Lecture 16 — Virtual Memory

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2024

Push it to the Limit

Paging helps, but there is a limit.

A program requiring more memory than the physical memory cannot run.

Maybe that seems ridiculous in the modern era of 16, 32+GB?

Server or supercomputer systems working on large datasets may require more memory than is available in the machine.

The Limits of Paging

What about multi-{user, processor, process} systems?

The sum of memory requirements exceeds the available memory.

A processor may be waiting because a process that is otherwise ready to run cannot proceed because there are insufficient free frames for it to run.

All or Nothing

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.

All or Nothing

Code to handle unusual situations 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.)

Partially in Memory?

Benefits if we could execute programs that are only partly in memory:

1. A program is no longer constrained by the size of physical memory.
2. Each program could use up less physical memory.
3. Less I/O is needed to swap user programs in or out.

Good News, Everyone!

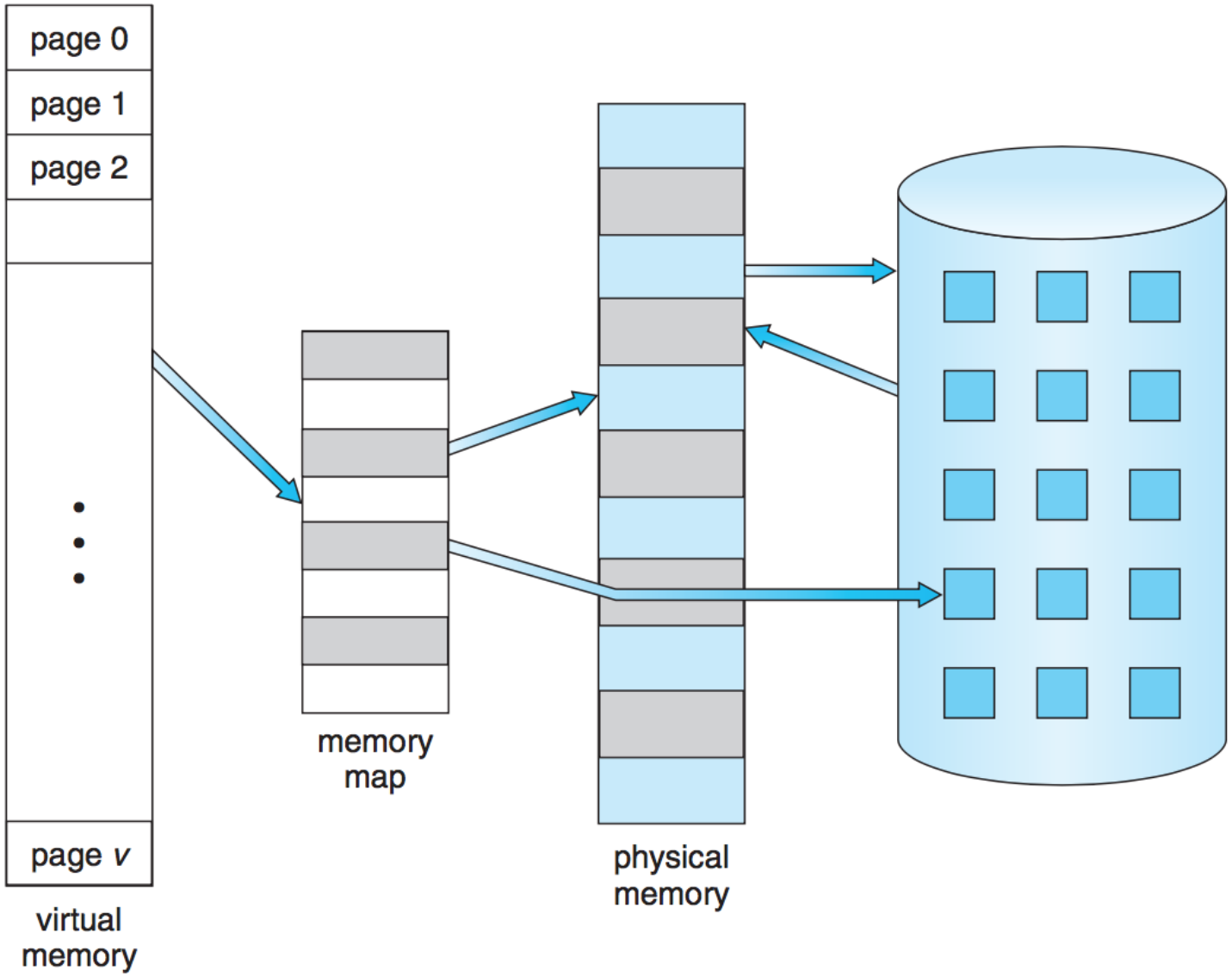
We have already discussed many of the key ideas to making this work.

Good News, Everyone!

The principle is really the same as caching.

Main memory can be viewed as yet another level of cache and the disk is the last stop where the data can be.

Virtual Memory and Physical Memory



Virtual Memory: Loading

The typical approach is also like that of the cache; demand paging.

A page is loaded into memory only if it is referenced or needed, thus preventing unnecessary disk accesses.

Virtual Memory: Loading

This is also called the “lazy” approach.

Though lazy is typically an insult; in this case it is not necessarily bad.

Goal: involve disk as little as possible, because disk is extremely slow.

Thrashing

The fact that disk is so slow means we can get into a state called thrashing.

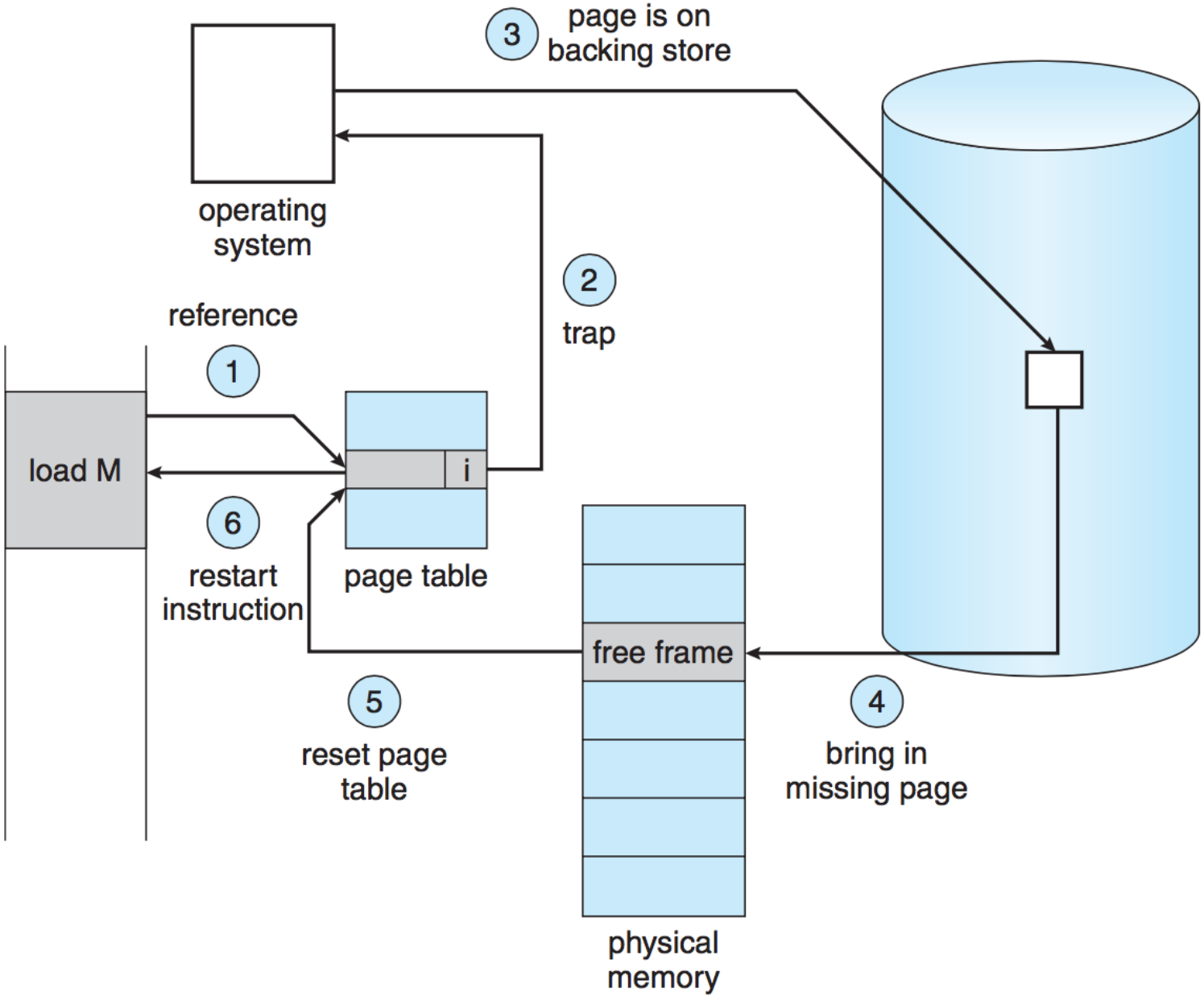
The operating system is spending most or all of its time swapping pages in and out of memory and very little actual work can get done.

We will come back to the subject of thrashing later.

Virtual Memory: Six Steps

1. Check if the memory reference is valid or invalid.
2. If the reference is invalid, terminate the program (segmentation fault).
   If it was valid, but the page referenced is not in memory, continue.
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.
6. Restart the instruction that referenced the page.

Virtual Memory: Handling a Page Fault



Don’t Waste Time

Note that between steps 4 and 5, a significant amount of time will take place.

While the slow disk operations are going on, the process is blocked on that I/O.

In the meantime, the processor can and should be working on something else.

Start Again (Nice Day for a..)

The key requirement is the ability to restart any instruction after 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.

Restart the process exactly where it was.

After the restart, the page needed is in memory and is accessible.

Restarting Instructions

Page faults can occur on any memory access, even fetching the next instruction.
    If it happens at that time, the fetch operation is done again.

If a 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.

Restarting an ADD Instruction

Imagine an ADD instruction that adds A to B and stores the result in C.

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.

Restarting an ADD Instruction

If a page fault occurs when trying to write to C, restart the instruction.

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.

Some work is repeated here: fetching and decoding the instruction, as well as taking the operands and doing the addition.

Round and Round We Go

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.

A random page replacement algorithm could result in that behaviour.

No Do-Overs

Can every instruction be restarted without affecting the outcome? Nope.

Example: an instruction modifies > 1 memory location. If we are moving a block of *n* bytes, it is possible those bytes will straddle a page boundary.

The move operation may not be restarted if the source and destination overlap.

No Do-Overs

Solution: try to access the start and end addresses before the move begins.

If one of the pages 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.

Virtual Memory Performance

Finding something in cache is significantly faster than 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:

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

Virtual Memory Performance

Let's convert this to formula for virtual memory.

Replace t_c with t_m.

Replace t_m and t_d (time to retrieve it from disk).

Redefine h as p, the chance that a page is in memory:

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

Virtual Memory Performance

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.

We expect p to be large if our paging algorithm is any good.

Handling a Page Fault

Handling a page fault is a “simple” 16 step process!

… You can see why it is something we wish to avoid.

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.
    1. If so, determine the location of the page on disk.
    2. If not, terminate the requesting program. Steps end here.

Handling a Page Fault

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?
    1. If so, schedule a disk write to flush that page out to disk. The disk write request is placed in a queue.
    2. 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.

Handling a Page Fault

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 necessary).
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.

Page Faults are Slow

The slow step in all this the amount of time it takes to load the page from disk.

Restarting the process and managing memory and such take 1 to 100 μs.

If a HDD has latency of 3 ms, seek time is 5 ms, and transfer time 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.

This is for servicing a request; don't forget that several requests may be queued, making the time even longer.

Page Faults are Slow

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}.

SSDs: The Way of the Future

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 with times that look more like memory reads.

(Take the technical elective Programming for Performance to learn why this, while faster, is not fast enough.)

Swap File

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.

Thrashing

This topic was previously introduced, and now we'll come back to it.

The quick definition of thrashing still applies.

Aside from intentionally depriving the system of RAM, how can we get into this state, and how can we get out of it?

Thrashing

In simple operating systems, the logic that controlled how many processes to run at a time would rely just on the CPU utilization.

If CPU utilization is low, the CPU needs more work to do!

Thrashing

Assign it more work by starting or bringing more processes into memory.

The global page replacement policy is used here, so when a process gets a page fault, it takes a frame from another process.

Under most circumstances, this works just fine.

Downward Spiral

This situation can run until one process starts to have a lot of page faults.

This is not unreasonable; a compiler might be finished with reading and parsing the input files and moving to generation of binary code.

This requires a whole bunch of new instructions pages, plus pages for output.

When this process does so, it starts taking pages from other processes.

Downward Spiral

The victim processes need the pages they had, so when they get a turn to run, they too start generating page faults.

So more and more requests are queued up for memory writes and reads, so the CPU is not very busy.

Here's the fatal mistake: seeing that the CPU is not very busy, the OS schedules **more** programs to run.

The Doom in Thrashing

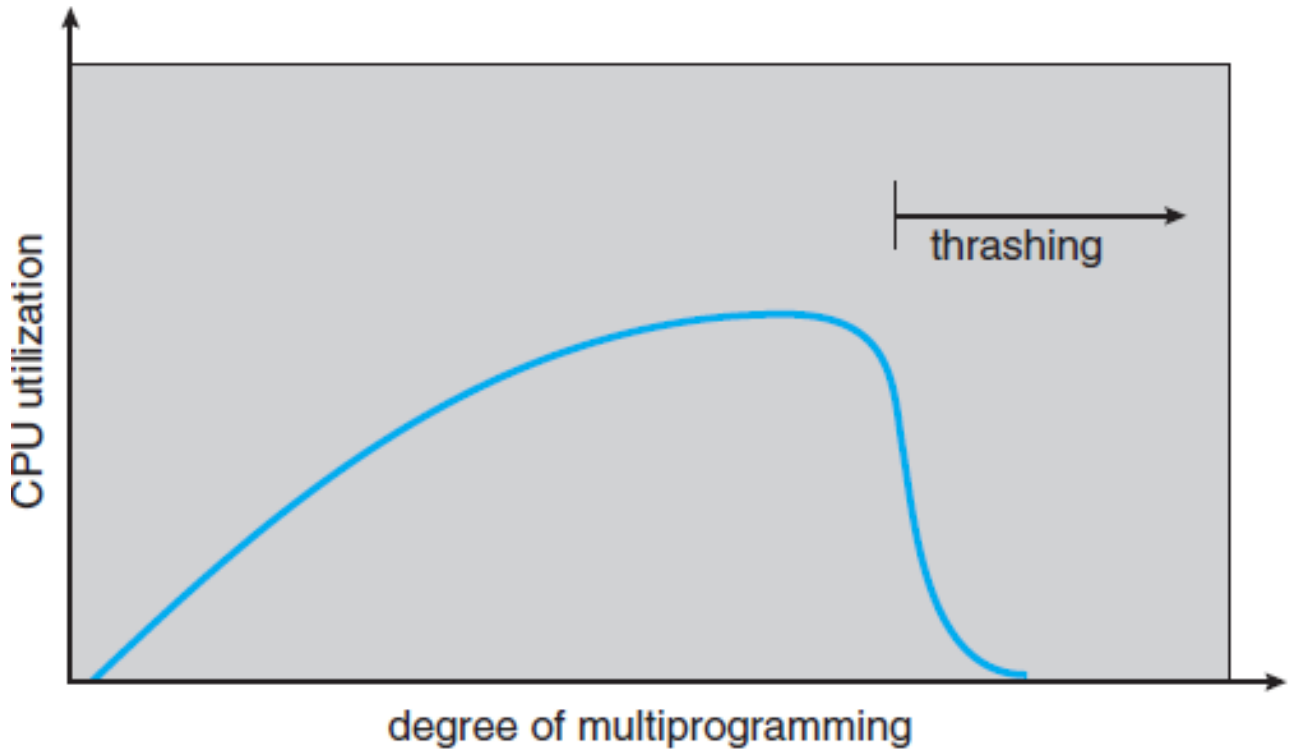
A new process getting started will need at least the minimum number of pages.

These have to come from somewhere, so they will necessarily come from the pages currently belonging to other processes.

This causes more page faults, more time spent paging, lower CPU utilization…
Prompting the OS to start more processes.

Whiplash!

No more work is getting done, because the system spends all its time moving pages into and out of memory, thrashing all around, acting like a maniac.



Breaking the Spiral

To increase CPU utilization we need to stop the thrashing…
which means we need **fewer** programs in memory at a time.

CPU usage alone is not a sufficient indicator of whether more or fewer processes need to be running right now.

It also matters *why* the CPU utilization is low. Another reason that might cause low CPU usage is, as you may recall, deadlock.

Check Assumptions

We've assumed so far that temporal and spatial locality are real.

Are they?

How would we know?

Check Assumptions