19 Virtual Memory

Creation, Propagation, and Destruction of Addresses

When a program is loaded into memory, the non-relative addresses in the program have to be adjusted depending on the load point. For example, consider the following program in the optimal instruction set:

```
           halt
a:        .word b   ; contains the address of b
b:        .word 23
```

It is translated to

```
Header
o
A     0001
C

Addr  Code       Source Code

0000  f000           halt
0001  0002 a:        .word b
0002  0017 b:        .word 23
```

The first `.word` directive specifies the label `b`. Thus, it is translated to the address corresponding to `b`, which is 0002. But this is correct only if the load point is 0. If the load point were 100, then the address corresponding to `b` would be 0102. Thus, the word corresponding to `a` would have to be adjusted—from 0002 to 0102. `lcc` can perform this adjustment because the A entry specifies the address of the word that needs adjustment. However, once a program starts executing, addresses can be anywhere. For example, if a program loads an address into `r0`, then `r0` contains an address. If it then stores the contents of `r0` on top of a constant, then that location subsequently contains an address. Moreover, because that location at assembly time did not contain address (because the `.word` directive for it specified a constant), *there will be no A entry for it*.

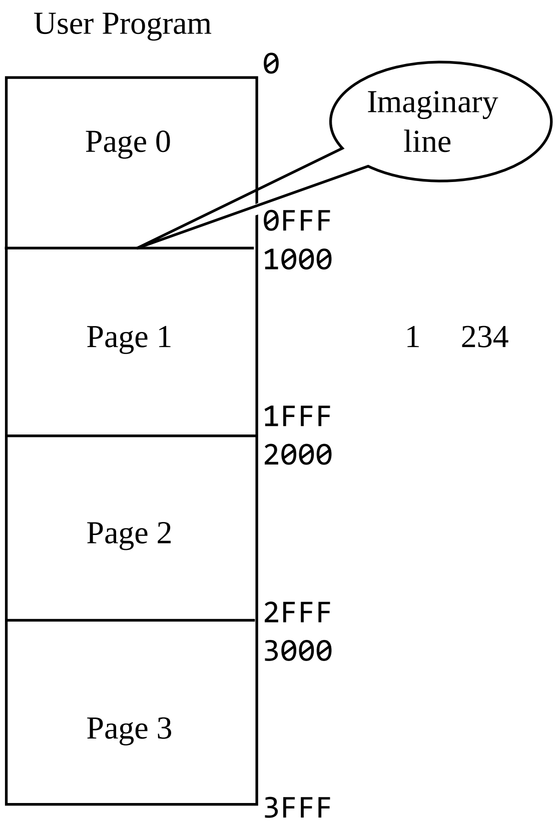
Rule: A entries cannot be relied on once a program starts executing.

Generally, this is not a problem. However, if a program is moved in memory to a new load point after it starts running, `lcc` has no way of determining where the addresses are in the program because the A entries are no longer valid. Thus, `lcc` has no way of adjusting the locations that contain addresses (because it does not know where they are). However, in a *virtual memory* system, programs can be moved in memory after they start executing. In fact, a program can be broken up into fixed length blocks called *pages*, and each page can be loaded into *any* available block of memory. Moreover, the pages in memory can be moved after execution starts. Even more impressive, a computer with virtual memory provides the user of a computer with more main memory than the computer really has. Thus, in a system with virtual memory, a user can load and execute a program whose size is bigger than all the main memory in the computer.

We start our investigation of virtual memory with a description of *simple paging*. Simple paging does not provide virtual memory, but it is the basis for virtual memory. Thus, it is an appropriate starting point.

Simple Paging

In a simple paging system, we view user programs as consisting of fixed length blocks called *pages*. Each page has the same length, typically a power of 2, such as $2^{10} = 1024 = 1K$, $2^{11} = 2048 = 2K$, or $2^{12} = 4096 = 4K$. For example, here is a program that consists of four pages, numbered 0 to 3 (we number pages starting from 0). The lines separating consecutive pages are imaginary. Within the program there is no marker of any sort dividing one page from the next.



Suppose our computer uses 16-bit addresses. Then each address can be represented with four hex digits. Further suppose that the page size for the program above is 4K. Then the addresses of page 0 run from 0000 to 0FFF hex (0 to 4095 decimal). The following table summarizes the address range for each page:

Page	Address Range (hex)
0	0000 to 0FFF
1	1000 to 1FFF
2	2000 to 2FFF
3	3000 to 3FFF

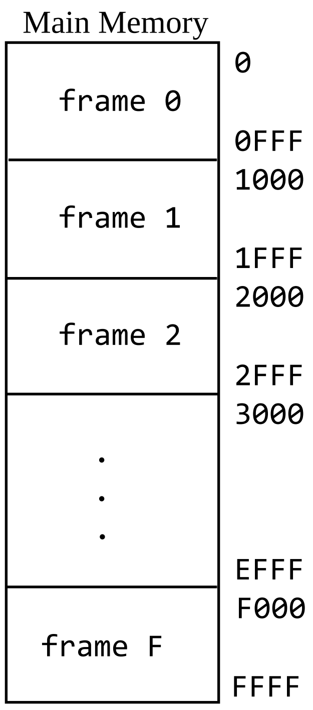
Notice that the leftmost hex digit always equals the page number of the page at that address. For example, the address 1FFF whose leftmost hex digit is 1 is in page 1. The three rightmost hex digits in an address specify the displacement into the page. For example, the address 213C in a byte-addressable memory system is 13C bytes into page 2. In a paging system, we always view addresses as consisting of two components: the page number and the displacement into that page:

$$\overbrace{\underbrace{2}_{\text{Page Number}}\qquad\underbrace{1\qquad 3\qquad C}_{\text{Displacement Within a 4K Page}}}^{\text{16-bit address}}$$

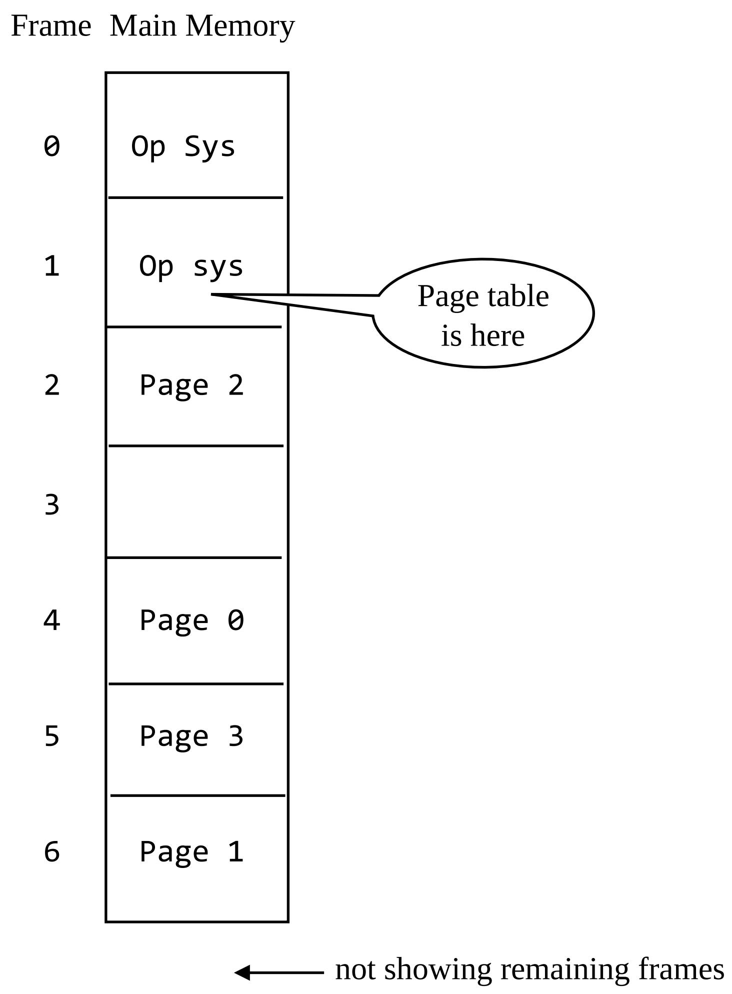
The displacement component of an address has to be big enough to hold any displacement within a page. Thus, for a 4096-byte (4K) page size, the displacement field has to hold displacements up to 4095 (displacements start from 0 so the maximum displacement is one less than 4096). Since the number 4095 in binary consists of 12 1-bits, a 4096 page size requires a 12-bit (3 hex digit) displacement field. Generalizing, we get that a page size of 2^n requires a n-bit displacement field.

The page-number component of an address contains all the bits in the address to the left of the displacement component. For example, in a 32-bit address with a 12-bit displacement field, the leftmost 20 bits make up the page number field.

In a paging system, we view main memory as consisting of fixed-length blocks. However, to distinguish them from the fixed-length pages that make up programs, we call the blocks of main memory *frames*:



In a paging system, when an operating system loads a user program into memory, it loads each page into *any* available frame. The pages do not have to appear in consecutive frames, nor do they have to appear in their normal order. Thus, page 2 can be loaded below or above page 1. For example, the operating system might load pages 0, 1, 2, and 3 of a program into frames 4, 6, 2, and 5, respectively:



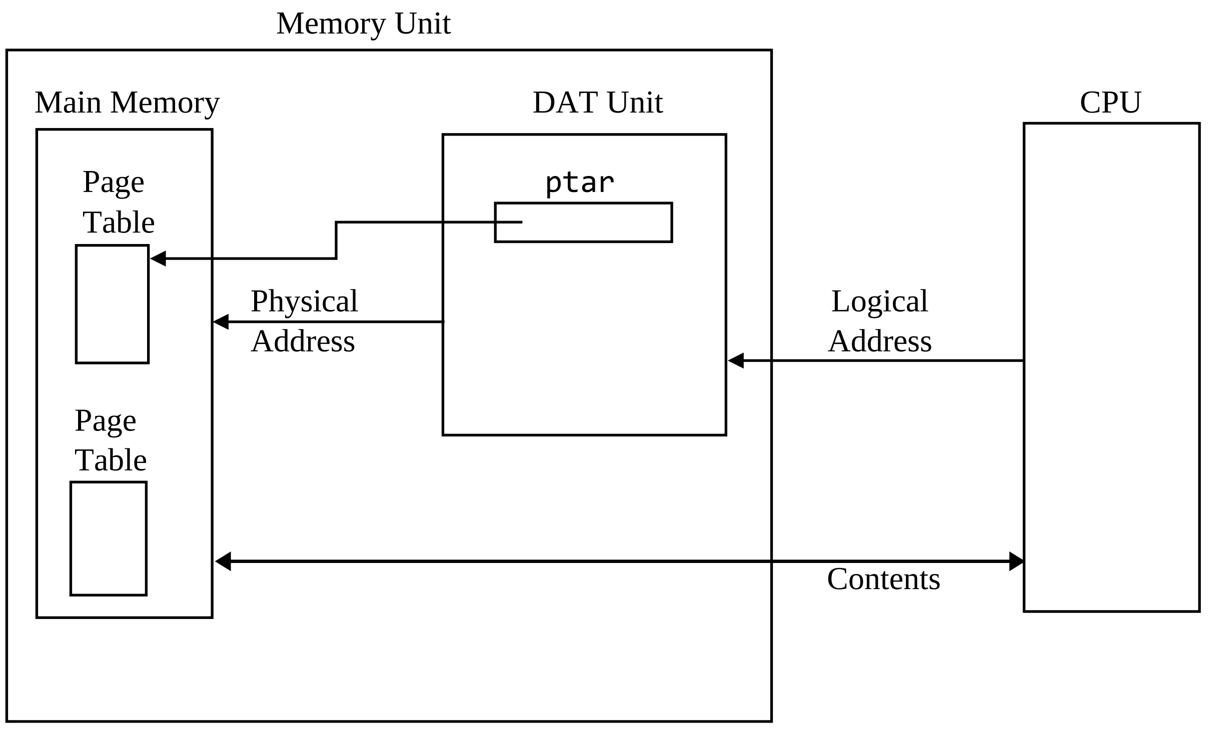
Typically, the operating system resides in the frames in low main memory, so these frames are not available for user programs. To keep the diagram above simple, we have shown the operating system as occupying only two frames. In reality, operating systems are large programs that would occupy many frames in low main memory.

A paging operating system does *not* adjust the addresses in the user program when loading the program into main memory. Address adjustment occurs *dynamically*—that is, during execution. As the operating system loads pages of a user program into memory, it keeps track of the location of each page by building a page table for that program. A page table is a one-dimensional array in main memory (in the operating system's area) that contains an entry for every page in the corresponding user program. Thus, for a user program which contains four pages, the operating system builds a page table containing four entries. Suppose the operating system loads frame 4 with page 0. Then it would record this by placing 4 into slot 0 of the page table. In a page table, each index represents a page number and the corresponding slot contains the frame number for that page. For example, the following page table indicates that pages 0, 1, 2, and 3 are in frames 4, 6, 2, and 5, respectively:

Page Table

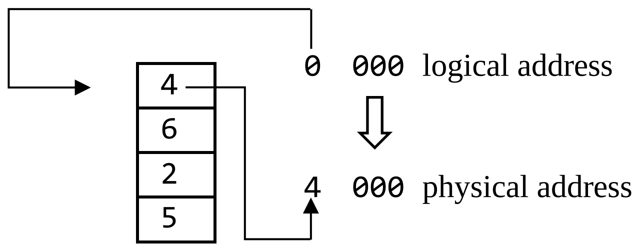
Page Number	Frame Number
0	4
1	6
2	2
3	5

When a program executes in a paging system, the addresses the CPU sends to memory first go to a *DAT unit* (dynamic address translation unit) where they are translated to physical addresses. Which unit? DAT unit. To do this translation, the DAT unit accesses frame numbers from the page table for the program currently executing:



Let's examine how the translation works. Suppose the program corresponding to the page table above is about to execute. Before it passes control to the program, the operating system loads the `ptar` (*page table address register*) in the DAT unit with the address of the page table of the program that is about to execute. Suppose the program's start address is 0000. Then to pass control to the program, the operating system loads the address 0000 into the `pc` register, causing the CPU to fetch the instruction at this address. In the fetch process, the CPU sends the address 0000 to the DAT unit. The DAT unit takes the left hex digit of this address (the page number) and uses it as an index into the page table pointed to by the `ptar`, obtaining 4 (the frame for page 0). It then substitutes 4 (the frame number) for the left hex digit (the page number) in the address to get the physical address 4000. 4000 is the physical address of page 0 in main memory. The DAT unit then sends this physical address to memory, which responds by returning the instruction at the beginning of page 0:

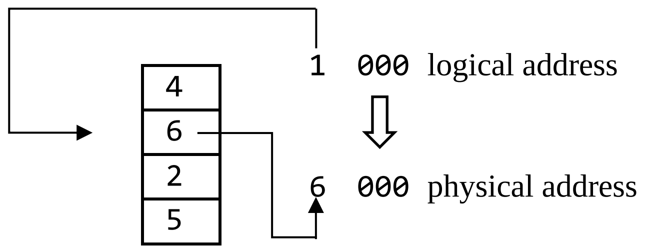
Use left hex digit as index into page table



Page table

Let's jump ahead in the execution of our program to the point at which the last instruction in page 0 is about to be fetched. Assume that this instruction is two bytes in length and starts at 0FFE. Thus, the `pc` register at this point contains the address 0FFE. In the fetch process, the CPU sends 0FFE to the memory system where it is translated to the physical address 4FFE. The `pc` register is incremented by the instruction length (to 1000) and the fetched instruction is executed. When the CPU repeats its fetch-increment-decode-execute cycle, it uses the address 1000 (the address now in the `pc` register). This time, however, the DAT unit accesses the page table using the index 1 (the left hex digit of the address) and receives frame number 6, the number of the frame holding page 1:

Use left hex digit as index into page table



Page table

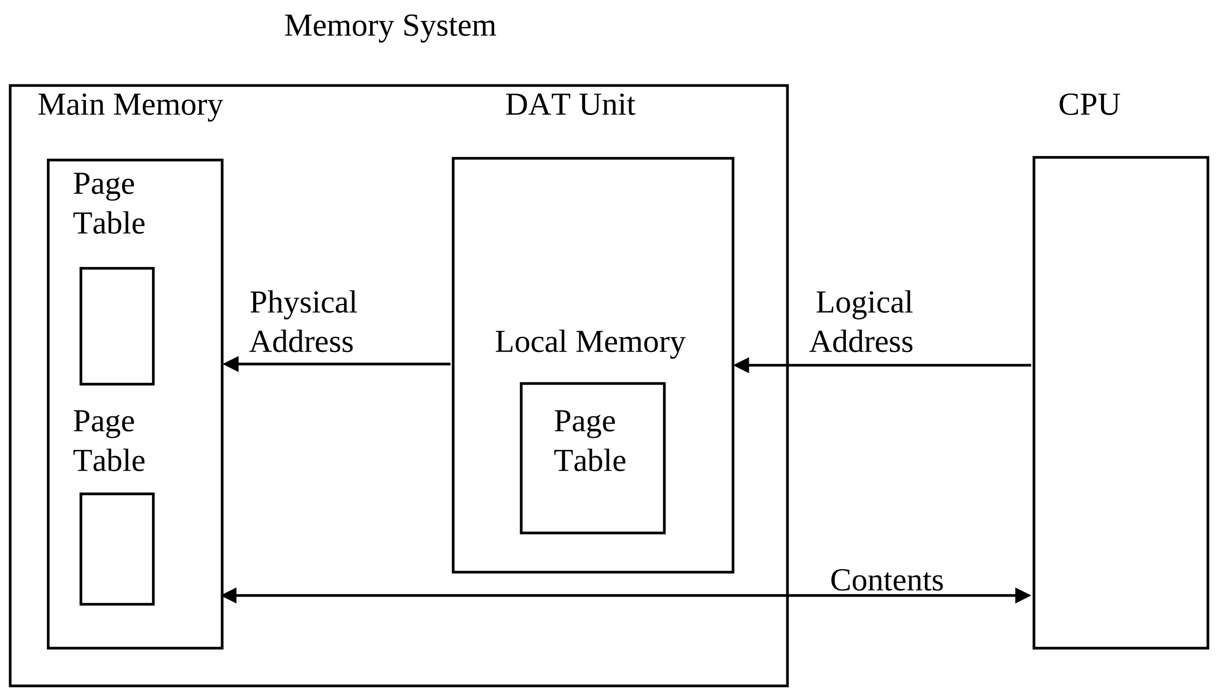
Thus, the physical address is 6000 (the address of page 1 in main memory). Although pages 0 and 1 and not contiguous (i.e., next to each other), their execution proceeds as if they are. It appears that page 0 starts at address 0 and page 1 is right after it.

With paging, from the program's point of view, it resides in memory starting at address 0. From memory's point of view, the program resides in a collection of frames that are not necessarily contiguous, and the starting physical address is not necessarily 0.

Paging systems suffer from a potentially serious problem. Each time the CPU reads an item from memory, two read operations are performed: the first to obtain the frame number from the page table (which is located in main memory) and the second to obtain the desired item from main memory at the computed physical address. The CPU can execute instructions no faster than it can get them from main memory. Thus, by increasing the access time for main memory (by requiring two reads), paging forces the CPU to execute instructions at a slower rate. If this problem were unfixable, the negative impact of paging on execution time would be worse than its positive effect on memory management. However, we will see in the next section that by carefully designing the DAT unit, we can eliminate virtually all the slow-down associated with the address translation mechanism in a paging system.

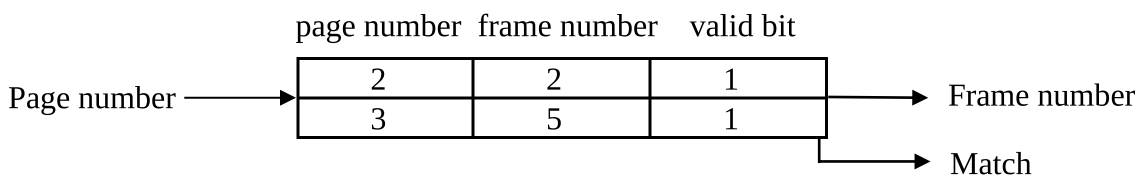
Associative Memory

The obvious solution to the problem with paging—two reads operations performed every time the CPU fetches an item from memory—is to keep a copy of the page table in a local memory area within the DAT unit:



Then each time the DAT unit translates an address from that program, it can get the frame number it needs very quickly from its local memory. If this local memory is very fast, the translation of addresses can also be very fast. Because the local memory within the DAT unit can hold information from only one page table at a time, the operating system still must maintain a page table in main memory for every user program currently in the system. Whenever the executing program goes into a wait state or terminates, the page table for the next program to executed is loaded into the DAT unit's local memory.

Because the local memory within the DAT unit has to be very fast, it is very expensive. A super-fast local memory that is large enough to hold the largest possible page would be too expensive. Because of cost considerations, the local memory within a DAT unit typically has only a few slots. Then it can be made both fast and still be inexpensive because of its small size. But what happens when the page table is too big to fit into the local memory in the DAT? Let's consider a simple example. Suppose the 4-page program is currently executing, and local memory in the DAT unit has only two slots. Although the whole page table cannot fit into the local memory in the DAT unit, local memory can hold the page table entries for those pages that are active at any given time, assuming no more than two pages are ever active at any time. For example, if only pages 2 and 3 are active, the system can keep their frame numbers (2 and 3) in local memory. The absence of the other two entries from local memory has no negative impact, as long as their corresponding pages remain inactive. There is, however, a serious problem with this approach. The index into local memory is not necessarily the page number as it is for the full page table. The solution for this problem is for each slot of local to hold *both* the page number and its corresponding frame number. For example, if pages 2 and 3 are in frames 2 and 5, respectively, then the local page table would contain



page number	frame number	valid bit
2	2	1
3	5	1

Each slot contains a page number and its corresponding frame number. In addition, each slot has a *valid bit* (a single bit) which, if equal to 1, indicates that its slot is valid (i.e., contains valid page and frame number).

Let's examine how the DAT unit translates a logical address usingthis local memory. It takes the left hex digit from the logical address (this is the page number) and inputs it to the local memory. Local memory then searches for a slot that contains this page number. When it finds it, it outputs the frame number in that slot, and sets the match output to 1 to indicate a successful search. For example, for the local memory above, if the DAT unit inputs 3, it gets back 5; if it inputs 2, it gets back 2. For both of these cases it also sets the match output to 1. If, however, the DAT unit inputs 1, then the match output is set to 0, indicating an unsuccessful search for page number 1.

The local memory in the DAT unit is unusual in two respects. First, unlike most memory units (which input an address and output the contents at that address), it inputs the contents of a subfield of one of its slots. It then outputs the contents of the other subfield of the same slot. Thus, we can describe the memory as *contents-in/contents-out memory*. In contrast, regular memory is *address-in/contents-out memory*. A common name for contents-in/contents-out memory is *content-addressable*. This name makes sense because we “address” a slot by the contents of a subfield of that slot. Another common name is *associative memory*. This name also makes sense because a subfield of a slot is addressed by the data in a subfield with which it is “associated.” To distinguish the associative memory in the DAT unit from associative memory that may be used elsewhere in a computer system, we call the associative memory in the DAT unit the *translation lookaside buffer* (TLB).

The second respect in which the associative memory above is unusual is that it searches *in parallel* every slot for the inputted page number. For example, if we input 3, it searches the first slot for 3 and, at the same time, it searches the second slot for 3. If a serial search were used here, it would generally take too long. Remember address translation has to be virtually instantaneous.

When an operating system loads a program into memory, it sets all the valid bits in the TLB to 0 (a machine instruction does this). It also loads the `ptar` with the address of the page table for the program that is about to executed. To translate the first address, the DAT unit supplies the page number in the address to the TLB. The TLB responds with a 0 on the match output (because all entries are not valid initially). Thus, the DAT unit for this address has to access the page table in main memory pointed to by the `ptar`. After it obtains the frame number from the page table, it enters this frame number and the page number into the TLB and sets the valid bit for that entry to 1. Thus, subsequent address translations for the same page use the data in the TLB. Similarly, if during the execution of a program, a page becomes active that was previously inactive, its page and frame number will not be in the TLB. Thus, on the first address translation for such a page, the match output will be 0. The unsuccessful search forces the DAT unit to get the frame number from the page table pointed to by the `ptar`. It then enters the page and frame number into the TLB, overlaying an entry for a page that hopefully has become inactive. Thus, only the first access of a page whose frame number is not in the TLB requires the DAT unit to access the page table in memory. As long as there are enough slots in the TLB to hold the page information for all the active pages at a given time, the system works quite well. A typical size for the TLB run is 64 slots.

The dynamic address translation that the DAT unit performs is a hardware function. The operating system (which is software) does perform some necessary initialization (creating the page table in memory, setting the valid bits in the TLB to 0, and loading the `ptar`). However, once program execution starts, the address translation mechanism—searching the TLB, accessing the page table in memory on an unsuccessful search, updating the TLB, and constructing the physical address—is all hardware. If any of these functions were performed by software, address translation would take too long.

Virtual Memory

Virtual memory is a memory management system that gives a computer the appearance of having more main memory than it really has (hence the name "virtual", which means “not real”). Virtual memory has two important advantages:

1. Memory is used more efficiently. In particular, only those pages of an executing program that are in use occupy main memory. For example, a 200-megabyte word processing program with only 128 kilobytes in use would occupy only 128 kilobytes of main memory.

2. Programs that are bigger than main memory can be executed.

At one point, some computer architects thought that computers would eventually have so much memory that virtual memory would be unnecessary. Memory sizes have, indeed, dramatically increased, but so has the size of programs. Thus, memory continues to be a critical resource that has to be used efficiently.

Virtual memory is implemented in one of three ways: demand paging (which is an extension of simple paging), segmentation, or, most commonly, a combination of segmentation and paging. We will examine the demand paging implementation of virtual memory.

Demand Paging

Simple paging is still very inefficient in its utilization of main memory. When a program executes, generally only a small portion of its code is executing at any given time. For example, when we use a word processor to simply enter data into a file, we do not use that portion of the word processor that performs spell-checking. A sophisticated word processor might contain 500 megabytes of code, but at any given time it may need only a small fraction of that. Suppose, for example, it needs at most one megabyte at any given time. Then 499 megabytes of memory—the main memory used to hold the unused portion of the word processor—is memory that is serving no useful purpose. For efficient memory use, main memory should contain only the code that is currently needed. Then our 500-megabyte word processor would never occupy more than one megabyte of real memory. This "loading only what is needed" is the idea that underlies demand paging.

Most programs generate memory references that tend to cluster in small localities within the program. We call this behavior the *locality of reference*. Programs that exhibit this behavior can run with only a small subset of its pages in memory—the subset that is currently in use. We call this subset of pages *the working set* of the program. As a program executes, its working set typically changes, and, therefore, the pages in memory also have to change. Keeping the working set in memory as a program executes is the job of a demand paging system.

Demand paging works like regular paging with a few additional twists. In demand paging, each page table entry contains two parts: the frame number and a valid bit. The valid bit indicates if the corresponding page is in memory. For example, the following page table shows that pages 2 and 3 are in frames 2 and 5, respectively. Pages 0 are 1 are not in memory (we know this because their valid bits are 0 in the page table).

Page Number	Frame Number	Valid Bit
0	0	0
1	0	0
2	2	1
3	5	1

When the operating system in a demand paging system is about to give control to a program for the first time, it allocates a page table in memory, but it neither loads the program nor loads the page table with frame numbers. Instead, it sets all the valid bits in both the page table and the TLB to 0 (it executes machine instructions to do this), and it loads the `ptar` with the address of the empty page table. It then gives control to the program by loading the `pc` register with the program's start address. When the DAT unit translates this start address, its search of the TLB fails, of course, because the TLB is initially empty. When the search fails, the DAT unit then attempts to access a frame number from the page table pointed to by the `ptar`, using the page number in the start address as an index. For example, suppose the start address is 0000 and its left hex digit is the page number. Thus, the DAT unit would use 0, its left hex digit (the page number), as an index into the page table. But because the valid bit for this page table entry is 0 initially (indicating the page is not in memory), this access fails. We call this situation—a page is needed that is not in memory—a *page fault*. Whenever a page fault occurs, the DAT unit generates a *page-fault interrupt*, causing the page fault interrupt handler in the operating system to get control. This interrupt handler loads into memory the page that is needed from an I/O device, enters its frame number into its page table and sets its valid bit to 1, and then restarts the program that caused the interrupt. This time, the DAT unit finds a valid page entry in the page table so it can complete the address translation and update the TLB with the page and frame number. Thus, subsequent accesses of this page now proceed with essentially no delay.

The reading in of a page during a page-fault interrupt is called a *page-in* operation. The I/O device that is used to hold the pages of an executing program in a demand paging system is called the *paging device*.

A page fault requires the intervention of the operating system to load the needed page into memory and update the page table accordingly. This process is time-consuming because it involves the execution of an interrupt handler in the operating system that contains many machine instructions. Thus, in a demand paging system, we want to keep page faults to a minimum. Some, page faults are necessary, and cannot be avoided. However, sometimes an excessive number of page faults occur, causing the system to spend most of its time performing paging operations instead of executing user programs. When this happens, we say the system is *thrashing*.

Page Replacement Policies for Demand Paging

When the operating system performs a page-in operation, it has to select a frame in which to load the page. If all the frames are occupied, it has to overlay a page in some frame with a new page. If the page to be overlaid has changed since it was paged in, its current state obviously must be saved. This is accomplished by writing it out to the paging device before the new page is paged in. We call this operation a *page-out* operation. The operating systems can determine if a page-out operation is necessary by examining the *dirty bit* attached to the frame holding the page to be replaced. The operating system sets this bit to 0 when a page is first paged in. The dirty bit is then automatically set to 1 by the hardware whenever the page is subsequently written to. Thus, if the dirty bit for a page to be replaced is 1, it must be paged out before a new page can overlay it.

When the operating system selects a page for replacement, it attempts to select one that will not be needed again in the near future, or even better, not at all. One possible page replacement policy is FIFO (first-in-first-out). With FIFO, the oldest page in memory is selected for replacement. The oldest page, however, is often one that is still in use, making it a bad choice for replacement. For example, many programs have a controlling routine that calls the various subroutines of the program. The controlling routine is needed during the execution of the entire program. The page containing the controlling routine is often the oldest (it typically executes first), and, therefore, would be selected for replacement by the FIFO policy.

FIFO is a policy that is easy to implement. It does not require special hardware support. Since page-in operations are performed by the operating system, the operating system can easily keep track of the time-in-memory order of the pages in memory. When a page must be replaced, the operating system simply uses the time-in-memory information it maintains to identify the oldest page. Because this order changes relatively infrequently (only on a page-in operation), maintaining the time-in-memory order of pages involves only a minimal amount of overhead.

The ideal page-replacement policy would select a page whose next reference is the least imminent. With stock funds, past performance is no guarantee of future gains. However, because most program exhibit the locality-of-reference behavior, past activity is, in fact, a good predictor of future activity. An active page will probably continue to be active; an inactive page will probably continue to be inactive. Of course, an active page might suddenly become inactive, and vice versa, but these shifts occur relatively infrequently. These observations suggest that a least recently used (LRU) policy—a policy that selects for replacement the page that has gone unused for the longest period of time—should work well. LRU requires that software and/or hardware keep track of the time-of-last-use order of the pages in memory. Because this order can change so frequently (it can change each time an instruction is executed), keeping track of it has to be essentially a hardware function. Consider what would occur if it were done purely in software: On every memory reference, a software routine would have to get control to update the order-of-use list it maintains. Clearly, a purely software approach would involve an exorbitant amount of overhead. Moreover, to implement it in hardware would be too costly. To minimize the hardware costs associated with LRU, computer designers typically implement a policy that only approximates true LRU. An LRU approximation does not necessarily select the least recently used page. However, it does select one that is not used recently (NUR).

Demand paging can dramatically increase the capabilities of a computer system. Suppose memory in a system without demand paging is big enough to hold five complete programs. With demand paging, the same memory could hold, perhaps, as many as 50 programs because only a fraction of every program is in memory at any given time.

Demand paging also allows a program that is too big to fit into memory to execute. For example, suppose a program contains 500 megabytes but memory has only 256 megabytes. Without demand paging, the system could not load and run the program. But with demand paging, the 500-megabyte program can run because only its active portion needs to be in memory at any given time. With demand paging, we can run a program of any size, up to the addressing capabilities of the computer. For example, if a computer used 32-bit addresses, we could run programs whose size is as large as 2^{32}.

Page Size Considerations

Which is better, a small page size or a large page size in a demand paging system? Let's examine the advantages and disadvantages of both sizes.

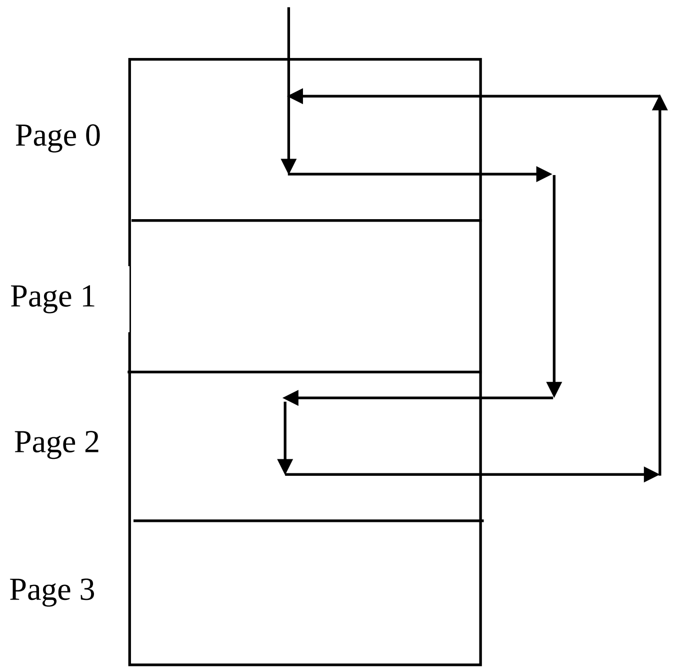
The length of a user program is usually not an exact multiple of the page size. Thus, the last page for most programs is partially unused. The last page, however, always occupies a complete frame when loaded into memory. Thus, its unused portion occupies, and, thereby, wastes main memory. We call this *internal page fragmentation* because it occurs "internal" to a page.

The unused portion of the last page of a program is, on the average, one half of a page. Thus, the smaller the page size, the less memory is wasted due to internal page fragmentation. Our conclusion: *a small page size is better*.

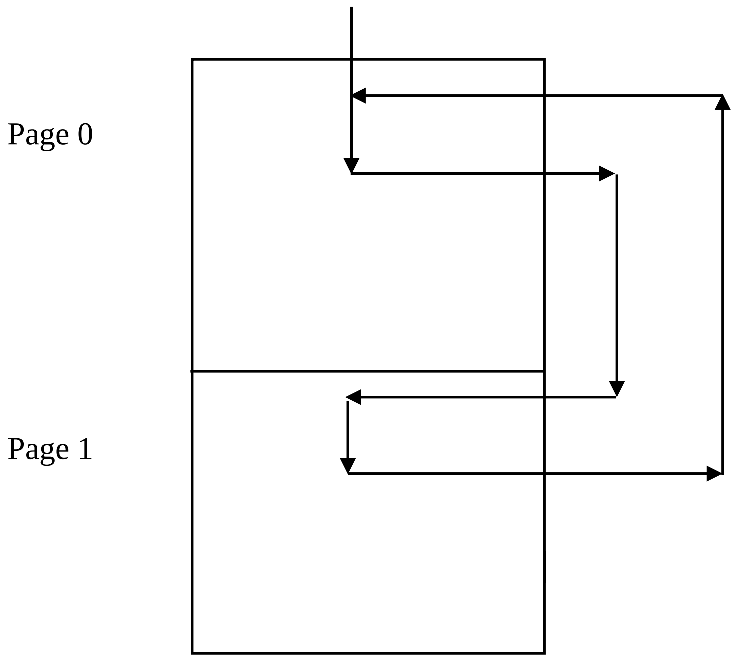
Suppose a program consists of two pages. If we reduce page size by a factor of eight, the same program would require 15 or 16 pages. Thus, the smaller the page size, the larger the page table. Our conclusion: *a large page size is better*.

Suppose a program executes from top to bottom and then terminates. Assume with large pages, the program consists of two pages, and with small pages the program consists of 16 pages. Then, in the large page case, only two page faults occur; in the small page case, 16 page faults occur. Our conclusion: *A large page size is better*.

Suppose a program's size is twice as big as the available memory. As a 4-page program, it executes a loop covering pages 0 and 2 that iterates 100,000 times:



The first time the loop body executes, two page faults occur—one for page 0 and one for page 2. Both pages can fit into memory at the same time. Thus, for the remaining iterations of the loop, no other page faults occur. But now suppose the page size is double, in which case only one page of the same program can fit into memory



The first time the program executes, two page faults occur as with the smaller page size case—one for page 0 and one for page 1. However, now when page 1 is paged in, it overlays page 0. Thus, on next iteration, page 0 has to be paged in again overlaying page 1. Thus, on each iteration of the loop, two page-in operations occur. If the loop iterates 100,00 times, a total of 200,00 page in operations occur compared to only two for the smaller page size case. Our conclusion: *A small page size is better*.

We have just seen scenarios in which large pages are better and other scenarios in which small pages are better. Because of the conflicting results, it is impossible to conclude or even guess from our analysis what the optimal page size is. This uncertainty is often the case when analyzing computer systems. Most proposed new designs have pros and cons. Thus, it is often hard to predict how well a new design will work. Sometimes an analysis clearly implies a new design is great (or terrible), and when the new design is implemented and tested, the analysis turns out to be completely wrong. Computer systems are difficult to analyze accurately because they are complex, interactive systems. Most analyses, on the other hand, are based on simple models which may yield different results than the actual system.

On the page size question, computer scientists debated the virtues of various page sizes for several years. The final consensus, which is based principally on tests performed on actual systems, is that large page sizes are better.

Problems

1) Write a simple assembly language program that contains a field that initially requires adjustment depending on the load point because it holds an address but becomes a constant during execution.

2) Determine the number of page faults using LRU for the following sequence of page references:

   0, 1, 0, 2, 0, 3, 0, 4, 0, 5, 0, 6, 0, 7, 0

   Assume only two frames are available. Do also for FIFO.

3) Same as problem 2, but for

   0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0

4) Same as problem 2 but use MRU (most recently used).

5) Same as problem 3 but use MRU (most recently used).

6) Give an example of a sequence of page references in which LRU outperforms FIFO. Give another example in which FIFO outperforms LRU.

7) Suppose a paging system has 32-bit addresses. If the page size is 4K, how many bits are in the displacement portion of a logical address? How many bits are in the page number portion of a logical address?

8) What is the difference between a logical address and a physical address?

9) Write a simple assembly language program that has only one address in memory initially but has exactly five when it terminates.

10) If memory becomes very cheap, will virtual memory become unnecessary?

11) Is there any advantage if the operating system loads successive pages into successive frames?

12) Under what circumstances must the page that is being overlaid in a page-in operation be paged out.

13) In what way is NUR (not used recently) better than LRU?

14) Why must the DAT mechanism have an off/on switch?

15) What are the symptoms of a system that is thrashing?

16) Should a user program be able to modify its own page table? Explain.

17) In a demand paging system, can a page initially be executed in one frame, and then later on in a different frame? Explain.

18) In a demand paging system, does it make sense to pre-load a program before it starts running?

19) Is thrashing more likely if a demand paging system uses a poor page-replacement policy? Why?

20) How many 4K pages does a modern word processing program like Microsoft Word consist of?

21) Does paging provide a memory protect feature—that is, a mechanism that prevents one user from accessing memory for another user or the OS?

22) Suppose a program consists of two parts—one part at low logical addresses and a second part at high logical addresses with nothing in between. What is the downside of this organization? How can it be remedied? *Hint*: Consider demand paging with segmentation.

23) Both super small pages or super big pages will not work well. Explain why. In between these two extremes, there must be an optimal page size. What factors might influence what the optimal page size is? For example, might the size of main memory influence what the optimal page size is?