Principles of virtual memory

by Carl Burch, Hendrix College, November 2012

Contents

1. Basics
  1.1. Address translation
  1.2. Page table format
2. Replacement algorithms
  2.1. FIFO (First in first out)
  2.2. LRU (Least recently used)
  2.3. Clock
3. Implementation issues
  3.1. The translation lookaside buffer
  3.2. Table directories
  3.3. Inverted page tables
4. Real-world virtual memory
  4.1. IA32 page table entries
  4.2. Applications of virtual memory

In a very simple operating system, each process is given its own segment of RAM for its own use. Though this system is straightforward and easy to implement, it brings up some issues.

- How does the operating system decide how much memory to allocate to each process?
- What if the total memory required by active processes exceeds the amount of RAM available?
- How can the system prevent a process from reading or altering the memory supposedly reserved for other processes?

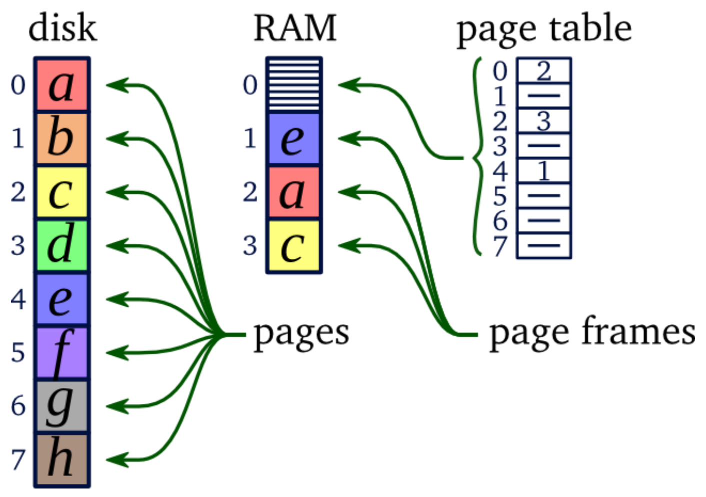
To address these issues, and to address several others that we'll discuss later, modern operating systems use *virtual memory*. In this document, we'll study how virtual memory works.

1. Basics

The idea of **virtual memory** is to create a *virtual address space* that doesn't correspond to actual addresses in RAM. The system stores the official copy of memory on disk and caches only the most frequently used data in RAM. To make this workable, we break virtual memory into chunks called **pages**; a typical page size is four kilobytes. We also break RAM into **page frames**, each the same size as a page, ready to hold any page of virtual memory.

The system also maintains a **page table**, stored in RAM, which is an array of entries, one for each page, storing information about the page. The most important piece of information in each page table entry is whether the corresponding page is loaded into a frame of RAM and, if so, which frame contains it. The CPU uses this page table in looking up data in virtual memory.

Consider the following illustration of a system using virtual memory.



Here, we have eight pages on disk pictured on the left and three page frames in the memory pictured at right. (This example is much smaller than in real life, where you would have thousands of pages and page frames.) In this example, each of the three frames holds a page from memory: The data of page 0 (in red) is in frame 1, page 2 (yellow) is in frame 3, and page 4 (blue) is in frame 2. You can see that the page table, also stored in RAM, contains this mapping from pages to frames.

1.1. Address translation

If each page held four kilobytes in our example illustrated above, each memory address would be 15 bits long. This is because the example includes eight 4K pages, for a total of 8 · 4K = 32K, and 2^{15} bytes is 32 kilobytes.

Whenever a program requests access to a memory address, the CPU will always work with this as a virtual memory address, and it will need somehow to find where the data is actually loaded. The CPU goes through the following process.

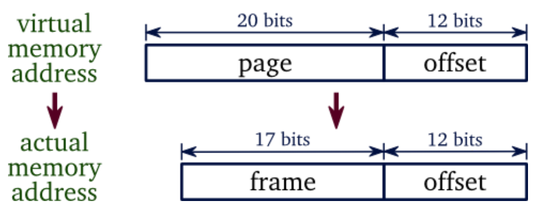
1. The CPU breaks the address into the first three bits giving the page number *page*, and the last twelve bits giving the offset *offs* within the page.

2. The CPU looks into the page table at index *page* to find which page frame *f* contains the page.

3. If the page entry says that the page is not in RAM, it initiates a **page fault**. This is an exception telling the operating system that it needs to bring a page into memory. After the operating system's exception handler finishes, it returns back to the same instruction so the CPU ends up trying the instruction over again.

4. Otherwise, the CPU loads from the memory address *offs* within page frame *f*.

In this way, the CPU translates the virtual memory address into an actual memory address.



Suppose we were in the above-illustrated situation, and a program says to load some information from address 010 1001 1101 0011. The computer breaks this into two parts, the page number 010 and the offset 1001 1101 0011. It looks into the page table at index $010_{(2)} = 2$, finds 3, and replaces the page index with the page frame index to get the actual memory address, $11\ 1001\ 1101\ 0011_{(2)}$. The CPU looks at this address in RAM to get the requested data.

Suppose that the program later says to load some information from address $110\ 1010\ 0000\ 1000_{(2)}$. In this case, the page number is $110_{(2)} = 6$. Looking in the page table at entry 6, we see that the page is not anywhere in memory. Thus the CPU causes a page fault, giving the operating system a chance to load the desired page into a page frame and repair the page table. When the OS completes this process, it returns from the interrupt handler for the CPU to try the program's instruction again. This time, the CPU should be successful, since the OS has just loaded the required page into memory.

Notice the separation of responsibilities in virtual memory between the instruction set layer and the operating system layer. For efficiency, the system must be designed so that, in the regular course of execution, the CPU does not often require entering the operating system. Since memory references occur often, the CPU should be able to complete most memory references on its own. Thus, the job of translating virtual memory addresses into actual memory addresses falls to the CPU. Since the CPU will be looking into the page table to accomplish this, the instruction set designer is the one who will define the page table format.

Page faults are rare, however, and so the process of handling page faults need not be efficient. (This is especially true because the process of loading a page from disk is bound to be slow anyway, simply because disks are slow.) Additionally, the process for handling them is relatively complicated. Thus, the instruction set designer simply defines how a page fault is raised, and it leaves the details of how to accomplish the job of loading a page into memory to the operating system. Among other things, this means that different operating systems for the same CPU can have different approaches for storing virtual memory on disk and different approaches for deciding which loaded page to replace when another page is needed.

1.2. Page table format

The page table is the primary data structure for holding information about each page in memory. In a typical system, a page table entry will include the following information about each page.

- a bit specifying whether that page is currently loaded into memory.
- several bits dedicated to specifying which page frame contains the page (if indeed it is in memory). In our illustration with just three page frames, the entry would require 2 bits for this purpose. Actual systems would be able to accommodate many more pages, and twenty bits would be more realistic.
- a bit, called the **dirty bit**, specifying whether the page in memory has been altered since being loaded into memory. This bit is useful because, when it is time to remove the page from memory, the OS must to store it back to disk if it has changed at all (that is, if it is “dirty”). Storing something on the disk takes additional time, so we only want to do this when necessary; the dirty bit allows the OS to avoid the cost when possible.
- a bit, called the **referenced bit**, that the CPU sets each time the page is accessed. The OS will reset this to 0 periodically. When it is 1, then, this indicates that the page has been accessed recently. The OS might use this information in deciding which page frame should be emptied to make room for a needed page.

2. Replacement algorithms

One of the most important issues in virtual memory is the paging algorithm. For the success of a virtual memory system, we need an algorithm that minimizes the number of page faults on typical request sequences while simultaneously requiring very little computation. Researchers have investigated many algorithms for this, and we'll examine a sample.

2.1. FIFO (First in first out)

With the FIFO algorithm, any page fault results in throwing out the oldest page in memory to make room for the new page. It is called FIFO after "First In, First Out," referring to the fact that the first among the current pages to enter memory is the first among the current pages to exit.

This technique has the virtue of being very simple to implement. We can simply keep a counter referring to a page frame, and with each page fault we step the counter forward one and replace the page found there with the requested page instead. (When the counter goes off the highest-numbered page frame, we circle around to the lowest-numbered frame.) By the time the counter reaches the page frame again, it will have circled around all other page frames, that page frame will hold the oldest page in memory.

Intuitively, FIFO makes sense as a way to avoid page faults: It makes sense that whatever page is oldest is likely to be the most "obsolete," and so ripest for replacement. However, in practice, the age of the page isn't a strong predictor of the future: Sometimes pages are accessed frequently over a long period of time, so throwing out the page means that it will simply need to be reloaded again in the near future.

2.2. LRU (Least recently used)

The LRU algorithm says that the system should always eject the page that was least recently used. A page that has been used recently, after all, is likely to be used again in the near future, so we should not eject such a page.

LRU is about the best people have found for avoiding page faults. People would use it universally, were it not also much less efficient computationally than FIFO. There are basically two alternatives for implementing it.

- **Linked list:** The system maintains a doubly linked list among the page frames. For any access to the page inside a frame, the system moves that frame's node to the front of the list. Later, when a fault occurs, the system chooses to eject from the frame at the list's end.

  This technique suffers from major updating costs for memory access: Moving a node to a list's front requires modifying at least four pointers. Thus, each memory request by a program requires four additional memory modifications (in addition to the extra page table access already necessary to determine the page's location in memory). These extra memory references add up to an unacceptable expense.

- **Counter:** The system associates a value for each page frame and it keeps a counter clocking the time. When the system accesses the page in a frame, it also copies the current clock into the frame's associated value. Later, when a fault occurs, the system goes through all frames and empties the one whose value is least.

  While this improves on the update cost per access, it dramatically increases the cost of selecting a page to eject: The OS must go through all page frames in order to determine the smallest value. Remember that a typical system may have many page frames (perhaps 100,000), so this cost is not small.

Thus, although LRU does well in avoiding page faults, people look for other algorithms because of its computational expense.

2.3. Clock

The Clock algorithm is a hybrid between the FIFO and the LRU algorithms. In this, the system maintains a referenced bit for each page, which is set by the CPU each time the page is accessed, and a counter of which page was last ejected. When it is time to eject a page, the OS repeatedly increments the counter until it finds a page whose referenced bit is 0; this is the page that ejected. As it increments the counter, it clears the referenced bit associated with each page it passes.

The below pseudocode summarizes the process. It uses `requested_page` for the page that has been requested, `frame_page` for an array saying which page is located in each frame, and `page_table` for the page table. Each entry of the `page_table` array includes the fields `referenced` for whether the page is marked as recently referenced, `in_memory` indicating whether the page is in memory, and `frame_location` telling which frame holds the page.

```
cur_page ← frame_page[cur_frame]
while page_table[cur_page].referenced = 1:
  page_table[cur_page].referenced ← 0
  if cur_frame < last_frame_index:
    cur_frame ← cur_frame + 1
  else:
    cur_frame ← first_frame_index
  cur_page ← frame_page[cur_frame]

Load page requested_page into frame cur_frame.
page_table[cur_page].in_memory ← 0
frame_page[cur_frame] ← requested_page
page_table[requested_page].frame_location ← cur_frame
page_table[requested_page].in_memory ← 1
page_table[requested_page].referenced ← 1
```

Let's look at a specific example, working with a virtual memory system that has just three page frames. Suppose our program accesses the following sequence of pages:

1, 2, 3, 1, 4, 2, 1, 5

page	frame			
ref	1	2	3	explanation
1	1'.	–	–	We load the first referenced page into the first empty page frame available. It is marked as “referenced,” which we represent with an apostrophe. The period represents where the last page was loaded.
2	1'	2'.	–	There is still an empty page frame, so the next referenced page goes into it. The period moves to mark this as the most recent place where a page as loaded.
3	1'	2'	3'.	There is still an empty page frame, so the next referenced page goes into it. The period moves to mark this as the most recent place where a page as loaded.
1	1'	2'	3'.	Page 1 is already loaded in memory, so the CPU simply ensures that page 1 is marked (using an apostrophe). As it happens, page 1 was already marked, so nothing changes.
4	4'.	2	3	Now we have a reference to a page that's not in memory. Starting from the period (which marks the most recent place where a page is loaded), we step to the right, circling around to the frame 1 when we move off the end. We see that the frame 1 contains a marked page, so we clear the mark and step again. Frame 2 also contains a marked page, so we clear the mark and step again. Likewise for frame 3. Then we

		circle back around and finally find a frame containing what is now an unmarked page; so we load page 4 into frame 1.
2	4'. 2' 3	Page 2 is already loaded in memory, so the CPU simply marks page 2 (using an apostrophe).
1	4' 2 1'.	Page 1 is not in memory, so we must decide what to replace. We start by stepping the period to the right, where we see a marked page 2. Since it is marked, we clear the mark and step the period once more. The period is now at the frame 3, which holds the page 3. It is unmarked, so we decide to replace it.
5	4 5'. 1'	Page 5 is not in memory, so we must decide what to replace. We start by stepping the period to the right, whereupon it circles back to frame 1. The page there (4) is marked, so we step to the right to get to frame 2. Frame 2's page is unmarked, so we decide to replace it.

(The name *Clock* for this algorithm comes from the fact that the counter iterates through pages like a clock's hand. Of course, this is true of FIFO too, but that algorithm was named *FIFO* already.)

Note that, for each page access, the Clock algorithm requires the CPU to set a single bit whether or not there is a page fault. This is an additional expense, but it requires only a single memory access — and often it will not be necessary at all, since the bit will already be set. (FIFO has no such expense, but it does not take a page's use into consideration.)

Computationally, the Clock algorithm takes much less time than LRU, though it is not as quick as FIFO. In terms of avoiding page faults, the Clock algorithm is far better than FIFO but somewhat worse than LRU. For virtual memory systems, it has proven an excellent compromise between the two, and so many operating systems base their page replacement algorithm on the Clock algorithm.

3. Implementation issues

David Wheeler coined a famous aphorism in computer science: “Any problem in computer science can be solved with another layer of indirection.” Virtual memory is a prominent examples supporting this: After all, with virtual memory, a program no longer uses direct memory addresses into RAM, but rather it uses virtual memory addresses which only indirectly reference RAM via the page table.

But Wheeler followed his aphorism with a warning: “But that usually will create another problem.” And so it proves with virtual memory. Virtual memory systems equipped with good algorithms and a reasonably large RAM have proven to be have very few page faults in realistic page request scenarios. (If programs accessed memory completely at random, then this would not be possible. Luckily, such programs are rare.) But there are three other concerns regarding virtual memory deserving attention.

- The address translation process must be supported by hardware, and this leads to a more complex CPU.

- Even if a memory location is already in RAM, accessing memory now requires *two* memory accesses for every memory reference — once to get the page table entry, and another time to get the actual data. Thus, in a straightforward implementation, virtual memory *halves* memory access time. (And that doesn't count the time to alter the referenced bit and/or dirty bit, which would require an additional memory access — nor does it count the possibility of a page fault.)

- The page table can be very large. For example, if a computer supports a full 32-bit address space (4GB), and each page is 4KB, then the page table itself must have 1M entries. Since each entry must have at least four bytes to hold the page frame index and other table entry data, the page table consumes 4MB of memory. This page table needs to be permanently in memory, so the CPU can always access it to determine the page frame containing any memory address.

In practice, as we'll see, each process will get its own page table; and these page tables will usually need to be in memory, since the time to switch processes is overwhelmingly expensive if it involves loading 4MB from disk. If we have 100 processes, then, we actually will spend 400MB of RAM on page tables. That is a large amount of memory to spend on mere bookkeeping.

Luckily, these issues can be addressed with additional effort. The issue of CPU complexity is relatively minor, since the benefits of virtual memory are substantial enough that added complexity is worth it. The other two issues — speed and page table size — are more interesting, and we'll look at some techniques for addressing each.

3.1. The translation lookaside buffer

As we've presented it, virtual memory requires two memory lookups — one to retrieve the page table entry, and another to actually perform the requested memory access. We can reduce the cost of retrieving the page table entry by using a cache. We might use the standard cache, but CPU designers have found that page table entries are accessed so distinctively and frequently that it is worthwhile creating a separate cache just for them. This cache is called the **translation lookaside buffer**, but most people refer to it simply as the **TLB**.

Each TLB entry has two pieces: the virtual page number and the page table entry. When the CPU wants to get a page table entry, it first looks into the TLB to find whether the virtual page requested matches a cached virtual page number. If it's there, then the CPU has the page table entry immediately. If not, then the CPU loads the page table entry from memory (and caches it for the future).

The TLB significantly increases the efficiency of virtual memory. Say that reading RAM takes 10 ns. Without a TLB, each virtual memory access requires 20 ns — one to read from the page table, and one to access the actual memory location. (We're neglecting page faults for the purpose of this example.) But with a TLB, the hardware can find the relevant TLB entry in negligible time. Thus, when the TLB happens to contain the requested page table entry, the memory access takes just 10 ns. If the TLB has a 90% success rate, the average time per memory access is $0.9 \cdot 10 + 0.1 \cdot 20 = 11$ ns. Thus, using a TLB, the system experiences just a 10% slowdown against using raw memory. Compared to the 100% slowdown without using a TLB at all, this is a much more manageable price to pay for capabilities provided by virtual memory.

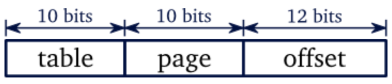
For the CPU designer, choosing the number of entries cached in the TLB is a choice of whether spending transistors for an additional cache entry will increase the overall system efficiency. A typical size would be 128 entries, though the TLB has grown larger as transistors have grown smaller.

In modern processors, the TLB hit rate tends to be quite high. Some CPU designers (including those designing the SPARC and Alpha chips) have decided that the TLB hit rate is high enough that the job of managing the page table should be the operating system's. In these systems, the operating system manages the TLB's contents: Any TLB miss causes a TLB fault, and the operating system is responsible for loading the table entry into the TLB (and ensuring that the requested page is in memory). By delegating TLB management to the operating system, the CPU no longer needs to specify how page faults occur or how page tables are structured. This reduces the complexity of the CPU, saving transistors for other purposes; and it gives more flexibility to the operating system designer for dealing with page tables.

3.2. Table directories

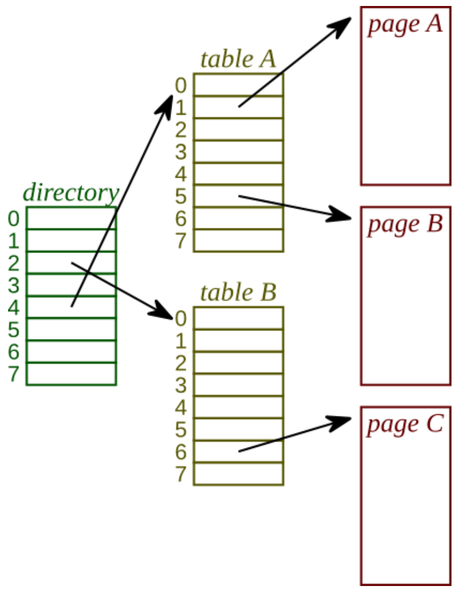
To get around the amount of memory taken by page tables, a common solution is to add another level of indirection, called the **page directory**. In this system, the page table is broken into small pieces, and we have a page directory that indicates where each of these pieces can be found. This reduces our space utilization when there are large unused portions of the address space; rather than have a page table for the portion, the page directory can simply indicate that this portion is invalid.

With page directories, each memory address has *three* pieces.



Say a program accesses a memory address which are broken into the three pieces *t*, *p*, and *o*. The CPU starts by examining element *t* of the page directory, which specifies where the page table is. Then it looks at element *p* of this page table to determine the memory address where the page starts. Finally, it looks at offset *o* in the page to get the actual information.

Suppose, for example, that we had a system which used 3 bits for *t*, 3 bits for *p*, and 3 bits for *o*, and memory looked as follows.



If a process requested the memory at the virtual address 010110011, then it would split this address into $t = 010_{(2)}$, $p = 110_{(2)}$, and $o = 011_{(2)}$. It would go to location $t = 2$ in the page directory, which says to go to table *B*, and it would go to entry $p = 6$ in this table, which says to go to page *C*, where it would find the data requested $o = 3$ bytes into this page.

The page directory has two virtues. First, it means that large unused portions of the virtual address space do not require page table entries, since such a page table can simply be omitted from the page directory. This can provide significant space savings. Just as significantly, the individual page tables can themselves be paged, so the system does not need to allocate memory for page entries of infrequently-used ranges of the address space.

One problem with page directories is that it exacerbates the speed issue: We've added yet another memory access to the process of decoding a virtual address. Fortunately, the TLB works effectively enough that this additional cost is not the major problem that it would otherwise be.

The IA32 bit architecture designed by Intel uses page directories. In their design, the page directory and each page table includes 1024 entries; since each entry requires four bytes, each page table takes 4KB, which conveniently matches the size of an IA32 page. A memory address is divided by taking the first 10 bits to determine which table to access, the next 10 bits to determine which page to access within that table, and the final 12 bits indicate the offset within that page.

3.3. Inverted page tables

An alternative approach to addressing the amount of memory used for page tables is based on the observation that the only entries that the CPU really needs are those for pages that are already in RAM. This is the idea behind the **inverted page table**, in which we replace the page table with a hash table mapping page indices to page table entries, including only those entries which correspond to pages that are in RAM. Thus, the inverted page table's size must only be big enough to accommodate the number of page frames, not the number of virtual pages (as with a regular page table).

As with page directories, the downside to an inverted page table is that accessing a hash table requires several memory accesses rather than the single access required for looking into a simple page table. An effective TLB, though, reduces the need to incur this expense.

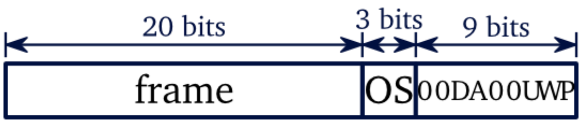
With the trend toward larger virtual memory sizes, especially with the movement into the 64-bit range, the page directory approach becomes less tenable, and so an inverted page table becomes more appealing.

4. Real-world virtual memory

Now let's look at page tables in the real world.

4.1. IA32 page table entries

In the IA32 design, each page table entry takes 32 bits. Twenty of these bits are used simply to identify which page frame contains the bits, leaving twelve bits for other purposes. Of these twelve, three are reserved for the operating system to use at it wishes. The final nine are the more interesting bits.



Of the final nine bits in the entry, four are required to be zero (labeled “0”); these are actually reserved for uses in future CPUs. (In fact, later CPU versions have used some of these bits, but the way in which they are used is not of interest here.)

The final bit (labeled “P”) indicates whether the page is present in RAM. If this bit is 0, then any access to the page will trigger a page fault.

The next-to-last bit (labeled “W”) indicates whether the page is writable. If this bit is 0, then any attempt to store into the page will trigger a protection fault. This can be useful for preventing erroneous or malicious programs from altering a page that should really remain constant — like a page containing the code of a program.

The third-from-last bit (labeled “U”) indicates whether the page can be accessed when the CPU is executing in user mode. Some portion of the address space will be used for storing information for the operating system, and these pages should not be accessible to the program.

The final two bits are the accessed bit (labeled “A”) and the dirty bit (labeled “D”). The CPU sets the accessed bit to 1 whenever a program accesses the page; this enables the Clock algorithm. And the CPU sets the dirty bit to 1 whenever a program alters a page; this enables the operating system to know whether to save the updated page when ejecting it from RAM.

4.2. Applications of virtual memory

Originally virtual memory was designed to allow a larger address space than supported directly by

RAM. This was very relevant when RAM was typically measured in megabytes or even kilobytes; but as RAM has grown to be measured in gigabytes, it has become less relevant. Nonetheless, virtual memory provides so many advantages that modern systems would have difficulty dispensing with them.

- The OS can devote a complete virtual address space to each running process. This saves the programmer from having to worry about memory limits at all, except for whatever very large limit the OS gives. In Windows 2000, for example, each process gets its own 4GB virtual address space (with 2GB reserved for Windows); if 100 processes are active (which would be normal), this amounts to 200GB of virtual address space. A process may use a small fraction of the address space given it, but the system does not need to allocate space for unused pages anywhere — not even on disk. Thus, the only loss is the space used for the page table — which is minimal if the system uses page directories or inverted page tables.

- The OS can easily arrange for multiple processes to share the same page of memory. Each process can have different page table entries refer to the same virtual page. This permits for processes to share data. Using this memory, one process can quickly transfer information to another. (Without this capability, the operating system would be forced to be the middleman, and it would slow communication considerably.)

- Suppose a Unix shell does a `fork` system call, with the child process quickly following it with an `execvp` system call. This seems like a waste of time, since all the shell's memory must be duplicated for the `fork` (and copying all that memory takes time), but it is used only briefly before being replaced with the new program. But if each process has its own page table, then the OS can duplicate the page table only, with the two processes sharing pages. But it can mark each page as being read-only, so that any attempt to access the page triggers a fault; and at that time, the OS can duplicate the page when it is actually needed. In fact, very few pages will be modified: The child will make its `execvp` system call very soon, at which time it no longer shares this memory; in the meantime, neither process will have the chance to modify much memory, so very few pages need to be duplicated. Thus the `fork` system call does not actually incur the cost of copying the process's memory in most cases.

- The most familiar way of accessing files is as a stream; in Linux this is done using `open`, `read`, `write`, and `close` system calls. An alternative interface a **memory-mapped file**, in which the loads the entire file into a portion of memory, so that the program can skip around the file as if it were one large array. One prominent example where this is useful is in servicing an `execvp` system call: We want to load the program file into memory so that it can be executed. But it could also be useful for applications like text editors and databases.

  Memory-mapped files sound very inefficient, but use of virtual memory can remove this inefficiency: The OS simply dedicates a portion of virtual address space to the file, but it doesn't actually load any portion of the file until a page fault occurs — and then it only loads the 4KB from the file that is required to service the page fault.

These many advantages of virtual memory retain its usefulness in systems with plentiful RAM.