ECE254: Operating Systems and Systems Programming Fall 2016

Lecture 23 — Virtual Memory

Jeff Zarnett

Virtual Memory

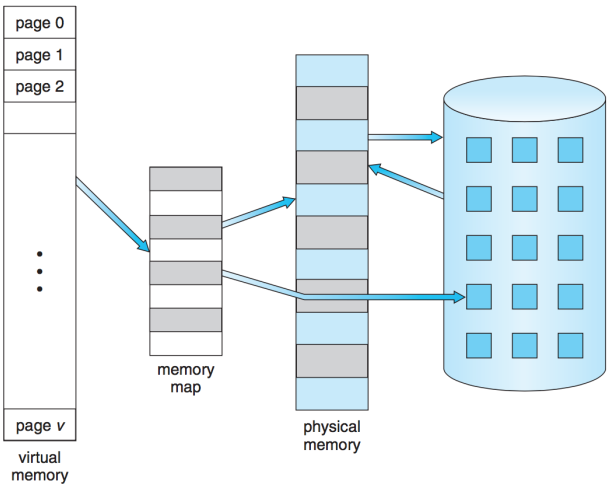
Even with paging and moving pages into and out of memory, there is a limit on what we can do with the system. Specifically, a program that requires more memory than the machine's physical memory cannot run. Maybe that seems ridiculous in the modern era of 4, 8, and 16 GB of memory, but server or supercomputer systems working on large datasets may require more memory than is available in the machine. Furthermore, when we have many programs running and a multiprocessor systems, we could have a situation where the sum of memory requirements exceeds the available memory. It is less than ideal to have a processor waiting for something to do because a process that is otherwise ready to run cannot proceed because there are insufficient free frames for it to run.

The problem is: a process must be entirely in memory or entirely on disk. In a lot of cases, the entire program is not needed at any given time. Code used to handle unusual situations (error handling, etc.) may not be needed except occasionally. Startup code is needed at the beginning of the program, but then never again. Data structures and collections may be declared to be very large even when it is not actually needed (e.g., the `ArrayList` in Java defaults to 16 elements, even if you might only need 4, wasting a bit of space).

If we could execute programs that are only partly in memory, there would be three major benefits [SGG13]:

1. A program is no longer constrained by the size of physical memory; programmers can use the entire virtual space without worrying about whether it fits.
2. Each program could use up less physical memory, allowing more processes to execute concurrently.
3. Less I/O is needed to swap user programs in or out.

Good news, everyone! We have already discussed many of the key ideas to making such a system work when we examined caching. The principle is really the same: main memory can be viewed as yet another level of cache and the disk is the last stop where the data can be. If a page is referenced and not currently in main memory, it is a page fault, and the page is loaded from disk into main memory. A page might need to be evicted from main memory to make way for it; thus a page replacement algorithm is needed to select the "victim" page and write it out to disk.



Virtual memory exceeding the size of physical memory, with some pages on disk [SGG13].

The typical approach is also like that of the cache, which is to use demand paging. A page is loaded into memory only if it is referenced or needed, thus preventing unnecessary disk accesses. This is also called the "lazy" approach in [SGG13], though lazy is typically an insult and in this case it is not necessarily bad. Clearly, we would like to involve disk as little as possible, because disk is, from the perspective of the CPU, extremely slow.

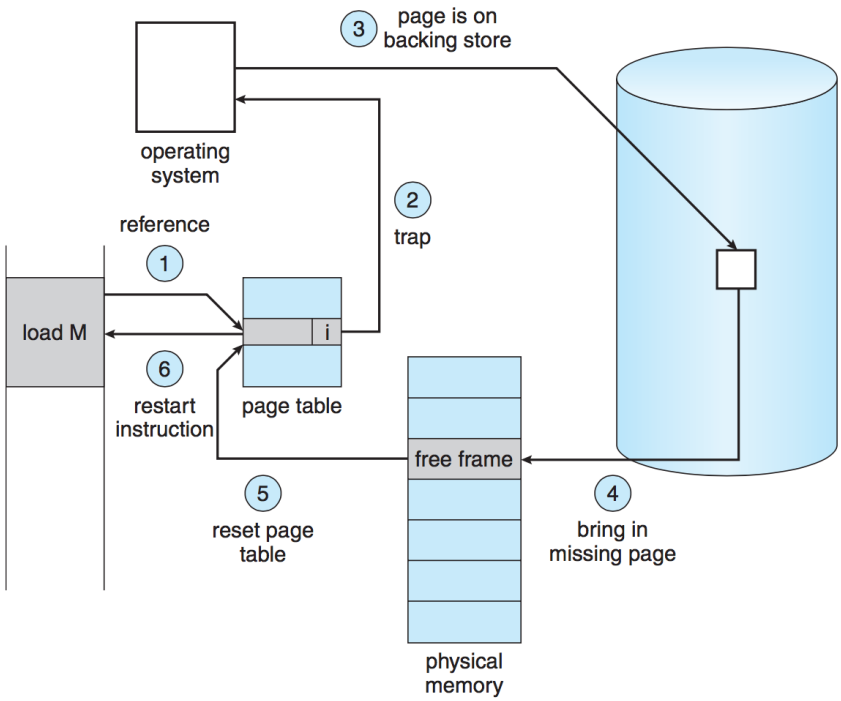
The fact that disk is so slow means we can get into a troublesome state called *thrashing*: the operating system is spending most or all of its time swapping pages in and out of memory and very little actual work can get done.

Apparently, if the NeXT operating system, NeXTStep[1] were booted on a machine with only 2 MB of RAM instead of the expected 4 MB, the steady state of the system would be a constant level of swapping [HZM14]. This was particularly bad because the most common cause of failure in the NeXT boxes was hard disk drive. But they did have cool magnesium cases, so after they died they at least made impressive conversation pieces and doorstops.

With virtual memory, each memory reference is a six step process [SGG13]:

1. Check if the memory reference is valid or invalid (just as we have done before).
2. If the reference is invalid, terminate the program (segmentation fault). If it was valid, but the page referenced is not in memory, we will need to retrieve it.
3. Find a free frame (or make one by evicting some other page).
4. Request a disk read (and possibly write) to bring in the new page.
5. When the disk read is complete, update the records to show the new page is in memory.
6. Restart the instruction that referenced the page that needed to be brought into memory.

Or, to view this visually:



Handling a page fault [SGG13].

[1] One of the three parents of Mac OS X: the classic Mac OS, BSD [UNIX], and NeXTStep.

Note that between steps 4 and 5, a significant amount of time will take place while the disk performs a read of the desired page and possibly a write of the page to be evicted if it was modified. While the slow disk operations are going on, the process is blocked on that I/O operation and, in the meantime, the processor can and should be working on something else.

The key requirement in the described workflow is the ability to restart any instruction following the page fault. We save the state of the process, including all the registers and instruction pointer and so on, when the page fault occurs, so we are able to restart the process exactly where it was. The difference is that after the restart, the page needed is in memory and is accessible. A page fault could occur on any memory reference, including fetching the next instruction. If it happens at that time, the fetch operation is done again. If a page fault happens when doing an operation that required fetching an operand, then we fetch and decode the instruction again and then fetch the operand. So a little bit of work may be repeated [SGG13].

Consider the `ADD` instruction that adds `A` to `B` and stores the result in `C`. First, we must fetch and decode the instruction. That tells us about the two operands, which must be retrieved themselves. Then we can add the two operands, and store the result in the target location. If a page fault occurs when trying to write to `C`, because that page is not currently in memory, we will restart the instruction. That means back to step one: fetch the `ADD` instruction again, then get the operands, then perform the addition, and finally write it into the destination location [SGG13].

As we can see, some work is repeated here: fetching and decoding the instruction, as well as taking the operands and doing the addition. This is not a big deal, because the time it takes the CPU to do such an operation is minuscule. CPUs are very good at executing instructions and doing this one a second time is not a big task.

While fetching the page containing `C` from disk, the page that contains `A` or `B` could get swapped out, meaning that the second run of the instruction will also produce a page fault. This is unlikely if using a sane replacement algorithm, because the page with `A` and `B` having just been referenced, it is a poor candidate for eviction. But a random page replacement algorithm could result in that behaviour on occasion. While hypothetically possible, it is very unlikely that a system would get stuck on the same instruction forever if the page containing `A` were constantly replaced in memory and cache by the page containing `C` and vice versa. But a system vulnerable to this problem would have some very significant design issues, to say the least.

The question is, can every instruction be restarted without affecting the outcome? The answer is no; an example is the situation that occurs when an instruction modifies more than one memory location. If we are moving a block of n bytes, it is possible those bytes will straddle a page boundary[2], either at the source or destination. We would like to avoid this situation, because the move operation may not be easily restarted if the source and destination overlap (i.e. the source is modified). One is for the CPU to try to access the start and end addresses before the move begins; if one of the pages needed is not in memory, the page fault is triggered before any data is changed, so we can be sure the move will succeed when it actually starts. Another solution is temporary registers to hold overwritten location; if a page fault occurs, then the temporary data is restored so the instruction may be restarted without affecting the operation's correctness [SGG13].

Virtual Memory Performance

As we have already seen, finding something in the cache is significantly faster than having to go to main memory. Retrieving something from disk is dramatically slower, but computing how long it takes to retrieve a given page will follow the same principle. Recall from earlier the effective access time formula:

$$\text{Effective Access Time} = h \times t_c + (1 - h) \times t_m$$

Where h is the hit rate, t_c the time to retrieve a page from cache, and t_m is the time to retrieve it from memory. If we replace the terms t_c and t_m with t_m and t_d (time to retrieve it from disk) respectively, and redefine h as p, the chance that a page is in memory, we can get an idea of the effective access time in virtual memory:

[2]Some CPUs and operating systems have boundary issues and take this kind of thing a bit more seriously, requiring alignment of data structures to byte and block boundaries... others don't seem to mind at all.

$$\text{Effective Access Time} = p \times t_m + (1 - p) \times t_d$$

And just while we're at it, we can combine the caching and disk read formulae to get the true effective access time for a system where there is only one level of cache:

$$\text{Effective Access Time} = h \times t_c + (1 - h)(p \times t_m + (1 - p) \times t_d)$$

This is good, but what is t_d? This is a measurable quantity so it is possible, of course, to just measure it[3]. We expect p to be small if our paging algorithm is any good. But what needs to take place to handle a page fault (miss) is [SGG13]:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Identify this interrupt as a page fault.
4. Check that the page reference was legal.
   (a) If so, determine the location of the page on disk.
   (b) If not, terminate the requesting program. Steps end here.
5. Figure out where to place the page in memory (use our replacement algorithm).
6. Is the frame we have selected currently filled with a page that has been modified?
   (a) If so, schedule a disk write to flush that page out to disk. The disk write request is placed in a queue.
   (b) If not, go to step 11.
7. Wait for the disk write to be executed. The CPU can do something else in the meantime, of course.
8. Receive an interrupt when the disk write has completed.
9. Save the registers and state of the other process if the CPU did something else.
10. Update the page tables to reflect the flush of the replaced page to disk. Mark the destination frame as free.
11. Issue a disk read request to transfer the page to the free frame.
12. As before, while waiting, let the CPU do something else.
13. Receive an interrupt when the disk has completed the I/O request.
14. Save the registers and state of the other process if the CPU did something else.
15. Update the page tables to reflect the newly read page.
16. Restore the state of and resume execution of the process that encountered the page fault, restarting the instruction that was interrupted.

The slow step in all of this, is obviously, the amount of time it takes to load the page from disk. According to [SGG13], restarting the process and managing memory and such take something like 1 to 100 μs. A typical hard drive in their example has a latency of 3 ms, seek time (moving the read head of the disk to the location of the page) is around 5 ms, and a transfer time of 0.05 ms. So the latency plus seek time is the limiting component, and it's several orders of magnitude larger than any of the other costs in the system. And this is for servicing a request; don't forget that several requests may be queued, making the time even longer.

[3] One of my favourite engineering sayings is "Don't guess; measure."

Thus the disk read term t_d dominates the effective access time equation. If memory access takes 200 ns and a disk read 8 ms, we can roughly estimate the access time in nanoseconds as $(1 - p) \times 8\,000\,000$.

If the page fault rate is high, performance is awful. If performance of the computer is to be reasonable, say around 10%, the page fault rate has to be very, very low. On the order of 10^{-6}.

And now you also know why solid state drives, SSDs, are such a huge performance increase for your computer. Instead of spending time measured in the milliseconds to find a page on disk, SSDs produce the data much quicker, with times that look more like memory reads.

We have not yet covered file systems, but files tend to come with a bunch of overhead for file creation and management. To avoid this, the system usually has a "swap file" which is just one giant file or partition of the hard drive. The system can get better performance by just dealing with the swap file as one big file (block) and not tiny individual files [SGG13].

Copy-on-Write

Recall that in UNIX we create a new process with a call to `fork()` and that this creates an exact duplicate of the parent process. That process may replace itself with the `exec()` system call immediately. Thus it seems like it might be a waste of time to copy the memory space of the parent if the child is just going to throw it away immediately. What UNIX systems do is use the *copy-on-write* technique: the parent and child share the same pages, but they are marked as copy-on-write [SGG13].

If there is immediately an `exec()` invocation then the new memory pages for the child are brought in and the unnecessary work is avoided. But suppose the child is to remain a clone of the parent.

Initially, all pages are shared but marked. As soon as either the parent or child attempts to modify a page, a copy of the page is made and the modification is then applied to the new copy of the page. All unmodified pages remain shared, but only the pages that are actually modified will be copied, so there is no unnecessary copying.

Some versions of UNIX, notably Solaris and Linux, have a system call `vfork()` which skips the copy-on-write procedure. The parent process is suspended and the child process uses the memory space of the parent [SGG13]. So any alterations the child makes will be visible to the parent when it resumes. Thus this is potentially dangerous, as the child can wreak a whole bunch of havoc. This is efficient if the intention is to immediately `exec()` and get a new memory space.

References

[HZM14] Douglas Wilhelm Harder, Jeff Zarnett, and Vajih Montaghami. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2014. Online; version 0.14.12.22.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.