. 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. 4-5(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 4K holes, the second entry is a pointer to a list of 8K holes, the third entry a pointer to 12K holes, and so on. Holes of say, 21K, could either be put on the 20K 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 is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.
4.3 VIRTUAL MEMORY

Many years ago people were first confronted with programs that were too big to fit in the available memory. The solution usually adopted was to split the program into pieces, called overlays. Overlay 0 would start running first. When it was done, it would call another overlay. Some overlay systems were highly complex, allowing multiple overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the operating system, dynamically, as needed.

Although the actual work of swapping overlays in and out was done by the system, the work of splitting the program into pieces had to be done by the programmer. Splitting up large programs into small, modular pieces was time-consuming and boring. 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 the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk. For example, a 16M program can run on a 4M machine by carefully choosing which 4M to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.

Virtual memory can also work in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for part of itself to be brought in, it is waiting for I/O and cannot run, so the CPU can be given to another process, the same way as for any other multiprogramming system.

4.3.1 Paging

Most virtual memory systems use a technique called paging, which we will now describe. On any computer, there exists a set of memory addresses that programs can produce. When a program uses an instruction like

```
MOVE REG, 1000
```

it is copying 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.

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 a Memory Management Unit (MMU), a chip or collection of chips that maps the virtual addresses onto the physical memory addresses as illustrated in Fig. 4-7.

![Diagram](image)

**Figure 4-7.** The position and function of the MMU.

A very simple example of how this mapping works is shown in Fig. 4-8. In this example, we have a computer that can generate 16-bit addresses, from 0 up to 64K. These are the virtual addresses. This computer, however, has only 32K of physical memory, so although 64K 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 64K, must be present on the disk, however, so that pieces can be brought in as needed.

The virtual address space is divided up into units called **pages**. The corresponding units in the physical memory are called **page frames**. The pages and page frames are always exactly the same size. In this example they are 4K, but page sizes from 512 bytes to 64K are commonly used in existing systems. With 64K of virtual address space and 32K of physical memory, we have 16 virtual pages and 8 page frames. Transfers between memory and disk are always in units of a page.

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

```
MOVE REG,0
```

the 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 board 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, an instruction

MOVE REG,8192

is effectively transformed into

MOVE REG,24576

because virtual address 8192 is in virtual page 2 and this page is mapped onto physical page frame 6 (physical addresses 24576 to 28671). 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. 4-8 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** in each entry keeps track
of whether the page is mapped or not.

What happens if the program tries to use an unmapped page, for example, by
using the instruction

```
MOVE 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 sys-
tem picks a little-used page frame and writes its contents back to the disk. It then
fetches the page 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 4K 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 4K and 8K. Then it would replace the cross
in virtual page 8’s entry with a 1, so that when the trapped instruction is re-
executed, it will map virtual address 32780 onto physical address 4108.

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. 4-9 we see an example of a
virtual address, 8196 (001000000000100 in binary), being mapped using the
MMU map of Fig. 4-8. The incoming 16-bit virtual address is split up into a 4-bit
page number and a 12-bit offset. With 4 bits for the page number, we can repre-
sent 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.

### 4.3.2 Page Tables

In theory, the mapping of virtual addresses onto physical addresses is as we
have just described it. The virtual address is split into a virtual page number
(high-order bits) and an offset (low-order bits). 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.

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

Despite this simple description, two major issues must be faced:

1. The page table can be extremely large.
2. The mapping must be fast.

The first point follows from the fact that modern computers use virtual addresses of at least 32 bits. With, say, a 4K 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.

The second point is a consequence of the fact that the virtual-to-physical mapping must be done on every memory reference. A typical instruction has an instruction word, and often a memory operand as well. Consequently, it is necessary to make 1, 2, or sometimes more page table references per instruction. If an instruction takes, say, 10 nsec, the page table lookup must be done in a few nanoseconds to avoid becoming a major bottleneck.

The need for large, fast page mapping is a significant constraint on the way computers are built. Although the problem is most serious with top-of-the-line machines, it is also an issue at the low end as well, where cost and price/performance are critical. In this section and the following ones, we will look at page table design in detail and show a number of hardware solutions that have been used in actual computers.

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. 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 potentially expensive (if the page table is large). Having to load the page table at every context switch can also hurt 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 memory 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. For this reason, this approach is rarely used in its most pure form, but below we will study some variations that have much better performance.

**Multilevel Page Tables**

To get around the problem of having huge page tables in memory all the time, many computers use a multilevel page table. A simple example is shown in Fig. 4-10. In Fig. 4-10(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 4K, and there are a total of $2^{10}$ 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. 4-10(b) we see how the two-level page table works in this example. On the left we have 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 represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 1024 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 address corresponds to \( PT1 = 1 \), \( PT2 = 3 \), and Offset = 4. The MMU first uses \( PT1 \) to index into the top-level page table and obtain entry 1, which corresponds to addresses 4M to 8M. 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 be zero, causing a page fault. If the page is in memory, the page frame number taken from the second-level page table is combined with the offset (4) to construct a physical address. This address is put on the bus and sent to memory.

The interesting thing to note about Fig. 4-10 is that although the address space contains over a million pages, only four page tables are actually needed: the top-level table, and the second-level tables for 0 to 4M, 4M to 8M, and the top 4M. The Present/absent bits in 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 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 \( PT1 = 1 \) and \( PT2 \) but in actual practice other values are also possible, of course.

The two-level page table system of Fig. 4-10 can be expanded to three, four, or more levels. Additional levels give more flexibility, but it is doubtful that the additional complexity is worth it beyond three levels.

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 is highly machine dependent, but the kind of information present is roughly the same from machine to machine. In Fig. 4-11 we give 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 locate 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.

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 scheme could include protection for reading, writing, and executing.

The MMU translates a virtual address written to the process to a physical address when the data has been modified or when it has been modified in the new state, as in a multiprocessor state.

The MMU also uses the Present/absent bit to detect page faults to a process and to look up the page number assigned to a process's virtual address. Page faults are common in operating systems. The term page fault is used to indicate the state of the system.

Finally, the MMU should be considered an important part of the hardware that the operating system runs on. The memory management instructions and the memory mapping instructions will be used by the operating system.

Note that the page table is not part of the virtual memory. Information about page tables is stored in the page table inside.

4.3.3 TLB

In most modern computer systems, it is possible to use a large number of page tables. Consider, for example, the system that we have been discussing, in which the absence of a page table entry is written for a page.
A typical page table entry.

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 writing. Its value is to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are 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.

4.3.3 TLBs—Translation Lookaside Buffers

In most paging schemes, the page tables are kept in memory, due to their large size. Potentially, this design has an enormous impact on performance. Consider, for example, an 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, additional memory references will be needed to access the page table. Since execution speed is generally limited by the rate the CPU can get instructions and data out of the memory, having to make two page table references per memory reference reduces performance by 2/3. Under these conditions, no one would use it.

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. 4-12. It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 64. Each entry contains information about one page, in particular, 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. Another bit indicates whether the entry is valid (i.e., in use) or not.

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>

**Figure 4-12.** A TLB to speed up paging.

An example that might generate the TLB of Fig. 4-12 is a process in a loop that spans virtual pages 19, 20, and 21, so 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.

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 in the TLB. If it is not, or if it has been modified after last used, the MMU will try to write it to the page table, just as it would if the TLB did not exist.

The instruction then proceeds to the TLB. The hardware then checks to see if the entry just loaded is already in the TLB. If it is, it is then used; if not, it results in a TLB miss, which requires a page table look-up. If the page modified before the TLB entry was loaded, the page table is modified before the TLB entry is loaded.

**Software TLB Management**

Up until now we have assumed that every process has a dedicated TLB. Because of performance reasons, this is not necessarily true. Traps to the MMU are taken at several stages during system boot.

In the first stage, the hardware checks to see if the current TLB contains any entries that are valid (in use). If the TLB is too small for the current operating system, the system halt is entered and the TLB is expanded. If the page table was not in the TLB, an exception is generated, and the system must find the page table, load it into main memory, and then find the instruction that caused the page fault. The system then checks all the instructions contained in the TLB.

Surprisingly, the number of TLB misses in modern systems is remarkably small. The main reason is that the TLB is very fast. The amount of a page table in the TLB would decrease the performance of the CPU by about 10% (1994).

Various designs for systems that do TLB management exist. Some systems have a single TLB that keeps track of all TLBs and some have two TLBs: one for instructions and one for data. To reduce TLB misses, a process can use one TLB for the instructions and the other for the data. Each process is then given a number of entries in the TLB. For example, if a process has 256 entries in the TLB, it is likely that the TLB will not need to be expanded because of TLB misses.
number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel). 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 same way as it would be from the page table itself.

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 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. 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, some modern RISC machines, including the MIPS, Alpha, and 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 just 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 reasonably large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be quite 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 at length by Uhlig et al. (1994).

Various strategies have been developed 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 does an RPC 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 RPC, the system can also check to see where the server’s code, data, and stack pages are, and map them in before they can 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., 4K) 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.

### 4.3.4 Inverted Page Tables

Traditional page tables of the type described so far require one entry per virtual page, since they are indexed by virtual page number. If the address space consists of $2^{32}$ bytes, with 4096 bytes per page, then over 1 million page table entries are needed. As a bare minimum, the page table will have to be at least 4 megabytes. On larger systems, this size is probably doable.

However, as 64-bit computers become more common, the situation changes drastically. If the address space is now $2^{64}$ bytes, with 4K pages, we need over $10^{15}$ bytes for the page table. Tying up 1 million gigabytes just for the page table is not doable, not now and not for decades to come, if ever. Consequently, a different solution is needed for 64-bit paged virtual address spaces.

One such solution is the inverted page table. 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 4K page, and 32 MB of RAM, an inverted page table only requires 8192 entries. The entry keeps track of which (process, virtual) page is located in the page frame.

Although inverted page tables save vast amounts 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 an 8K table on every memory reference is not the way to make your machine blazingly fast.

The way out of this dilemma is to use 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. Using a hash table as an index into the inverted page table, this search can be made reasonably fast, however. Inverted page tables are currently used on some IBM and Hewlett-Packard systems.

### 4.4 PAGE TABLE MAINTENANCE

When a process is running on a computer, the pages of the working set are kept in physical memory. Other pages of the process are on the disk and are kept only when the pages are needed. The working set of a process is defined as that set of pages of that process that are needed in the near future.

While in an accessing (read or write) state, the pages of the working set are never swapped out. If these pages are needed again, they can be rapidly loaded from physical memory. Some pages may change state to no longer be part of the working set, and these pages may be swapped out to disk.

The only exception to this rule is when a page fault occurs. A page fault occurs when a process attempts to access a page that is not currently in the TLB. In this case, the page is brought in from disk and added to the TLB.

### 4.4.1 The Page Fault

The best way to minimize the number of page faults is to cache frequently used pages in the TLB. If a process attempts to access a page that is already in the TLB, the page is quickly loaded from the TLB and returned to the process. If a page is not in the TLB, the page is brought in from disk and added to the TLB.

The number of page faults is inversely proportional to the efficiency of the TLB. When a process is running, the number of page faults is a good indicator of the efficiency of the TLB.
Hewlett-Packard workstations and will become more common as 64-bit machines become widespread.

Other approaches to handling large virtual memories can be found in (Huck and Hays, 1993; Talluri and Hill, 1994; and Talluri et al., 1995).

4.4 PAGE REPLACEMENT ALGORITHMS

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. 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., a page 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 replace 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 algorithms.

4.4.1 The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to 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 algorithm simply 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.
For $s = 128K$ and $e = 8$ bytes per page table entry, the optimum page size is 1448 bytes. In practice 1K or 2K would be used, depending on the other factors (e.g., disk speed). Most commercially available computers use page sizes ranging from 512 bytes to 64K.

### 4.5.4 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. 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. 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.

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 and Hudak, 1989; 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.

### 4.6 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. 4-19.

![Virtual address space diagram](image)

**Figure 4-19.** In a one-dimensional address space with growing tables, one table may bump into another.

Consider what happens if a program has an exceptionally large number of variables. The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables. The compiler could, of course, simply issue a message saying that the compilation cannot continue due to too many variables, but doing so does not seem very sporting when unused space is left in the other tables.

Another possibility is to play Robin Hood, taking space from the tables with an excess of room and giving it to the tables with little room. This shuffling can be done, but it is a great deal of extra work for the compiler.
be done, but it is analogous to managing one's own overlays—a nuisance at best and a great deal of tedious, unrewarding work at worst.

What is really 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 extremely general solution is to provide the machine with many completely independent address spaces, called segments. Each segment consists of a linear sequence of addresses, from 0 to some maximum. The length of each segment may be anything from 0 to the maximum 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 4–20 illustrates a segmented memory being used for the compiler tables discussed earlier.

Figure 4–20. A segmented memory allows each table to grow or shrink independently of the other tables.

We emphasize 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 up 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 next to each other, with no address space between them. Consequently, changing one procedure's size can affect the starting address of other, unrelated procedures. 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 much more complicated. In effect, these systems do it by simulating segmentation.

Because each segment forms a logical entity of which the programmer is aware, such as a procedure, an array, or a stack, different segments can have different kinds of protection. A procedure segment can be specified as execute only, prohibiting attempts to read from it 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 programming errors.

You should try to understand why protection makes sense in a segmented memory but not in a one-dimensional paged memory. In a segmented memory the user is aware of what is in each segment. Normally, a segment would not contain a procedure and a stack, for example, but one or the other. Since each segment contains only one type of object, the segment can have the protection appropriate for that particular type. Paging and segmentation are compared in Fig. 4-21.

The contents of a page are, in a sense, accidental. The programmer is unaware of the fact that paging is even occurring. Although putting a few bits in each entry of the page table to specify the access allowed would be possible, to utilize this feature the programmer would have to keep track of where in his address space the page boundaries were. That is precisely the sort of administration
that paging was invented to eliminate. Because the user of a segmented memory has the illusion that all segments are in main memory all the time—that is, he can address them as though they were—he can protect each segment separately, without having to be concerned with the administration of overlaying them.

### 4.6.1 Implementation of Pure Segmentation

The implementation of segmentation differs from paging in an essential way: pages are fixed size and segments are not. Figure 4-22(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. 4-22(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. 4-22(c), and segment 3 is replaced by segment 6, as in Fig. 4-22(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. 4-22(e).
4.6.2 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 that are actually needed have to be around. 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 Pentium.

MULTICS ran on the Honeywell 6000 machines and their descendants and provided each program with a virtual memory of up to $2^{18}$ segments (more than 250,000), each of which could be 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 is being used) with the advantages of segmentation (ease of programming, modularity, protection, and sharing).

Each MULTICS program has a segment table, with one descriptor per segment. Since there are potentially more than a quarter of a million entries in the table, the segment table is itself a segment and is paged. A segment descriptor contains an indication of whether the segment is in main memory or not. If any part of the segment is in memory, the segment is considered to be in memory, and its page table will be in memory. If the segment is in memory, its descriptor contains an 18-bit pointer to its page table [see Fig. 4.23(a)]. Because physical addresses are 24 bits and pages are aligned on 64-byte boundaries (implying that the low-order 6 bits of page addresses are 000000), only 18 bits are needed in the descriptor to
store a page table address. The descriptor also contains the segment size, the protection bits, and a few other items. Figure 4-23(b) illustrates a MULTICS segment descriptor. The address of the segment in secondary memory is not in the segment descriptor but in another table used by the segment fault handler.

![Diagram](image)

(a) The descriptor segment points to the page tables. (b) A segment descriptor. The numbers are the field lengths.

Each segment is an ordinary virtual address space and is paged in the same way as the nonsegmented paged memory described earlier in this chapter. The normal page size is 1024 words (although a few small segments used by MULTICS itself are not paged or are paged in units of 64 words to save physical memory).
An address in MULTICS consists of two parts: the segment and the address within the segment. The address within the segment is further divided into a page number and a word within the page, as shown in Fig. 4-24. When a memory reference occurs, the following algorithm is carried out.

1. The segment number is used to find the segment descriptor.
2. A check is made to see if the segment’s page table is in memory. If the page table is in memory, it is located. If it is not, a segment fault occurs. If there is a protection violation, a fault (trap) occurs.
3. The page table entry for the requested virtual page is examined. If the page is not in memory, a page fault occurs. If it is in memory, the main memory address of the start of the page is extracted from the page table entry.
4. The offset is added to the page origin to give the main memory address where the word is located.
5. The read or store finally takes place.

![Figure 4-24. A 34-bit MULTICS virtual address.](image)

This process is illustrated in Fig. 4-25. For simplicity, the fact that the descriptor segment is itself paged has been omitted. What really happens is that a register (the descriptor base register), is used to locate the descriptor segment’s page table, which, in turn, points to the pages of the descriptor segment. Once the descriptor for the needed segment has been found, the addressing proceeds as shown in Fig. 4-25.

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 contains a 16-word high-speed TLB that can search all its entries in parallel for a given key. It is illustrated in Fig. 4-26. When an address is presented to the computer, the addressing hardware first checks to see if the virtual address is in the TLB. If so, it gets the page frame number directly from the TLB and forms 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 are kept in the TLB. Programs whose working set is smaller than the TLB size will come to equilibrium with the addresses of the entire working set in the TLB and therefore will run efficiently. If the working set is actually larger than the TLB size, future references will be included in the TLB, and the TLB keeps track of which pages are currently in the TLB. Each page frame used by the program is kept in the TLB for some particular amount of time.
run efficiently. If the page is not in the TLB, the descriptor and page tables are actually referenced to find the page frame address, and the TLB is updated to include this page, the least recently used page being thrown out. The age field keeps track of which entry is the least recently used. The reason that a TLB is used is for comparing the segment and page number of all the entries in parallel.
4.6.3 Segmentation with Paging: The Intel Pentium

In many ways, the virtual memory on the Pentium (and Pentium Pro) resembles MULTICS, including the presence of both segmentation and paging. Whereas MULTICS has 256K independent segments, each up to 64K 36-bit words, the Pentium has 16K independent segments, each holding up to 1 billion 32-bit 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 segments holding megabytes.

The heart of the Pentium virtual memory consists of two tables, 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, a Pentium 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. 4-27.

![Figure 4-27. A Pentium selector.](image)

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. A descriptor consists of 8 bytes, including the segment’s base address, size, and other information, as depicted in Fig. 4-28.

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 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 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.

It then checks to see if the offset is beyond the end of the segment, in which case a trap also occurs. Logically, there should simply be a 32-bit field in the descriptor giving the size of the segment, but there are only 20 bits 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. The Pentium page size is fixed at 4K bytes, so 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 Pentium 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. 4-29. 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.

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 permitted to overlap, incidentally, 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 4K 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. 4-30.

![Diagram showing two-level memory mapping](image)

**Figure 4-30.** Mapping of a linear address onto a physical address.

In Fig. 4-30(a) we see a linear address divided into three fields, Dir, Page, and Off. 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, Off 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 4K page frames, so a single page table

handles 4 megabytes of memory. A segment shorter than 4M will have a page di-

rectory 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 Pentium, like MULTICS,

has a small TLB that directly maps the most recently used Dir–Page combina-

tions onto the physical address of the page frame. Only when the current com-

bination is not present in the TLB is the mechanism of Fig. 4.30 actually carried out

and the TLB updated.

A little thought will reveal the fact that when paging is used, there is really no

point in having the Base field in the descriptor be nonzero. All that Base does is

cause a small offset to use an entry in the middle of the page directory, instead of

at the beginning. The real reason for including Base at all is to allow pure (non-
paged) segmentation, and for compatibility with the 286, which always has paging

disabled (i.e., the 286 has only pure segmentation, but not paging).

It is also worth noting that if some application does not need segmentation but

is 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.

All in all, one has to give credit to the Pentium 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 effi-
ciently, the resulting design is surprisingly simple and clean.

Although we have covered the complete architecture of the Pentium virtual

memory, albeit briefly, it is worth saying a few words about protection, since this

subject is intimately related to the virtual memory. Just as the virtual memory

scheme is closely modeled on MULTICS, so is the protection system. The Pentium

supports four protection levels with level 0 being the most privileged and level 3

the least. These are shown in Fig. 4.31. At each instant, a running program is at a

certain level, indicated by a 2-bit field in its PSW. Each segment in the system

also has a level.

As long as a program restricts itself to using segments at its own level, every-
thing works fine. Attempts to access data at a higher level are permitted.
Attempts to access data at a lower level are illegal and cause traps. Attempts to

call procedures at a different level (higher or lower) are allowed, but in a carefully
controlled way. To make an interlevel call, the CALL instruction must contain a selector instead of an address. This selector designates a descriptor called a call gate, which gives the address of the procedure to be called. Thus it is not possible to jump into the middle of an arbitrary code segment at a different level. Only official entry points may be used. The concepts of protection levels and call gates were pioneered in MULTICS, where they were viewed as protection rings.

A typical use for this mechanism is suggested in Fig. 4-31. At level 0, we find the kernel of the operating system, which handles I/O, memory management, and other critical matters. At level 1, the system call handler is present. User programs may call procedures here to have system calls carried out, but only a specific and protected list of procedures may be called. Level 2 contains library procedures, possibly shared among many running programs. User programs may call these procedures and read their data, but they may not modify them. Finally, user programs run at level 3, which has the least protection.

Traps and interrupts use a mechanism similar to the call gates. They, too, reference descriptors, rather than absolute addresses, and these descriptors point to specific procedures to be executed. The Type field in Fig. 4-28 distinguishes between code segments, data segments, and the various kinds of gates.

4.7 OVERVIEW OF MEMORY MANAGEMENT IN MINIX

Memory management in MINIX is simple: neither paging nor swapping is used. The memory manager maintains a list of holes sorted in memory address order. When memory is needed, either due to a FORK or an EXEC system call, the