Hint: In C++, use `new` and `delete`, and in C, use `malloc` to allocate big 20,000 bytes at a time.

19. Write a program that allocates a lot of space, then deallocates it. Does the memory it uses go up and then down, or does it stay up? *Hint:* In C++, use `new` and `delete`, and in C, use `malloc` and `free`.

20. Give the advantages and disadvantages in using a doubly linked list for the free list.

21. Write a C++ subroutine that finds a block of N units in a memory allocation bitmap. Write a C++ subroutine that finds a block of N units of memory in a system that uses a free list and first fit. Compare the two procedures.

22. Explain why it is easy to dynamically allocate blocks of memory if all the blocks are the same size. Why don't we have to use a dynamic memory allocation algorithm in this case?

23. Describe a kind of operating system where it would be reasonable to assume that all processes were about the same size.

24. Why is the space allocated to a stack automatically increased when the stack overflows it? Why is there a limit to how much it can be increased altogether?

25. Describe an application that would be much easier to implement on a computer that used the lock/key form of memory protection than it would on a computer that used the *base/bound* register form of memory protection.

26. Design an extension of the lock/key method of memory protection that allows different protection for reading, writing, and executing in an area of memory. Can you think of a similar extension for the *base/bound* method of memory protection? Can it solve part of the problem, if not all of it?

27. Many modern OSs will dynamically load parts of the operating system. Give one example of a part of the operating system that would be a very good candidate for dynamic linking, and give one example of a part of the operating system that you would not want to dynamically load.

28. Consider the problem of dynamically loading operating system modules. What problems would you encounter? What would be the benefits of doing this?

29. Suppose you wanted to implement a per-process memory manager that returned excess unused space to the operating system. Say that, as soon as 75 percent of the dynamic memory was free, it would halve the size of the memory pool. What techniques would you use to accomplish this goal? How hard would it be? What kind of overhead would you incur?

30. Modify the code for memory allocation in Section 10.16 to use block headers at the beginning of each block rather than a separate list of block structures like it uses now.

31. Modify the code for memory allocation in Section 10.16 to use a next-fit algorithm rather than the first-fit algorithm it uses now.

32. Modify the code for memory allocation in Section 10.16 to use a best-fit algorithm rather than the first-fit algorithm it uses now.

chapter

11

Virtual Memory

In the last chapter, we looked at ways to allocate memory dynamically out of a pool of free memory. In this chapter, we will look at ways to divide a program into smaller pieces so they are easier to allocate and more programs will fit into memory at the same time. Then we will see how we can keep some of these small pieces on disk, and thereby fit even more program in memory at the same time.

11.1 FRAGMENTATION AND COMPACTION

compaction

There is a way to eliminate fragmentation after it occurs, and that method is called compacting storage, or *compaction*. You compact storage by moving all the allocated blocks to one end of storage to create one big block (see Figure 11.1). This eliminates all fragmentation, but it has one major problem: it is expensive to move all that memory around. With today's larger memories, compaction is even more expensive and less attractive.

11.2 DEALING WITH FRAGMENTATION

We have observed that fragmentation is inevitable in dynamically allocated memory. What can we do to reduce the effects of fragmentation? The smaller the pieces we allocate, the easier it is to find space and the smaller the loss due to fragmentation. So far, we have assumed that a program runs in a big block of physical memory and that

contiguous

that memory was *contiguous*, that is, consisting of consecutive physical addresses. Compilers and linkers assume contiguous memory, but memory mapping has given

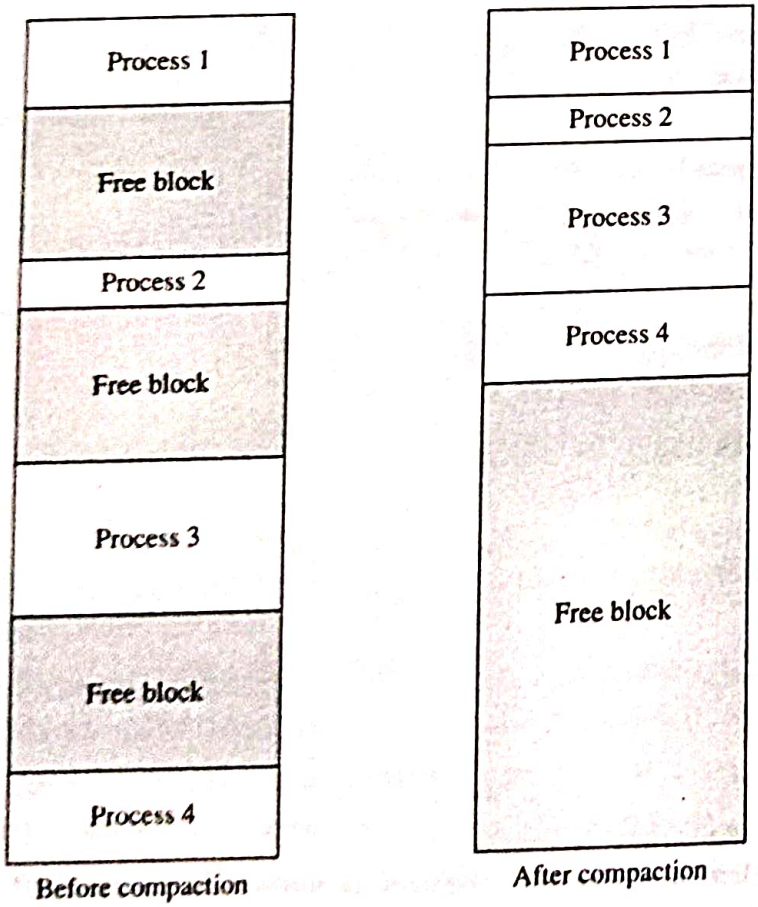


Figure 11.1 Compacting memory

us some flexibility. We need the logical address space to be contiguous, but it is not necessary for the physical address space to be contiguous.

The operating system memory manager allocates physical memory space, not logical address space, so we can divide the program into several parts and put each part into a separate area in physical memory. Each part will be smaller and hence will fit into smaller free blocks, and so the effects of fragmentation will be reduced. We will need some hardware assistance to get this to work, but first we have to decide on what basis we will divide up the program into parts. We will provide a series of answers to this question.

11.2.1 Separate Code and Data Spaces

The processor knows when it is fetching an instruction and when it is fetching data (instruction addresses come from the program counter). So the processor can use two relocation registers, one for code addresses and one for data addresses. Figure 11.2 shows how the relocation works, and Figure 11.3 shows the two address spaces. This allows us to put the code and data parts of a program into two different blocks of physical memory, so it will be easier to find blocks that are large enough.

11.2.2 Segments

But what about programs where the data is much larger than the code, or the code is much larger than the data? In these cases, this split will not help much. Also, why stop at only two parts of the program? Why not more? This generalization is called *segmentation*, and Figure 11.4 shows how it would work with 16-bit addresses and four segments. In Figure 11.4, there are four segment registers, each with two parts:

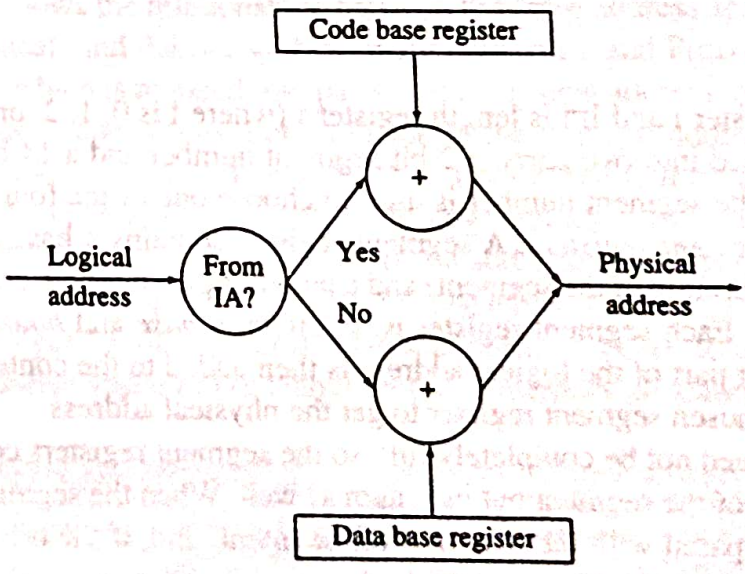


Figure 11.2 Memory relocation with separate code and data spaces

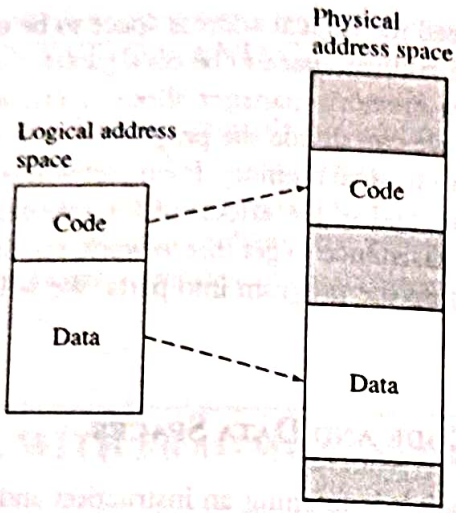


Figure 11.3 Separate code and data spaces

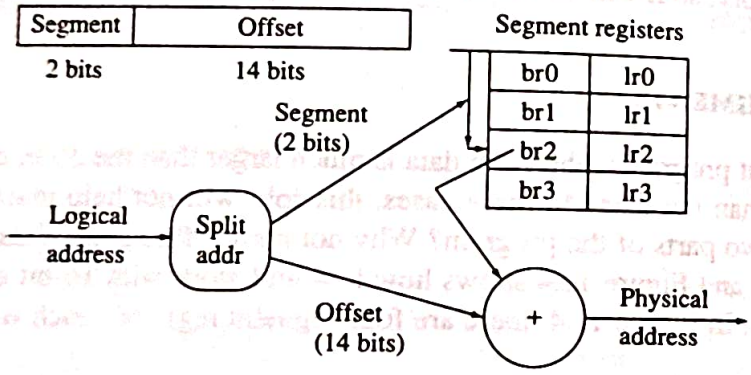


Figure 11.4 Four segments

segment register

`bri` is base register `i` and `lri` is length register `i` (where `i` is 0, 1, 2, or 3). The logical address is divided into two parts, a 2-bit segment number and a 14-bit offset within that segment. The segment number is used to choose one of the four relocation registers, called *segment registers*. A segment register contains a base register (which points to the beginning of the segment) and a length register (which records how long the segment is). Each segment register is, in effect, a *base* and *bound* register pair. The 14-bit offset part of the logical address is then added to the contents of the base register of the chosen segment register to get the physical address.

Segments need not be completely full, so the segment registers contain not only the base address of the segment but its length as well. When the segment is accessed, the offset is compared with the length of the segment, and, if the offset is larger (or equal), a segmentation fault is generated.

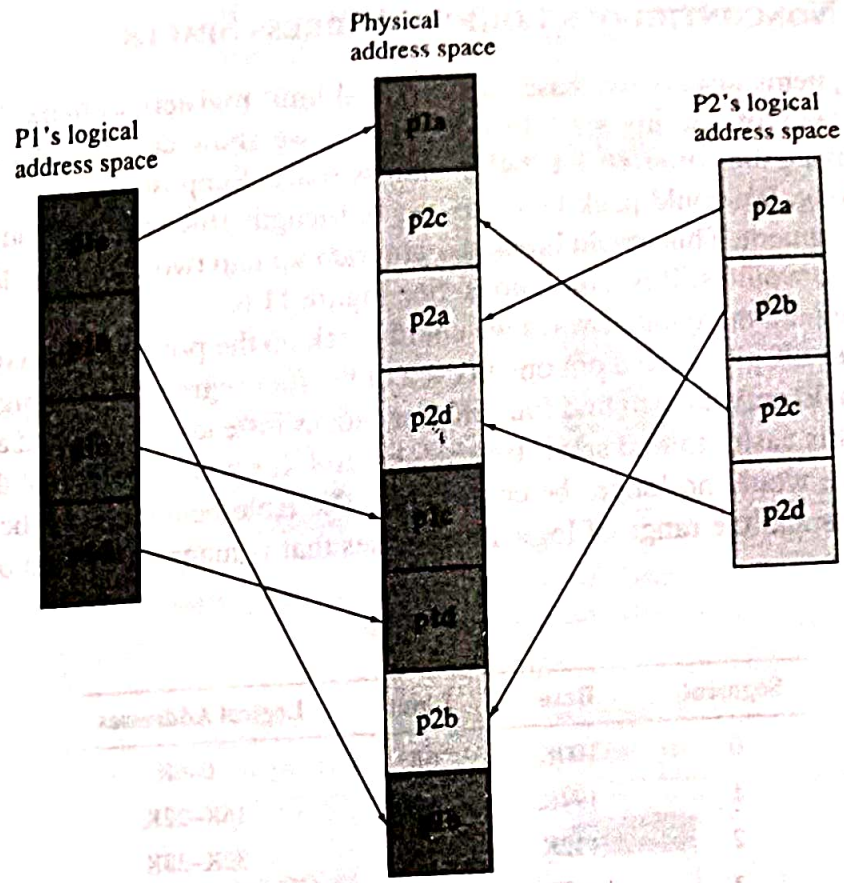


Figure 11.5 Two logical address spaces

This method has several advantages over the code/data split. First, we have divided the program into four pieces instead of just two, so each part will be that much smaller. Second, each piece can be the same size because the split is an arbitrary one by address, rather than being based on the content of the program (whether something is code or data).

Figure 11.5 shows the relationships between the three address spaces (two logical and one physical) and the two address maps. Process 1 and Process 2 each have an address space which is mapped into parts of the physical address space. Each logical address space is contiguous, but its image in the physical address space is not contiguous. The address map is defined by the values of the four base registers.

This method can be easily generalized to larger numbers of segments. The legendary DEC PDP11 used this method with 16-bit addresses and eight segments. This divided the 64K address space into eight 8K segments. Later, PDP11s also used a code/data split to get 16 8K segments. This was done, however, to increase the size of the logical address space rather than to improve memory allocation.

The logical memory of a process should be contiguous. The physical memory of a process does not need to be.

11.2.3 Noncontiguous Logical Address Spaces

Segment systems always use base registers and limit registers in pairs, and hence allow segments of varying size. In Figure 11.5, we show each of Process 1 and Process 2 using the entire 16-bit (64K) address space. Suppose that a program was only 24K long. We could pack this into one full-length 16K segment and one half-length 8K segment. This would break the program up into two parts, and the logical address space would still be contiguous. See Figure 11.6.

But there are many other ways we could break up the program. We could break it up into four 6K pieces, and put one in each of the four segments. (See Figure 11.7.) Then we would only have to find four 6K free blocks instead of a 16K and an 8K free block, and it is easier to find small free blocks. But this would mean that the logical address space would no longer be contiguous. The table below shows the base and limit registers and the range of logical addresses that is mapped by each of the four segments.[1]

Segment	Base	Limit	Logical Addresses
0	100K	6K	0–6K
1	194K	6K	16K–22K
2	132K	6K	32K–38K
3	240K	6K	48K–54K

If a logical address of, say, 8K is generated, the two high-order bits (00) will place it in the first segment, but the 14 low-order bits will be larger than the limit register and so the address will be flagged as illegal by the memory mapping hardware. This is true of all logical addresses in the ranges 6K–16K, 22K–32K, 38K–48K, and 54K–64K. This is what we mean by the logical address space being noncontiguous—there are holes in the logical address space.

Unfortunately, it is unlikely that this kind of mapping will work if we use typical compilers and linkers. Compilers and linkers assume that the logical address space is contiguous, and are not set up to deal with gaps in the address space. Now it is possible that these tools could be modified to handle noncontiguous address spaces, but it would be a lot of work and extra complication. It is possible to use noncontiguous address spaces for dynamically allocated storage. We will discuss this more in Section 12.13.9.

So we are really not free to split up the program into pieces that are evenly distributed across the segments, but we must first fill up segment 0, then fill up segment 1, and so on. For a 16 bit (64K) address space, this is not a problem, since the 16K pieces are fairly small and most programs are larger than 16K. But suppose we want to move to a 32-bit (4Gbyte) address space and still use four seg-

[1]The base register values are chosen arbitrarily.

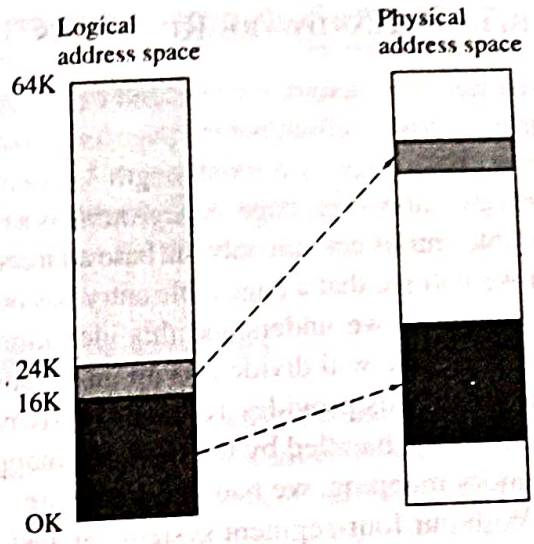


Figure 11.6 Contiguous logical address spaces (uses two segments)

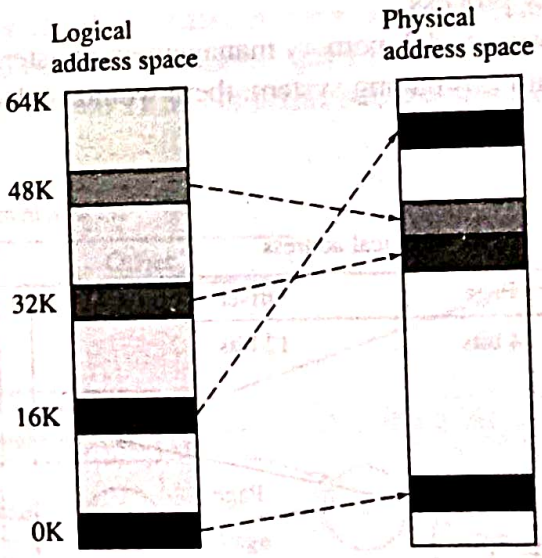


Figure 11.7 Noncontiguous logical address spaces (uses four segments)

ments. Each segment would be 1 Gbyte of memory. Very few programs would fill up even one segment, and so the segmentation hardware would be essentially unused and no benefits would be gained from it.

Since we know the computer industry is moving to larger and larger address spaces, the only solution to the problem is to have much smaller segments that we can fill up completely (and many more of them, of course). But if we are going to fill up each segment, what do we need the length registers for? The answer is that we don't. We will call this new fixed-size segment a "page." The next step in breaking up our program into smaller pieces is to go from segments to pages.

11.2.4 Page Tables in Hardware Registers

page
page table entry
page table

A segment has a base register and a length register, and can be any length from 0 up to some maximum. A *page* has a base register, but the page has a fixed length and so it does not need a length register. Thus pages are fixed-length segments. A *page table entry* (*PTE*) is a word of information about one page. A *page table* is a collection of page table entries. So far, our page table entries contain only the base address of the page frame that holds the page, but later we will see that a page table entry has other fields as well.

Let's look at an example so we understand this idea more easily. Suppose the logical address is 16 bits long. We will divide this up into 4 bits of page number and 12 bits of offset into the page. This provides 16 pages of 4Kbytes each. Figure 11.8 shows how the logical address is handled by the memory-mapping hardware.

With base/limit memory mapping, we had two hardware registers: a base register and a limit register. With our four-segment system, we had eight hardware registers: four segment base registers and four segment limit registers. With this paging system, we have 16 hardware registers: the 16 page base registers. Each time we switch between processes, we have to save the registers of the old process and load the registers of the new process.

Our original hardware had 2 memory management registers and 34 other registers to save and restore. With this paging system, there would be 16 memory management

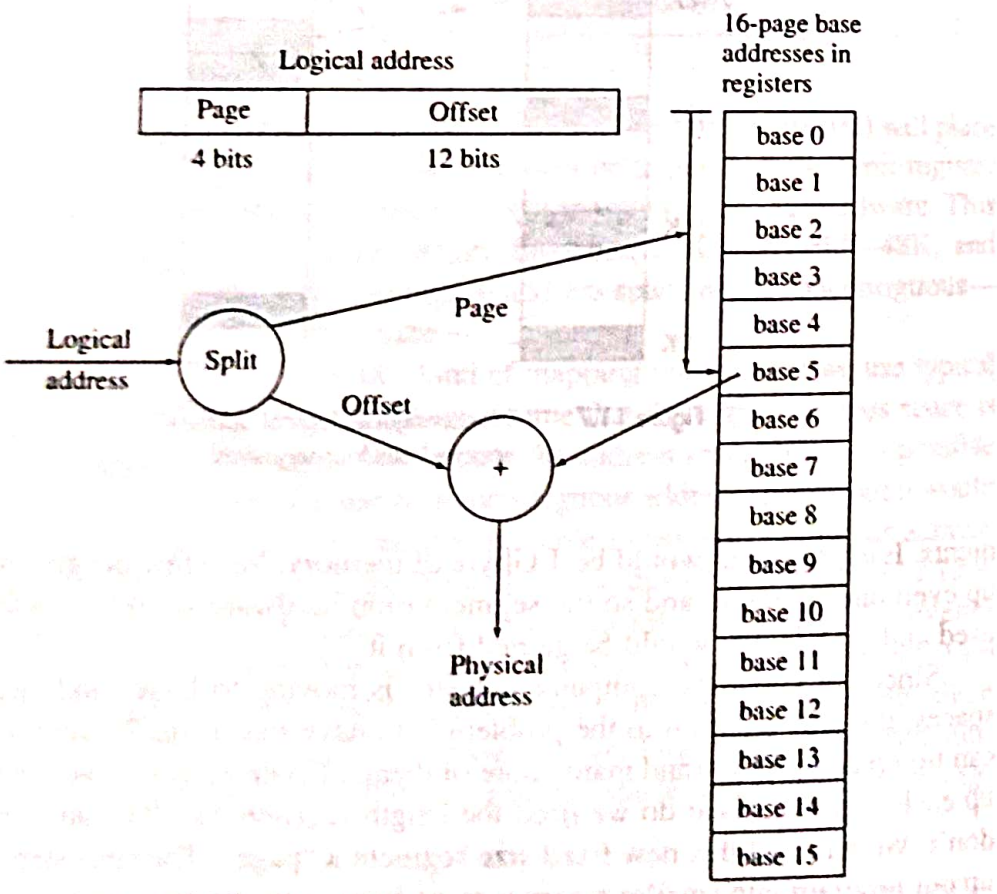


Figure 11.8 Page table in registers

registers. This is still a reasonable number, but if we are going to have very many more pages, then this will get to be an intolerable overhead on process switching.

For example, suppose we kept the 4K pages and decided to expand to a 20-bit (1 Mbyte) address space. This would require 256 page base registers. The hardware cost of providing 256 more fast registers is actually fairly small. The more significant cost is 512 memory accesses (256 stores of the old page registers and 256 loads of the new page registers) for every process switch.

11.2.5 Page Tables in Memory

We have to find some method of reducing the number of registers that we need to save and restore on a context switch. One solution is to store the page base addresses in main memory. We allocate 256 words of memory, and keep the page base addresses there. Page information in memory is also called a *page table*. Now we only need one hardware register—a page table base register which tells us where in memory to find the page table. Figure 11.9 shows how this changes the mapping. The only differences are the addition of the page table base register and the moving of the page table to memory. Figure 11.10 is a different view of the mechanisms of paging.

Suppose we extend our logical address space to 22 bits (4 Mbytes) and divide it into 10 bits of page number and 12 bits of offset into the page. This keeps the 4 Kbyte

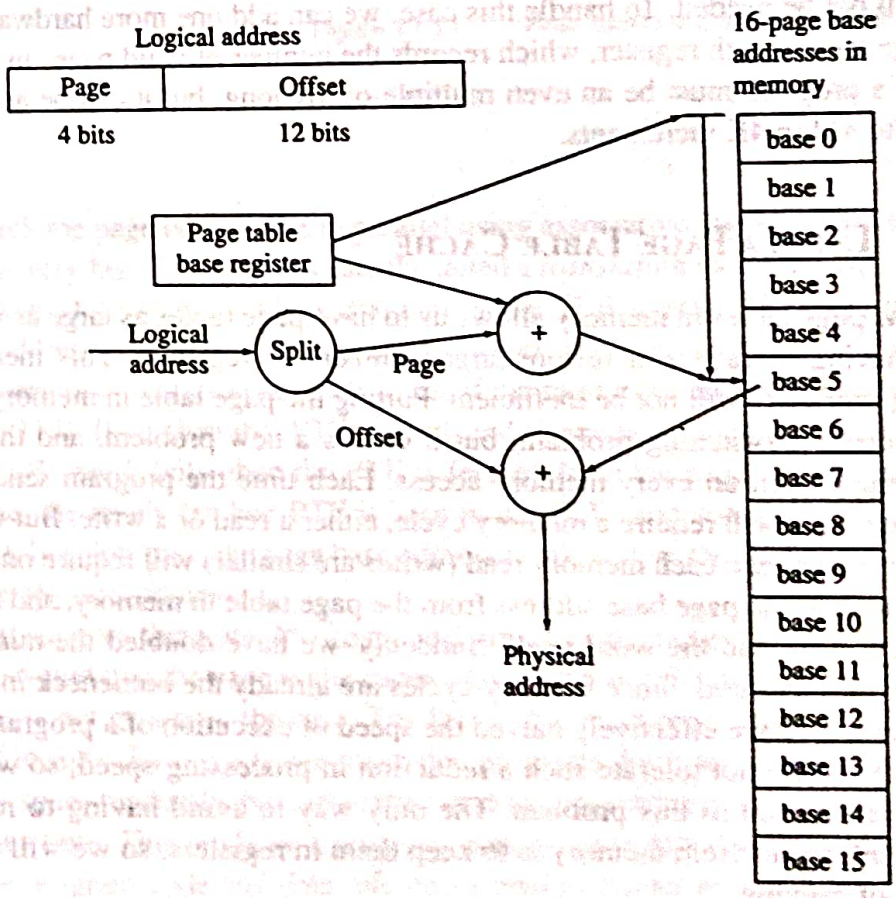


Figure 11.9 Page table in memory

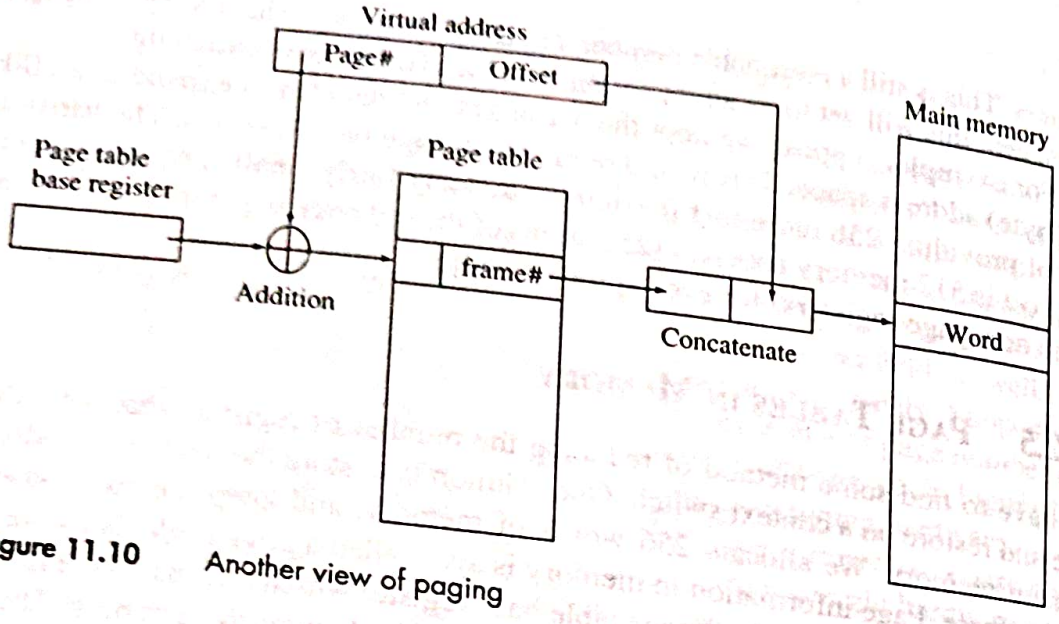


Figure 11.10 Another view of paging

pages we have been using, and makes the page table 1K entries long. Since each page table entry will be 32 bits (4 bytes), that means the page table will be exactly one page long also, which makes things neat and tidy.

What if a program does not take up the whole address space? Then some of the pages will not be needed. To handle this case, we can add one more hardware register, a page table length register, which records the number of valid pages in the page table. So a program must be an even multiple of 4K long, but it can be any length from 4K to 4M in 4K increments.

11.2.6 Using a Page Table Cache

Putting the page tables in memory allows us to have page tables as large as we want without having to save and restore large numbers of registers. This means that switching processes will not be inefficient. Putting the page table in memory solves the slow context switching problem, but it causes a new problem, and that is an extra memory fetch on every memory access. Each time the program sends out a logical address, it will require a memory cycle, either a read or a write. But with the page table in memory, each memory read (writes are similar) will require one memory cycle to read the page base address from the page table in memory, and another memory cycle to read the word itself. Suddenly, we have doubled the number of memory cycles required. Since memory cycles are already the bottleneck in almost all programs, we have effectively halved the speed of execution of a program.

Clearly we cannot tolerate such a reduction in processing speed, so we must figure out a solution to this problem. The only way to avoid having to read the page base addresses from memory is to keep them in registers, so we will use the technique of caching.

The idea is that we keep, in processor registers, the last few page base addresses that we have looked up in the page table. Figure 11.11 shows a flow chart for caching of page table entries. The algorithm in C++ is shown in Figure 11.12. The hardware

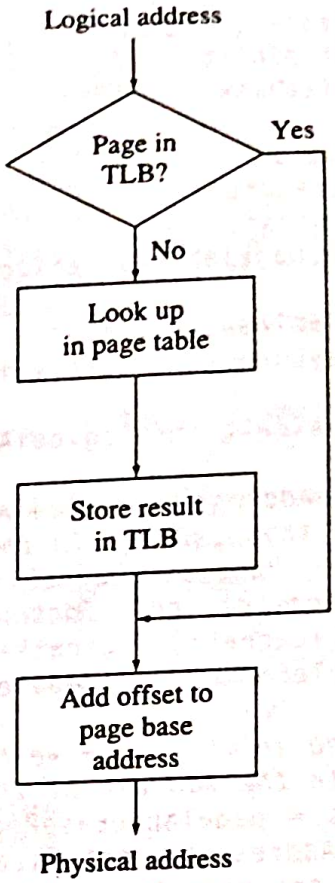


Figure 11.11 Flow chart for paging with a TLB

will search the page table cache in parallel using associative registers, and so this operation is very fast. This cache is usually called a *translation lookaside buffer*, or *TLB*. translation (TLB)

Figure 11.13 shows a block diagram of the TLB lookup. The page part of the logical address is isolated and used in two places. It is used to do a hardware parallel (hence very fast) search in the TLB to see if the page table entry (PTE) for this page is in the TLB. If so, then that PTE is used, and no memory fetch is required to get it. If the TLB search fails, then the PTE is fetched from the page table in memory. In that case, the newly fetched PTE is kept in the TLB (and some other TLB entry is evicted). In either case, the page base address is taken from the PTE and used to compute the physical address.

But suppose there are 256 pages and only 8 page cache entries. It would seem that we would find the page in the cache only $8/256 = 1/32$ or 3 percent of the time. But this is not generally the case. The fact is that most programs exhibit a high degree of locality. By *locality*, we mean that programs don't just jump all around their address space randomly, but rather they tend to concentrate in small areas of the program at a time. This might be a program loop, or some data that they are using heavily. The program code and data references tend to cluster around a few pages for a while, then they might move on to another few pages, and so on. The result is that only a few pages are in use at a time, and once they are loaded into the page table cache, you usually find the PTE on the cache. locality

```
const int PageTableCacheSize = 8;
const int pageSizeShift = 12;
const int pageSizeMask = 0xFFF;

struct CacheEntry {
  int logicalPageAddress;
  int pageBaseAddress;
} PageTableCache[PageTableCacheSize];

extern int pageTableBaseRegister;
int LeastRecentlyUsedCacheSlot( void );

int LogicalToPhysical( int logicalAddress ) {
  int i;
  int logicalPageAddress = logicalAddress & ~pageSizeMask;
  // first see if this page is in the cache
  for( i = 0; i < PageTableCacheSize; ++i ) {
    // in the hardware, this lookup is done in parallel
    if( PageTableCache[i].logicalPageAddress == logicalPageAddress )
  return PageTableCache[i].pageBaseAddress;
  }
  // if not found in the cache we have to look it up in memory
  // first compute the address of the page table entry
  int pteAddress = pageTableBaseRegister
     + (logicalAddress >> pageSizeShift);
  // then fetch the page base address from the page table
  int pageBaseAddress = MemoryFetch( pteAddress );
  // now update the cache by replacing the entry that has not been
  // used in the longest time (the least recently used one)
  // with this new entry
  i = LeastRecentlyUsedCacheSlot();
  PageTableCache[i].logicalPageAddress = logicalPageAddress;
  PageTableCache[i].pageBaseAddress = pageBaseAddress;
  // and then return the physical address of the page
  return pageBaseAddress;
}
```

Figure 11.12 Caching the page table entries

Each time the page base register is found in the cache, we call it a *cache hit*. If it is not in the cache we call it a *cache miss*. The *cache hit rate* is the percentage of times we get a cache hit, so if the cache hit rate is 90 percent, then 9 memory accesses out of 10 we find the page base address in the cache. Thus the effective memory access time is $0.9 * 1 + 0.1 * 2 = 1.1$. In other words, the average memory access takes 1.1 memory cycles, so our effective slowdown is 10 percent, which is probably acceptable.

We need to be sure to invalidate the cache when we change processes, since we will be entering a new logical address space and all the logical addresses will refer to different physical pages. The hardware does not normally keep an address space identifier in the page cache, and depends on the operating system to invalidate the cache when it does a process switch. Some modern memory management

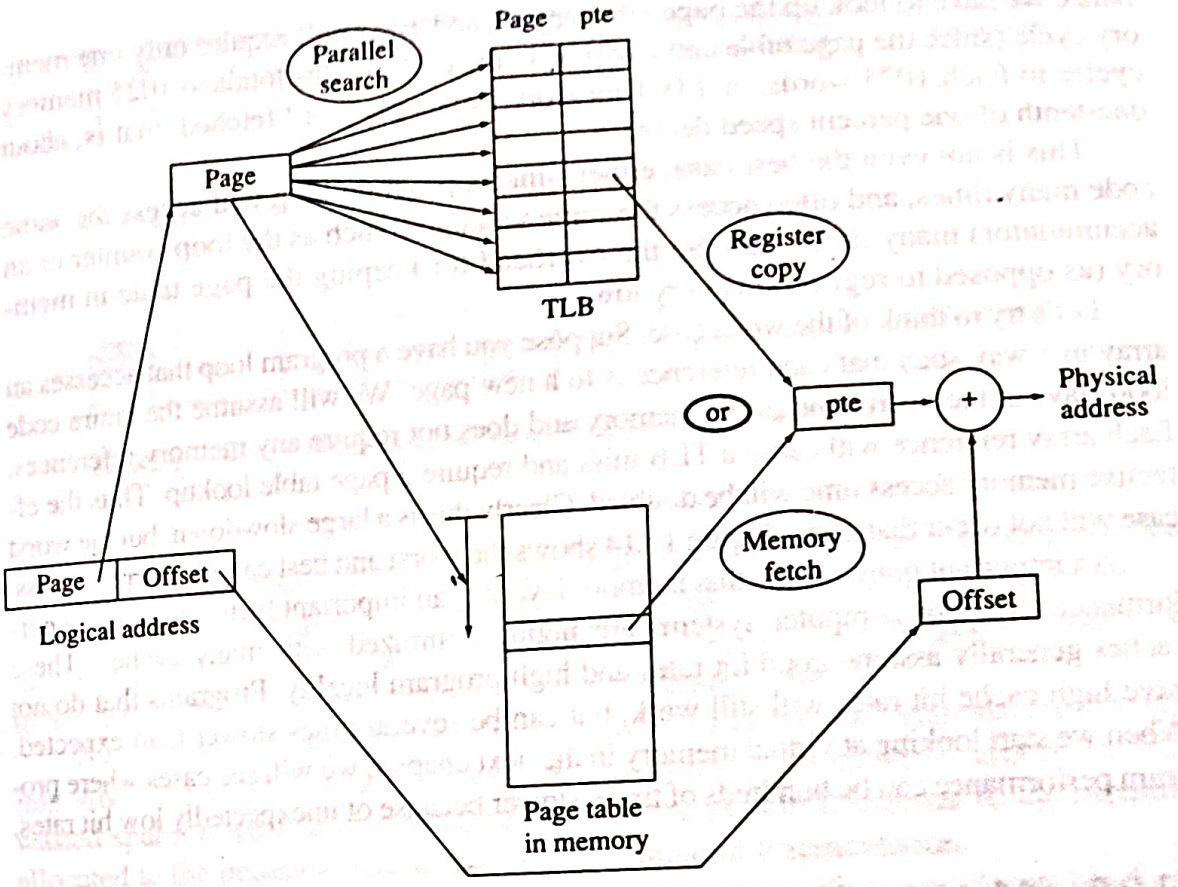


Figure 11.13 TLB lookup

chips actually do keep the address space numbers in their cache, and do not require invalidating the cache. This is especially useful for very large caches where entries might stay valid across several context switches.

Let's review what we did here. We found that it was too expensive (in context switch time) to keep the page table in hardware registers. When we moved the page table into memory, we found that it was too expensive (in extra memory cycles), so we moved some of the page table back into registers. But, we did not move all of the page table back into registers, only a small portion of it, and locality ensured that the effective memory access time was acceptable. The TLB is still a source of inefficiency in context switching, and modern processors have other techniques to deal with that problem.

Caching can solve many operating system performance problems.

11.2.7 Analysis Models of Paging with Caching

Suppose we are going through a large array that covers several hundred pages, and that each page has 1024 array words in it (4 K byte pages). The first word of each page will cause a page cache miss, but the next 1023 words will have their page table entry in the cache. So over the 1024 words fetched, 1 will require two memory cycles

(since we have to look up the page table entry), and 1023 will require only one memory cycle (since the page table entry will be in the cache). This totals to 1025 memory cycles to fetch 1024 words, or 1.001 memory cycles per word fetched, that is, about one-tenth of one percent speed decrease.

This is not even the best case, either, since a program loop will access the same code many times, and often access the same variables (such as the loop counter or an accumulator) many times also. So the overhead for keeping the page table in memory (as opposed to registers) is very low.

Let's try to think of the worst case. Suppose you have a program loop that accesses an array in a way such that each reference is to a new page. We will assume the entire code loop stays in the instruction cache memory and does not require any memory references. Each array reference will cause a TLB miss and require a page table lookup. Thus the effective memory access time will be doubled. Clearly this is a large slowdown, but the worst case will not occur that often. Figure 11.14 shows the worst and best cases for array access.

An important point here is that memory layout is an important factor in program performance. Modern computer systems are highly optimized with many caches. These caches generally assume good hit rates and high program locality. Programs that do not have high cache hit rates will still work, but can be several times slower than expected. When we start looking at virtual memory in the next chapter, we will see cases where program performance can be hundreds of times slower because of unexpectedly low hit rates.

11.2.8 Memory Allocation with Paging

Each program is divided up into pages, and each page is the same size, so memory management becomes almost trivial. All pages are the same size, and all programs will be allocated an integral number of pages. The extra bytes left over at the end of the last page are left unused. Physical memory is divided up into blocks of memory, called *page frames*, that can hold a page. A page frame is the size of a page, and is always at an even page boundary. For example, if the pages are 4K, then the page frames are at physical addresses 0, 4K, 8K, 12K, 16K, 20K, etc.

page frame

The advantage of paging is that a page can go into *any* page frame. All we have to do is set the base address in the page table. This means that memory is divided into a bunch of fixed-size blocks that are all the same. So memory management consists of keeping a list of the free page frames, handing them out when a page frame is requested, and putting them back on the list when a page frame is freed.

DESIGN TECHNIQUE: SIMPLE ANALYTIC MODELS

We did some simple calculations to determine the cost of paging with and without a TLB, and found that we needed a TLB with a 90 percent or better hit rate. We will do some calculations on page faults, and determine that we need a 99.9999 percent hit rate on page references. These simple models help us determine goals for our designs. Simple analytic models are useful in other parts of operating systems and computer science. To learn more about the use of simple analytic models, see Section 13.5.

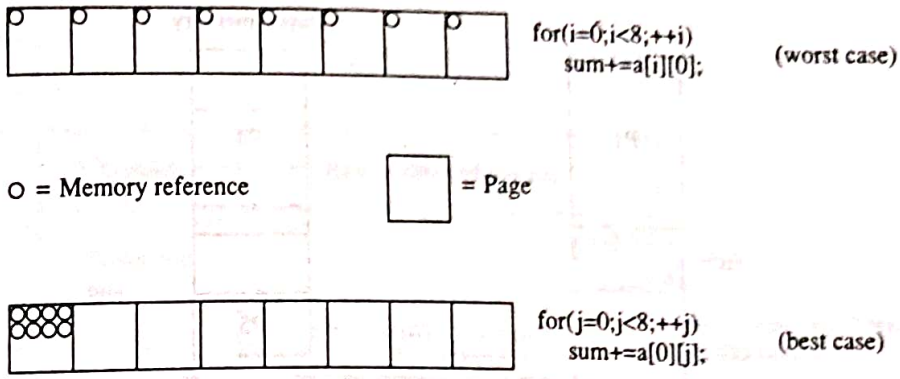


Figure 11.14 Worst and best cases for paging of array accesses

There is apparently no fragmentation at all, since all blocks are of fixed size, but this is a bit of an illusion. A problem that comes up with pages is *internal fragmentation*, that is, the wasted space at the end of the last page of a program. Pages tend to be smaller than segments to reduce internal fragmentation.

internal fragmentation

The fragmentation in dynamic memory allocation is called *external fragmentation* since the fragments are external to the memory allocated to processes. The wasted space at the end of a program in a paging system is internal to the memory allocated to the program, and so this is called internal fragmentation.

external fragmentation

Figure 11.15 shows internal and external fragmentation. When you allocate blocks of memory, then the holes in between are the external fragments. With paging, all the memory is allocated, but the last page of a process might not be filled and that creates internal fragments. This means that the amount of the internal fragmentation is fixed by the page size, and so goes down as a percentage of program size as programs get bigger. External fragmentation, on the other hand, tends to be a fixed percentage of the memory allocated.

11.2.9 Terminology: Page and Page Frame

It is important to understand the distinction we are making between pages and page frames. A page is 4 Kbytes (or 1 Kbyte, or whatever the page size) of information. It is the same page whether it is stored on disk or in memory, since the page is the information, not the medium that holds the information. A page frame is 4K of physically contiguous bytes that start on an even 4K boundary, no matter what the content of that memory. A page frame is a container for a page.

Pages are logical entities that contain information. Page frames are physical entities that contain pages.

11.2.10 Page Tables

We have taken a simplified view of page tables so far, and viewed them as simply an array in memory of page base addresses. Actual page tables contain more information

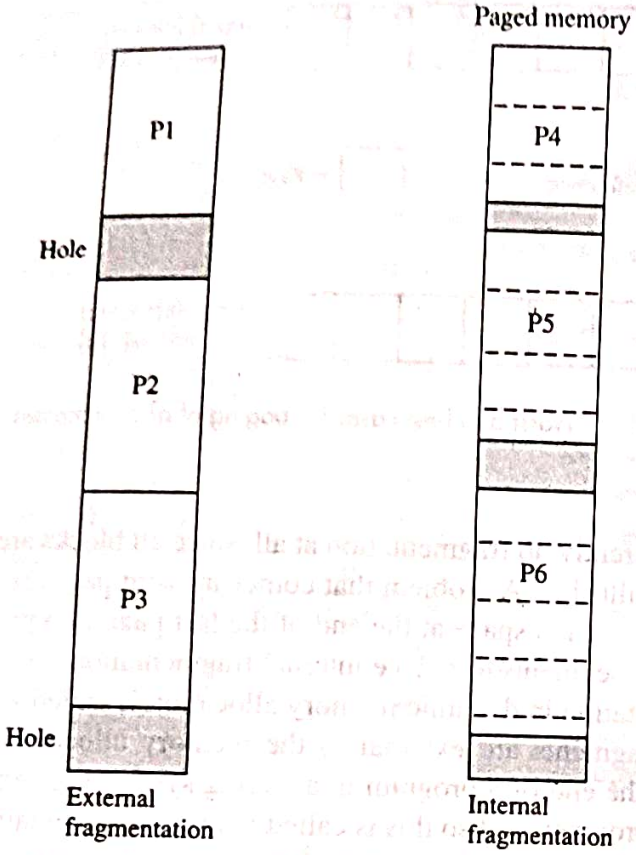


Figure 11.15 Internal and external fragmentation

protection field

about the page than the base address. A typical page table entry (see Figure 11.16) contains the base address where the page is located in memory and a *protection field* which describes the allowed uses of the page.

The base address always ends with a string of 0 bits, since page frames begin at multiples of the page size. It is not really necessary to store these zeros, because the hardware can put them back on when it uses the page table entry. So if we have, for example, 32-bit physical addresses and 12-bit page offsets (4K pages), then the page base address need only be 20 bits long. Figure 11.16 illustrates this idea.

The protection field tells how the page can be validly used. For example, there might be three bits in the protection field:

- *bit 0*—The page can be read.
- *bit 1*—The page can be written.
- *bit 2*—The page can be executed.

This allows eight possible protection values:

- *000*—The page cannot be used for any purpose.
- *001*—The page can only be read. This is used for read-only data.
- *010*—The page can only be written.
- *011*—The page can be read or written. This will detect invalid jumps into read-write data.

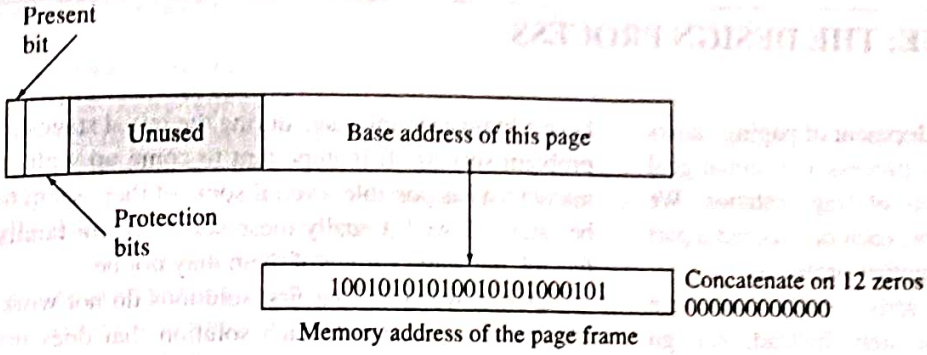


Figure 11.16 Page table entry

- *100*—The page can only be executed. This will detect code being used as data.
- *101*—The page can be read or executed. This will prevent the code from being written.
- *110*—The page can be written or executed. This is not likely to be used.
- *111*—The page can be read, written, or executed. This means there is no protection at all.

The protection information is a bonus we get from paging that allows extra hardware protection for the program. With page protection fields, it is not possible to write into the instruction area. Since the instructions are usually loaded first in memory, they occupy the low addresses, and those pages are set to execute only. This tends to find the use of 0-valued pointers (uninitialized pointers often have a value zero). It is also possible to ensure that constants are not written over by marking the pages they are in as read-only.

In Section 11.7.1, we will talk about the use of the "present" bit (but we will bet that you can guess what it is used for).

11.2.11 Paging Summary

With paging, the process sees a contiguous, linear address space that is easy to work with. The paging is transparent to the process in that the process does not have to do anything different than it would in a nonpaged system. The physical memory is divided up into a large number of small chunks (page frames) that are all the same size. These are simple to manage because no dynamic memory allocation (as described in Section 10.4 and following sections) is required, there is no external fragmentation, and everything fits in fixed-size slots.

Memory management has been made much easier. But we have had to get considerable hardware help to achieve this end.

We have done as much as we can with sharing memory space. If we want to increase the efficiency of memory use, we have to think about preempting memory like we did the processor, and that is what we will do in the next few sections.

Paging eliminates external fragmentation and the need to do dynamic memory allocation.

DESIGN TECHNIQUE: THE DESIGN PROCESS

Our presentation of the development of paging shows a typical example of a design process. Our initial goal was to ameliorate the effects of fragmentation. We went through a series of steps: each one solved a part of the problem, but caused another problem.

One does not generally arrive at a solution to a design problem in a single step. Instead, you go through a series of attempts, none of which quite work, but some of which have the seeds of the eventual solution. This is why it is recommended that you have a brainstorming stage during the initial stages of problem solving. It is important to come up with as many ideas as possible, even if some of them seem to be fatally flawed. Usually these solutions *are* fatally flawed, but some variant of them may not be.

Do not worry if your first solutions do not work. This is to be expected. Each solution that does not work teaches you a little more about the problem and helps you understand it a little better. Plan to iterate on solutions.

Iteration is a natural and necessary part of design.

*11.3 MEMORY ALLOCATION CODE WITH PAGES

In this section, we will redo the memory allocation code from the previous chapter, but with pages to show how much simpler memory allocation is when you have fixed-size pages instead of variable-sized segments.

The memory request structure is almost the same, except we keep the address of an array in which to put the base addresses of the pages allocated. The block list becomes a list of free pages. Initially, all the pages are in the free page list.

PAGED MEMORY ALLOCATOR: DATA

```
const int PageSize = 4096;

// The structure for memory requests
struct MemoryRequest {
   int nPages;     // size of the request in pages
   Semaphore satisfied;    // signal when memory is allocated
   int * pageTableArray; // store page numbers here
   MemoryRequest *next, *prev;  // doubly linked list
};

// The memory request list
// keeps a front and back pointer for queue discipline
MemoryRequest * RequestListFront, *RequestListBack;

// The structure for the free page list
struct FreePage {
   int pageNumber;
```

```
    FreePage *next;
};
// The free page list
FreePage * FreePageList;
int NumberOfFreePages;

// The initialization procedure needs to be called before
// any requests are processed.
void Initialize( int npages ) {
    RequestListFront = 0;
    NumberOfFreePages = npages;
    FreePageList = 0;
    // put all the pages on the free page list
    for( int i = 0; i < NumberOfFreePages; ++i ) {
        FreePageList = new FreePage( i, FreePageList );
    }
}
```

Memory requests are handled nearly the same way as before.

PAGED MEMORY ALLOCATOR: TRYALLOCATING

```
// The request procedure: request a block to be allocated
void RequestBlock( int npages, Semaphore * satisfied,
        int * pageTableArray ) {
    MemoryRequest * n = new MemoryRequest( npages, satisfied,
        pageTableArray, 0 , 0);
    if( RequestListFront == 0 ) {// list was empty
        RequestListFront = RequestListBack = n;
    } else {
        RequestListBack->next = n;
        RequestListBack = n;
    }
    TryAllocating();
}

// The allocation procedure
void TryAllocating( void ) {
    MemoryRequest * request = RequestListFront;
    // look through the list of requests and satisfy any ones you can
    while( request != 0 ) {
        // can we allocate this one?
        if( CanAllocate( request ) {// yes we can
            // remove from the request list
            if( RequestListFront == RequestListBack ) {
                // it was the only request on the list
                // of the request list is now empty
                RequestListFront = 0;
                // make sure we drop out of the loop
                request = 0;
            } else {
                // unlink it from the list
```

```
                request->prev->next = request->next;
                request->next->prev = request->prev;
                MemoryRequest * oldreq = request; // save the address
                // get the link before we delete the node
                request = request->next;
                delete oldreq;
            }
        } else {
            request = request->next;
        }
    }
}
```

We free a page table by freeing each page.

PAGED MEMORY ALLOCATOR: FREEPAGES

```
// Free a set of pages
void FreePages( int npages, int pageTable[] ) {
    for( int i = 0; i < npages; ++i ) {
        FreePageList = new FreePage( pageTable[i], FreePageList );
    }
}
```

We allocate a request if there are enough free pages for it. We take the free pages off the free page list.

PAGED MEMORY ALLOCATOR: CANALLOCATE

```
// See if we allocate one request
int CanAllocate( MemoryRequest * request ) {
    if( request->npages >= NumberOfFreePages ){
        NumberOfFreePages -= request->npages;
        int * p = request->pageTableArray;
        for( int i = 0; i < request->npages; ++i ) {
            *p++ = FreePageList->pageNumber;
            FreePage * fpl = FreePageList;
            FreePageList = FreePageList->next;
            delete fpl;
        }
        return True;
    }
    return False;
}
```

*11.4 SHARING THE PROCESSOR AND SHARING MEMORY

In previous chapters, we have examined the processor resource as the primary *time resource* in a computer system. The processor is a serially reusable resource, that is, the processor can only be used by one process at a time. But the processor can be switched between processes rapidly and cheaply. This is called time-multiplexing. Managing a time-multiplexed resource only involves scheduling decisions, since keeping track of the resource is easy: you just remember who has it currently.

time resource

Memory, on the other hand, is a *space resource* that can be divided up into pieces and shared between several processes at the same time. This is called space multiplexing.

space resource

Of course, a space resource can also be time multiplexed. For example, a hotel is divided into rooms and can have a number of guests at one time, each in their own room (space multiplexing). (See Figure 11.17.) A guest in the bridal suite might be moved to another room when a newlywed couple comes to stay, and moved back in again when they have left (time multiplexing).

A space resource is more complicated to manage than a time resource, since both time and space multiplexing are possible with a space resource. In the last chapter, we looked at various techniques for sharing memory between processes, that is, space multiplexing of memory. But memory can also be time multiplexed, just like a processor. We preempt a processor by saving the processor state into memory. The state of memory is its contents. We preempt memory by saving the memory contents onto disk. In this chapter, we will look at techniques for time-multiplexing memory. We will start with the historical methods of swapping and overlays to set the scene, and then discuss the main topic of this chapter, virtual memory.

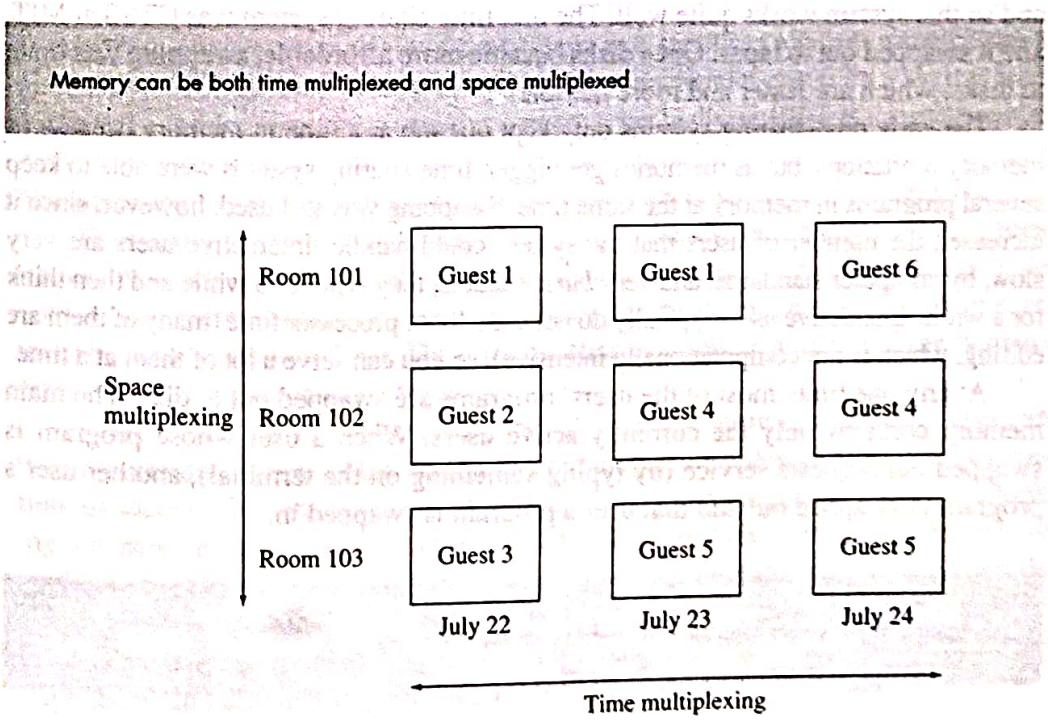


Figure 11.17 Time and space multiplexing of hotel rooms

DESIGN TECHNIQUE: MULTIPLEXING

Time and space multiplexing in general was discussed in Section 1.2.2., time multiplexing of the processor was discussed in Chapter 5, space multiplexing of memory was discussed in Chapter 10, and time multiplexing of memory was discussed in this chapter.

Multiplexing is common wherever resources are shared and used for multiple purposes. For example, we use time and space multiplexing when we use windows on the display screen.

For more discussion of multiplexing, see Section 13.1.

*11.5 SWAPPING

The first operating systems were batch systems that ran only one program at a time. The memories at the time were so small that it was not practical to have more than one program in memory at a time. But soon people starting thinking about time-sharing systems, and it became desirable to run more than one program at a time. Since it was not possible to fit more than one program in memory at the same time, it was necessary to time-multiplex the memory. Here is how they did it.

A program would be running and interacting with a user at a terminal. After that user's time allotment was up, the entire program was written out to secondary memory (tape or disk), and another user's program would be read back in from secondary memory. Then that user would run for a while. (See Figure 11.18.) This mechanism is called *swapping* because one process was "swapped" (exchanged) with another. It is really quite fast to write a job out and read another job in (a fraction of a second), and so this system works quite well. The first time-sharing system was CTSS at MIT, and it swapped out to tapes. Once disks became more affordable, swapping was done to disks, which are faster and more flexible.

swapping

The early time-sharing systems only kept one job at a time in memory because of memory limitations, but as memories got bigger, time-sharing systems were able to keep several programs in memory at the same time. Swapping was still used, however, since it increased the number of users that the system could handle. Interactive users are very slow, by computer standards, and very *bursty*, that is, they type for a while and then think for a while. Interactive users typically do not use a lot of processor time (many of them are editing, which is not computationally intensive), so you can serve a lot of them at a time.

bursty

At any one time, most of the users' programs are swapped out to disk. The main memory contains only the currently active users. When a user whose program is swapped out requests service (by typing something on the terminal), another user's program is swapped out and that user's program is swapped in.

Swapping allows you to keep the most useful processes in memory.

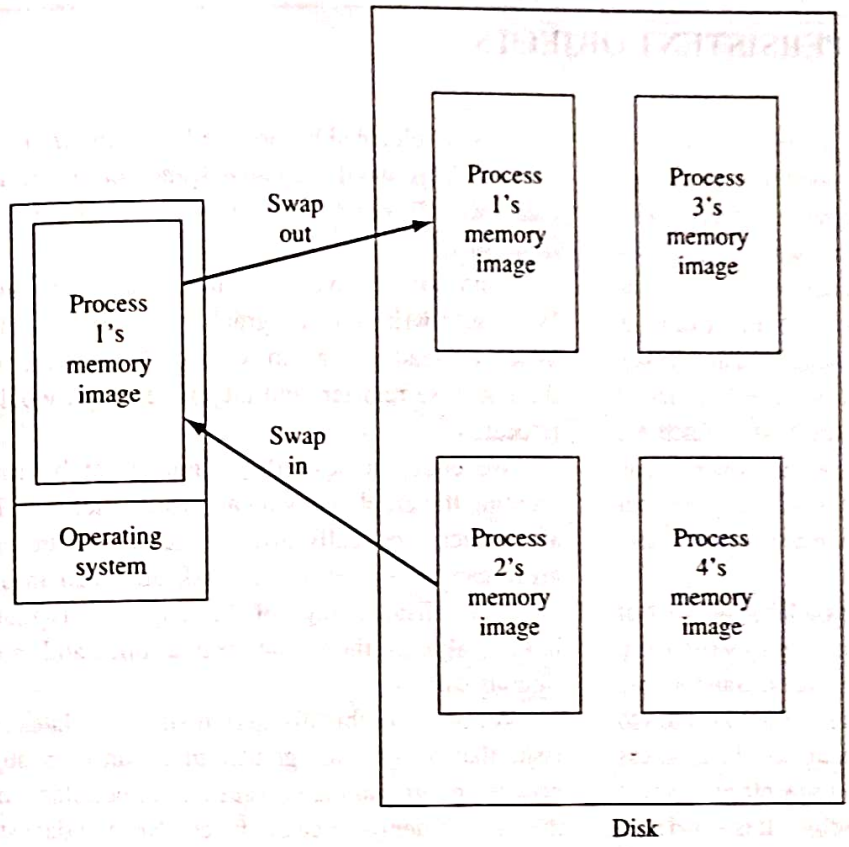


Figure 11.18 Swapping

11.5.1 Efficient Resource Use and User Needs

In swapping, we see an example of a technique that sacrifices efficient resource use to meet the needs of the users. Swapping jobs in and out of memory takes up resources without actually causing programs to progress, and so is not really an efficient use of resources. A batch system is more efficient in terms of use of the computer resources. However, users like interactive systems and feel it is worth it to sacrifice efficiency for convenience.

The history of operating systems has shown a steady trend away from a concentration on efficient resource use and towards ease of use. Today's workstations devote a tremendous amount of their resources to graphical user interfaces that are there because users like them. The concentration is on the efficiency of the human's resources, not the computer's resources.

There is another way to look at this which might be more illuminating. There are two important activities in the history of operating system design. The first is a continuous increase in services to the user. Operating system designers are always looking for new services they can provide that will help their users program and use applications. Once the desired functionality is settled, then the operating system designer tries to implement this functionality as efficiently as possible. As the services to users have gotten more elaborate, they have also gotten more resource-intensive to implement, and efficient implementation has remained very important.

DESIGN TECHNIQUE: PERSISTENT OBJECTS

Swapping pretty much requires dynamic relocation of processes. When a job is swapped out and later swapped in, it might be swapped into a different place in memory than where it was when it was swapped out.

What about extending the idea of swapping to data structures instead of entire programs? For example, suppose you had a large graph implemented as a linked structure. Each node in the graph is a C++ structure or class and is the linked list of arc nodes. Each arc node points to another node. Such a data structure can be used to represent general graphs with any number of outgoing and incoming arcs. Such a structure is also filled with pointers.

Suppose we were dealing with a large number of such graphs, and we wanted to be able to write them out to disk and read them in again later. Suppose we took the naive approach and just wrote out the nodes to disk, and then, later, read them in again. Well, unless we read each node into exactly the same place as it was before, the pointers would be all wrong. It is a bad idea to write pointers out to disk in any case.

Also, if we wanted to write out such a graph from one program and read it into another, there is very little possibility that the pointers will remain correct after doing this.

How can we be more clever about this? Well, one way is to convert the graph into a more portable form when we write it out. We can convert pointers into unique character string names, and write out pointers as strings. When we read the graph in again, we have to convert those strings back into pointers. This requires keeping a map of strings to pointers. This approach will work, but it involves a lot of processing.

Another strategy would be to use the technique used to relocate whole programs. In a program, every address is relocated with a relocation register. In the data structure, all addresses could be offsets from a base pointer that is some address below all the pointers in the graph. Each address in the graph would have to be relocated before use by adding in the base pointer. This would involve some inefficiency because we will not be able to rely on the hardware to do it for us.

But it would be very easy to relocate such a graph. We would write out the graph with just the offsets. When we read it in again, we would just have to set the one base register, and the entire graph would be relocated.

We could achieve this same effect by implementing the graph using an array of structures. Then all pointers are really array indices, and the entire array can be written out to disk and read in again later. The disadvantage of this approach is that we have to allocate the whole array at once and in contiguous storage.

We bring up this discussion since it relates to an issue that is assuming greater importance as object-oriented programming becomes more popular. This is the issue of **persistence** of objects. Normal data structures only exist as long as the program that defines and manipulates them exists. If you define a binary tree in a program, that tree goes away when the program exits. If you want to save any information, it must be saved as a file in the file system. Normally, the file system is the only means of persistent storage in the computer system.

But now databases also provide persistent storage of data, and, with the increase in interest in object-oriented programming, there has been an increase in interest in object-oriented databases. Now a true object-oriented database would allow you to store objects. But objects contain data and, usually, pointers. For example, a graph might be an object. Therefore, we have to find a way to store these data structures on persistent storage, and read them back in again later. Using the relocation technique would allow this to be done efficiently. This is an example of applying the relocation idea in another context.

The important point to understand is that efficiency is not the most important criterion for what services to offer, but it is a very important criterion in deciding how to implement those services.

User needs are the primary concern in designing what operating system services to offer. Efficiency is the primary concern in designing how to implement operating system services.

DESIGN TECHNIQUE: MULTIPLE DESIGN GOALS

We decided that we wanted to provide user processes with large virtual address spaces, and then we went to a lot of trouble trying to implement them efficiently.

All design projects have two steps, the design of the interface and the design of the implementation. The interface determines what functionality the system will have, and the implementation determines a good structure to implement that functionality.

When we are designing the interface, our primary duty is to the users of the interface. We want to make the interface as convenient to use as possible. We look at user needs and try to meet them. Once the functionality has been decided upon, we start designing the implementation. In this stage, our duty is to design as efficient an implementation as possible.

There are several caveats we have to add to this idealized view of design. First, "efficiency" is not a single concept. In implementing an interface, we might be concerned with any or all of the following forms of efficiency:

- How much space the programs uses.
- How much processor time the program uses.
- How much engineering time the development takes.
- How much computer time the development takes.
- How expensive it is to make changes in the product after it is delivered.
- How much delay there is until the product is ready.

In fact, the measure of efficiency we are interested in is probably some combination of these measures. We say this to emphasize that by "efficiency" we do not mean only how fast the program runs.

We described the process in two stages: first you design and specify the interface, and then you do the implementation. In real projects, it is not so cut and dried. If we find that a certain feature is very expensive to implement, we will probably go back to the interface design and see if we can modify it a little to something that is just as useful (or almost as useful), but easier to implement. User needs are the primary consideration, but not the only consideration. And besides, one of the things the user needs is likely to be a reasonably efficient implementation.

So, things are not as simple as we indicated, but the basic idea is still one to keep in mind: the two stages, interface design and implementation design, are logically separate and have different goals. There is mixing because that is how life is, but you should keep them separate as much as you can.

Design goals critically affect design choices.

*11.6 OVERLAYS

Swapping was used to allow several programs to run at the same time, even though the memory was not large enough to hold several programs. But sometimes a single program by itself was too large to fit into memory, and that presented a more serious problem. Swapping is a convenience that saved time. It wasn't necessary because you could always run the programs one at a time, but if a single program was too large to fit into memory, then you could not run it at all.

If the data area was too large, the program could read it in from disk (or tape or cards) in pieces and work on it a little at a time. But what if the program code itself were too large to fit into memory? *Overlays* were a solution to this problem.

Overlays

To understand the idea of overlays, let's take a specific example. Suppose you have a statistical program that offers four different kinds of statistical tests. Each statistical test is independent of the others, and the user only asks for one at a time. Suppose the code for all four tests will not fit into the memory, but the code for each individual test will fit into memory. What you do is link the program so that the code for all four tests is placed in the same memory area. The program consists of a small core program that controls overlays and reads user input, and then a large overlay area for the code for a statistical test. The core code determines which test is needed, loads in that code, and then jumps to it. When the test is completed, the core code looks for the next test desired, loads in that code, and then jumps to it.

Figure 11.19 shows an example with four overlays. The core program takes 100K, and the overlays take 300K, 250K, 100K, and 300K. The full program is 1050K, but it will run in 400K.

Overlay facilities were developed that were generic, that is, that would work for any program. You just had to tell it which parts of the program should share the same memory areas, and the overlay loader would take care of everything. Each time a procedure call was made to an overlay area, the code would check to see if the correct overlay was present. If not, it would read in the correct code and then make the call.

More complex programs had a tree of overlays, and it was often quite difficult to design the overlay structure for a program, even when the overlay system was handling the mechanics of overlaying.

The overlay technique worked, but it was a lot of effort for the programmer to figure out how the overlay structure should work, and there was a lot of run time overhead to getting overlays to work. But the idea was very good. In the next section we will look at virtual memory, which is a more general and more effective solution to this problem.

Overlays allow you to run a program that is larger than the address space you are running it in.

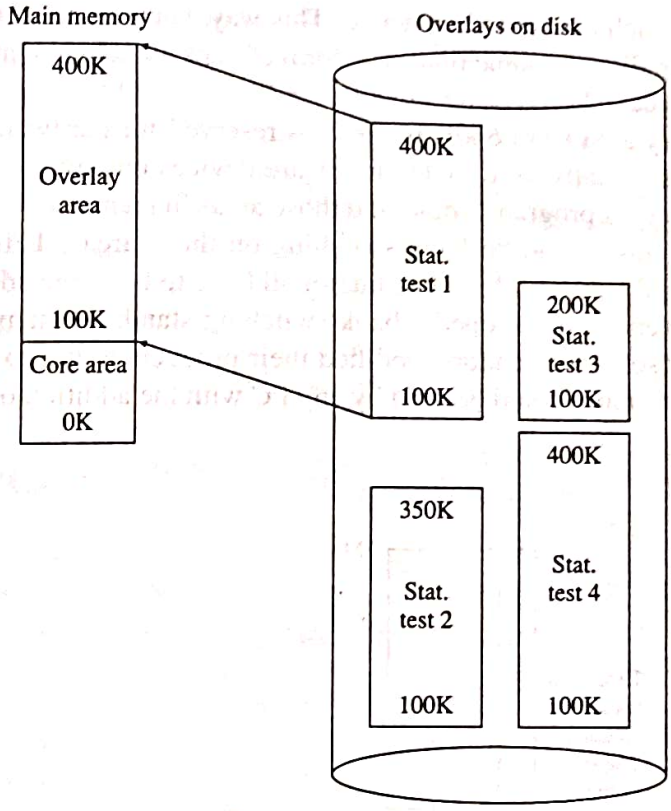


Figure 11.19 Overlays

11.6.1 Overlays in PCs

IBM PCs (and their clones) did not have enough memory for some programs. The problem was not that the memory was too expensive, but that the architecture of the 8086 and 8088 machines did not have a large enough logical address space. The total logical address space was 1 Mbyte (20 bit addresses), but even that was hard to use because of the awkward segment architecture it used. The IBM PC architecture allowed programs to use 640 Kbytes, and reserved the rest of the 1 Mbyte for hardware devices such as video boards that had frame buffers in the address space.

As PCs became more powerful, this memory limitation became very constraining, and it limited what could be done on PCs. PC vendors looked for ways around the problem. One method was overlays. Overlays are the traditional software method of running a large program in a small address space. Many popular programs used overlays.

But overlays were not the only solutions to the memory problem that was used in PCs. There were also more hardware-oriented solutions. The traditional hardware solution was the idea of *bank switching*. In bank switching, you have two memories that both are set to the same addresses. For example, you might have two memories that cover the range from 768K to 896K. There is an instruction to switch them, that

bank switching

is, to indicate which one you want active. This way, you can have more available memory, but not all at the same time. The form of bank switching that became popular on PCs was called *expanded memory*. (See Figure 11.20.)

The memory area from 640K to 1M was reserved for hardware devices, but not all of it was actually used by the hardware devices on any one system. With expanded memory, a program would find those areas in memory between 640K to 1M that were unused, and do bank switching on those areas. This method was good for things like disk buffers that did not all have to be in the address space at one time. The vendors developed a bank switching standard, many boards were sold, and many software vendors modified their products to use expanded memory. Expanded memory could be used by any PC with the addition of an expanded memory board.

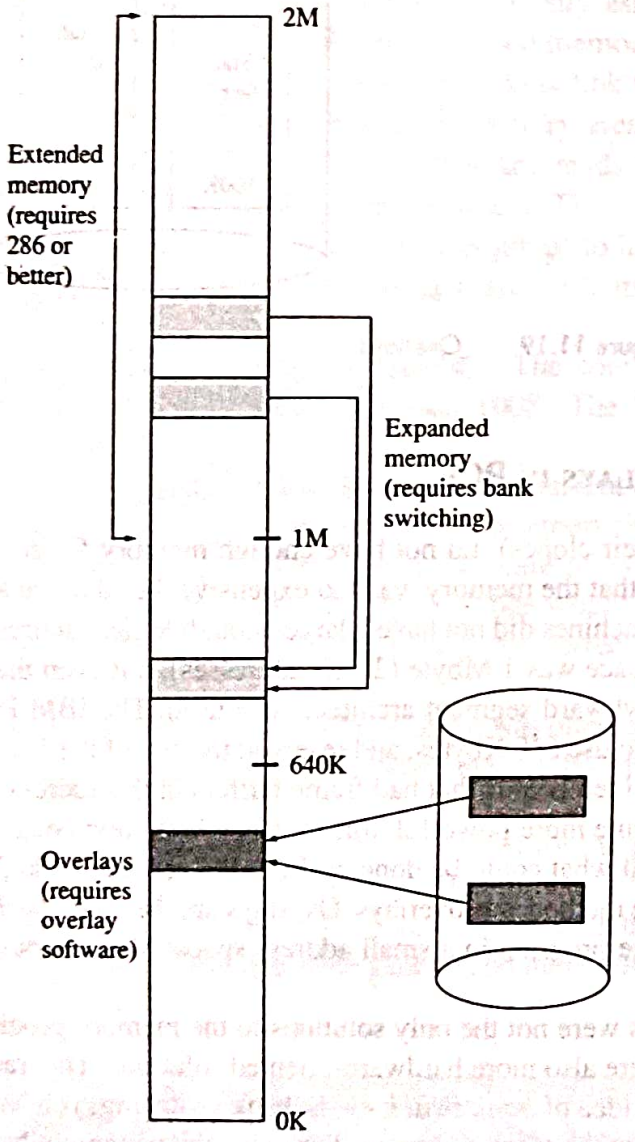


Figure 11.20 Overlays, expanded memory, and extended memory in PCs

The other method used to increase the address space in PCs was called *extended memory*. Extended memory was just memory whose physical addresses were above 1 Mbyte. This meant that it was not accessible to 8086 machines, but it could be addressed by 80286 machines. The problem was that DOS required the 80286 machines to run in 8086 mode while doing DOS system calls. So systems were developed where the machines would run in 80286 mode while in the user program, and then switch back to 8086 mode when making DOS system calls. This method was complicated by the awkward way that 80286 machines would switch modes. Figure 11.20 shows the three methods.

11.7 IMPLEMENTING VIRTUAL MEMORY

Swapping is a good solution because it allows a number of programs to run at the same time and it is handled entirely by the system. But swapping only applies to whole programs; and so we cannot be selective enough about what we keep in memory. Overlays are a good solution because they allow us to move small parts of program in and out of memory. But overlays require the programmer to design the overlay structure. What we would like is a method that combines the good features of both ideas, but without their problems. There is such a method, and it is called *virtual memory*.

virtual memory

We almost have virtual memory already in our paging system. The paging system divides the program up into a number of small parts, and this division is done based purely on logical addresses so the programmer does not have to make any decisions about how to divide up the program. In fact, paging is entirely transparent to the programmer. The hardware is already checking each memory reference and picking up the page base address to use. Using the protection bits, it is already checking each page reference for validity. To achieve virtual memory, we simply need to drop the requirement that every page be in memory while a program is running. When a process references a page that is not in memory, an interrupt is generated. This interrupt does not end the program: instead, it is intercepted by the operating system. The operating system figures out which page was found to be missing, reads it in from disk, changes the page table to indicate that the page is now in memory, and restarts the program at the instruction that caused the exception. The program does not even know that this happened. From the point of view of the process, all the pages are in memory, and that is why it is called *virtual* memory.

Virtual memory combines the good points of swapping and overlays. It is like overlays in that only part of the program is in memory at any one time, and the rest of it is out on disk. It is like swapping in that the missing parts of the program are brought in automatically by the system, and the programmer does not have to worry about how this happens or how to logically divide up the program.

Figure 11.21 show the components of a virtual memory system. The user sees a large linear virtual address space. Only parts of the virtual address space are in physical memory. The rest of it is "virtual" and is kept on the disk until needed. The disk contains an image of the entire virtual address space, even the parts that are in

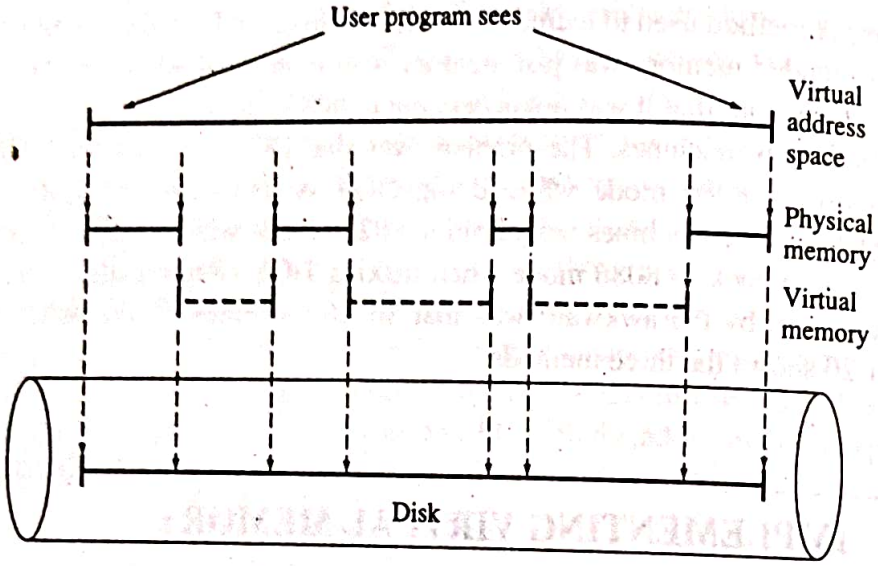


Figure 11.21 Virtual memory

physical memory. The virtual memory system module of the operating system maintains the illusion of the virtual memory by moving pages from disk to physical memory when they are needed.

With virtual memory, some of the memory is simulated with disk.

11.7.1 Hardware Required to Support Virtual Memory

present bit

page fault interrupt

To implement virtual memory, we will add a new field to each page table entry. This field will be a single bit, and it will be called the *present bit*. If the present bit is 1, the page is present in memory and the base address field is valid. If the present bit is 0, the page reference is a legal program reference but the page is not currently in memory. When a page is accessed, the hardware checks the present bit, and if the present bit is 0 it generates an interrupt, usually called a *page fault interrupt*. The operating system handles the page fault exception by reading in the page and fixing up the page table.

To make this clearer, let us first look at the algorithm the hardware will implement for making memory accesses. The following code gives the algorithm that the hardware uses.

THE HARDWARE VIRTUAL MEMORY ACCESS ALGORITHM

```
const int LogicalPages = 1024;
const int ByterPerPage = 4096;
const int OffsetShift = 12;
```

```
const int OffsetMask = 0xFFF;
const int PhysicalPages = 512;
enum AccessType { invalid = 0, read = 1, write = 2, execute = 3 };
struct PageTableEntry {
   int pageBase : 9;
   int present : 1;
   AccessType protection : 2;
   int fill : 4; // fill to 16 bits;
};

PageTableEntry UserPageTable[LogicalPages];

int MemoryAccess( int logicalAddress, AccessType how, int dataToWrite
= 0 ) {
   int page = logicalAddress >> OffsetShift;
   int offset = logicalAddress & OffsetMask;
   PageTableEntry pte = UserPageTable[page];
   // check if the access is valid
   if( how != pte.protection ) {
      // you CAN read a read/write page
      if( !(how = read && ptr.protection = write) ) {
         // otherwise it is a protection violation
         CauseInterrupt( ProtectionViolation );
         return 0;
      }
   }
   if( pte.present == 0 ) {
      GenerateInterrupt( PageFault, page );
      return 0;
   }
   int physicalAddress = (pte.pageBase << OffsetShift) + offset;
   switch( how ) {
      case read:
      case execute:
         return PhysicalMemoryFetch( physicalAddress );
      case write:
         PhysicalMemoryStore( physicalAddress, dataToWrite );
         return 0;
   }
}
```

First, the algorithm checks if the access is a valid one according to the protection bits, and if not, it causes an interrupt. Then, it checks if the page is present in memory, and if it is not, it causes a page fault interrupt. Otherwise, it executes the appropriate memory operation.

11.7.2 Software Required to Support Virtual Memory

The operating system will reserve an area on disk to hold the image of the virtual address space of each process. This is called the *swap area*. Some (or all) of the pages in the swap area will be in memory at any one time.

swap area

There are four times when an operating system has to consider page tables and page faults.

- When a process is created.
- When a process is dispatched.
- When a page fault occurs.
- When a process exits.

Figure 11.22 show the parts of the virtual memory system that are affected by each event, and the sections below describe in detail how each event is handled.

Process Creation To create a process *P*, the operating system must:

1. Let *N* = the size of *P* in bytes (which is obtained from the load module). Round *N* up to the next highest multiple of `pagesize`.
2. Allocate *N* bytes for *P* in the swap area on disk.
3. Allocate space in main memory for *P*'s page table. This will require *N*/*pagesize* page table entries.
4. Initialize the swap area by copying in the machine code and initialized data sections from the load module.
5. Initialize the page table by marking all the pages as not present and setting the protection bits appropriately for each page (execute-only for code, read-only for constant data, read/write for other data).
6. Record this information in the process descriptor (the location in memory of the page table and the location in the swap area of the program image).

Process Dispatch To dispatch process *P*, the operating system must:

1. Invalidate the TLB (since we are changing page tables).
2. Load the hardware page table base register with the address of the process's page table (kept in the process descriptor).

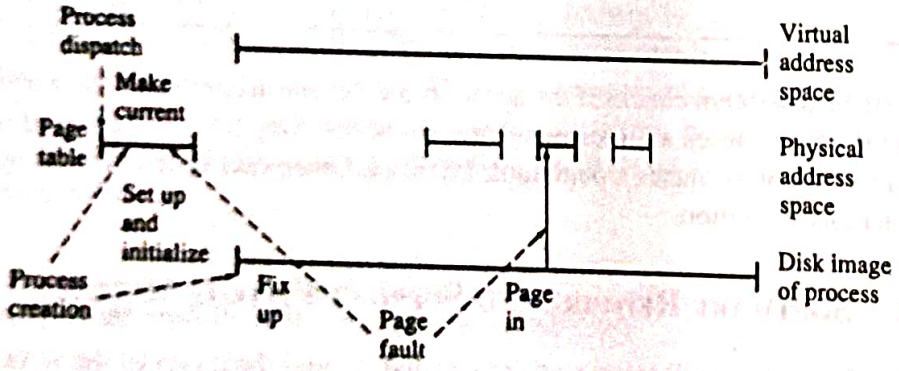


Figure 11.22 Virtual memory events

Page Fault To handle a page fault for process *P*, the operating system must:

1. Let *K* = the number of the page that caused the fault (the hardware will provide this information with the page fault).
2. Find an empty page frame. This involves finding a page to replace, and writing it out to the swap area (*if* it has been changed since it was read in from the swap area).
3. Read in page *K* from disk (it will be on the swapping disk at disk location `SwapOrigin(process P) + (K*pagesize)`.
4. Fix up the page table entry by: (1) setting the present bit to 1; and (2) setting the frame number to the address of the newly allocated frame.
5. Restart the instruction that caused the page fault (the hardware will have already undone any effects of the partial execution of that instruction, so it will be safe to reexecute it).

Process Exit When a process exits, the operating system must:

1. Free the memory used by its page table.
2. Free the disk space used by its memory image.
3. Free the page frames the process was using.

As you can see, the basic idea and implementation of virtual memory is quite simple. The hard part is deciding which page to evict from memory when a page fault requires you to bring in a new page. We will spend a considerable amount of time on that question, since it is the critical issue in the design of a paging system. But first we will look into whether virtual memory is a practical idea or not.

The events important in virtual memory are: process creation, process dispatching, page fault, and process exit.

11.8 WHAT IS THE COST OF VIRTUAL MEMORY?

We do not want to get a page fault on every memory access, or else the program would be hideously slow. The percent of memory references that causes page faults is called the *page fault rate*. The page fault rate must be very low for paging to be a practical method. page fault rate

Let's do some calculations to see what a reasonable page fault rate would be. Suppose a memory cycle normally takes 200 nanoseconds if the page is in memory. If we get a page fault, we will need to execute a few hundred to a few thousand instructions to handle the page fault interrupt, and we will have to read in the page from disk. On a 50 MIPS machine, 2000 instructions takes 40 microseconds. Reading 4K

from disk will take, say, 20 milliseconds. The 20 milliseconds dominate the page fault time, so let's ignore the other delays. This means that an instruction that normally takes 200 nanoseconds will take 20 milliseconds, or 100,000 times longer.

Obviously, we don't want to have very many page faults, or the program will be slowed down by a huge factor. If one instruction out of 100,000 caused a page fault, then the average memory access time would be double what we would get if there were no page faults, since 100,000 memory accesses would take 200,000 memory cycle times. This would cause the program to run at half speed, but even this is an unacceptable speed penalty. Suppose we wanted to run only 10 percent slower than unpaged memory. To achieve this, we would need to limit the average page fault rate to, at most, one page fault for every 1,000,000 memory accesses.

The following table shows a range of page fault rates and the corresponding slowdown they will cause. If a slowdown factor is 2, that means that the average memory access will take twice as long as it would in an unpaged system (where the entire program is in memory).

Page Fault Rate	Slowdown Factor
1/1,000,000	1.1
1/100,000	2
1/10,000	11
1/1,000	101
1/100	1001
1/10	10001
1	100001

The higher page fault rates and slowdowns (like a slowdown factor of 1001) are clearly in the unacceptable range.

11.8.1 Paging More than One Process

In the calculations in the previous section, we assumed that the paging disk was free to read in the page we needed. If there are other processes being paged, then the slowdown due to paging will be even greater.

Sometimes paging is done to a disk that is dedicated to paging, but if this is not the case then other disk I/O will also interfere with paging and increase the slowdown even more.

11.8.2 Locality

Now let us suppose, for purposes of argument, that page references were uniformly distributed throughout the address space. This would mean that each time an address

is generated, all possible addresses are equally likely. In order to achieve a page fault rate of one in 1,000,000, we would have to keep 999,999 out of every 1,000,000 pages in memory.

This would not give us much of an advantage over unpaged memory, but fortunately our supposition that page references are uniformly distributed throughout the address space is very far from being true. In fact, page references are extremely nonuniform. It is intuitively obvious why this would be so. After we execute an instruction, the next sequential instruction is almost always executed next. This instruction is almost certainly on the same page, and so the page references for instructions will almost always be the same page as the previous instruction. Even when you do not execute the next sequential instruction, you are branching at the end of a loop, and branching to a nearby instruction that is also probably on the same page.

spatial locality

temporal locality

Sequential instruction execution generates a type of locality called *spatial locality* because the next reference is spatially close (close in address) to the previous reference. Instruction loops generate a type of locality called *temporal locality* because the next reference to an instruction word is temporally close (close in time) to the previous reference to the same word. The loop executes the same set of instructions over and over. Spatial locality means that words near each other tend to get accessed in a short space of time, and temporal locality means that the same word will be accessed more than once in a short space of time. Since a page contains a large number of words that are close in address and are in memory for a period of time, paging benefits from both kinds of locality.

Data references are similar. If you are accessing an array sequentially, you will almost always be accessing a data word on the same page as the last word. Thus, data references exhibit spatial locality. In addition, programs tend to access the same data items over and over, that is, data references also exhibit temporal locality.

We saw this concept of locality before, when we talked about translation lookaside buffers (TLBs) in Section 11.2.6. The same locality that ensures that instruction and data references cluster in a few pages also ensures that the page table entry for these few pages will be in the TLB.

Since both instruction and data references exhibit a high degree of locality, we can expect that almost all page references will be to a few pages.

But how far is it from the vague "almost all" in the previous paragraph to the 999,999 in 1,000,000 that we need to make paging practical? Experience has shown that, for typical programs, if you have from 25 percent to 50 percent of the pages in memory at any one time, you will get acceptably low page fault rates, that is, on the order of one in a million.

So we see that virtual memory is not magical. With virtual memory, we can fit two to four times as many programs into memory. But we cannot normally fit 10 times as many programs in memory. On the other hand, a program might use a large data area that is used in a very localized fashion. Such a program might be able to have only a few percent of its data pages in memory at a time, and still have a fairly low paging rate. Such programs exist, but they are special cases and are not typical.

DESIGN TECHNIQUE: LOCALITY

The locality principle is used many times in these chapters. Locality allows TLBs to work, and it allows paging to work.

In fact, it is a generalization of the locality principle that is the basis behind any system that uses caching. Caching works because questions are not asked randomly. The same question, or closely related questions, are asked in clumps. Caching works because it takes advantage of this knowledge about the questions that will be asked.

The current directory concept allows the use of short file names in the most common cases. It works because the files asked for are localized, usually in the same directory as the previous file asked for, or at least in a nearby directory.

DESIGN TECHNIQUE: LATE BINDING

With virtual memory we do not bind pages to page frames until the first time the page is used. This is a form of lazy creation where the page in memory is not created until it is first needed. In programming languages we do not allocate stack space for a procedure until it is called. These are both examples of late binding. Early versus late binding is an example of the static versus dynamic tradeoff.

To learn more about late binding see Section 13.2.

11.9 VIRTUAL MEMORY MANAGEMENT

We will talk about page replacement algorithms in the next chapter. In this section we will discuss the modules and tables that must exist in an operating system in order to implement virtual memory. Figure 11.23 shows a block diagram of the data structures of a virtual memory subsystem. The data structures shown are:

- *Page frames*—This is the part of physical memory reserved for user pages. These are the page frames the virtual memory manager is allocating.
- *Page tables*—These are the page tables for all the processes in memory.
- *Free page frame list*—This is a list of all the page frames that are not currently in use by any process.
- *Modified free page frame list*—This is a list which contains the free pages whose current contents need to be written out to disk.

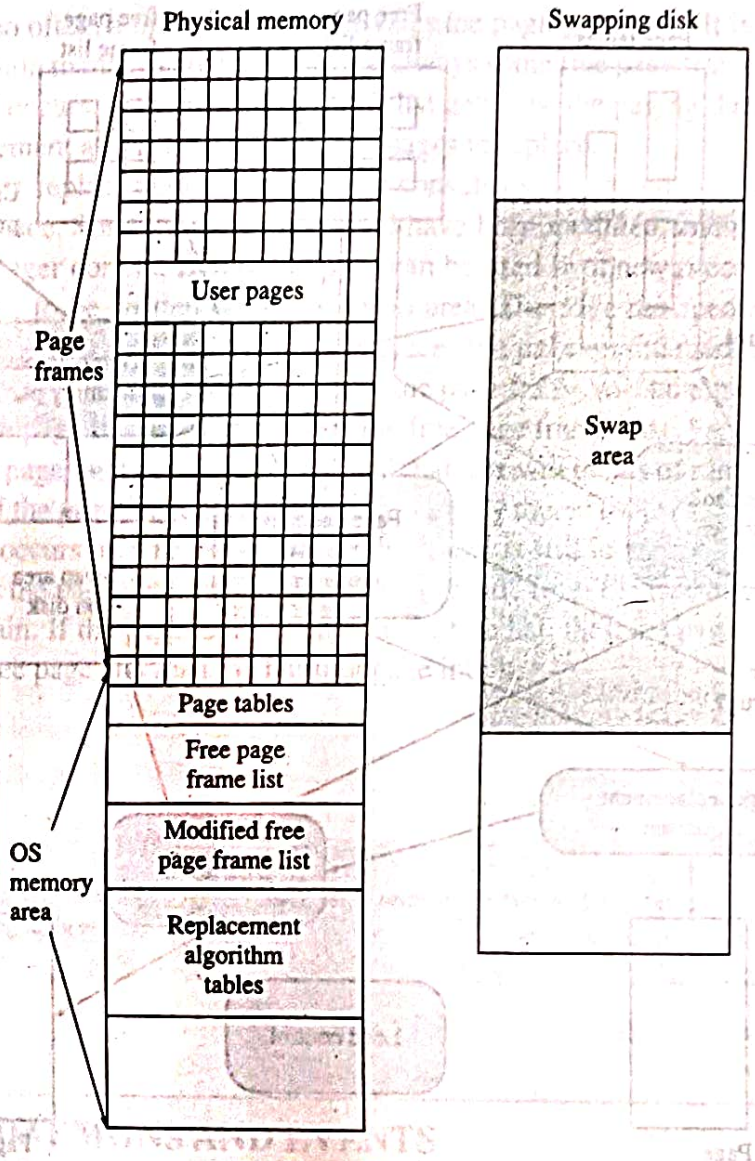


Figure 11.23 Virtual memory data structures

- *Replacement algorithm tables*—All page replacement algorithms keep tables of information about the pages. They use this information to decide which pages to replace.
- *Swap area*—This is the area on disk where pages are written when they are removed from memory. A complete image of the memory for every process is kept in the swap area.

Figure 11.24 shows a block diagram of the data structures, the procedures that use and modify them, and the events that cause these procedures to be executed. Events that cause changes in the tables are named with bold text. There are three such events: a page fault exception, a timer interrupt, and a create process system call interrupt. Memory management procedures are drawn as rectangles with rounded corners. Data tables are shown as rectangles.

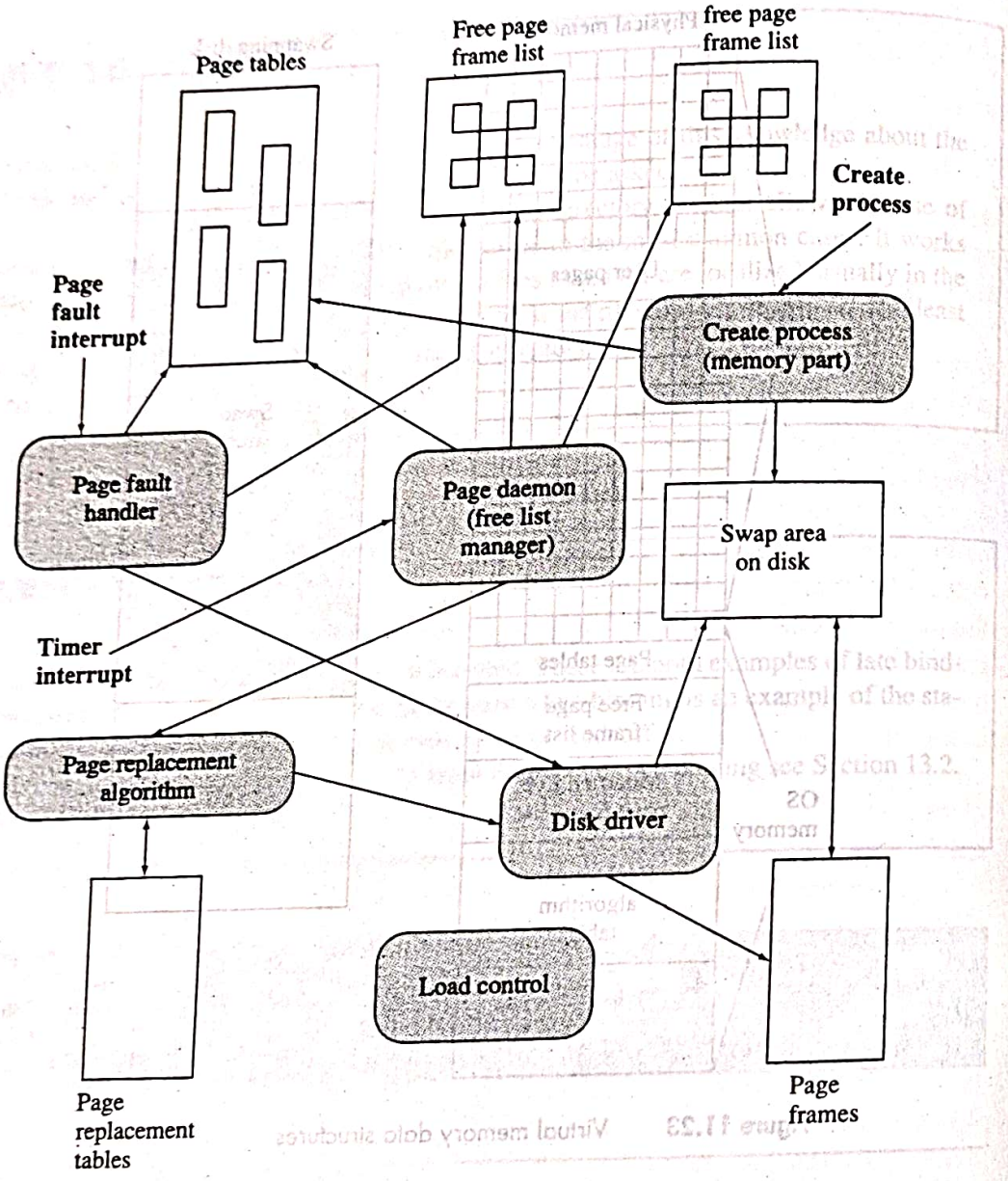


Figure 11.24 Virtual memory events and procedures

When a create process system call is made, the create process procedure is called. It sets up the page tables and the swap area for the process.

When a page fault occurs, the page fault handler is called. This examines the page tables and figures out which page needs to be read in. It allocates a page frame and calls the disk driver to read the page into the page frame.

paging daemon

An operating system may not wait until a page fault occurs to look for a page to replace. Operating systems often keep a pool of free page frames so a page fault can immediately be granted a page frame. There is a *paging daemon* that wakes up every so often and makes sure there are sufficient page frames in the free page frame pool. The paging daemon will call the page replacement algorithm to determine which pages to replace.

Every so often, a timer interrupt activates the paging daemon. It is the job of the paging daemon to make sure that there are always some free page frames to use when a page fault occurs. If the free page frame list gets low, the paging daemon calls the page replacement algorithm to pick some pages to replace.

The page replacement algorithm will consult its tables and determine the best pages to replace. Some of these pages will have been modified, and so the copy on disk is no longer correct. Before the page can be used by a new process, its current contents have to be written out to the swap area. The page replacement algorithm calls the disk driver to do this write, and places the page on the modified free page frame list. When the disk write completes, the page frame will be removed from the modified free page frame list and put on the free page frame list.

When a page fault occurs, the page fault handler will look in the free page frame pool to see if the page it needs happens to be already there. This would happen when a page fault occurs on a page that was replaced but is still in a "free" page frame. It still contains the page data, and so it can be "rescued" from the free page frame pool and used again. If the page is not found on the free list, then the page fault handler must get a free page and read the faulting page into it.

An operating system is largely a table manager. It responds to events by looking things up in tables and changing table entries.

11.10 DAEMONS AND EVENTS

Operating systems respond to events—a system call interrupt, a device interrupt, the arrival of a message over the network, etc. An operating system is principally made up of event handlers that are activated by these events. As such, the operating system is a passive entity that does not initiate actions on its own, but only responds to the actions of devices and the processes it is running.

The model of an operating system as an event manager is a good one, but it does not tell the entire story. Sometimes the operating system does do things on its own, instead of in response to external events. One example is the paging daemon we discussed in the last section. The operating system knows that free pages will be required because processes get a continual stream of page faults. So it prepares for this by keeping a pool of free pages to use when a page fault occurs. The paging daemon wakes up every so often and makes sure that the free page pool has enough free pages in it. If the pool is too low, then it starts freeing some pages.

Technically, the paging daemon is responding to a timer interrupt. But this is just so it can gain control periodically and fit into the interrupt-driven model of the operating system.

DESIGN TECHNIQUE: SYSTEM MODELS AND DAEMONS

Operating systems are essentially reactive systems, that is, they react to events. They are not proactive, in the sense that they do not go out looking for things to do where they can be useful.

Operating systems react to interrupts. But, if we include timer interrupts, then an operating system process can be activated every so often and go looking for useful things to do. The timer allows us to add proactive elements to a basically reactive operating system.

The concept of a daemon is a proactive part of an operating system. A daemon might wake up at 3 A.M. every night and see if parts of the disk need optimizing (to move files into contiguous blocks). Another daemon might wake up every 30 seconds and make sure there are enough free pages in the free page pool.

There are advantages to both the reactive and the proactive models, and you get the best of both worlds by combining them.

User interfaces are another example of this. The way user interfaces are implemented in graphical user interface systems is reactive. The interface consists of a number of screen objects which react to user input by calling parts of the application. So the application becomes a reactive system that reacts to user input. Generally this is a very good model and corresponds to the accepted user interface principle that the user should feel in control of the interface. But some things do not fit into this model, such as receiving messages from other parts of the system. In fact, a recent trend in user interfaces is the idea of an *agent*, that is, a daemon that is given a general task and goes off on its own, and is proactive in achieving its goals.

DESIGN TECHNIQUE: POLLING, SOFTWARE INTERRUPTS, AND HOOKS

There is another issue with daemons, and that is how one process can know when some event of interest had occurred. There are two general approaches to that problem: polling and interrupts. In the polling approach, you check over and over again to see if the event has occurred. Often you use a timer, and check at periodic intervals. For example, our paging daemon woke up every so often and checked to see if there were enough free page frames. A scheduling daemon might wake up every so often and examine the performance statistics to see if there is a need to change the scheduling parameters of the system.

Suppose you have a program that is displaying a directory. You would like the display to be kept up to date, even when other programs change the directory by deleting or adding files. One solution is to check every few seconds and see if the directory has changed. This is easy in UNIX because it keeps a time of last modification for each directory.

The interrupt solution would be a mechanism that would allow you to register interest in the directory, so that the operating system will inform you when a change has been made to it. This assumes that the operating system has a facility that allows you to register interest in a directory. Few operating systems have such a facility, and so you are usually stuck with the polling solution.

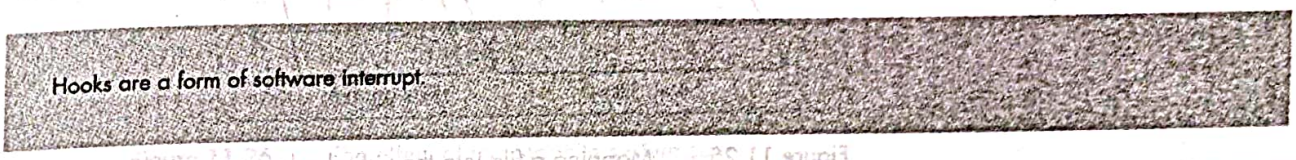
The ability to register interest in certain events is often called a **hook**. The idea is that any event in a computer system is caused by the action of some software, or else some software detects and handles the event. Since this code is executed anyway, it is handy if it can send a message to any interested processes that the event has occurred. This allows any process to get control when the event occurs.

This idea can be used inside a single program as well. For example, the emacs text editor defines a number of useful hooks. You can, for example, say you want a certain procedure to be called whenever a file name lookup fails. This gives you a chance to find the file in some other place and allows the file name lookup to succeed. Or you can get control when a file is written, and do the writing yourself in some special way.

Continued

The hook idea allows you to get control when certain events happen, and is a form of **software interrupt.** Hooks are more efficient than polling, and more flexible, too, since they allow you to get control before an event happens and to change the system's response to the event.

Notice that the direction of the call is reversed with a hook. Normally, the entity that is interested in the event would make the call, but with a hook, the entity that causes the event makes the call. This is useful since that is the entity that knows when the call should be made.



It is called a *daemon* because it acts on its own and does things, instead of responding to specific events. Modern operating systems have dozens of different daemons.

daemon

11.11 FILE MAPPING

A virtual memory system allows you to have a larger logical address space than physical address space. A copy of the process's logical address space is kept on disk because it may not always be in physical memory. Another way to look at virtual memory is that it creates a logical address space out of data stored on disk, and takes care of moving the parts of the address space that you are currently using into memory.

If you think about it for a moment, you will see that file I/O does exactly the same thing. Parts of the data on disk are moved into the logical address space so they can be used in the computation. File I/O is not as convenient as virtual memory because the user has to worry about when the data needs to be brought in memory and when it needs to be written back out again.

Virtual memory, as we have defined it, only maps process images from the swap area into memory. It will not map a file into virtual memory. It would simplify the operating system and the job of the user process to extend the virtual memory system to allow it to map any file into the virtual address space. Most newer operating systems allow this and the facility is called *file mapping*. Another name for file mapping is *memory-mapped files*. Once this is done, then all file reading and writing is accomplished with ordinary memory operations. That is, changing the file data in the virtual memory is like writing to the file, and reading the file data in virtual memory is like reading the file. This means that no special I/O operations are necessary; they are subsumed by ordinary memory operations.

file mapping
memory-mapped files

Figure 11.25 shows a file mapped into the virtual address space. Assume that the C++ pointer `file` refers to the beginning of the file when it is mapped into the address space. Then you can access the file with array operations:

- `x = file[912]; // file read`
- `file[912] = 12; // file write`

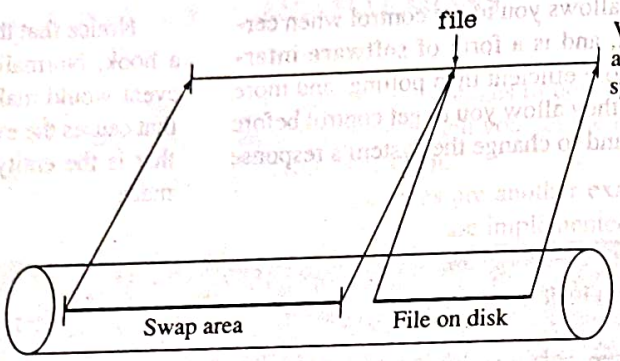


Figure 11.25 Mapping a file into the virtual adress space

Normally, we would map the file into a part of the virtual address space that is not used, for example, into the address space after the end of our program, data, and stack areas. The file-mapped area will not be copied to the swap area, like the other parts of the address space, because the file on disk acts as the disk backup for that part of the virtual address space.

File Mapping the Code Space When we described how a process is loaded in a virtual memory system, we noted that the code is copied from the load module into the swap area before we began execution of the program. If we have file mapping, we can avoid that step and use the load module itself as the disk backup of the program code. The code does not change, so we do not have to worry about damaging the load module by writing changes to it. If we already have the mechanism for file mapping, then any parts of the load module that are read-only can use this facility, and do not need to be copied into and take up space in the swap area. This will speed up loading because it avoids a disk-to-disk copy.

11.11.1 The System Call Interface

Let us look at what the system call interface to file mapping would look like. There will be two system calls:

`int MapFile( int openFileID, char * startAddress = 0, int startOffset = 0, int length = 0 )`—The open file `openFileID` is mapped into the caller's virtual address space. All or part of the file might be mapped in. The first byte in the file to be mapped is at file offset `startOffset` (this is usually 0), and `length` bytes will be mapped. If `length` is zero, then the rest of the file (from `startOffset` to the end of the file) is mapped in. The selected bytes of the file are mapped in the caller's virtual address space, starting at virtual address `startAddress` and continuing for `length` bytes. If `startAddress` is 0 (the default), then the operating system picks the virtual address where the file will be mapped. If the return value is negative, then an error occurred. Otherwise the return value is the virtual address the file is mapped into. Possible error returns are:

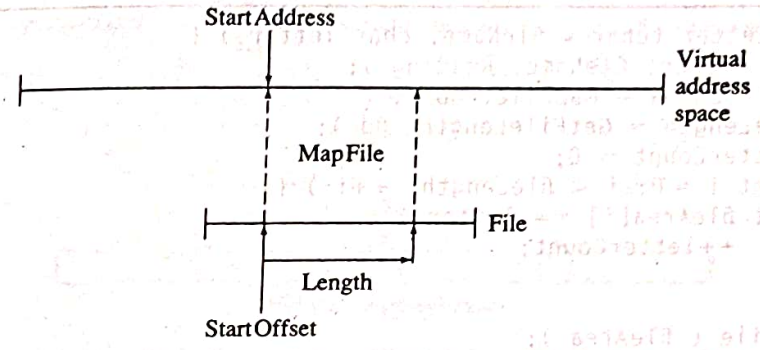


Figure 11.26 The effect of the MapFile system call

-1 ⇒ `fileName` could not be found.

-2 ⇒ `startAddress` is not a legal address.

-3 ⇒ the file would not fit in the allocated address space (from `startAddress` to the end of your allocated address space).

Figure 11.26 shows the effect of the MapFile system call.

- `int UnMapFile( char * startAddress )`—The mapped file is unmapped.

You map a file with `MapFile`, read and/or write the data, and then `UnMapFile` will unmap it.

11.11.2 An Example of Using File Mapping

As an example of how you would use the file-mapping interface, suppose we had a file that contained some text, and you wanted to count the number of lowercase 'a's in the text. The program in Figure 11.27 shows how you would use file mapping to do this.

The `MapFile` call allows you to specify a length and an offset in case you want to map the file in pieces. It is simpler to map an entire file, but sometime you will want to do it in pieces. This makes the mapping more like I/O, and is useful for very long files.

File mapping is most useful with very large address spaces where you can afford to use up address space for large files. It is likely it will be more used with the new generation of hardware with address lengths over 32 bits.

11.11.3 Advantages of File Mapping

File mapping means that the user does not have to do explicit I/O, and so the interface to the operating system is easier. You access and modify data in memory on disk the same way.

File mapping is more efficient than file I/O. Ordinary I/O requires a system call for every I/O operation. With file mapping, there are only two system calls. All file access is handled by the virtual memory system.

```
int CountLetter (char * fileName, char letter ) {
   int fid = open( fileName, Reading );
   char * fileArea = MapFile( fid );
   int fileLength = GetFileLength( fid );
   int letterCount = 0;
   for( int i = 0; i < fileLength; ++i ) {
      if ( fileArea[i] == letter )
         ++letterCount;
      }
   }
   UnmapFile ( fileArea );
   return letterCount;
}
```

Figure 11.27 Count letters

Another advantage of file mapping is that it reduces the number of copies of the data in physical memory. Figure 11.28(a) shows what happens normally. When a process reads a page of file data, it is first read into the operating system address space (we'll see this in Chapter 16). Then it copies the data into the user's address space. If two users read the same data, there will be two copies. Since normally address spaces do not share pages, there will be three copies of the page in physical memory.

Figure 11.28(b) shows what happens with file mapping. The page is brought into memory once, and mapped into the virtual address space of the process that mapped the file. The operating system never needs to map it. If a second process maps the same file, it will see the same physical page as its copy of the file.

This example shows one difference between the semantics of file mapping and ordinary file I/O. Changes to the file are seen immediately by all processes mapping the file. Ordinarily, a process would have to read the data in to see any changes in the file. With file mapping, you have a "live," shared copy of the file.

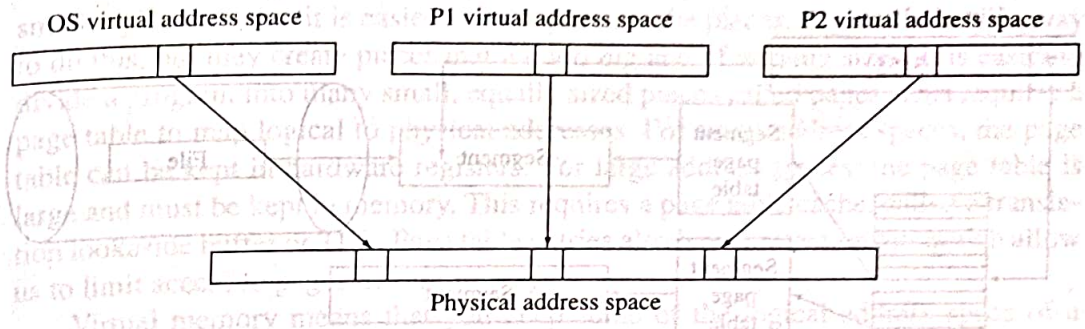
Even with file mapping, you still need the normal file I/O operations. One advantage of file I/O is that it is atomic. While you are reading or writing, the file is locked, and no other process can read or write the file at the same time. In addition, you need ways of increasing or decreasing the length of a file.

File mapping integrates virtual memory with the file system and allows you to place any file on disk into your address space.

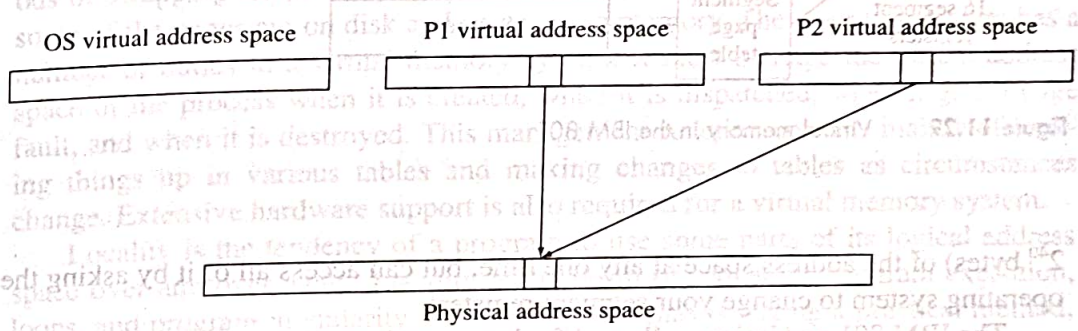
11.11.4 Memory and File Mapping on the IBM 801

The IBM 801 was the first RISC computer, and it was innovative in a number of ways, both in hardware and software. The designers of the 801 system used a segmented virtual memory that supported file mapping.

The virtual address space in the 801 consists of up to 2^{36} segments, where each segment can be up to 2^{28} bytes long. The IBM PC/RT used 4096 segments, so we will assume that number for the rest of the next few paragraphs. Each segment is mapped



(a) Each process has a physical copy of the data



(b) The processes share a physical copy of the file data

Figure 11.28 Multiple copies of file data in memory

with a page table. There is a systemwide segment table containing 4096 entries. Each entry in the segment table is a segment descriptor which contains the protection bits and a pointer to the page table for the segment. So the virtual address space consists of up to 2^{40} bytes (2^{12} segments of 2^{28} bytes each). This virtual address space is common to all processes running in the system (including the operating system). The protection bits in the segment table entries protect the segments from unauthorized use. Figure 11.29 shows the memory addressing and file-mapping facilities of the IBM 801 computer and operating system.

A running process does not have immediate access to all 4096 segments in the virtual address space (even if it does have the proper permissions). Virtual addresses on the 801 are 32 bits long. The 801 has 16 hardware segment registers, and the high-order four bits of a virtual address are the number of the segment register to use. The segment registers contain copies of the segment descriptors for 16 segments. Modifying the contents of the segment registers is a privileged operation, so if a user process wants to access a new segment, it must ask the operating system to do it.

So we have an interesting situation here that is different from the virtual memory systems we have seen before. First, there is only one address space for all processes, instead of each process having its own address space. All the segments are shared by all the processes. We depend on protection rather than different address spaces to limit access. Each process can only access a small portion (2^{32} bytes out of

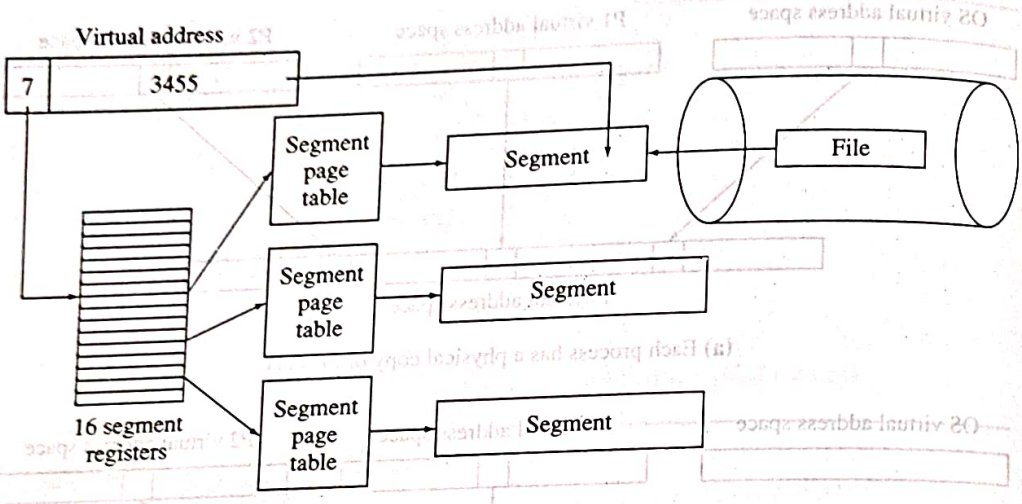


Figure 11.29 Virtual memory in the IBM 801

2^{40} bytes) of the address space at any one time, but can access all of it by asking the operating system to change your segment registers.

The IBM 801 architecture allowed for large numbers of segments. For example, it could expand the address space from 40 bits to 48 bits. This would allow 2^{20} or 1M segments, and a total address space of 2^{48}. With this many segments, it would be necessary to page the segment table.

File I/O is done by mapping a file into a segment and then accessing the memory in that segment. This means that you can only access, at most, 16 files at a time directly, although you can access more by changing segment registers.[2] File segments are automatically shared because all segments are shared.

11.11.5 File Mapping Examples

File mapping has so many benefits that almost all modern operating systems use it. This includes OS/2, Windows NT, Mach, OSF/1, SVR4, and most other versions of UNIX. In Section 12.13, we will discuss the virtual memory system of several operating systems, and there is more discussion there of file mapping in a range of operating systems.

11.12 SUMMARY

Memory allocation is harder when we have to find physically contiguous space for the whole program. Compaction creates a large, physically contiguous block, but it requires a lot of processor time. There are a number of ways to divide a program into

[2] Actually, the maximum will be smaller than 16 because you must use a few of your segment registers to map your code, data, and stack.

smaller pieces so that it is easier to find space for the pieces. Segments are one way to do this, but they create pieces that are too big and of varying sizes. It is easier to divide a program into many small, equally sized pieces called pages. This requires a page table to map logical to physical addresses. For small address spaces, the page table can be kept in hardware registers. For large address spaces, the page table is large and must be kept in memory. This requires a page table cache, called a translation lookaside buffer or TLB. Page table entries also have protection bits which allow us to limit access to pages in various ways.

Virtual memory means that you keep some of the logical address space of a process on disk. This idea was started in the early days of computing with the methods of swapping and overlays. Modern virtual memory systems use paging, where some of the pages are on disk and some are in memory. The operating system has a number of duties in a virtual memory system. It must manage the virtual address space of the process when it is created, when it is dispatched, when it gets a page fault, and when it is destroyed. This management function consists mainly of looking things up in various tables and making changes to tables as circumstances change. Extensive hardware support is also required for a virtual memory system.

Locality is the tendency of a program to use some parts of its logical address space over and over again. Locality is the result of sequential program execution, loops, and program modularity. It is locality that makes paging a practical method, and very low page fault rates are required for paging to be fast enough.

Most newer operating systems allow you to map files into the virtual address space. This makes for convenient I/O, allows the operating system to unify virtual memory and I/O, and have a single system for disk access.

11.12.1 Terminology

After reading this chapter, you should be familiar with the following terms:

- bank switching
- bursty
- cache hit
- cache hit rate
- cache miss
- compaction
- contiguous
- daemon
- external fragmentation
- file mapping
- hook
- internal fragmentation
- locality
- memory-mapped files

- overlay
- page
- page fault interrupt
- page fault rate
- page frame
- page table
- page table entry (PTE)
- paging daemon
- persistent objects
- present bit
- protection field (in a PTE)
- segmentation
- segment register
- software interrupt
- space resource
- spatial locality
- swap area
- swapping
- temporal locality
- time resource
- translation lookaside buffer (TLB)
- virtual memory

11.12.2 Review Questions

The following questions are answered in the text of this chapter:

1. What are the benefits of compaction?
2. Compare segments and pages.
3. How can the logical address space be contiguous if the physical address space is not contiguous?
4. Compare segment registers to base and limit registers.
5. What are the problems with noncontiguous logical address spaces?
6. Compare page tables in registers with page tables in memory.
7. What is the purpose of a TLB?
8. Why do programs exhibit locality?
9. What is the relationship between program locality and the TLB hit rate?
10. Why does paging make memory allocation easier for the operating system?

11. What is the difference between a page and a page frame?
12. What is the difference between internal fragmentation and external fragmentation?
13. What is the purpose of the protection bits in the page table entry?
14. Why does design require iteration?
15. Compare swapping and overlays.
16. Why is it called *virtual* memory?
17. Describe the hardware changes required to switch from a base/limit register system of memory mapping to a paging system.
18. Describe the actions an operating system must take when a page fault interrupt occurs.
19. What is a typical page fault rate?
20. What is a paging daemon? What does it do?
21. Relate the terms daemon, polling, interrupts, and hooks.
22. What is a page replacement algorithm?
23. How does file mapping make using files easier and more efficient?

11.12.3 Further Reading

See Satyanarayanan and D. Bhandarkar (1981) for a discussion of TLB design trade-offs. See Section 12.15.3 for references on virtual memory systems.

11.13 PROBLEMS

1. In the chapter, we talked about the idea of compacting memory in the physical address space. Consider compacting memory in the virtual address space. Can you think of a situation where you might want to do this?
2. Suppose you have a memory of 128Kbytes, and you can move a word (four bytes) of memory every microsecond.
   a. If memory is 80 percent full, how long will it take to compact it? (Assume you have to move all allocated memory.)
   b. Do the same calculation with a memory of 32Mbytes and assuming that you can transfer 64 bytes per microsecond.
   c. Now suppose that a system contains, on the average, three processes, and processes execute an average of 10 seconds. How long will it be until the next process completes (on the average)?
   d. Redo the calculation with 100 jobs that complete in 3 seconds (on the average).

3. A linker knows how to combine object modules together into a single linear address space. It knows how to resolve external references, find object modules, and read libraries. It knows how to read and write object modules and load modules.

   Suppose you wanted to modify such a linker to link modules into an address space that was not contiguous. For example, suppose the logical address space consists of 1 Mbyte in 16 segments of 64K each. For each program, you want to divide it up into 16 pieces, one in each segment, where all the pieces are approximately the same size. Describe some of the tasks and problems you would encounter in modifying the linker to do this. How would the load module format have to change in order to support this?

4. What TLB hit rate would be required to achieve an effective memory access time in a paged system only 1 percent slower than the equivalent unpaged system?

5. Suppose we have a paging system with 10 pages, and the likelihood that a page is referenced is based on the exponential probability distribution. We can simulate the page number of the next memory reference by generating a uniform random number, x, between 0 and 1, and then computing $10e^{-10x}$ and rounding it down to the nearest integer. Most of the time you will get 0, implying that page 0 is referenced. Some of the time you will get 1, less often 2, less often 3, etc. Suppose you have a TLB of 2 pages. When you have a TLB miss, you must replace one of the pages in the TLB. There are three strategies you can use for this. First, you can replace a random page (get a uniform random integer from 0 to 1). Second, you can replace them in order: first slot 0, then slot 1, then slot 0 again, then slot 1 again, etc. This is called first-in, first-out (or FIFO). Third, you can replace the one that has not been used for the longest time. This is called least recently used (or LRU). That is, when you get a TLB hit, you exchange it to the first slot (if it is not already there). Assume the slots start out with pages 0 and 1 in them.

   Write a simulation program that computes the TLB hit rate for this system.

6. Suppose that the average amount of external fragmentation in a dynamic memory allocation system is 20 percent of the space. Suppose that 70 percent of the blocks are allocated and 30 percent are free, and that the average free block size is one-half of the average allocated block size. In a paging system, the average internal fragmentation is one-half of a page. Suppose the average allocated block is 100Kbytes. At what page size will the internal fragmentation and the external fragmentation be (about) equal?

7. Suppose we have a multiprogramming operating system, and a process asks for more memory while it is running. It is possible that you will have to move the process around in physical memory to satisfy that request. Explain why this is. Explain why it depends on the kind of memory mapping you use (*base/bound*, code/data split, segments, pages).

8. Explain the differences between logically contiguous memory (that is, memory that is contiguous in the logical address space) and physically continuous memory.

9. Can pages overlap? Can page frames overlap? Why or why not?

10. State and explain two major advantages of paged virtual memory over the base/limit register method of memory mapping.

11. Some computers have several possible page sizes. Do you think it is possible to switch between two processes that use different page sizes? That is, is it possible to have page size a parameter that changes as you switch processes? If it is not possible, explain why. If it is possible, explain what problems the operating system would face in dealing with it.

12. Explain why small page sizes necessitate keeping the page table in memory rather than in hardware registers.

13. With page tables in hardware registers, you have to copy the page table registers each time you switch processes. With page tables in memory and using a TLB, you also have a "register filling" overhead when you switch processes. Explain why that is. Explain why there is no "register saving" overhead in this case.

14. What changes would be required in the program in Figure 11.12 to change the page size from 4 Kbytes to 1 Kbytes?

15. Is it reasonable for segment tables to have the same kind of protection bits as page tables (as shown in Figure 11.16)?

16. Could we keep the segment table in memory, as we did the page tables? Why or why not? Would there be any difference in how they were handled?

17. Bank switching is the hardware equivalent of overlays: both place two (or more) parts of a program in the same addresses. Suppose a typical instruction takes 1 microsecond, the bank switching hardware instruction takes 10 microseconds to switch banks, and bringing in an overlay takes 100 milliseconds. We want the program to run at no more than 110 percent of the speed it would run with no swapping or bank switching. What percent of instructions can require an overlay to achieve this? What percent of instructions can require a bank switch to achieve this?

18. Suppose it takes a disk an average of 10 milliseconds to get to a specific sector, and that, once it is there, it can transfer 2Mbytes per second, and that the average job is 300,000 bytes long. How long does it take to swap one job out and another job in?

19. Suppose we wanted to design an automatic overlay system. This system would lay out the overlays and ensure, at run time, that the right overlay was always in memory. It would do this based on a specification by the programmer. The programmer would provide a list of the procedures and global data structures that would go in each overlay. What code needs to be added to each procedure call for this to work? Where would this code be placed? What data is kept by the overlay system? What are we assuming about the use of global data by procedures? Decide what computations can be done at compile (or load) time, and which ones have to be done at run time.

20. Suppose we had an extremely fast paging device that took only 2 milliseconds to store or fetch a page. Redo the calculations in Section 11.8 based on this new paging device. Do the calculations one more time for a paging device that can read or write a page in 200 microseconds.

21. Rewrite the hardware virtual memory access algorithm (in Section 11.7.1) to use a TLB.

22. Procedure calls will reduce the locality of a program. How can the linker reduce this loss of locality?

23. Most programming languages allocate procedure activation records (the record of the saved registers, local variables, parameters, etc.) on the stack rather than allocating them from the per-process memory manager. Which method would cause more program locality, using the stack or the memory allocator? Explain your reasoning.

24. Suppose we have an average of one page fault every 100,000 instructions, a normal instruction takes 1 microsecond, and a page fault causes the instruction to take an additional 10 milliseconds. What is the average instruction time, taking page faults into account? Redo the calculation assuming that a normal instruction takes 0.5 microseconds instead of 1 microsecond.

25. Using the same assumptions as we gave for the previous problem, suppose we used a paging algorithm that took an additional one millisecond per page fault, but decreased the page fault rate to one in 120,000 instructions. Would this result in an increase in overall performance or not? Explain your reasoning.

26. Three things take up time in a swapping system: (1) swapping out the old job, (2) swapping in the new job, and (3) running the new job. When the new job is ready to be swapped out again, it becomes the old job, and the cycle starts over again. If we can overlap these activities, we can speed up the system. There are three levels of overlap:

a. No overlap.

b. You can overlap running a program with either swapping in or swapping out (but not both).

c. You can overlap all three things.

Suppose we have a computer system where each job is 200Kbytes long, the CPU can execute 1M instructions per second, the I/O channels can transfer 1Mbytes per second, and a job runs exactly 300K instructions before being swapped. (K=1024; M=K*K) We only consider the CPU time spent on running the jobs to be useful. The *effective computation rate* is the ratio of useful computation time to total time. So if the CPU was busy only half of the time, the effective computation rate would be 0.5. For each of the three overlap cases above, compute the effective computation rate. Now assume that you can speed up either the CPU or the I/O channels by 100 percent (double their speed). For each of the three overlap cases, indicate which one would be better to speed up (you can speed up only one, not both) and how that speedup will affect the effective computation rate. If your results seem anomalous, explain why.

27. Explain how the UNIX process system call `exec` could be used to implement overlays. Would it be the same as the kind of overlay system described in Section 11.6?

28. Consider the following structure for a page table entry and skeleton of the C function to translate addresses:

```
struct pte {
    int present          : 1; /* = 1 if page is in memory */
    int readAllowed      : 1; /* = 1 if the page can be read */
    int writeAllowed     : 1; /* = 1 if the page can be written */
    int executeAllowed   : 1; /* = 1 if the page can be executed */
    int unused           : 12; /* unused bits */
    int pageFrame        : 16; /* page frame number (if present=1) */
};

struct VirtualAddress {
    /* 32 bit virtual address with 16K pages */
    /* physical addresses are also 32 bits */
    int pageNumber       : 18;
    int pageOffset       : 14;
};

/* global variable that contains the physical address of the first */
/* level page table for the running process */
struct pte * PageTableBaseAddress;

/* accessType is (one of) 1 for read, 2 for write and 4 for execute */

/* This procedure will translate the virtual address AND fetch the */
/* word addresses. Assume no translation lookaside buffer */
int mmu( struct VirtualAddress va, int accessType ) {
    /* algorithm for translation of the virtual address and fetch */
    /* of the word addresses */
    /* use the procedure "fetch(physicalAddress)" to fetch a word */
    /* from the physical memory. use this to fetch page table entries */
    /* and data words */
}
```

Complete the mmu procedure to do the address translation.

29. Do pages and page frames have to be the same size? Why or why not?

30. In a paged OS, any allocation whose size is a page or greater is allocated by whole pages. So dynamic memory allocation is only used for blocks whose size is less than one page (say 4 Kbytes). Explain why this fact makes a buddy system allocator more attractive than it would be if it were doing all allocations.

31. Consider a system call that will insert N pages of virtual memory at a specific virtual address. What are the problems in implementing such a system call? What uses would it be put to? What happens to all the virtual addresses after the addresses are inserted?

32. Suppose we had 30 Gbytes of disk storage on our computer. How large a virtual address (in bits) would it take to map the entire file system into your virtual address space? Then do the same calculation for 250 Gbytes of disk storage.

Now suppose there are 15,000,000 computer installations in the world, and each has 20 Gbytes of disk storage. How much virtual address space (in bits of virtual address) would be required to map all of that storage into your virtual address space?

33. The IBM 801 has only 16 segment registers, even though 4096 segments are possible. Give some reasons why they did not provide more segment registers.

34. Suppose you wanted to extend the file mapping interface to allow you to modify the length of a file that is mapped. What would you add to the existing system calls, and what new system calls would you add to do this? Be sure it is clear how they are used to change the length of a mapped file.

chapter

12

Virtual Memory Systems

In the last chapter, we looked at the mechanisms of virtual memory in an operating system. In this chapter, we will look at page replacement algorithms and at some extensions and variations of the paging idea.