CACHING AND CONCURRENCY

ALT-TAB W25: SAT ARORA

INTRODUCTION

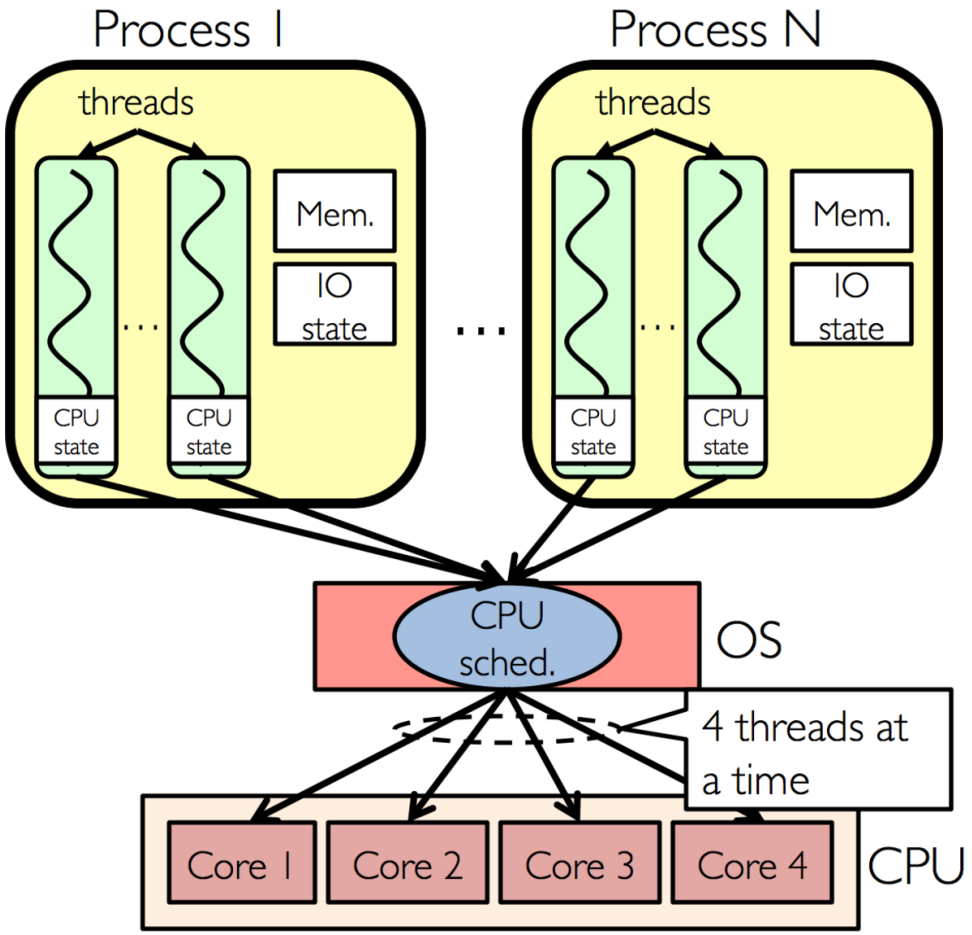
- 4A Computer Science
- Past Experience: Product Backend @ Ramp, Infrastructure @ Matroid
- Current Interests: Programming Performance, Running, Guitar

WHAT IS CONCURRENCY?

- "C*oncurrency* means multiple computations are happening at the same time."
- Applications:
  - Multiple computers in a network
  - Multiple applications in one computer
  - Multiple processors in a computer
- Without concurrency, your computer would wait for every operation. It's thus essential for both performance and responsiveness.

CONCURRENCY FROM THE CPU

- Thread: A unit of execution within a process.
- Core: Runs one or more threads – an individual processing unit on a CPU chip.
- Context Switching (key to concurrency!): Swapping out one thread for another to execute on the core.
- Shared memory: Exists between multiple threads, also exists between multiple cores.



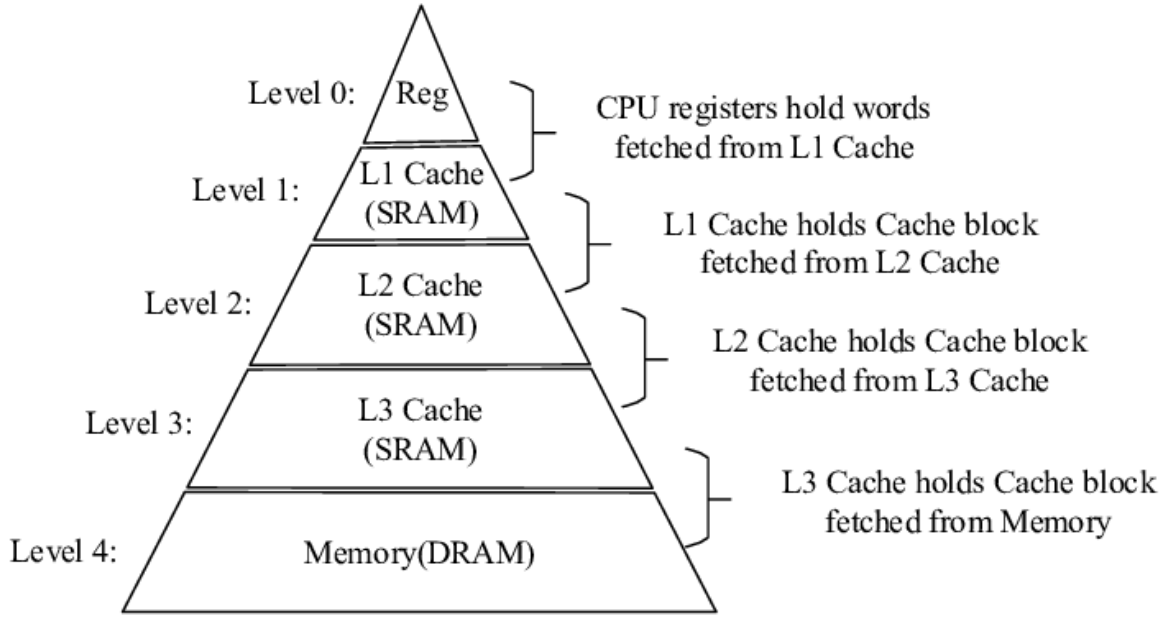
A DIAGRAM

- Source: this medium article
- The **scheduler** manages how threads from different processes are allocated and executed across the available cores

CONCURRENCY IS HARD

- Code is non-deterministic.
  - Example: Get 2 threads to increment the result of a counter
- Issues with accessing **critical sections**.
  - Sections of code that only one thread should have access to at a time.
  - Deadlocking, race conditions, live lock, etc.
- To make matters worse, ChatGPT isn't that good at concurrent programs.

DEBUGGING LIKE



CACHE

WE HAVE A FIXED AMOUNT OF CACHE. WAIT, WHAT IS CACHE? DON'T CONFUSE IT WITH CASH. I KNOW THEY'RE DIFFERENT BECAUSE I'M RUNNING LOW ON CASH, BUT NOT CACHE.

WHAT IS CACHING?

- Caching: The temporary storage of program data that have been previously used, that may need to be retrieved again shortly.
  - Controlled typically by the OS
- Big advantage: Cache lives on the CPU, so access time is much faster than access time for memory (RAM), which is much faster than that of disk (SSD)
- Typical Levels of cache: L1, L2, L3, Memory

LATENCIES OF DIFFERENT CACHE LAYERS

Layer	Approx. Latency	Relative Latency
L1 (private to each thread)	0.5-1 ns	1s
L2 (private to each core)	3-10 ns	5-20s
L3 (shared across all cores)	10-20 ns	40s
RAM	100 ns	~3 minutes
SSD	50-150 µs	~1.5 days

WHAT CAN GO WRONG?

- Typically, cache is divided into **cache lines**. If we need data, it tries to retrieve the entire cache line (it's the smallest unit of transfer from memory)
- If multiple variables live on the SAME cache line, and multiple threads try to write onto those cache lines, they **invalidate** it from other readers, causing a **cache miss** when the other thread tries to read/write to that cache line.
- A cache miss means that the thread must reload the values from memory.
- This is a phenomenon that we call **false sharing**.

SAMPLE STRUCTURES

```
// Struct that causes false sharing
struct FalseSharing {
    long long x;
    long long y;
};
```

```
// Struct that avoids false sharing with padding
struct NoFalseSharing {
    long long x;
    char padding[128 - sizeof(long long)];
    long long y;
    char padding2[128 - sizeof(long long)];
};
```

- An instance of the struct on top will put x and y on the same cache line.
- Adding **padding** in between x and y will cause them to not be put on the same cache line.
- What happens if we run a program on both of these? How do they compare?

DEMO!

- **perf** is a performance analysis tool that we can use to see how algorithms perform
- **perf stat** is used for statistics, and can track certain events (specified by -e)
- We want to look at statistics related to caching, and so our program is:
  perf stat -e cache-references,cache-misses [cmd]

RESULTS

```
 mountain  s97arora  ~  perf stat -e cache-references,cache-misses ./demo false
=== Running False Sharing Demo ===
False sharing time: 0.723614s

 Performance counter stats for './demo false':

        19,078,288      cache-references
         7,786,032      cache-misses              #   40.811 % of all cache refs

       0.730657338 seconds time elapsed

 mountain  s97arora  ~  g++ -pthread false_sharing_demo.cpp -o demo
 mountain  s97arora  ~  perf stat -e cache-references,cache-misses ./demo no-false
=== Running No False Sharing Demo ===
No false sharing time: 0.219734s

 Performance counter stats for './demo no-false':

            59,301      cache-references
            19,887      cache-misses              #   33.536 % of all cache refs

       0.225124613 seconds time elapsed
```

WHAT DID WE SEE?

- By separating the cache lines that the two threads interact on, we allow them to always stay in their cache without getting booted off by the other thread.
- This means that things that run in parallel should be on different cache lines?
- Issues:
  - Padding takes more memory.
  - Threads that must share a variable will always have cache misses.
  - Depending on the mechanics of the machine that algorithms are being run on, you may not always see way less cache misses. An example of this is the **adjacent line prefetcher.**

DEBRIEF

- There's a lot more to concurrency and caching than what we've discovered today – not everything is solved by just padding.
- But, you hopefully have more of an idea with how small changes in how we write code can save a lot of CPU cycles, cache misses, etc.
- The algorithms and possibilities for optimization are very specific to certain use cases, and are considered both when writing software and when designing hardware (Intel, AMD, etc.)

RECOMMENDED COURSES AND APPLICATIONS

- CS 798: Advanced Topics [Multicore Programming]
  - Undergrads would have to override in…
- CS 343: Concurrency
- Graphics programming (CUDA) – relies heavily on parallel + concurrent execution

- CS 350