



NTNU

Norwegian University of
Science and Technology

TDT4102 Bonus Lecture: A Threaded Future

Bart Iver van Blokland

Today

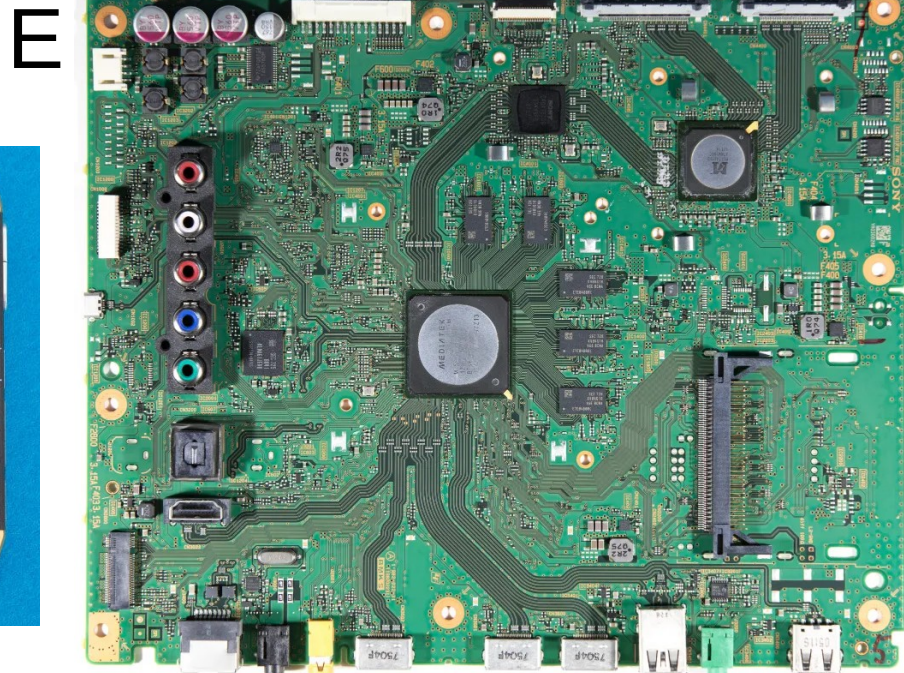
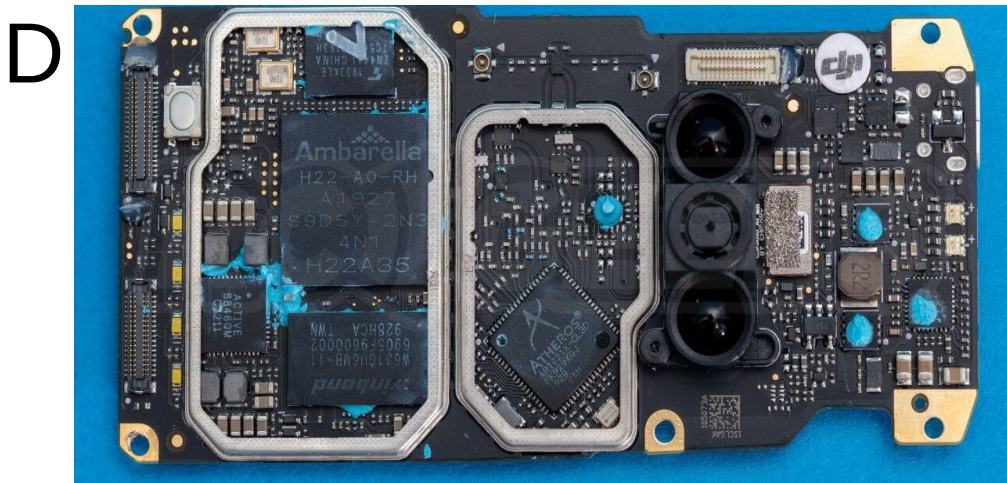
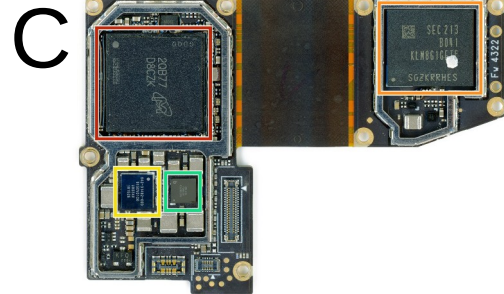
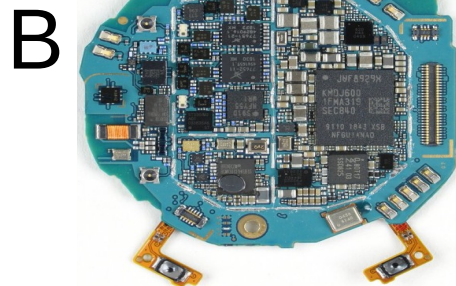
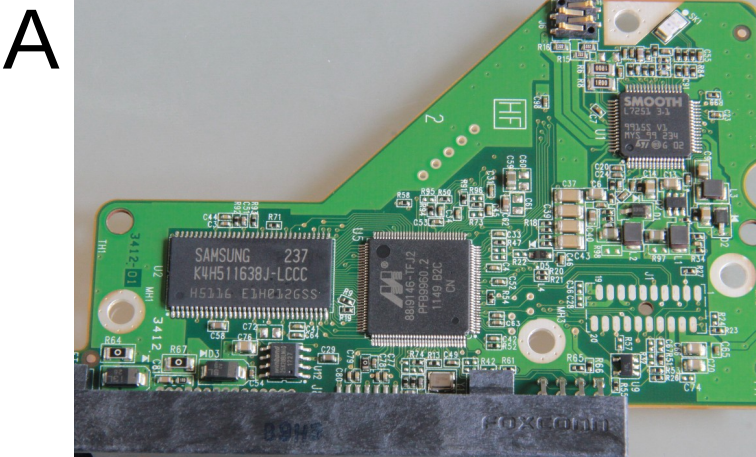
- Why do we care about parallel computing?
- How do we write multithreaded code in C++?
- What are problems that can be encountered when writing multithreaded code?

Parallel processors are everywhere..



Product Name	Launch Date	Total Cores	Processor Base Frequency	Cache	TDP
<input type="checkbox"/> Intel® Quark™ Microcontroller D2000	Q3'15	1	32 MHz	0 KB	
<input type="checkbox"/> Intel® Quark™ SE C1000 Microcontroller	Q4'15	1	32 MHz	8 KB	
<input type="checkbox"/> Intel® Quark™ Microcontroller D1000	Q3'15	1	33 MHz	0 KB	0.025 W
<input type="checkbox"/> Intel® Quark™ SoC X1001	Q2'14	1	400 MHz	16 KB	2.2 W
<input type="checkbox"/> Intel® Quark™ SoC X1011	Q2'14	1	400 MHz	16 KB	2.2 W
<input type="checkbox"/> Intel® Quark™ SoC X1020	Q2'14	1	400 MHz	16 KB	2.2 W
<input type="checkbox"/> Intel® Quark™ SoC X1021	Q2'14	1	400 MHz	16 KB	2.2 W
<input type="checkbox"/> Intel® Quark™ SoC X1021D	Q2'14	1	400 MHz	16 KB	2.2 W
<input type="checkbox"/> Intel® Quark™ SoC X1010	Q1'14	1	400 MHz	16 KB	2.2 W
<input type="checkbox"/> Intel® Quark™ SoC X1020D	Q1'14	1	400 MHz	16 KB	2.2 W
<input type="checkbox"/> Intel Atom® Processor E3815	Q4'13	1	1.46 GHz	512 KB L2 Cache	5 W
<input type="checkbox"/> Intel® Quark™ SoC X1000	Q4'13	1	400 MHz	16 KB	2.2 W
<input type="checkbox"/> Intel® Celeron® Processor G470	Q2'13	1	2.00 GHz	1.5 MB Intel® Smart Cache	35 W
<input type="checkbox"/> Intel® Celeron® Processor 927UE	Q1'13	1	1.50 GHz	1 MB Intel® Smart Cache	17 W

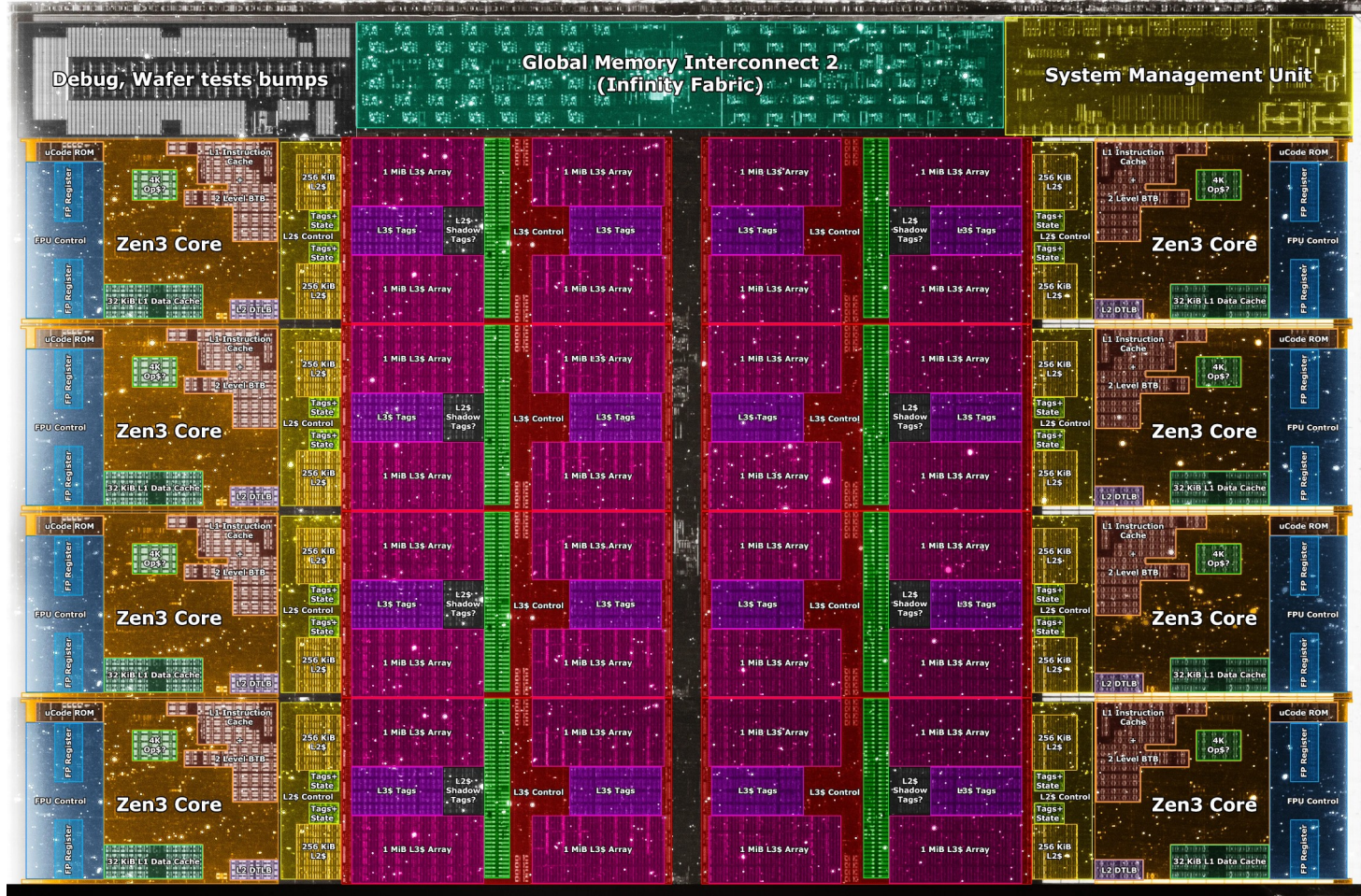






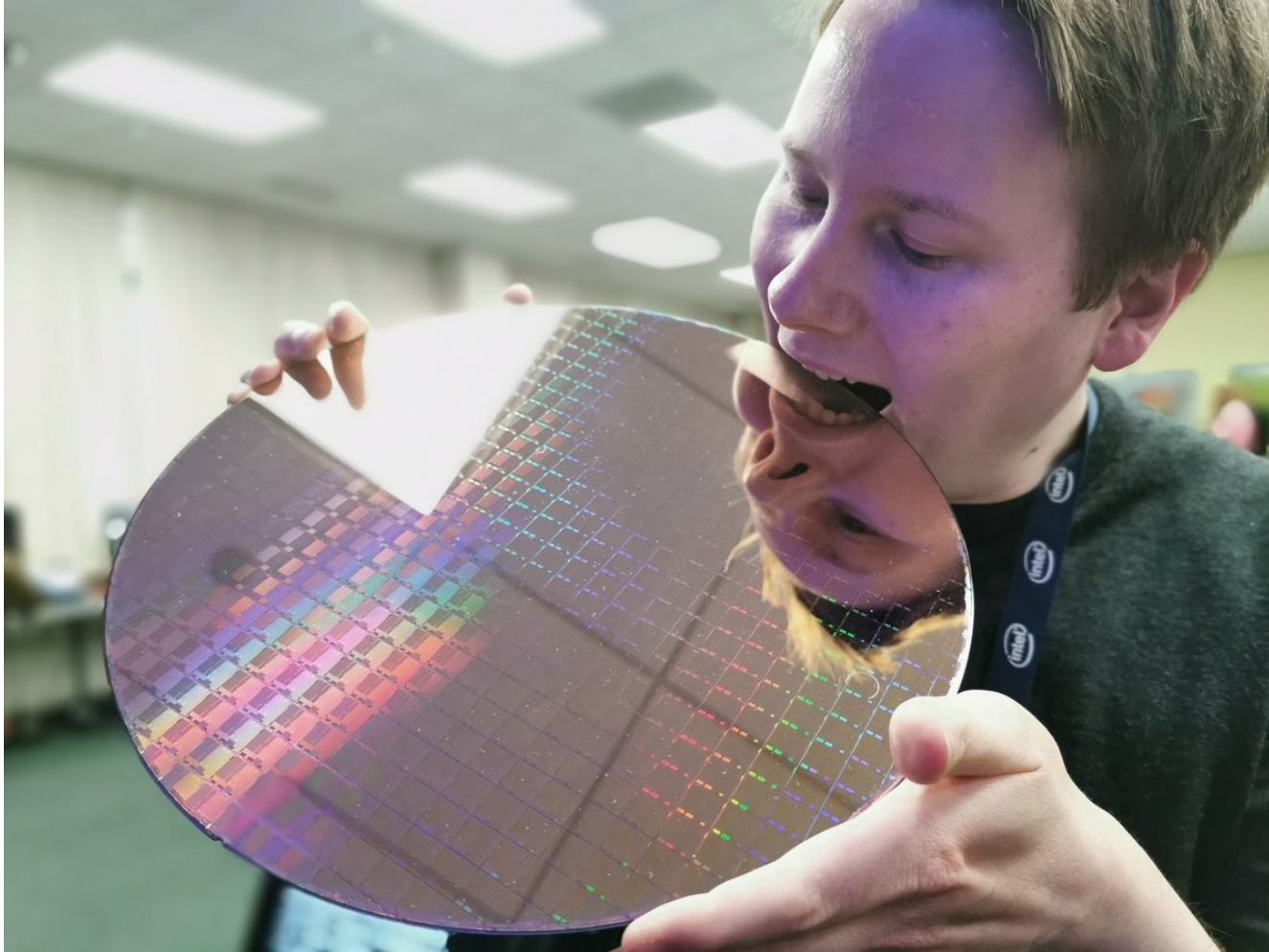
Why parallel computing?

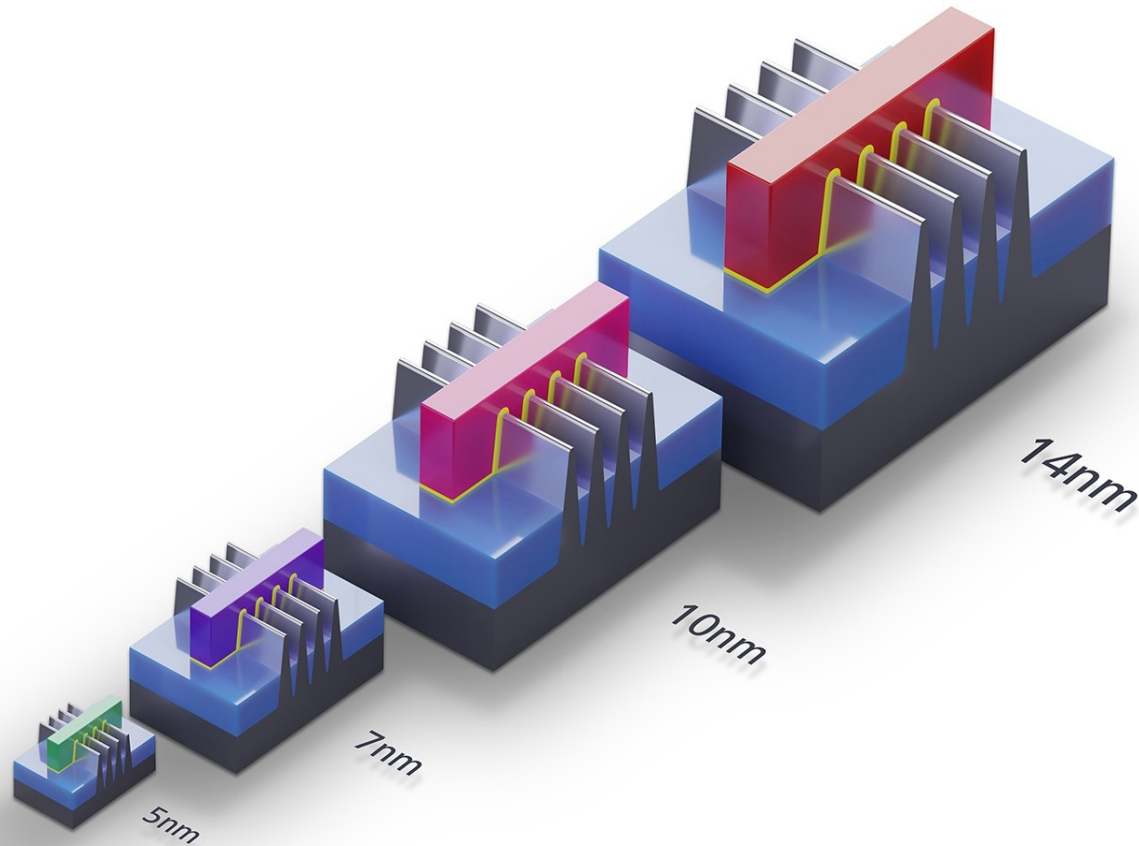
- **Multicore processors are ubiquitous**



Why parallel computing?

- Multicore processors are ubiquitous
- **Needed to fully utilise all cores of a processor**





Intel Pentium 4 Northwood

Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

Floating Point, MMX, SSE2 Renamed Register File 128 entries of 128 bit.

uOp Schedulers

FP Move Scheduler: (8x8 dependency matrix)

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)

Load / Store uOp Scheduler: (8x8 dependency matrix)

Load / Store Linear Address Collision History Table

Integer Execution Core

- (1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle
- (2) Integer Renamed Register File 128 entries of 32 bit + 6 status flags 12 read ports and six write ports
- (3) Databus switch & Bypasses to and from the Integer Register File.
- (4) Flags, Write Back
- (5) Double Pumped ALU 0
- (6) Double Pumped ALU 1
- (7) Load Address Generator Unit
- (8) Store Address Generator Unit
- (9) Load Buffer (48 entries)
- (10) Store Buffer (24 entries)

Execution Pipeline Start

Register Alias History Tables (2x126)
Register Alias Tables uOp Queue

Micro code Sequencer
Micro code ROM & Flash

Instruction Trace Cache

Trace Cache Fill Buffers
Distributed Tag comparators 24 bit virtual Tags

Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 512 entries.

Return Stacks (2x16 entries)

Trace Cache next IP's (2x)

Miscellaneous Tag Data

Instruction Decoder

Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle) Instructions with more than four are handled by Micro Sequencer

Trace Cache LRU bits

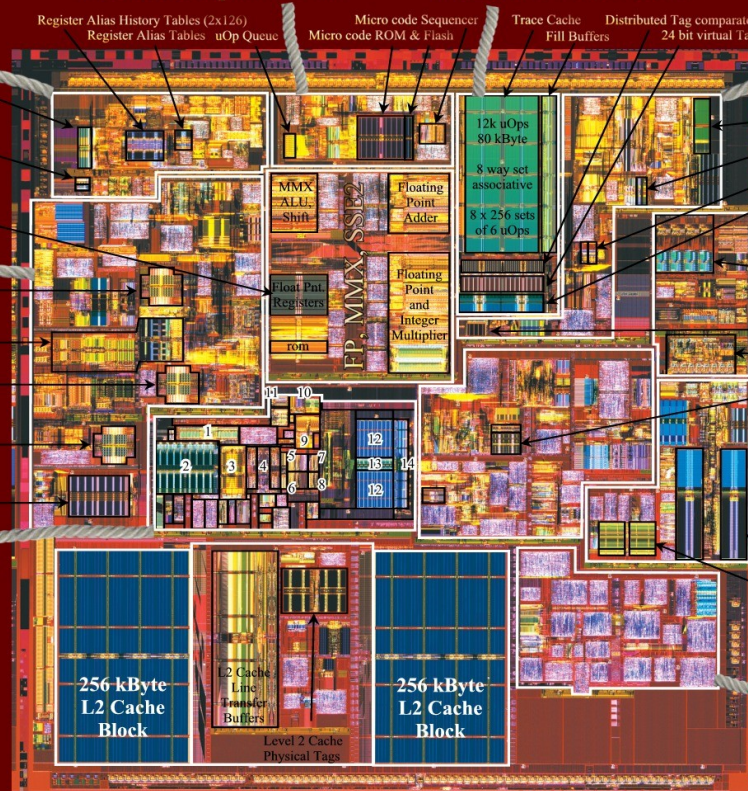
Raw Instruction Bytes in Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Instruction Fetch from L2 cache and Branch Prediction

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

Instruction TLB's 2x64 entry, fully associative for 4k and 4M pages. In: Virtual address [31:12] Out: Physical address [35:12] + 2 page level bits

Front Side Bus Interface, 400..800 MHz



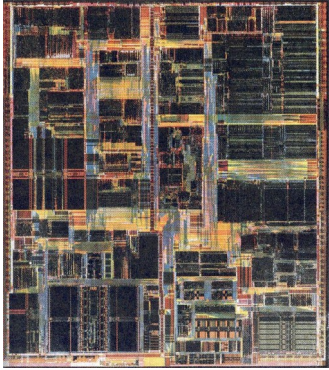
- (11) ROB Reorder Buffer 3x42 entries
- (12) 8 kByte Level 1 Data cache four way set associative. 1R/1W

- (13) Summed Address Index decode and Way Predict
- (14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

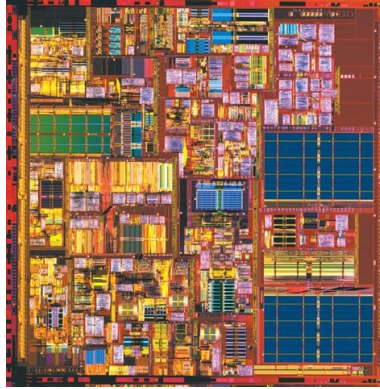
April 19, 2003 www.chip-architect.com

- Improving single core performance linearly requires an exponential number of transistors
- At some point it becomes worth it to spend those transistors on multiple independent cores instead

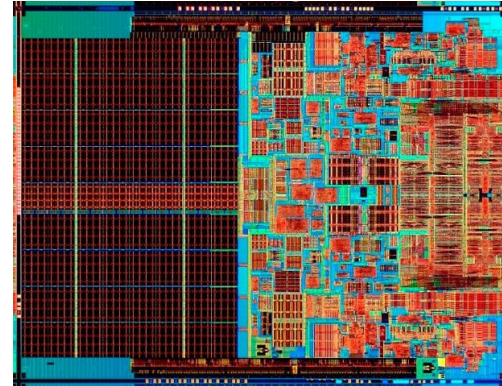
Pentium III



Pentium 4



Pentium D



Core 2 Duo

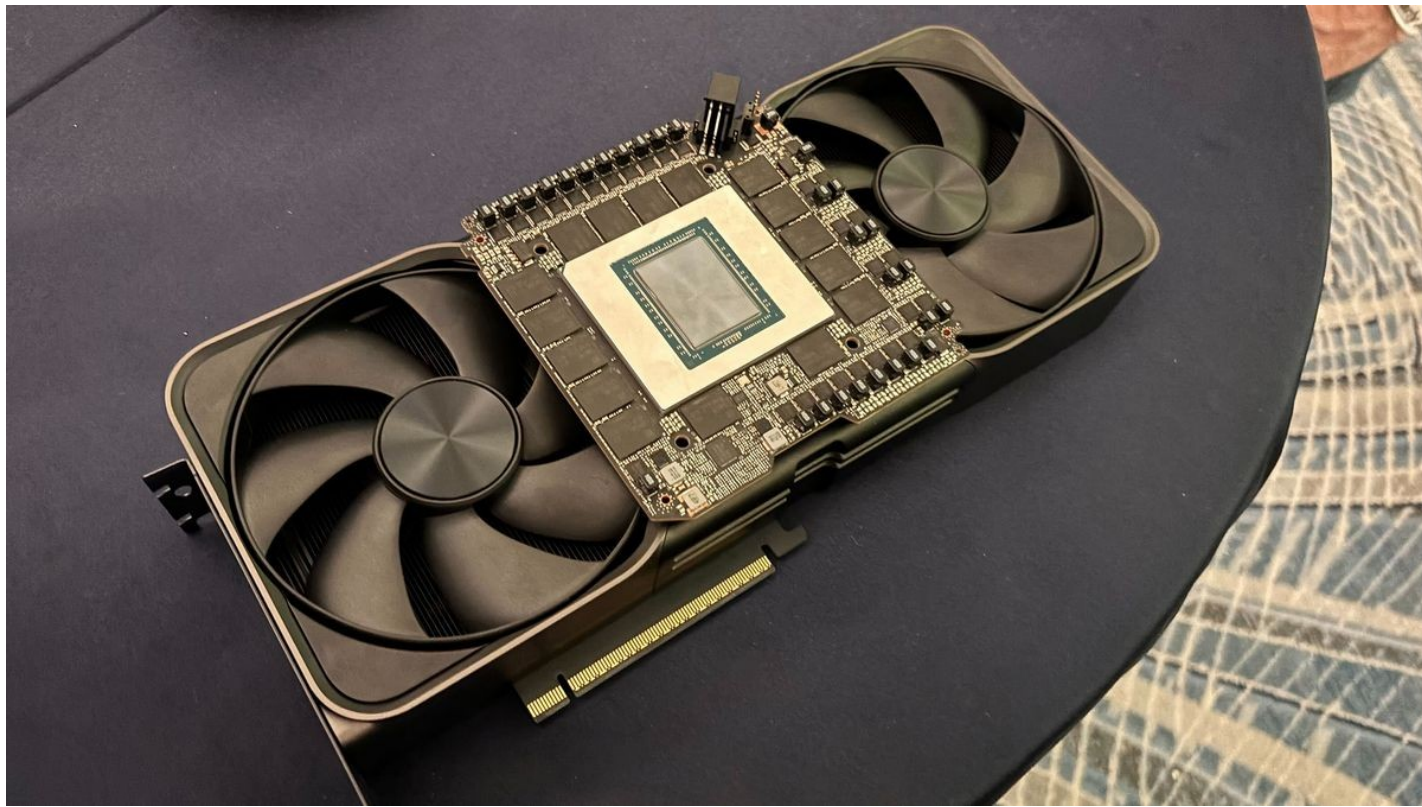
Why parallel computing?

- Multicore processors are ubiquitous
- Needed to fully utilise all cores of a processor
- **Only way for chip manufacturers to improve performance**

Why parallel computing?

- Multicore processors are ubiquitous
- Needed to fully utilise all cores of a processor
- Only way for chip manufacturers to improve performance
- **Some processors are practically useless without it**

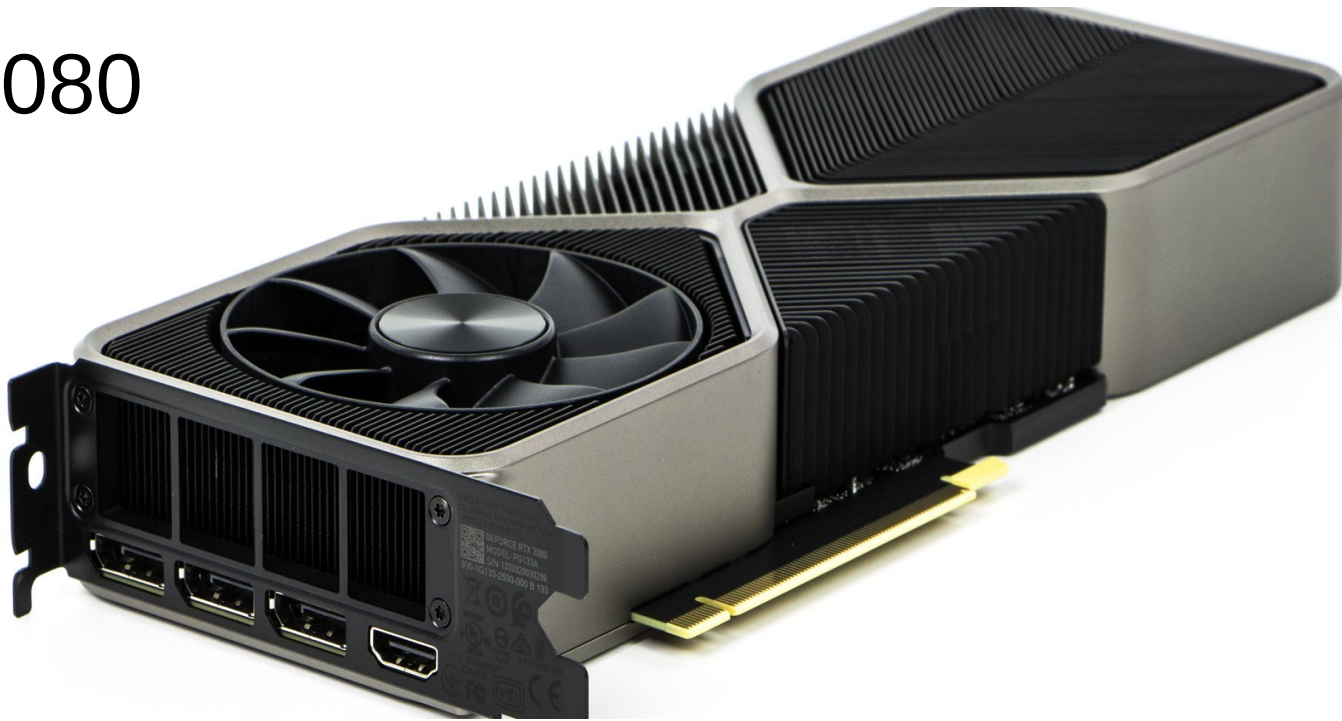
RTX 5090



Power consumption (load): 575W

Theoretical performance: 104,800,000,000 Flops

RTX 3080



Power consumption (load): 320W

Theoretical performance: 29,770,000,000 Flops

Theoretical performance: 29,770,000,000 Flops

Theoretical performance: 29,770,000,000 Flops

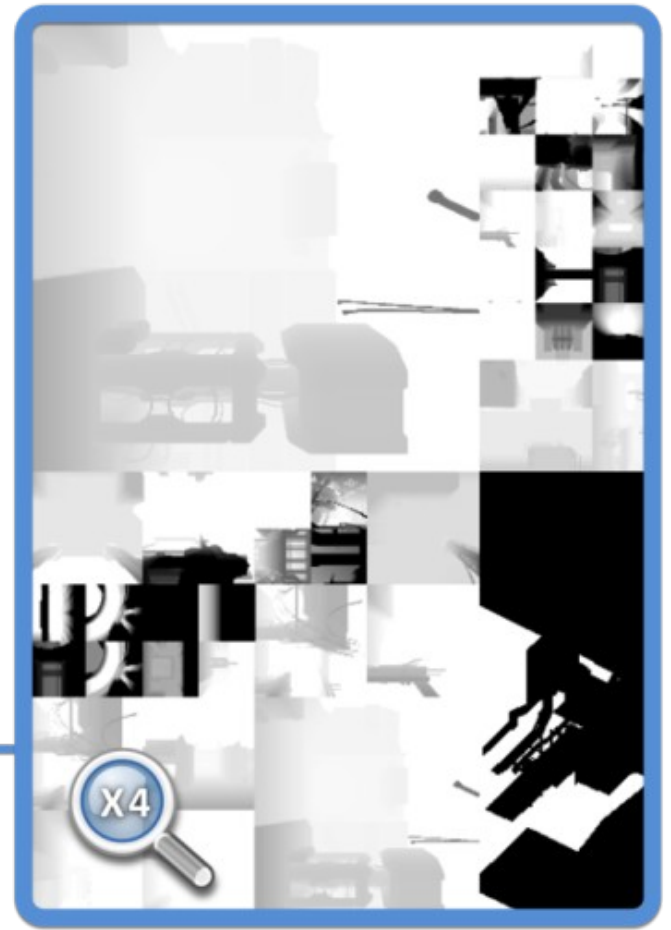
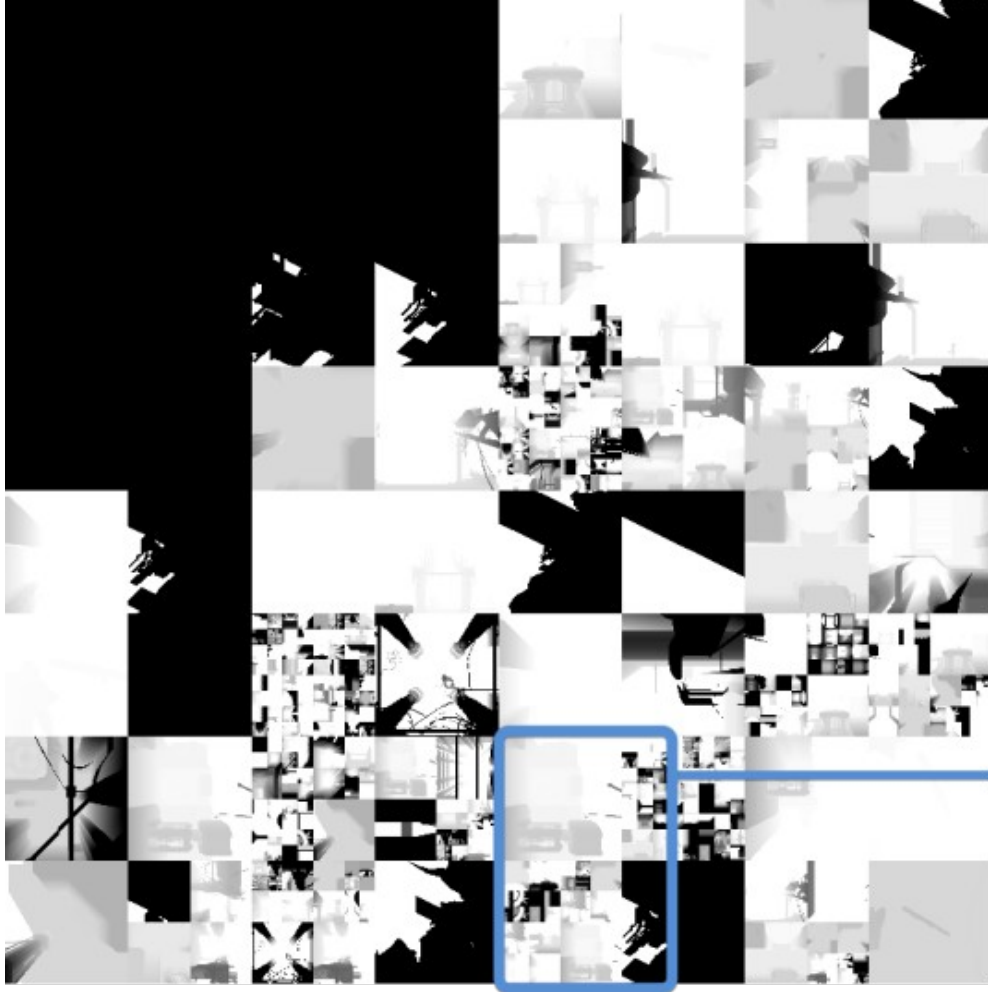
If you'd do one calculation per second,
the same number of calculations would take:

944 years

A more realistic workload..

DOOM













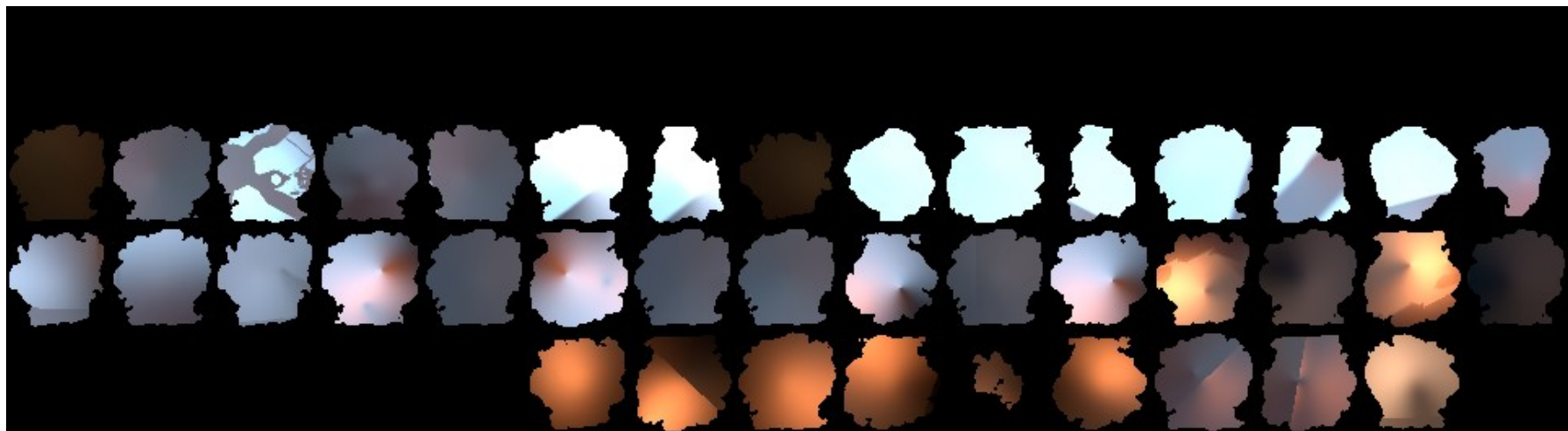






















At a resolution of 2560x1440 pixels,
the GPU does all of that 200 times per second.

(and even gets to take short breaks)

Why parallel computing?

- Multicore processors are ubiquitous
- Needed to fully utilise all cores of a processor
- Only way for chip manufacturers to improve performance
- Some processors are practically useless without it
- **Problem is too large to fit one machine**



Why parallel computing?

- Multicore processors are ubiquitous
- Needed to fully utilise all cores of a processor
- Only way for chip manufacturers to improve performance
- Some processors are practically useless without it
- Problem is too large to fit one machine
- **Can even be useful on single core machines**



Crash Course Parallel Computing

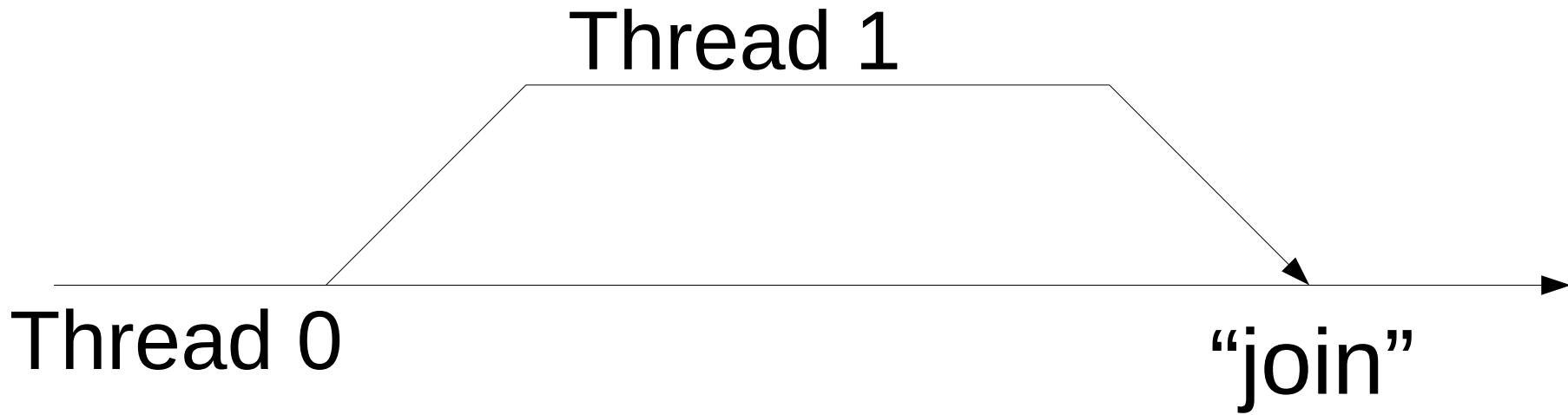
- Using `std::thread`
- Using OpenMP

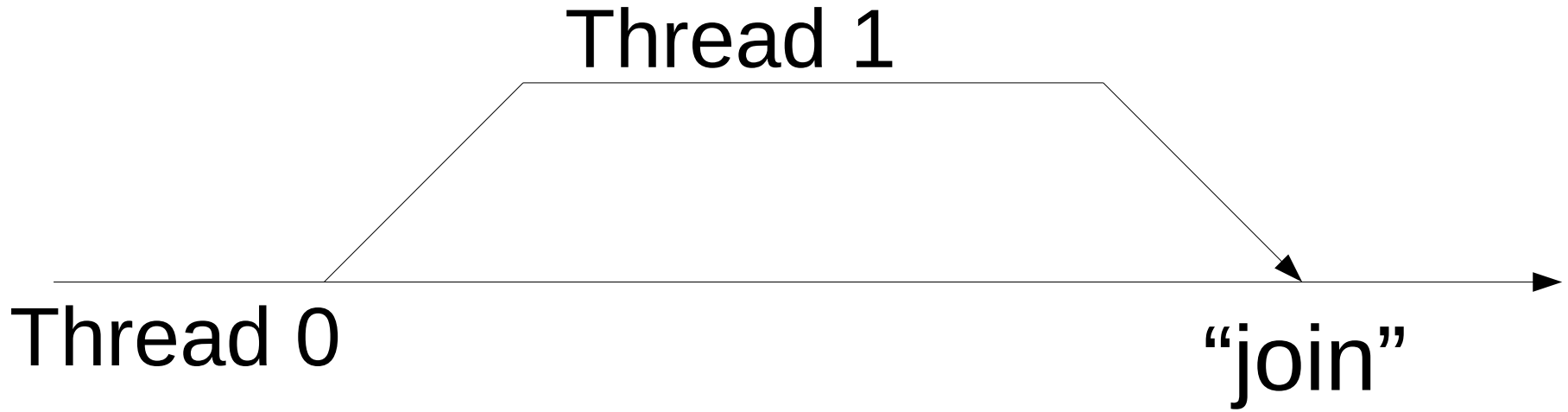
Using MacOS? Run 'brew install libomp' in a terminal

```
#include <thread>
#include <iostream>

void doSomething(int a, int b) {
    std::cout << "I'm from another planet." << std::endl;
}

int main() {
    std::thread uselessThread(doSomething, 4, 2);
    uselessThread.join();
    return 0;
}
```





Thread 0 “joins” Thread 1


→ Thread 0 waits for Thread 1 to complete and exit.



```
#include <iostream>
#include <thread>
```

```
void doSomething(int* value) {
    for(int i = 0; i < 1000000; i++) {
        (*value)++;
    }
}
```

```
int main() {
    int* sum = new int{0};
    std::thread threads[10];
    for(int i = 0; i < 10; i++) {
        threads[i] = std::thread(doSomething, sum);
    }
    for(int i = 0; i < 10; i++) {
        threads[i].join();
    }
    std::cout << "Sum: " << *sum << std::endl;
    return 0;
}
```



What is the value of sum?

131571

116638

113056

121983

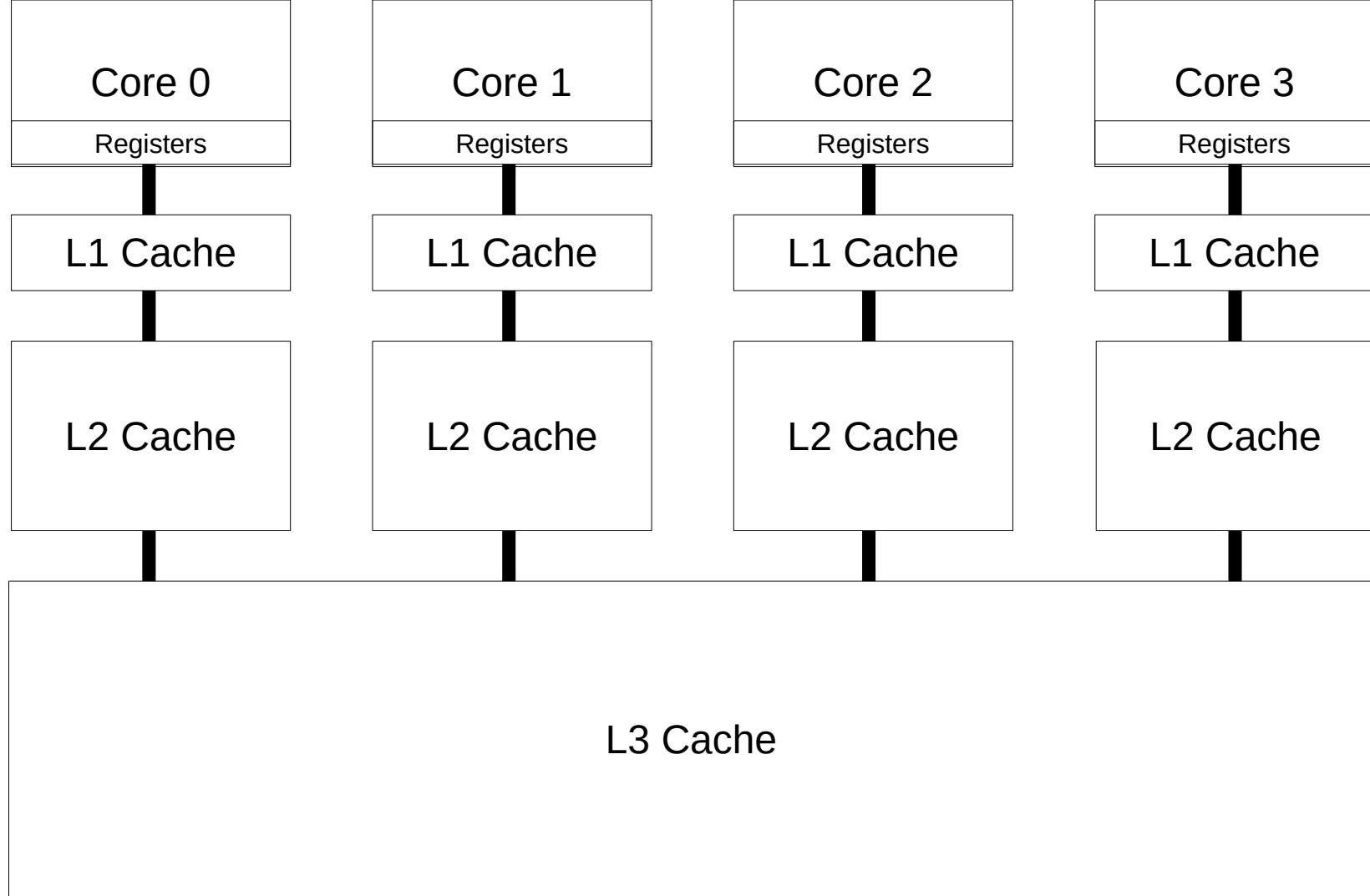
170546

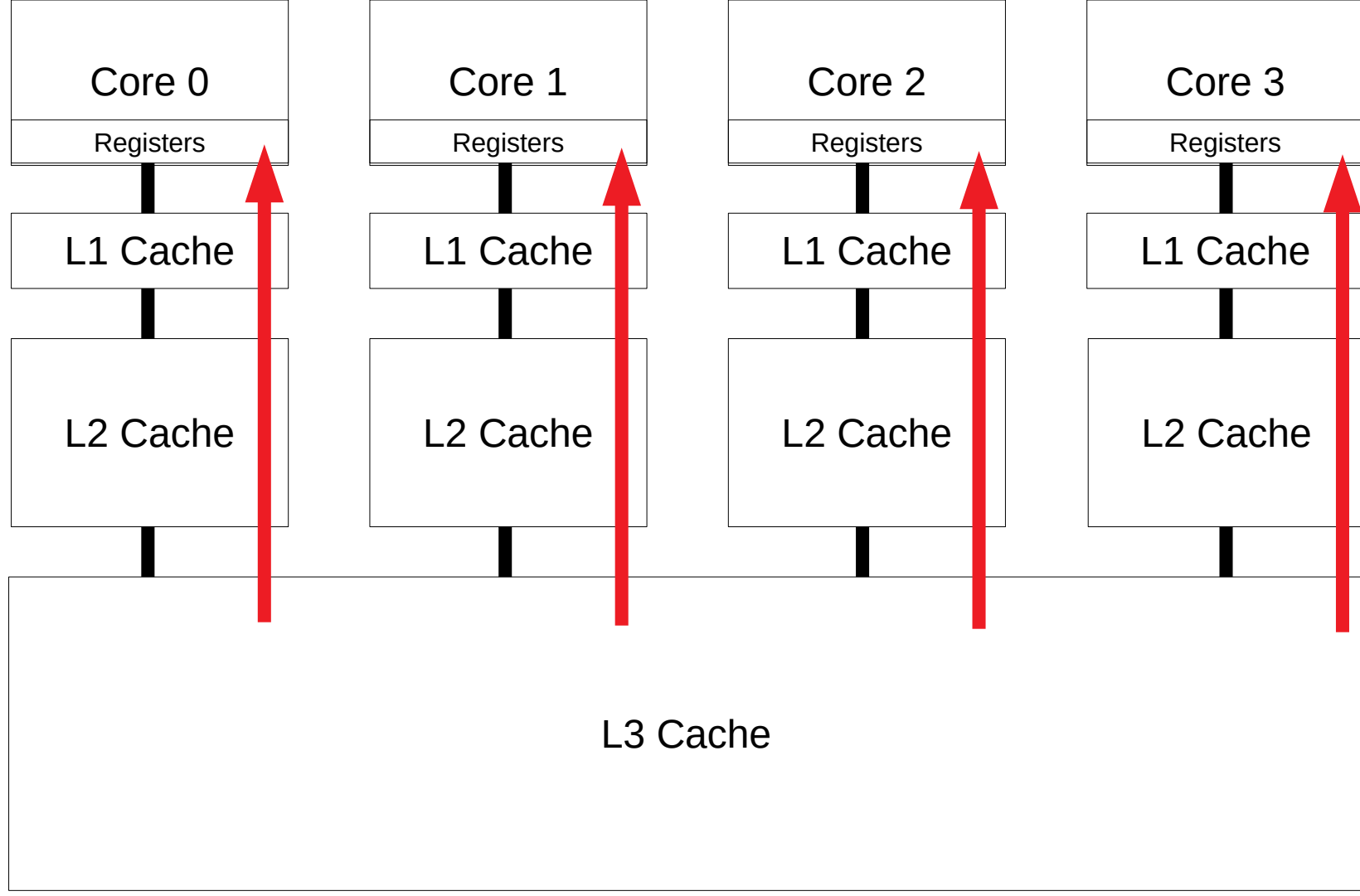
124298

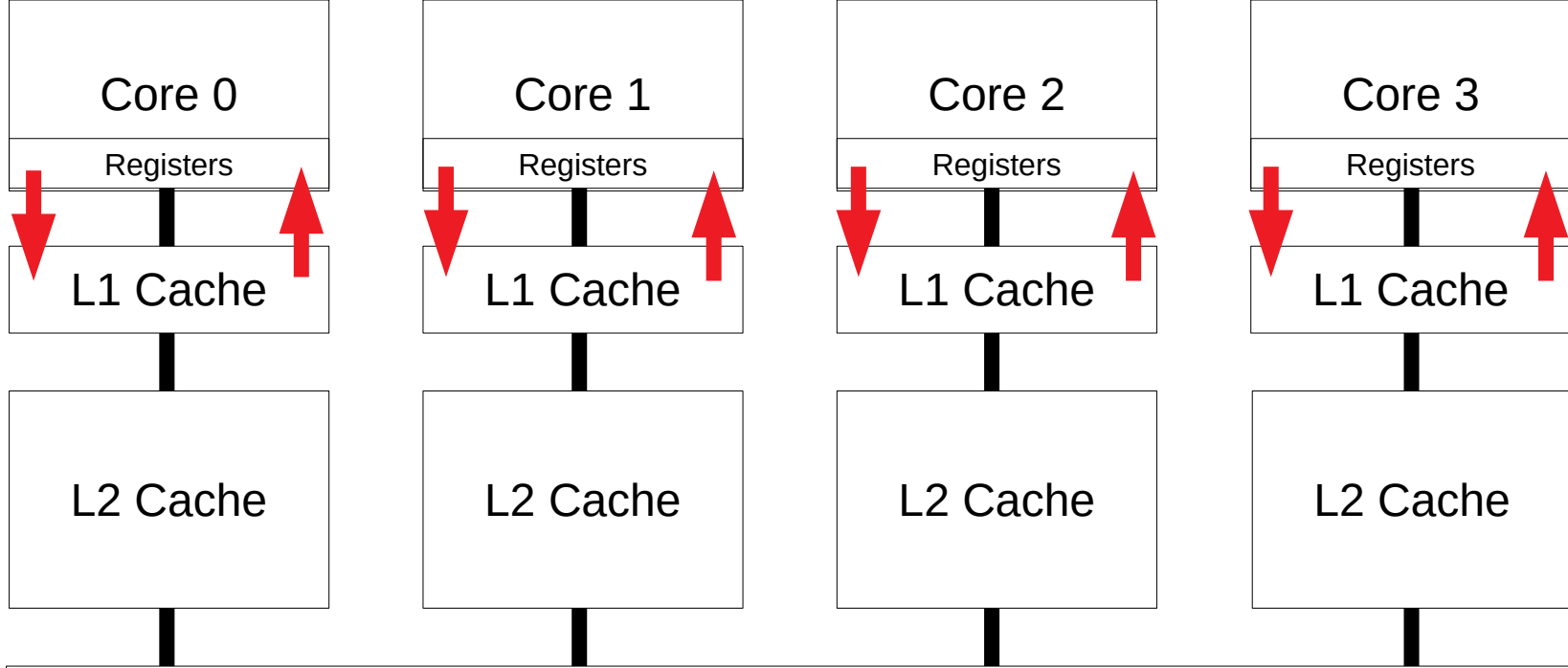
127607

170724

Cache







Problem: Cache Coherence

Race Condition:

A part of a program whose outcome is nondeterministic due to factors outside the control of the program itself.

What causes nondeterministic results in a race condition?

- Operating System scheduler
- Cache
 - And associated cache coherency protocols
- CPU Interrupts
- Variations in accessing RAM memory banks

And more..

Main problem:

Threads can be interrupted
at *any* time, in *any* order.

```
// SHARED variable  
int a = 5;
```

```
// thread 0:
```

```
a = 3;
```

```
if(a < 5) {  
    a = 2;  
}
```

```
// thread 1:
```

```
if(a == 5) {  
    a = 1;  
} else {  
    a = 4;  
}
```

```
// SHARED variable  
int a = 5;
```

```
// thread 0:
```

```
a = 3;  
  
if(a < 5) {  
    a = 2;  
}
```

```
// thread 1:
```

```
if(a == 5) {  
    a = 1;  
} else {  
    a = 4;  
}
```



```
// SHARED variable  
int a = 5;
```

```
// thread 0:
```

```
a = 3;
```

```
if(a < 5) {  
    a = 2;  
}
```

```
// thread 1:
```

```
if(a == 5) {  
    a = 1;  
} else {  
    a = 4;  
}
```

```
// SHARED variable  
int a = 3;
```

```
// thread 0:
```

```
a = 3;
```

```
if(a < 5) {  
    a = 2;  
}
```

```
// thread 1:
```

```
if(a == 5) {
```

```
    a = 1;  
} else {  
    a = 4;  
}
```

```
// SHARED variable  
int a = 1;
```

```
// thread 0:
```

```
a = 3;
```

```
if(a < 5) {  
    a = 2;  
}
```

```
// thread 1:
```

```
if(a == 5) {  
    a = 1;
```

```
} else {  
    a = 4;  
}
```

```
// SHARED variable  
int a = 2;
```

```
// thread 0:
```

```
a = 3;
```

```
if(a < 5) {  
    a = 2;
```

```
}
```

```
// thread 1:
```

```
if(a == 5) {  
    a = 1;
```

```
} else {  
    a = 4;
```

```
}
```

```
// SHARED variable  
int a = 5;
```

```
// thread 0:
```

```
a = 3;  
  
if(a < 5) {  
    a = 2;  
}
```

```
// thread 1:
```

```
if(a == 5) {  
    a = 1;  
} else {  
    a = 4;  
}
```



```
// SHARED variable  
int a = 3;
```

```
// thread 0:
```

```
a = 3;
```

```
if(a < 5) {  
    a = 2;  
}
```

```
// thread 1:
```

```
if(a == 5) {  
    a = 1;  
} else {  
    a = 4;  
}
```

```
// SHARED variable  
int a = 3;
```

```
// thread 0:
```

```
a = 3;
```

```
if(a < 5) {  
    a = 2;  
}
```

```
// thread 1:
```

```
if(a == 5) {  
    a = 1;
```

```
} else {  
    a = 6;
```

```
}
```

```
// SHARED variable  
int a = 6;
```

```
// thread 0:
```

```
a = 3;
```

```
if(a < 5) {  
    a = 2;  
}
```

```
// thread 1:
```

```
if(a == 5) {  
    a = 1;  
} else {  
    a = 6;  
}
```

```
// SHARED variable  
int a = 6;
```

```
// thread 0:
```

```
a = 3;  
  
if(a < 5) {  
    a = 2;  
}
```

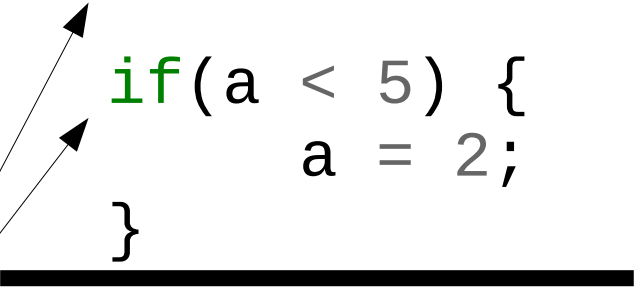
```
// thread 1:
```

```
if(a == 5) {  
    a = 1;  
} else {  
    a = 6;  
}
```

```
// SHARED variable  
int a = 6;
```


```
// thread 0:
```

```
a = 3;  
if(a < 5) {  
    a = 2;  
}
```



```
// thread 1:
```

```
if(a == 5) {  
    a = 1;  
} else {  
    a = 6;  
}
```



Consecutive lines!

What can happen?

- <https://en.m.wikipedia.org/wiki/Therac-25>

The **Therac-25** is a computer-controlled [radiation therapy](#) machine produced by [Atomic Energy of Canada Limited](#) (AECL) in 1982 after the Therac-6 and Therac-20 units (the earlier units had been produced in partnership with Compagnie générale de radiologie (CGR) of France).^[1]

The Therac-25 was involved in at least six accidents between 1985 and 1987, in which some patients were given massive [overdoses of radiation](#).^{[2]:425} [Because of concurrent programming errors](#) (also known as race conditions), it sometimes gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury.^[3]

These accidents highlighted the dangers of software [control](#) of safety-critical systems.

The Therac-25 has become a standard case study in [health informatics](#), [software engineering](#), and [computer ethics](#). It highlights the dangers of engineer overconfidence^{[2]:428} [after the engineers dismissed end-user reports](#), leading to severe consequences.

So how do we solve race conditions?





Semaphore
(mutex)

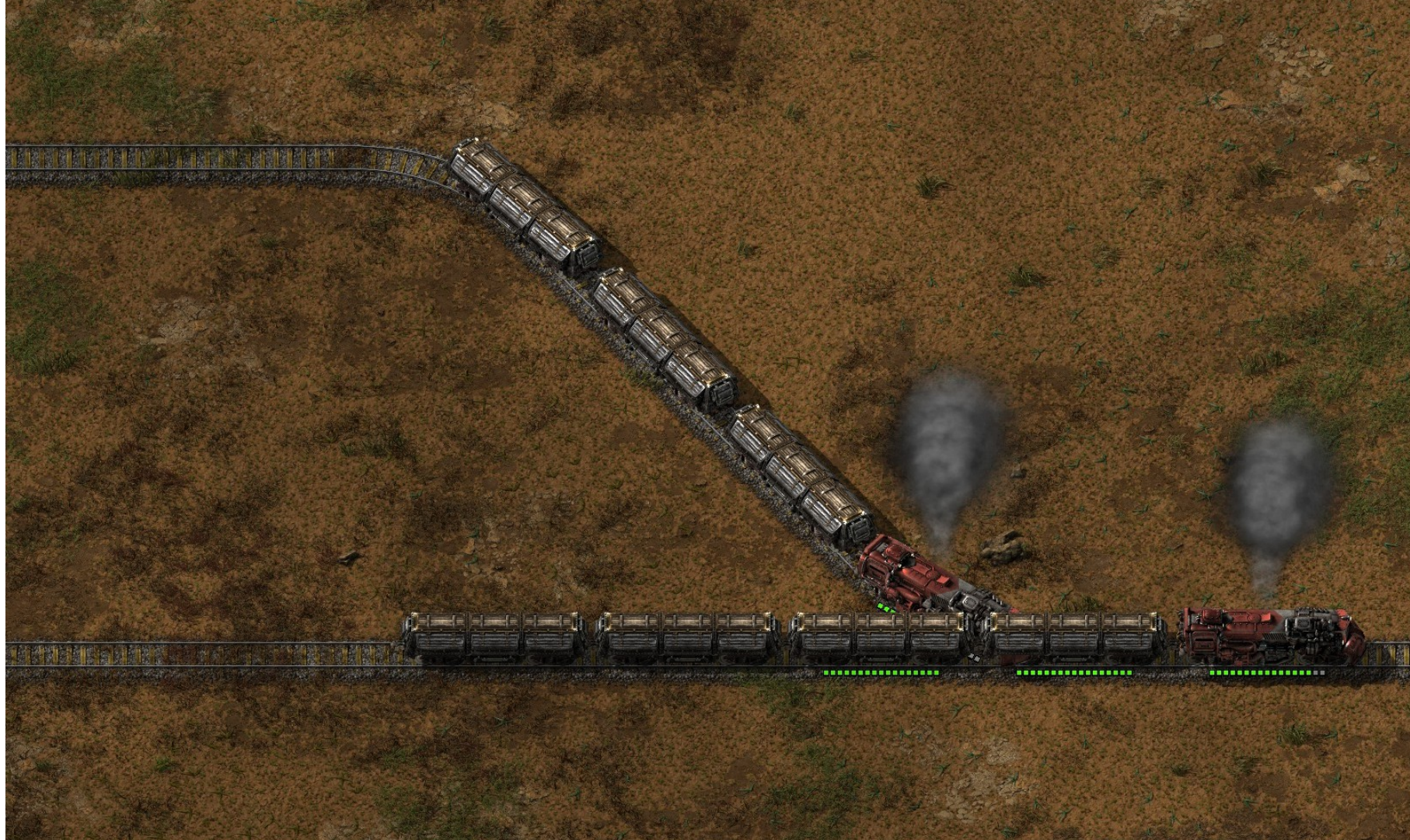


Shared variable or resource





Thread





Mutex



Threads check lock
before going in



And wait whenever the resource
is occupied by another thread



When multiple threads are waiting,
only one at a time is given access

How do we implement a semaphore?

```
unsigned int semaphore = 5;
```

```
struct semaphore {  
    unsigned int count = 1; // 1 makes it a mutex  
    void down() {  
        while(count == 0) {}  
        count--;  
    }  
    void up() { count++; }  
};
```

```
void increment(semaphore* lock, int* value) {  
    lock->down();  
    (*value)++;  
    lock->up();  
}
```

100

100

100

100

100

99

100

100

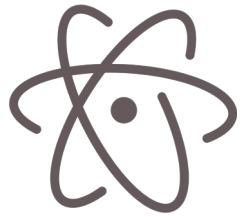


A black and white image of a stage. At the top, eight spotlights are arranged in a semi-circle, casting bright beams of light down towards the center. In the center of the stage is a white, three-tiered circular pedestal. The text 'CMPXCHG' is written in a large, white, sans-serif font, centered horizontally and positioned just above the pedestal. The background is dark, and the overall atmosphere is dramatic and focused.

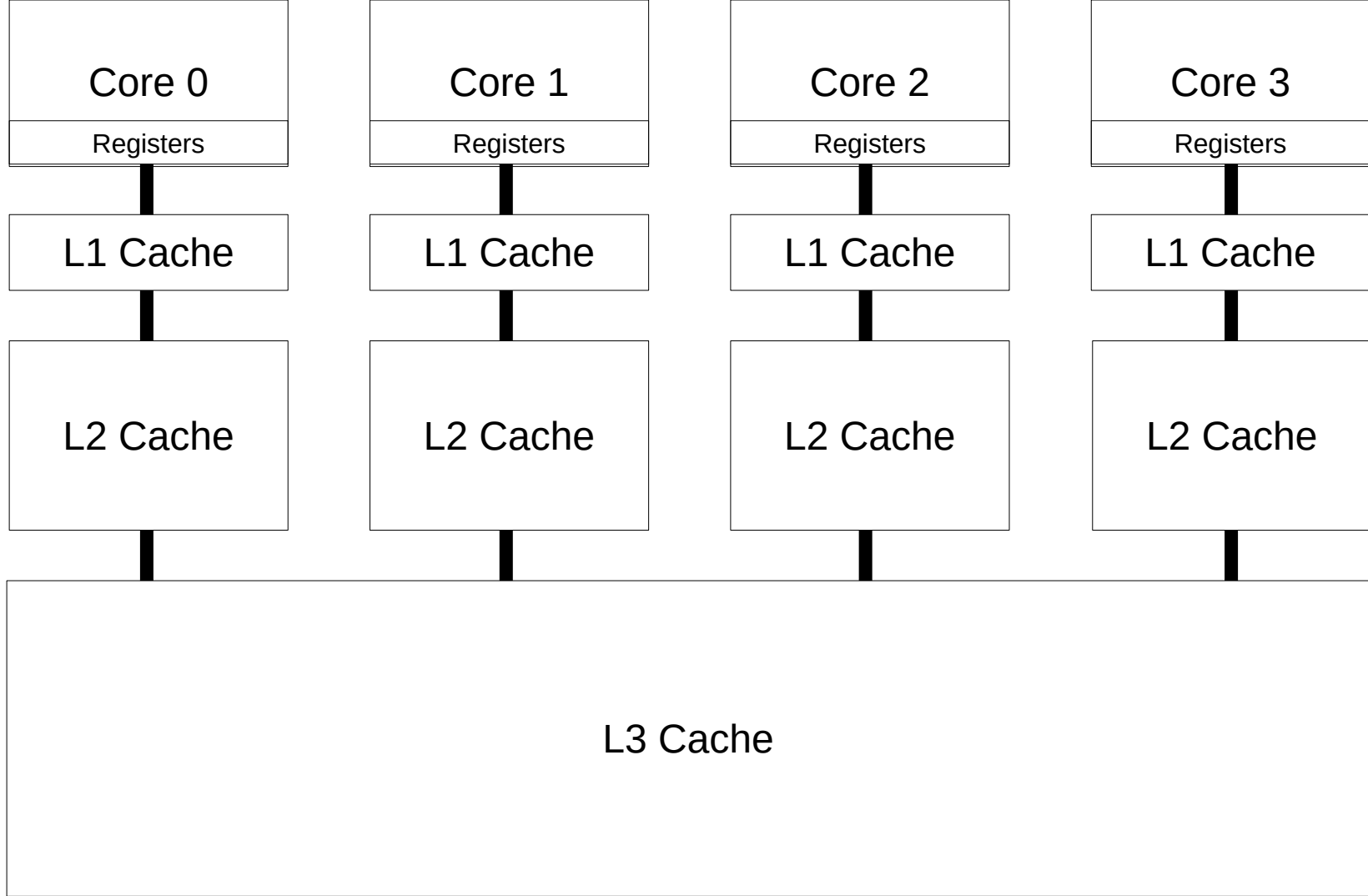
CMPXCHG

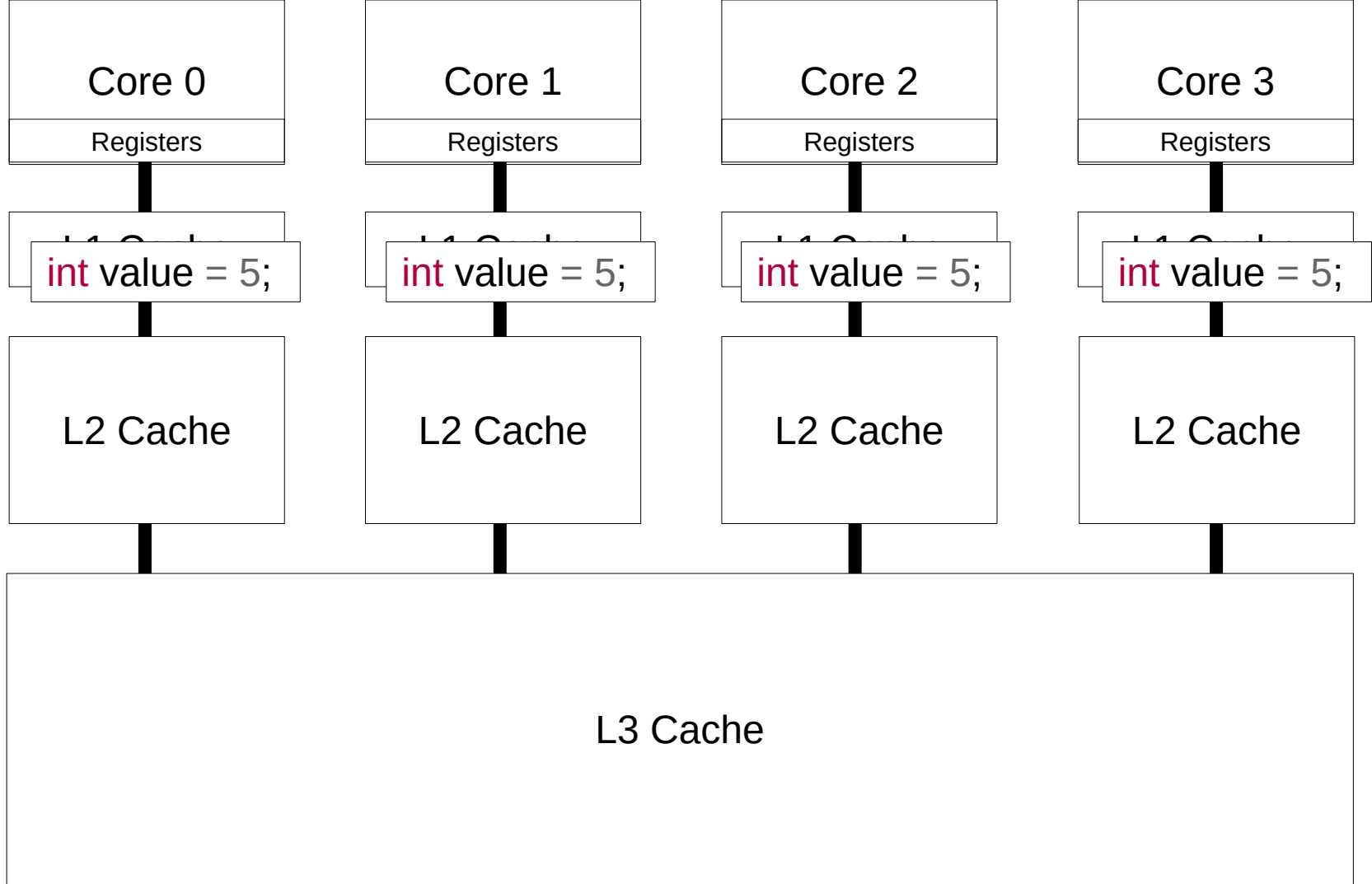
CMPXCHG

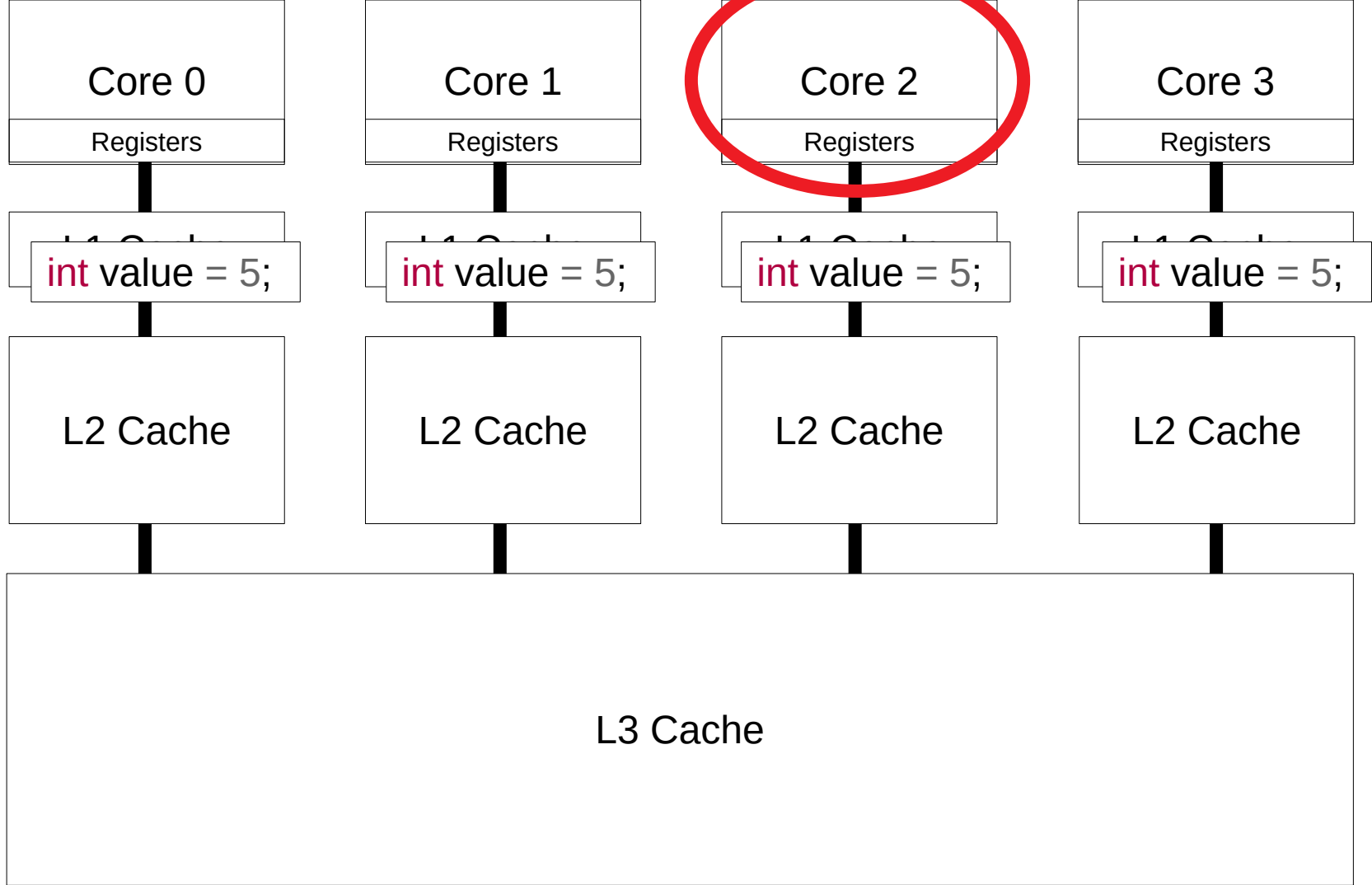
```
int compareAndSwap(int* location, int old, int new)
{
    int currentValue = *location;
    if (currentValue == old)
        *location = new;
    return currentValue;
}
```

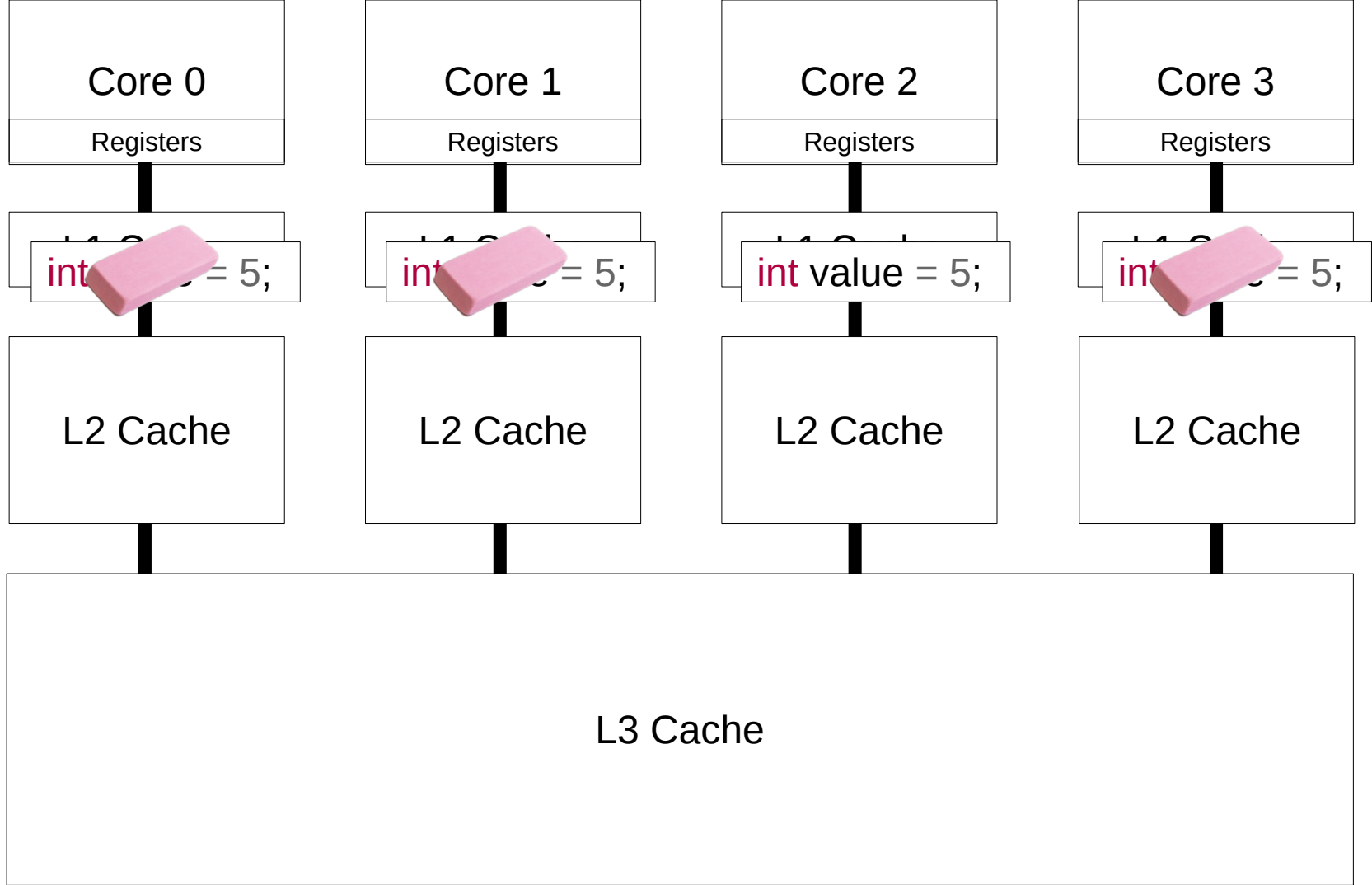


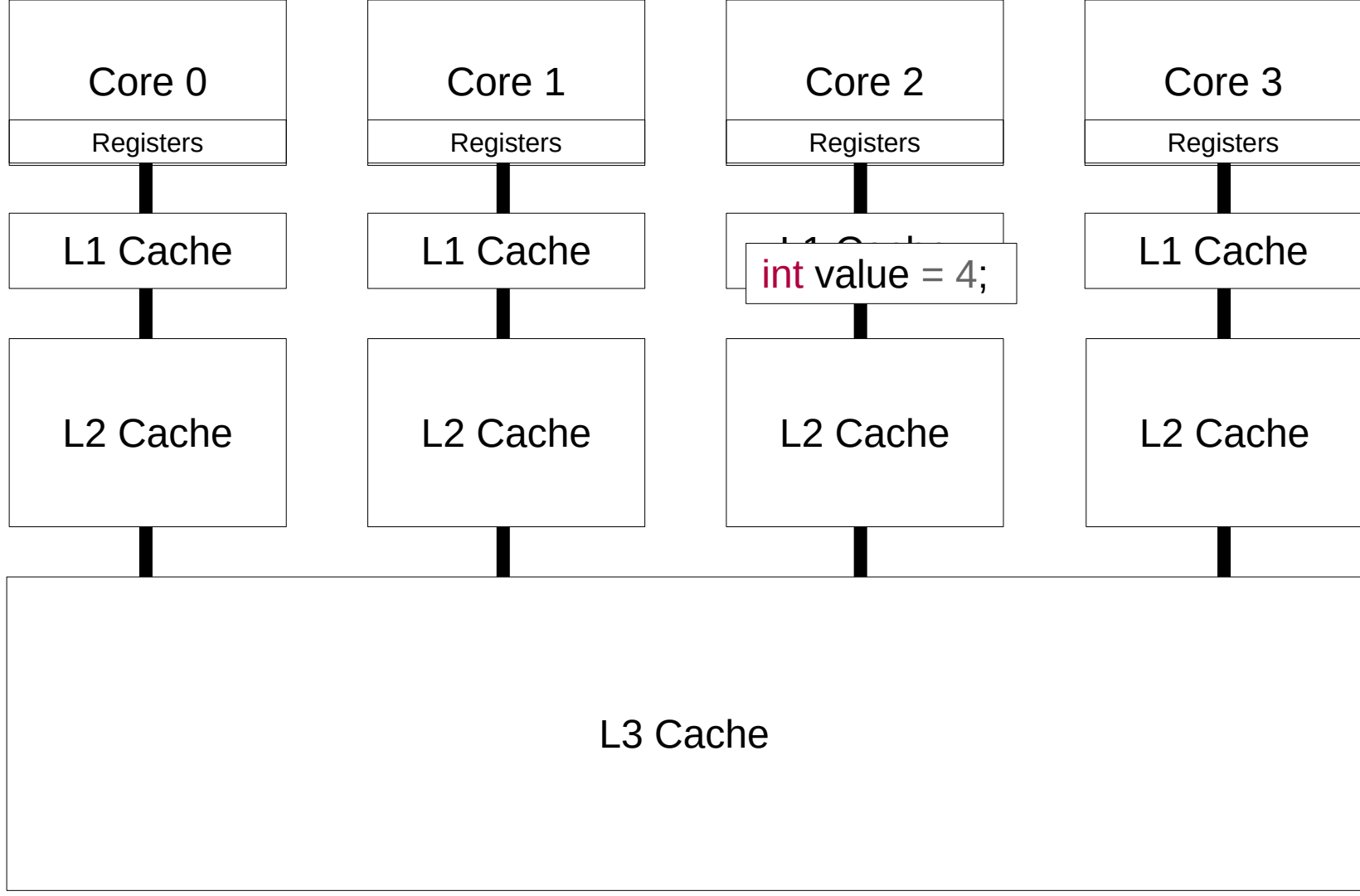
Atomic Operations

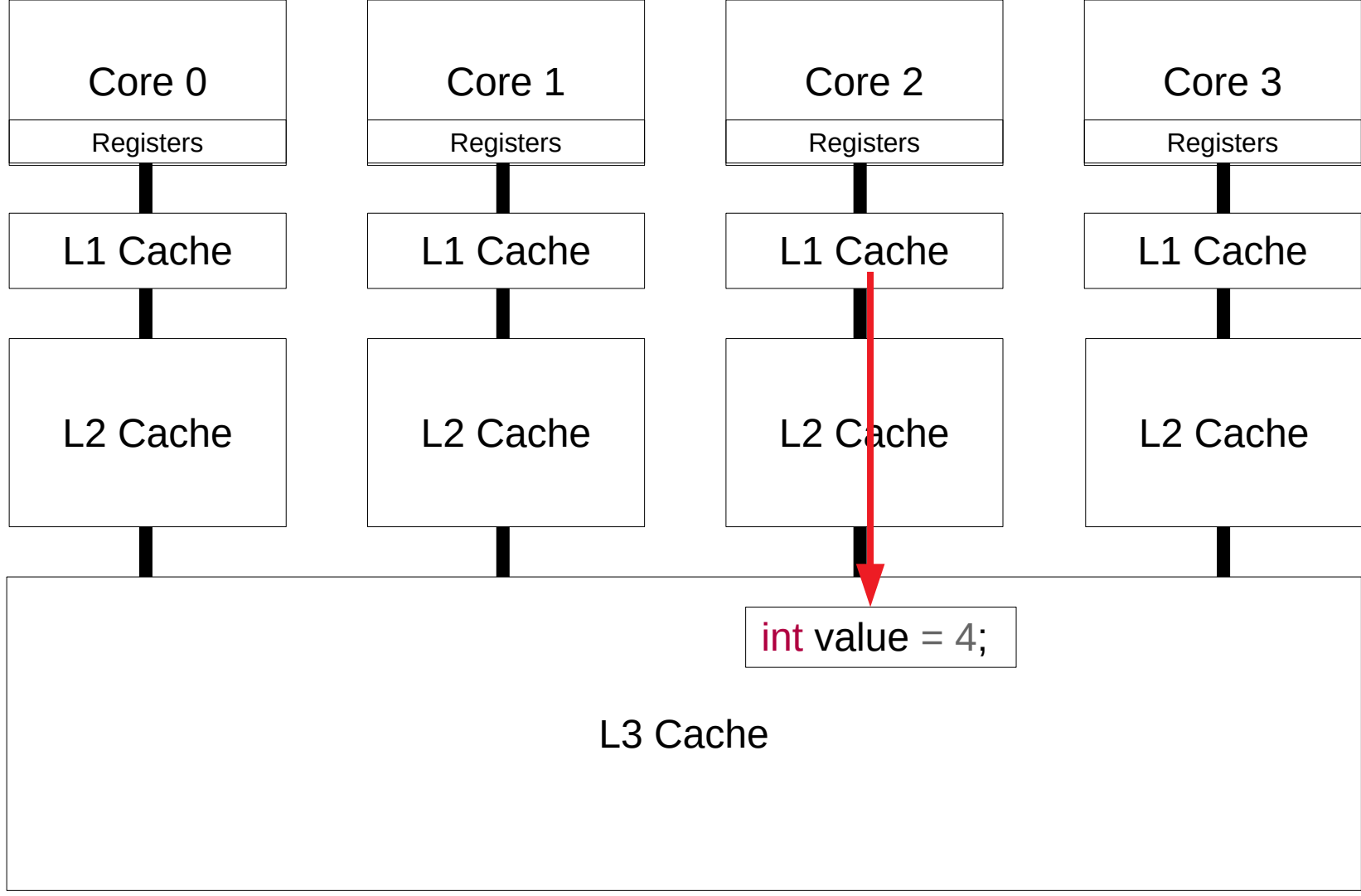


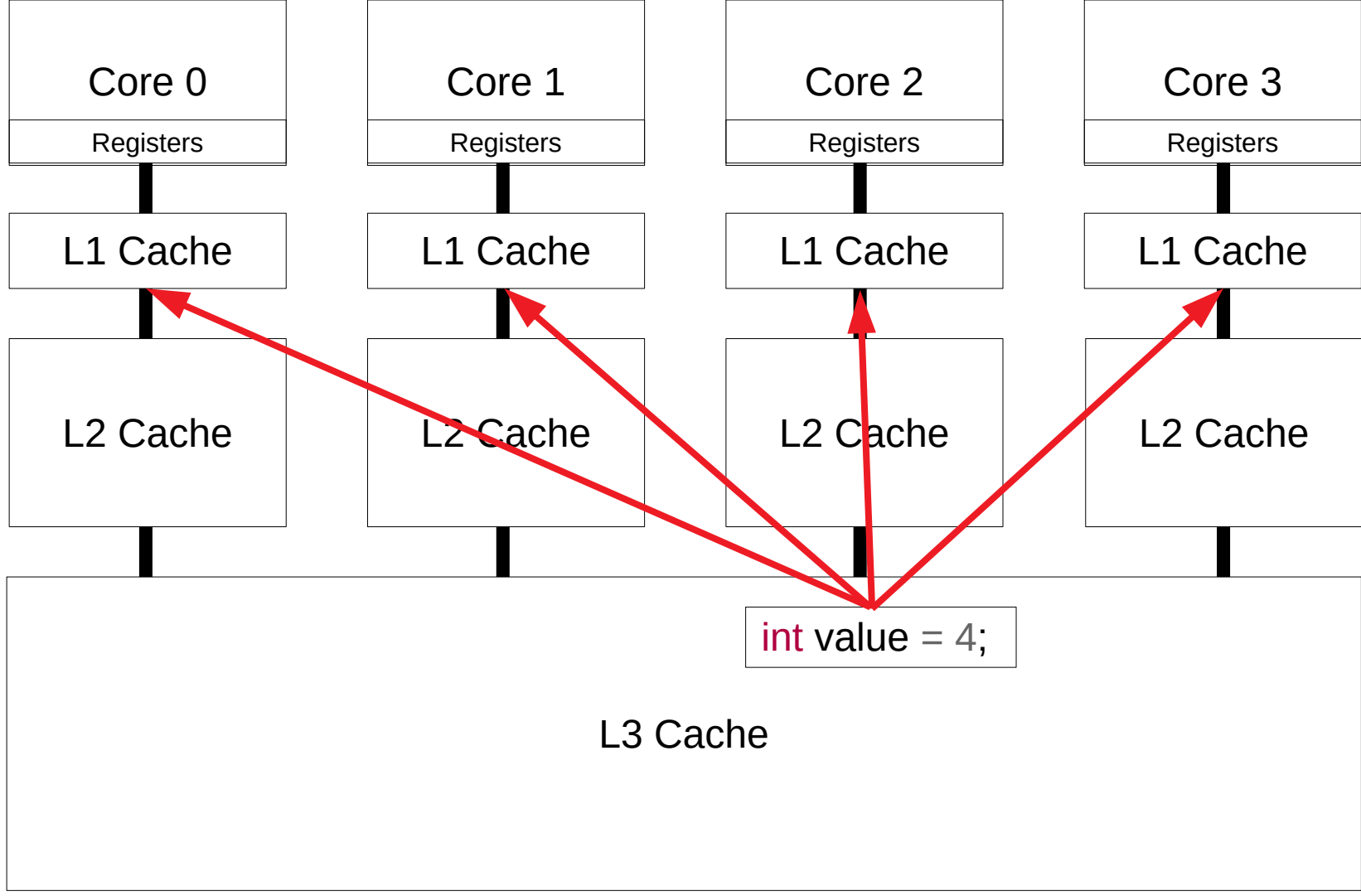












Involves complex interaction between cores.

Result: Expensive instructions

But: No race conditions! (if used properly)

Available atomic operations (x86)

Arithmetic:

ADD, ADC, DEC, INC, NEG, SUB, SBB

Bitwise:

AND, BTC, BTR, BTS, NOT, OR, XOR

Exchange:

CMPXCHG, CMPXCH8B, XADD

In practice:

Mutexes in C++

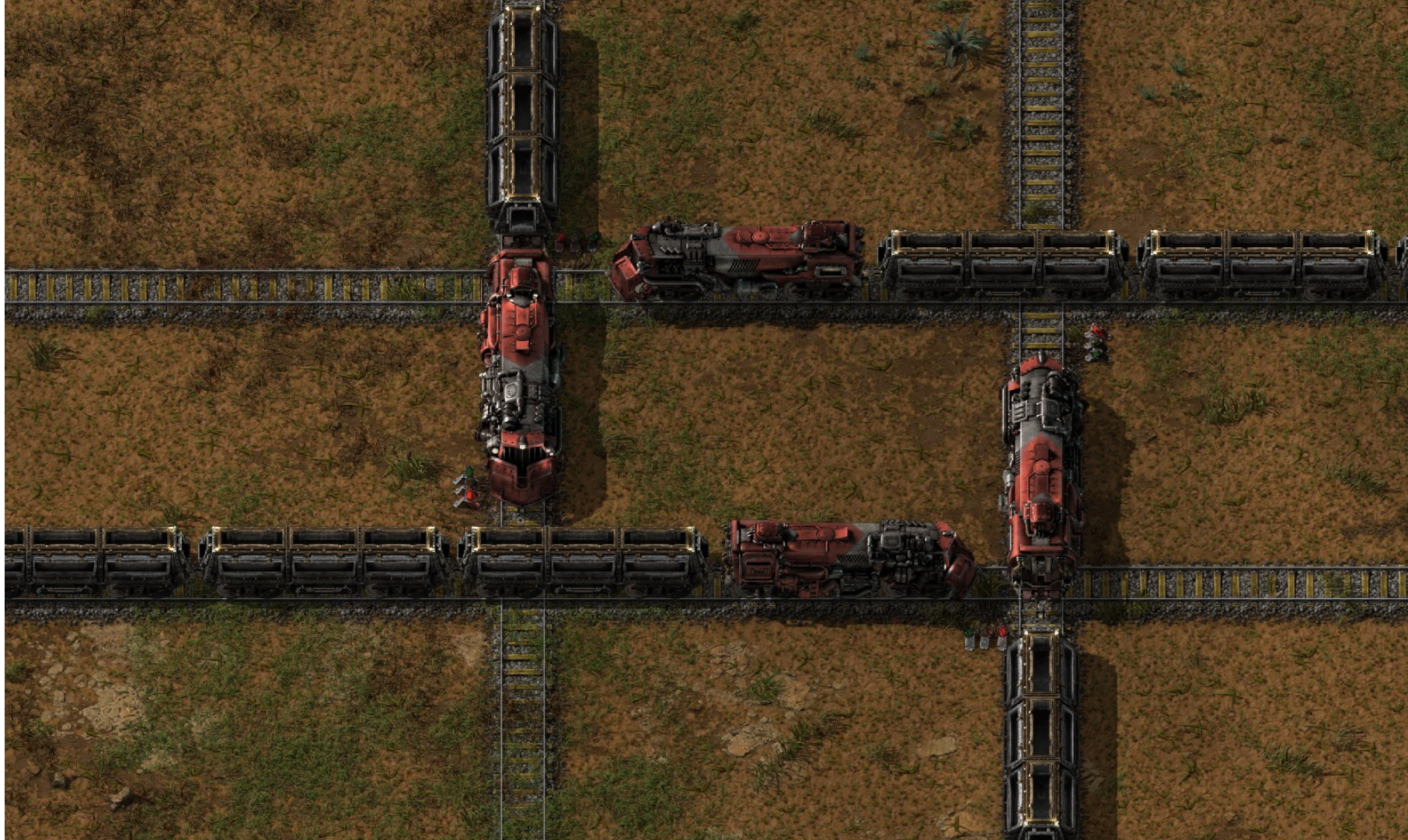

```
#include <thread>
#include <iostream>
#include <mutex>
#include <atomic>

void increment(std::mutex* lock,
               std::atomic<int>* atomicInt, int* value) {
    // Alternative 1: mutex with normal int
    lock->lock();
    (*value)++;
    lock->unlock();

    // Alternative 2: atomic increment
    (*atomicInt)++;
}
```

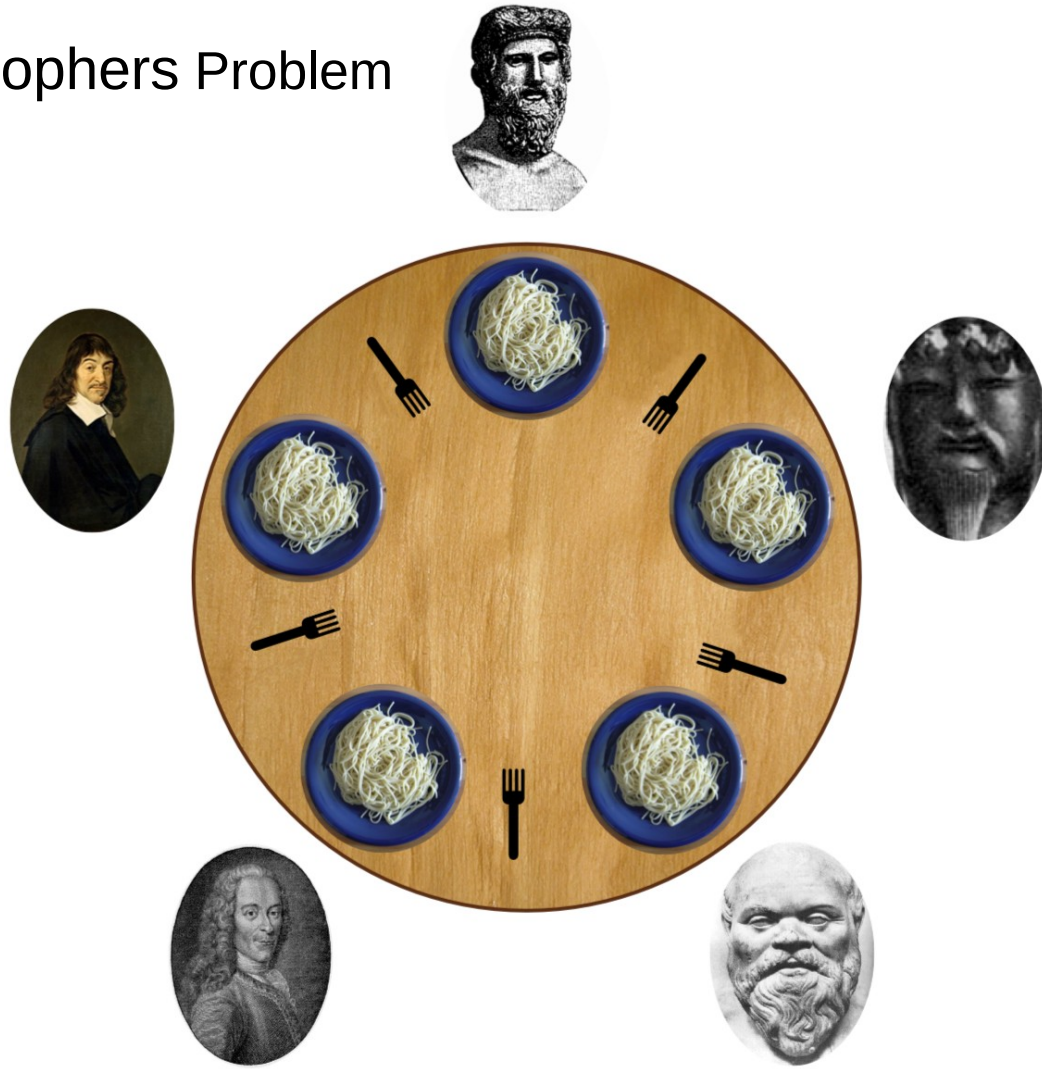
Locks and Atomics are awesome!

All our problems are solved!

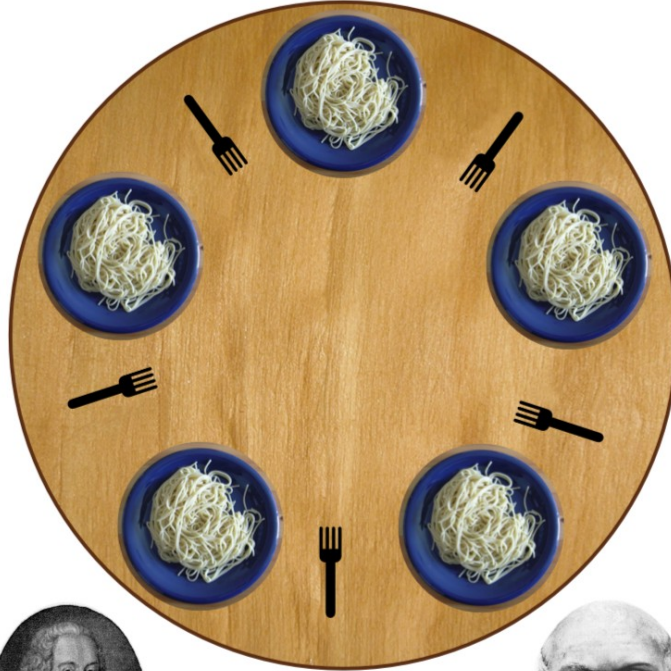


Deadlocks

The Dining Philosophers Problem



The Dining Philosophers Problem



Rules:

- Only allowed to eat with two forks
- Can only put down a fork *after* eating

Locking does not necessarily mean
no race conditions exist!

```
#include <thread>
#include <iostream>
#include <mutex>

void increment(int* value) {
    (*value)++;
}

int main() {
    std::mutex lock;
    int value = 0;

    std::thread thread0(increment, &value);

    lock.lock();
    value++;
    lock.unlock();

    thread0.join();

    std::cout << value << std::endl;
    return 0;
}
```

Finally:

Thread Architecture and Best Practices

Critical Section:

A block of code that updates a shared resource which should only be accessed by one thread at a time

Run in serial: do as little as possible in them

Minimise resources shared between threads

Atomic operations are
generally faster than mutexes

Summary:

**Threads can be interrupted
at *any* time in *any* order.**

Use locks and atomics to avoid race conditions.

If you need to use locks,
minimise the time they are locked.

Crash Course Parallel Computing

- Using `std::thread`
- **Using OpenMP**

OpenMP - Setup

- MacOS: You may need to run 'brew install libomp'
- In meson.build:

```
26
27 src = []
28
29 omp = dependency('openmp')
30
31 exe = executable(
32     'program',
33     src,
34     'main.cpp',
35     dependencies : [animationwindow_dep, sdl2_dep, std_lib_facilities_dep, omp],
36     cpp_args : compiler_flags
37 )
```

<http://spritesmods.com/?art=hddhack&page=2>
<https://www.briandorey.com/docs/2020-01-15-sony-55xe9005-teardown/pcb-main.jpg> <https://www.rcgeeks.co.uk/blogs/news/dji-mavic-mini-teardown-whats-inside>
<https://www.komplett.no/img/p/800/1219388.jpg>
<https://www.ebay.com/itm/402730890541>
<https://www.ifixit.com/Teardown/GoPro+HERO11+Black+Mini+Teardown/155069>
<https://news.satnews.com/2020/08/04/xiphos-reveals-their-new-space-processor-board-for-sdr-applications/>
<https://www.hpc.ntnu.no/idun/>
https://cdn.benchmark.pl/uploads/backend_img/c/newsy/2020-09/PM/nvidia-rtx-3080_05.jpg
<https://pbs.twimg.com/media/ELcxWo0U8AAAsiR.jpg>
http://www.chip-architect.org/news/Northwood_130nm_die_text_1600x1200.jpg
<https://www.zeiss.com/spectroscopy/applications-industries/oem-applications/semiconductor.html>
<https://wp.technologyreview.com/wp-content/uploads/2018/08/googlecbf009-11.jpg>
<https://tpucdn.com/cpu-specs/images/chips/2817-die-shot.jpg>
<https://www.techrepublic.com/wp-content/uploads/2011/11/22inteldieshot1997.jpg>
http://brainstones.narod.ru/collection/intel/intel_pentium_d_925_sl9ka_wo_lid.jpg