TDT4200 Parallel Programming

Bart van Blokland

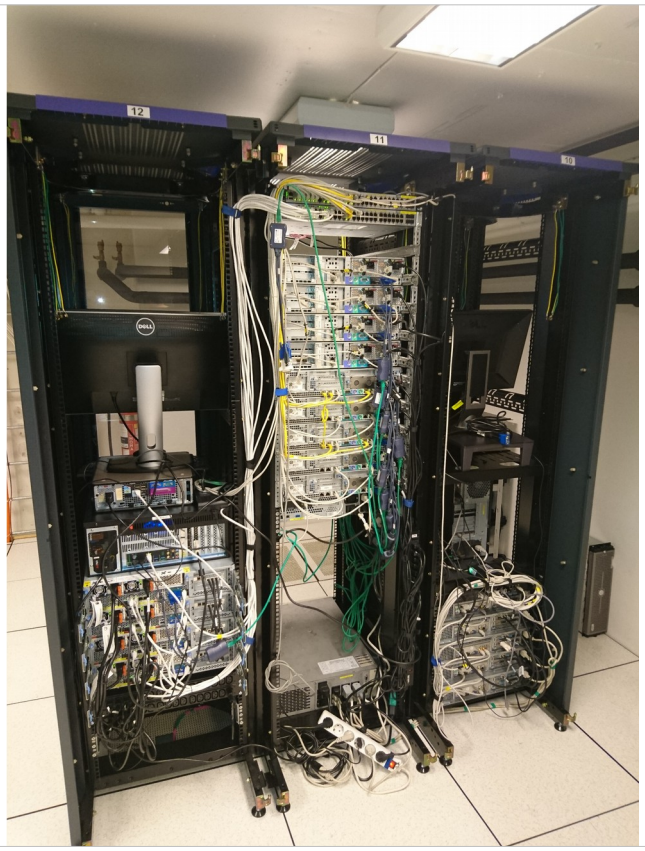
Lecture 6

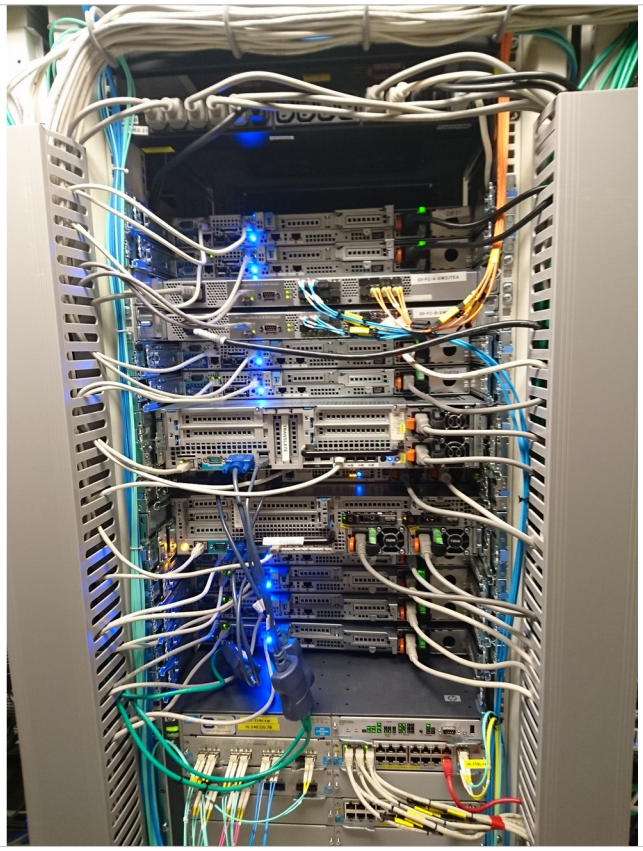
Image from:
https://i.kym-cdn.com/entries/icons/original/000/002/361/maxresdefault.jpg

Last two weeks:

MPI

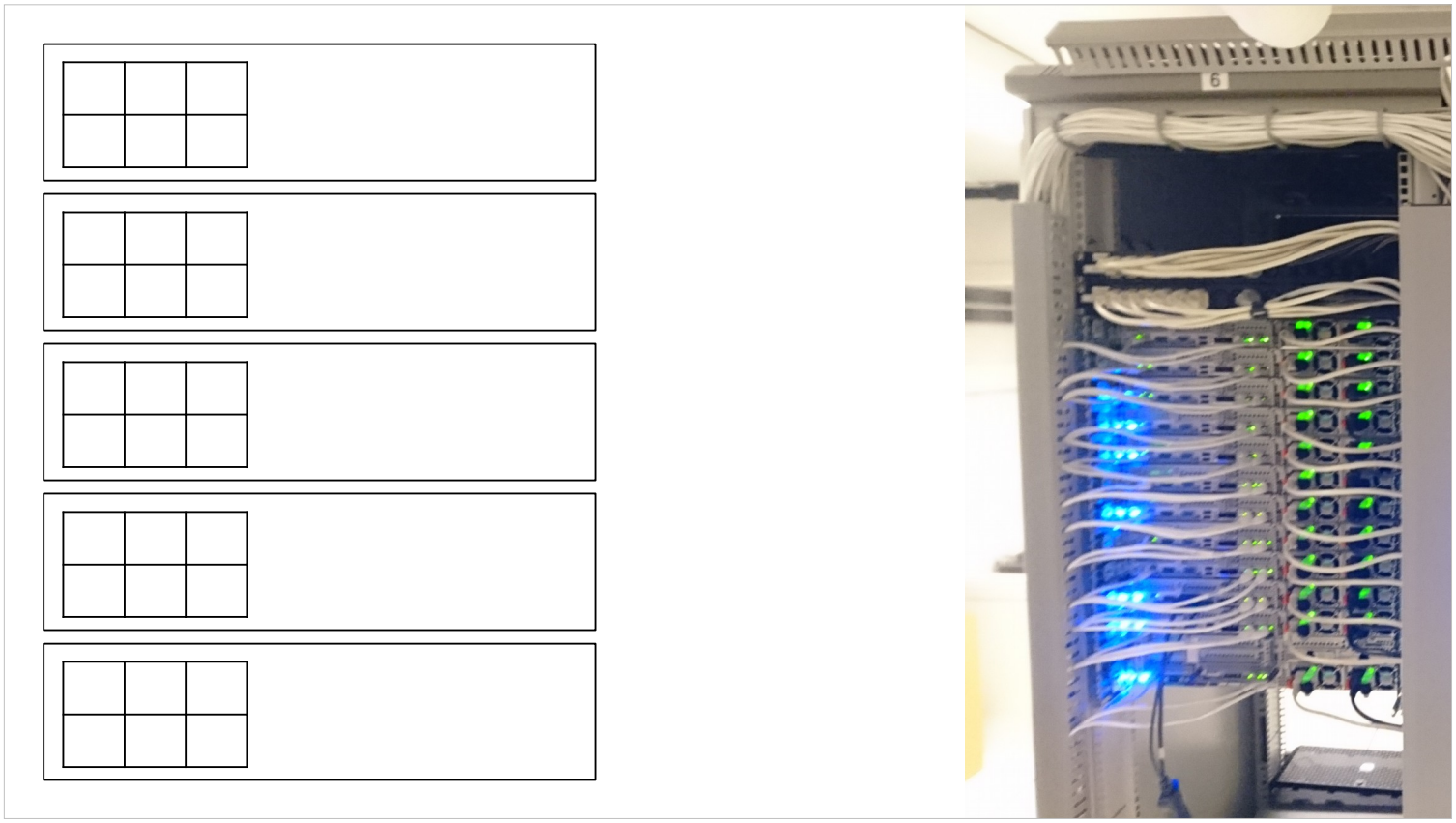
These are some of the server racks we have in the basement of the IDI building. This is a typical environment in which you would use MPI.



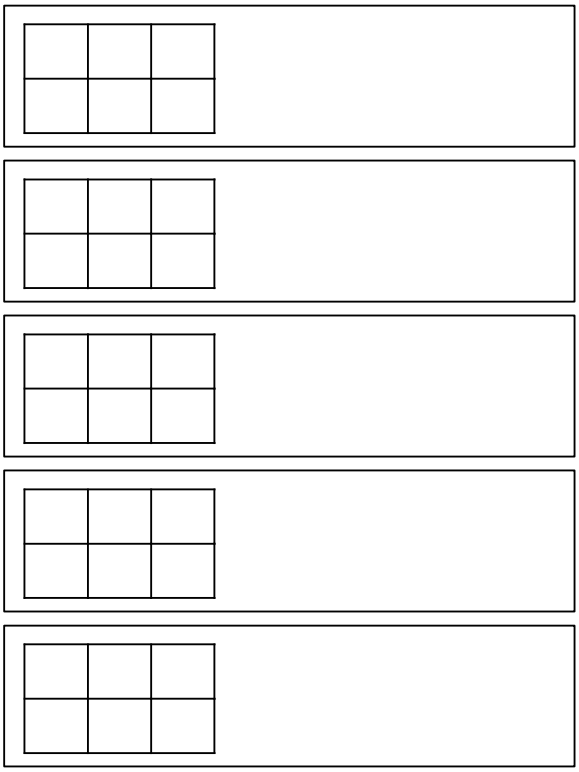


More pictures. And cables!

Here’s a good example from the rack in the back. There are a number of equivalent servers stacked on top of each other.

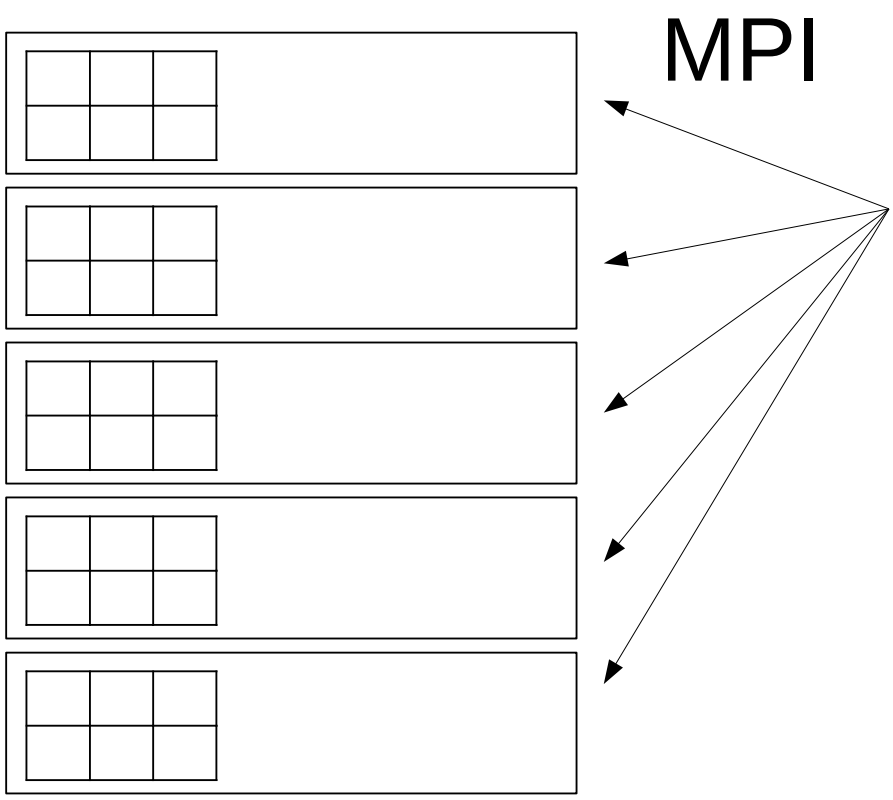


On the left a rough schematic version of the picture on the right.

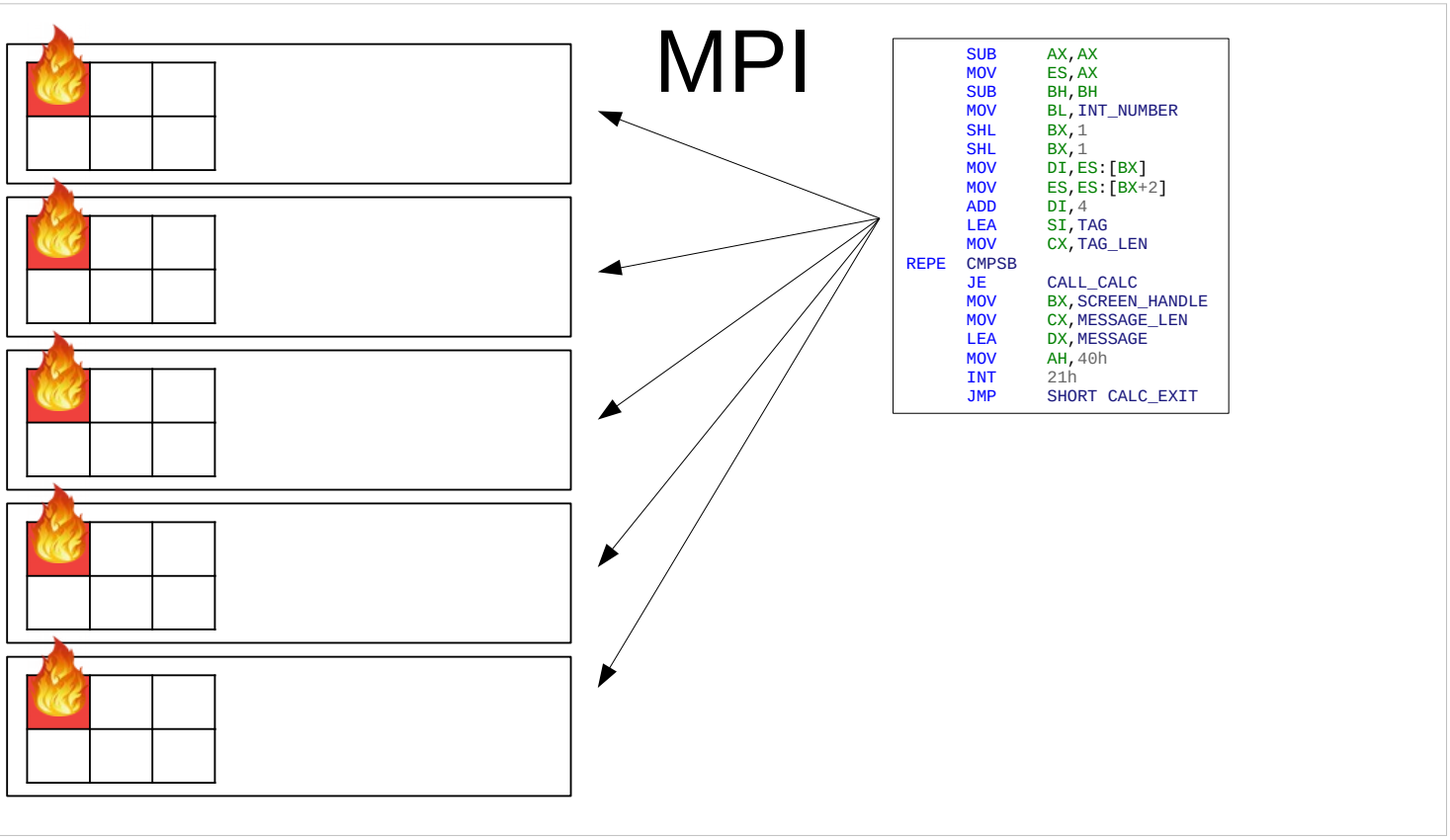


```
        SUB     AX,AX
        MOV     ES,AX
        SUB     BH,BH
        MOV     BL,INT_NUMBER
        SHL     BX,1
        SHL     BX,1
        MOV     DI,ES:[BX]
        MOV     ES,ES:[BX+2]
        ADD     DI,4
        LEA     SI,TAG
        MOV     CX,TAG_LEN
REPE    CMPSB
        JE      CALL_CALC
        MOV     BX,SCREEN_HANDLE
        MOV     CX,MESSAGE_LEN
        LEA     DX,MESSAGE
        MOV     AH,40h
        INT     21h
        JMP     SHORT CALC_EXIT
```

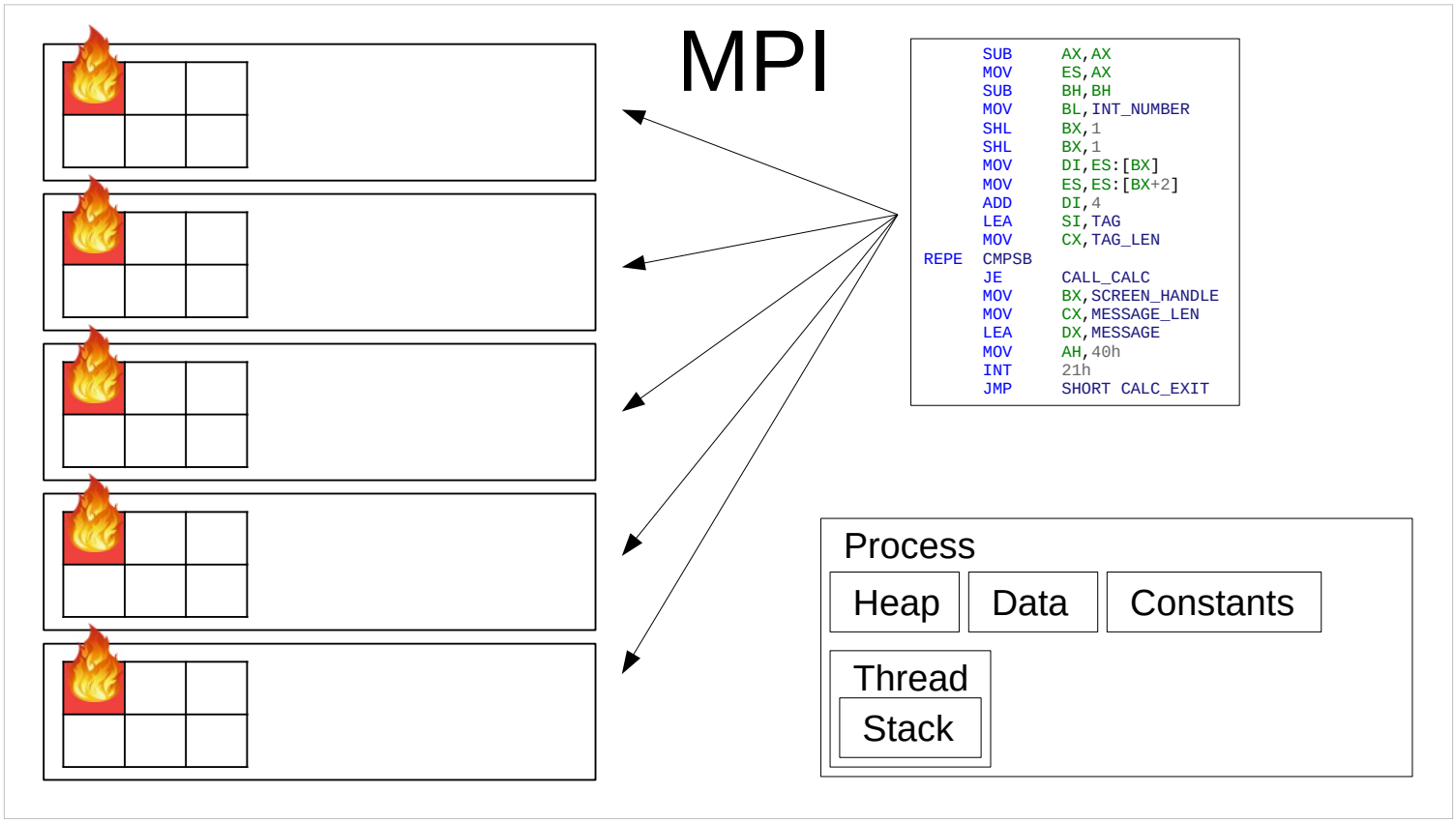
Let’s run a program on them!



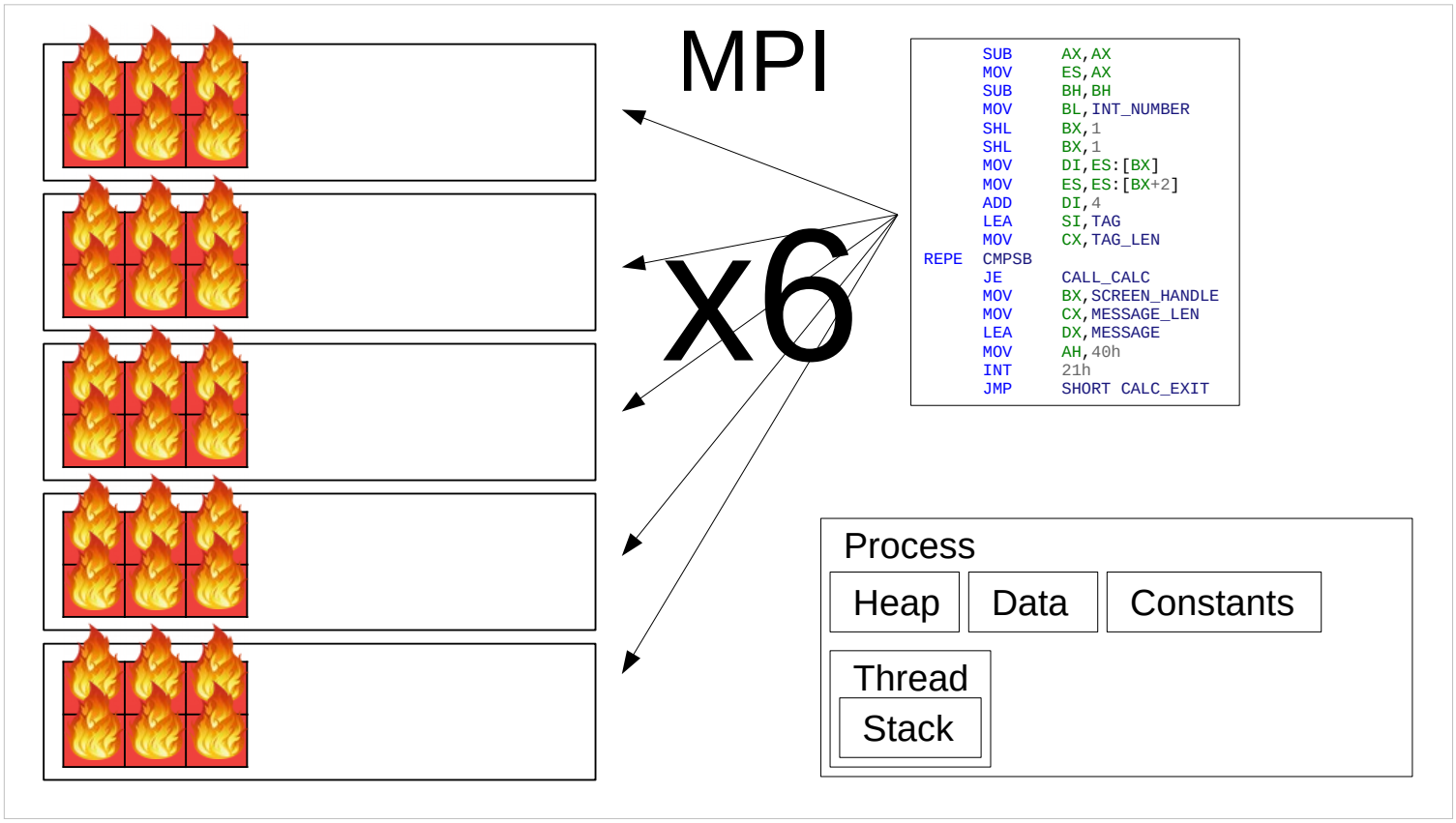
We can use MPI for this!



Hm. Weird. Only one core appears to be doing something when we run one instance of our program on each node.



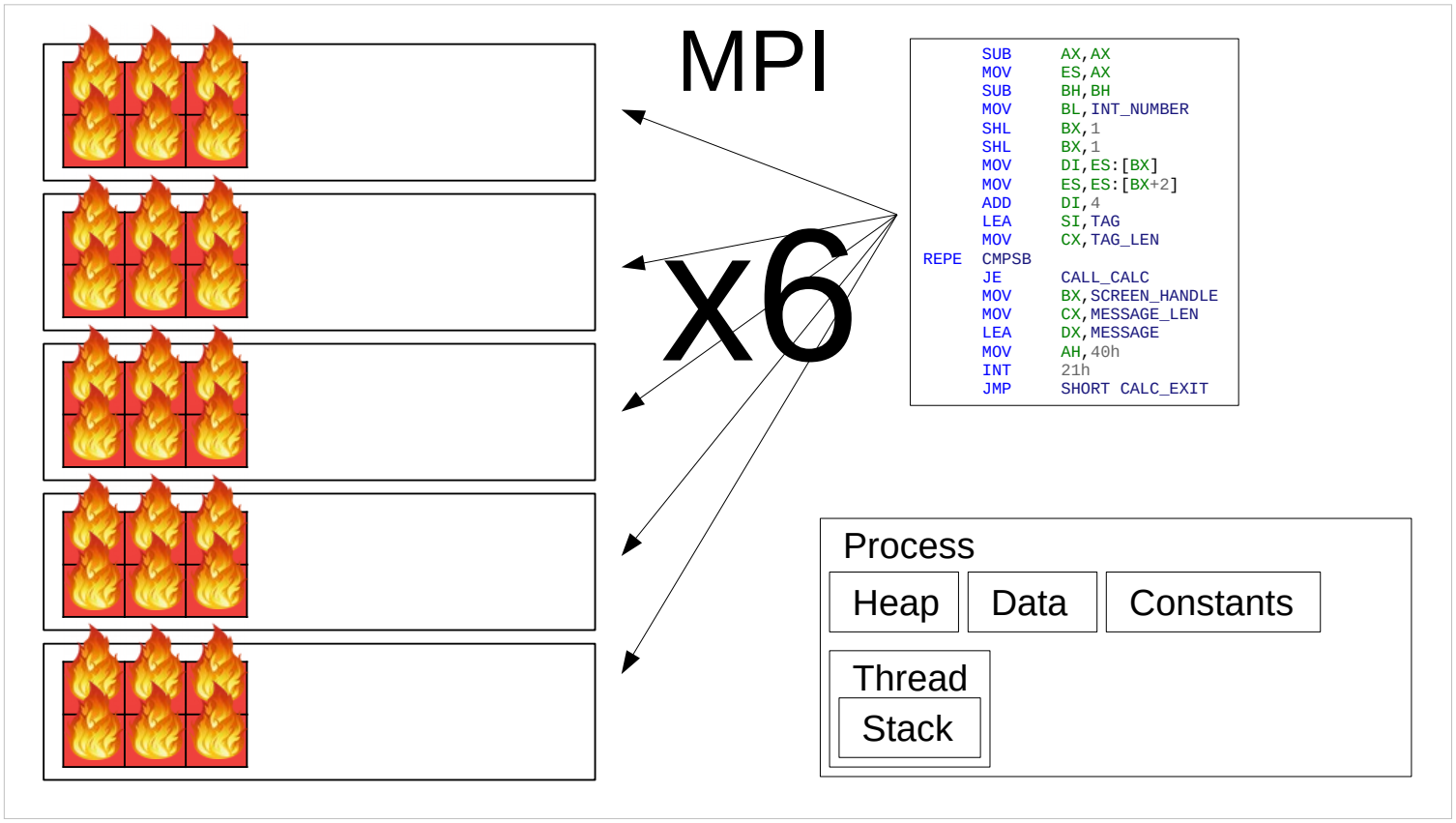
Problem: each process only has one thread.

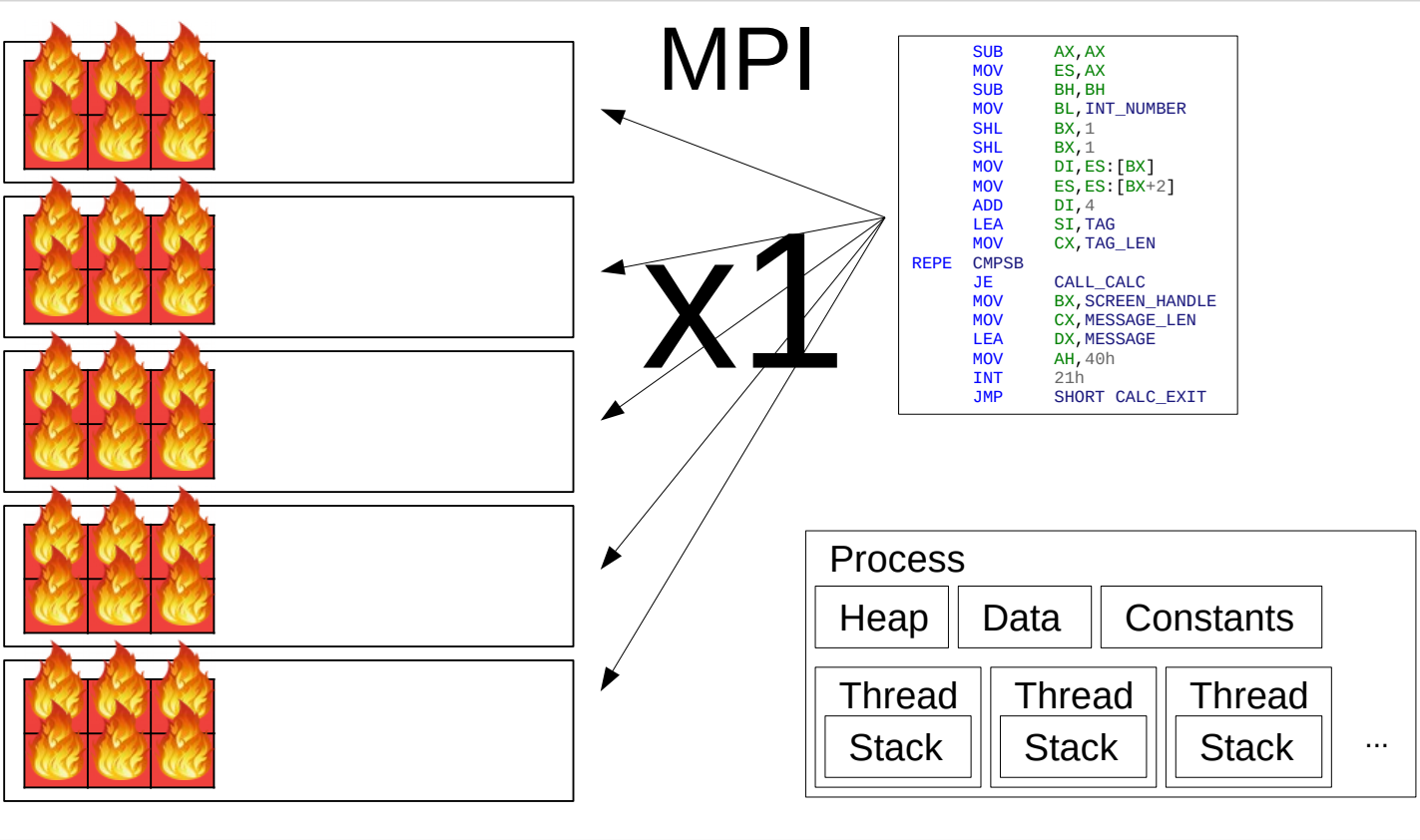


One solution is to just spawn processes equal to the number of cores present on each node. However, this usually results in very large overheads. Both due to interprocess communication being expensive, as well as the need for memory duplication (in turn making the cache less efficient because it needs to hold memory from all processes separately).

May duplicate memory

Communication between processes is expensive


MPI
x6
SUB AX,AX
MOV ES,AX
SUB BH,BH
MOV BL,INT_NUMBER
SHL BX,1
SHL BX,1
MOV DI,ES:[BX]
MOV ES,ES:[BX+2]
ADD DI,4
LEA SI,TAG
MOV CX,TAG_LEN
REPE CMPSB
JE CALL_CALC
MOV BX,SCREEN_HANDLE
MOV CX,MESSAGE_LEN
LEA DX,MESSAGE
MOV AH,40h
INT 21h
JMP SHORT CALC_EXIT
Process
Heap
Data
Constants
Thread
Stack



The far better solution is to spawn one process per node, then creating multiple threads inside of them. The operating system's scheduler tends to allocate these to different processing cores.

Threads:

+ More lightweight than processes

+ Make better use of available cores

The result is a system that makes better use of resources, but comes at a significant cost: having to share memory between other threads. If not properly managed, this can introduce problems which can only appear sporadically. This makes them hard to diagnose, let alone solve.

Threads:

+ More lightweight than processes

+ Make better use of available cores

- Cause their own set of problems

Thread Lightly

So the topic for this lecture and the next one is managing threads on the CPU. We'll particularly take care to look at the issues you may encounter, and how to counteract or avoid them.

Threading issues? Oh dear..

Procrastination time!

std::thread

We'll be using std::thread in this course. It's a part of the C++ standard library that allows you to create threads and manage their interactions.

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

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

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

This is all the code you need to create a thread.

```
#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;
}
```

You can also give it parameters.

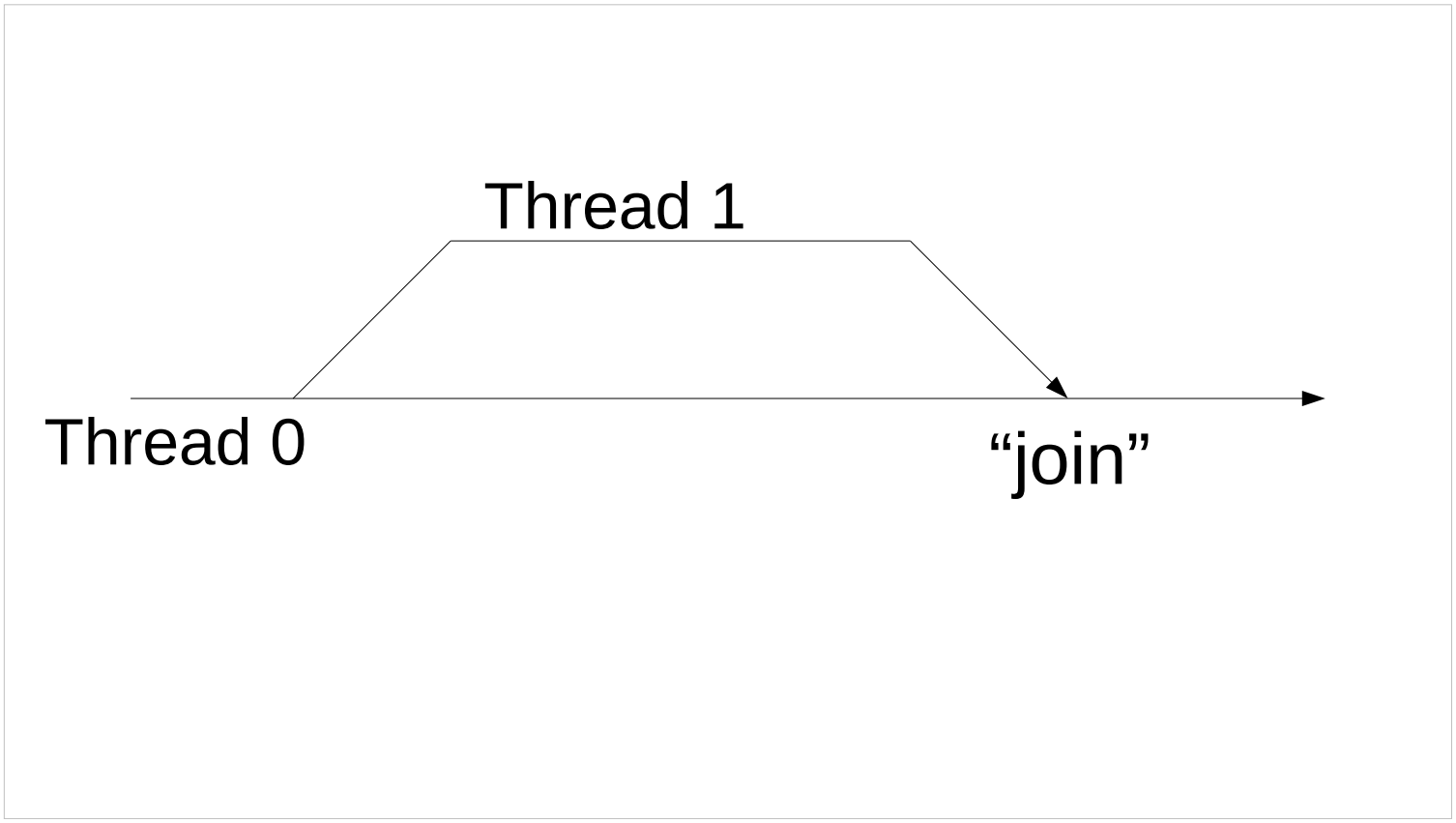
```
g++ -pthread threads.cpp -o threads
```

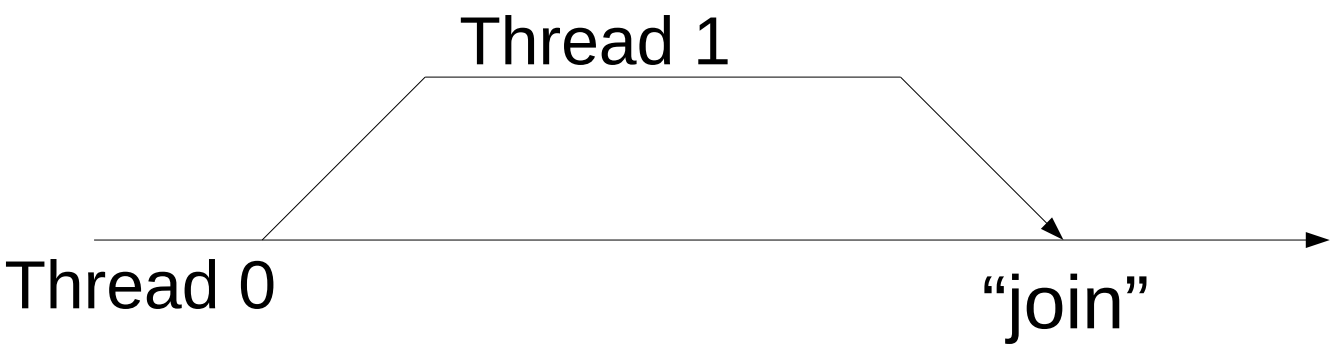
This is how you compile this program on linux. Note that std::thread depends on pthreads on UNIX systems, which requires a special flag to enable them.

```cpp
#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;
}
```

```
#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 a joins thread b means a waits until b is completed.


Thread 1
Thread 0
“join”



Thread 0 “joins” Thread 1

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

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

void doSomething(int* value) {
    for(int i = 0; i < 100000; i++) {
        (*value)++;
    }
}

int main() {
    int* sum = new int;
    *sum = 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;
}
```

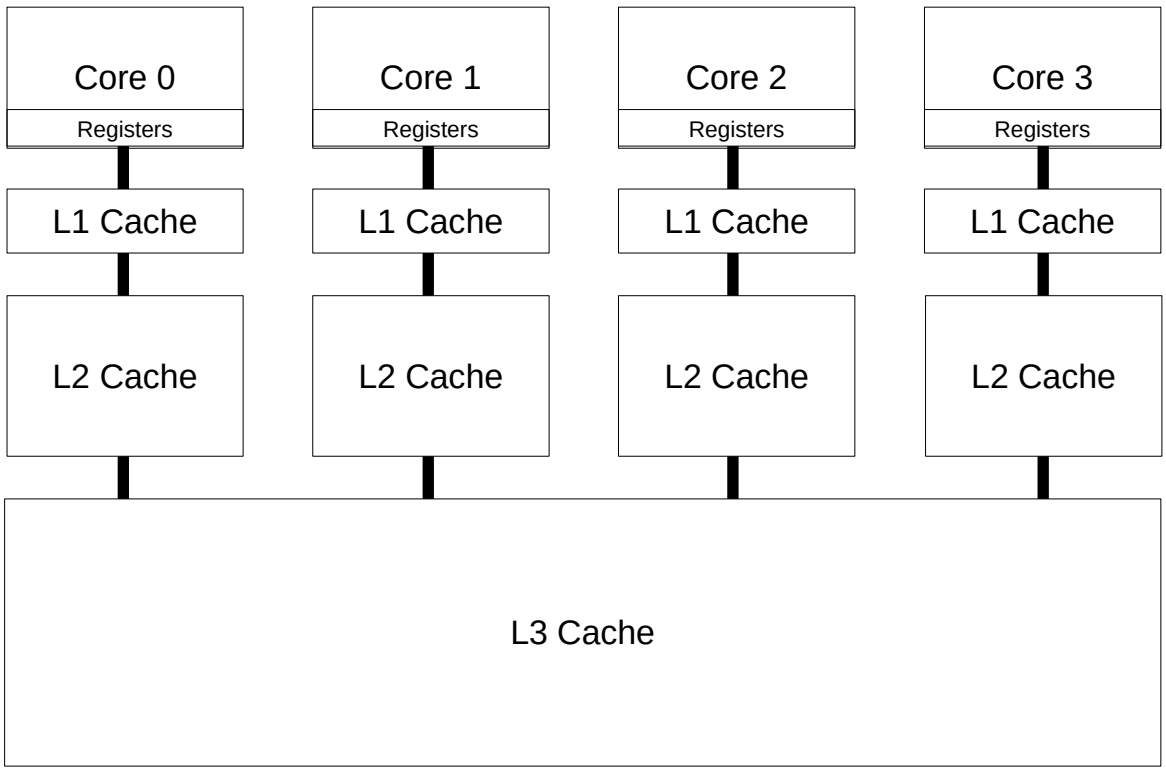
← What is the value of sum?

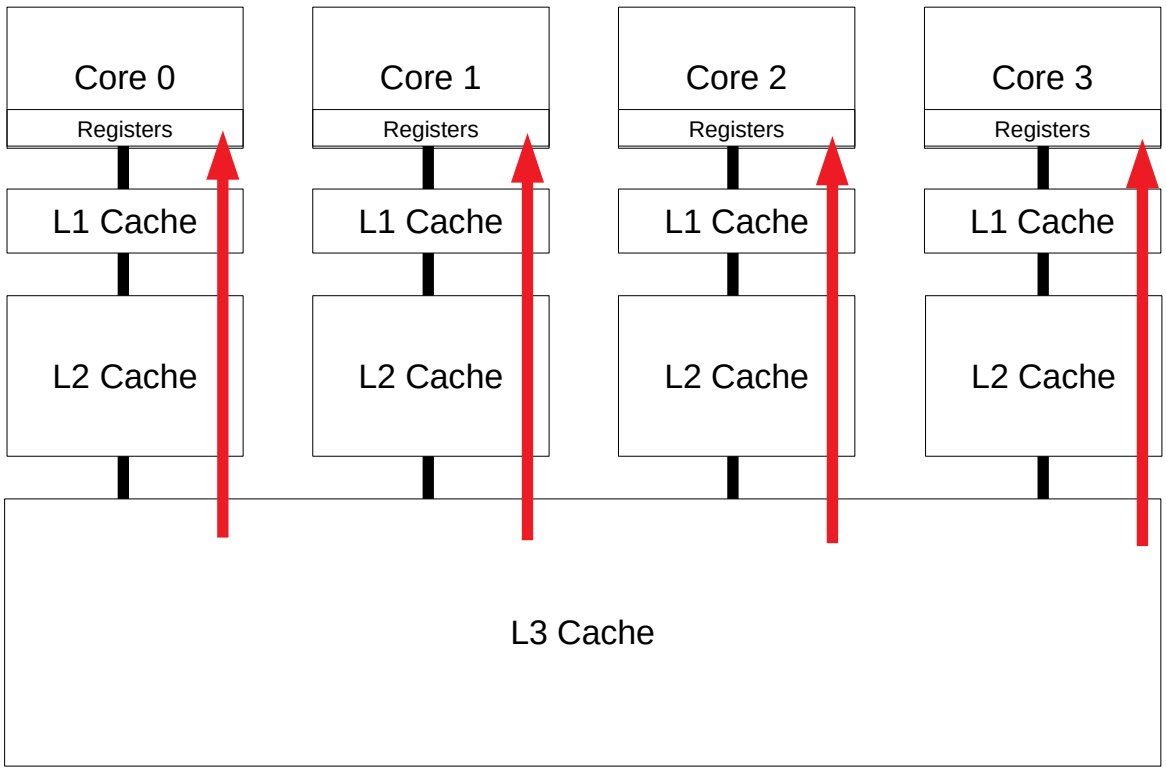
Conceptually, this code should print 1000000

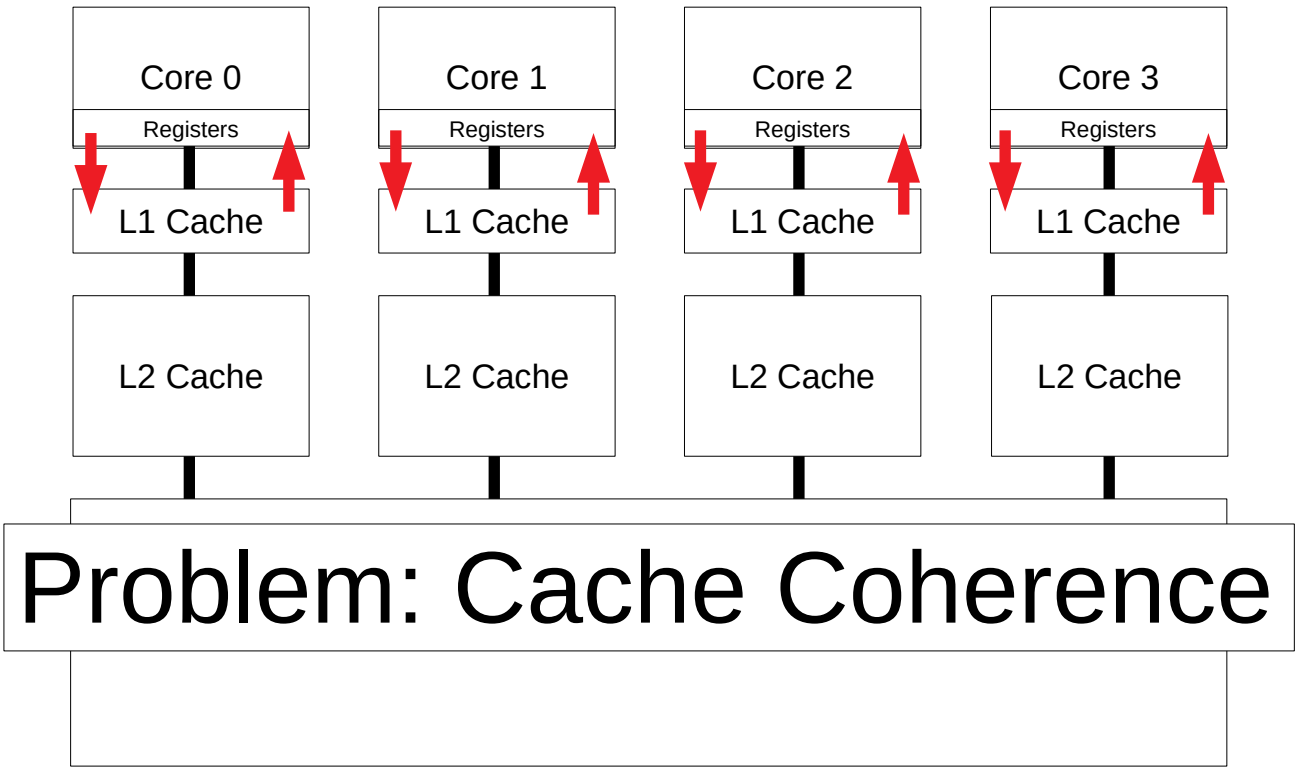
131571
116638
113056
121983
170546
124298
127607
170724

But it doesn’t

Cache


Core 0
Registers
L1 Cache
L2 Cache
Core 1
Registers
L1 Cache
L2 Cache
Core 2
Registers
L1 Cache
L2 Cache
Core 3
Registers
L1 Cache
L2 Cache
L3 Cache


Core 0
Registers
L1 Cache
L2 Cache
Core 1
Registers
L1 Cache
L2 Cache
Core 2
Registers
L1 Cache
L2 Cache
Core 3
Registers
L1 Cache
L2 Cache
L3 Cache



The problem is that lower level caches do not syncrhonise their changes very often. This means that if a value is modified by a core, the other threads do not hear about it. So the issue you saw previously is caused by threads working on their “own local copy”, and not updating the value stored in L3 cache. Once it is time to write it back, it just overwrites whatever was there.

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
- CPU Interrupts
- Variations in accessing RAM memory banks

And more..

This is by no means an exhaustive list. There are many other reasons.

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;
```

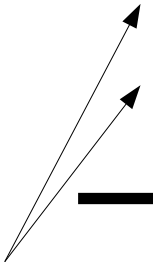
```
// thread 0:

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;
}
```

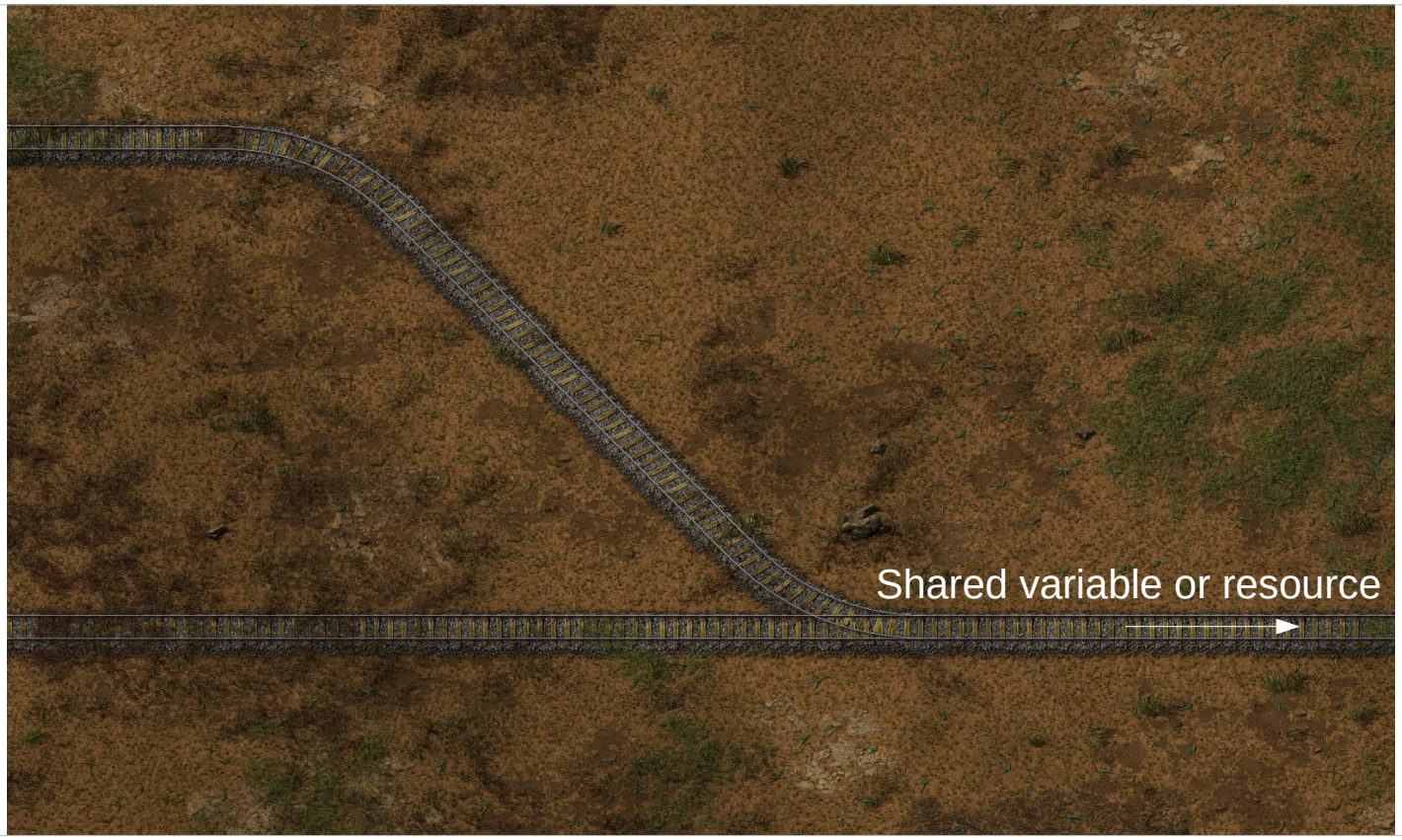


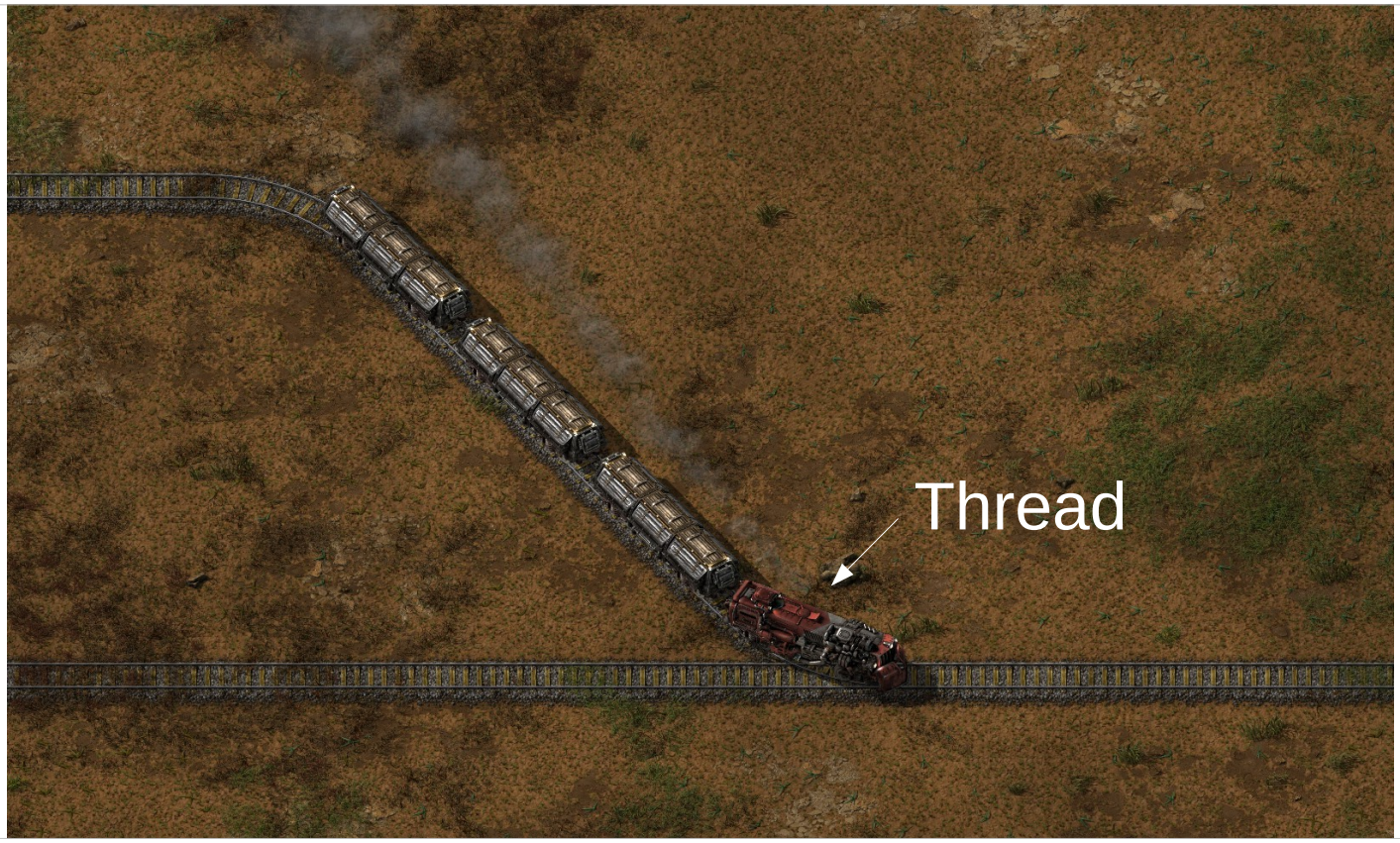
Consecutive lines!

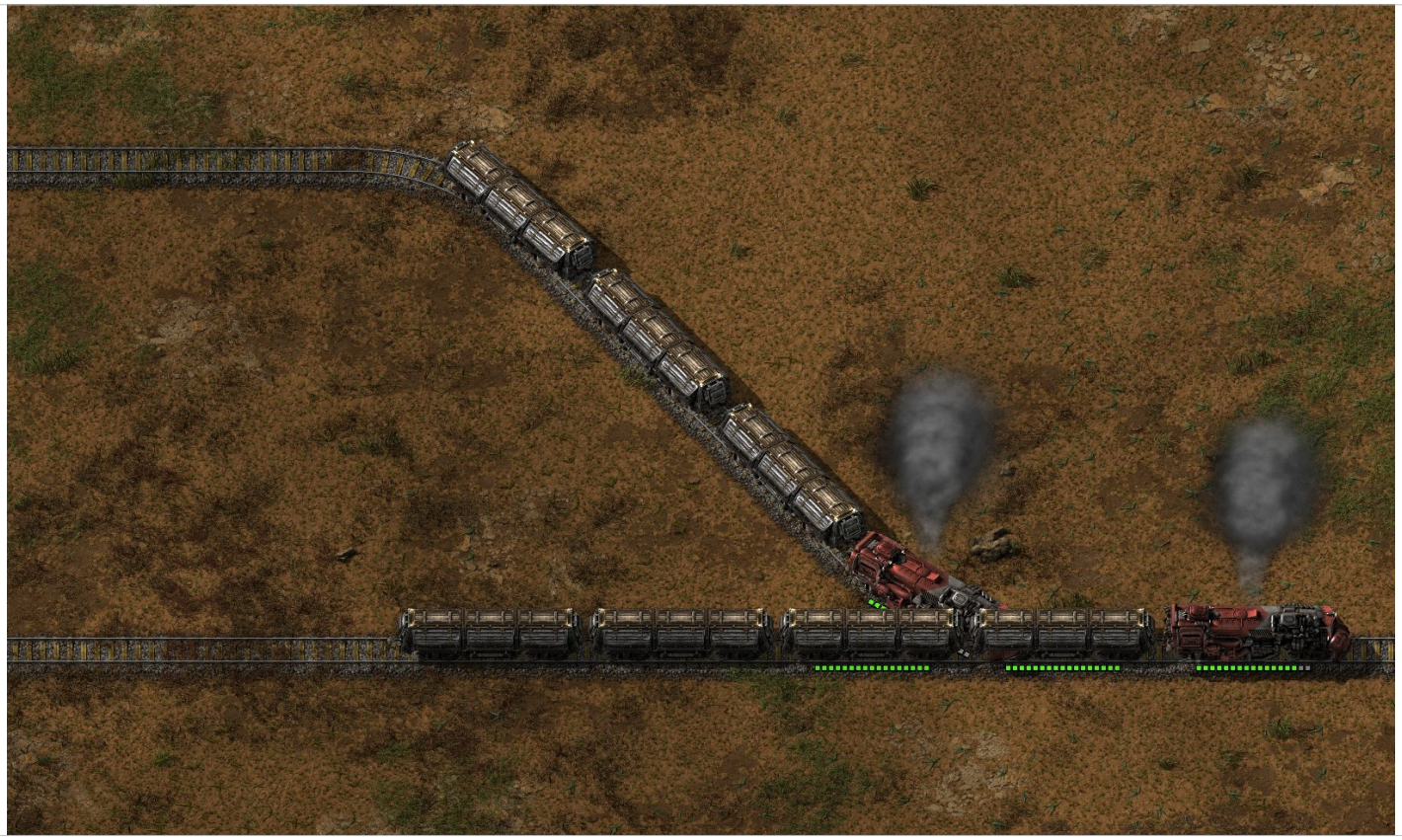
So how do we solve race conditions?

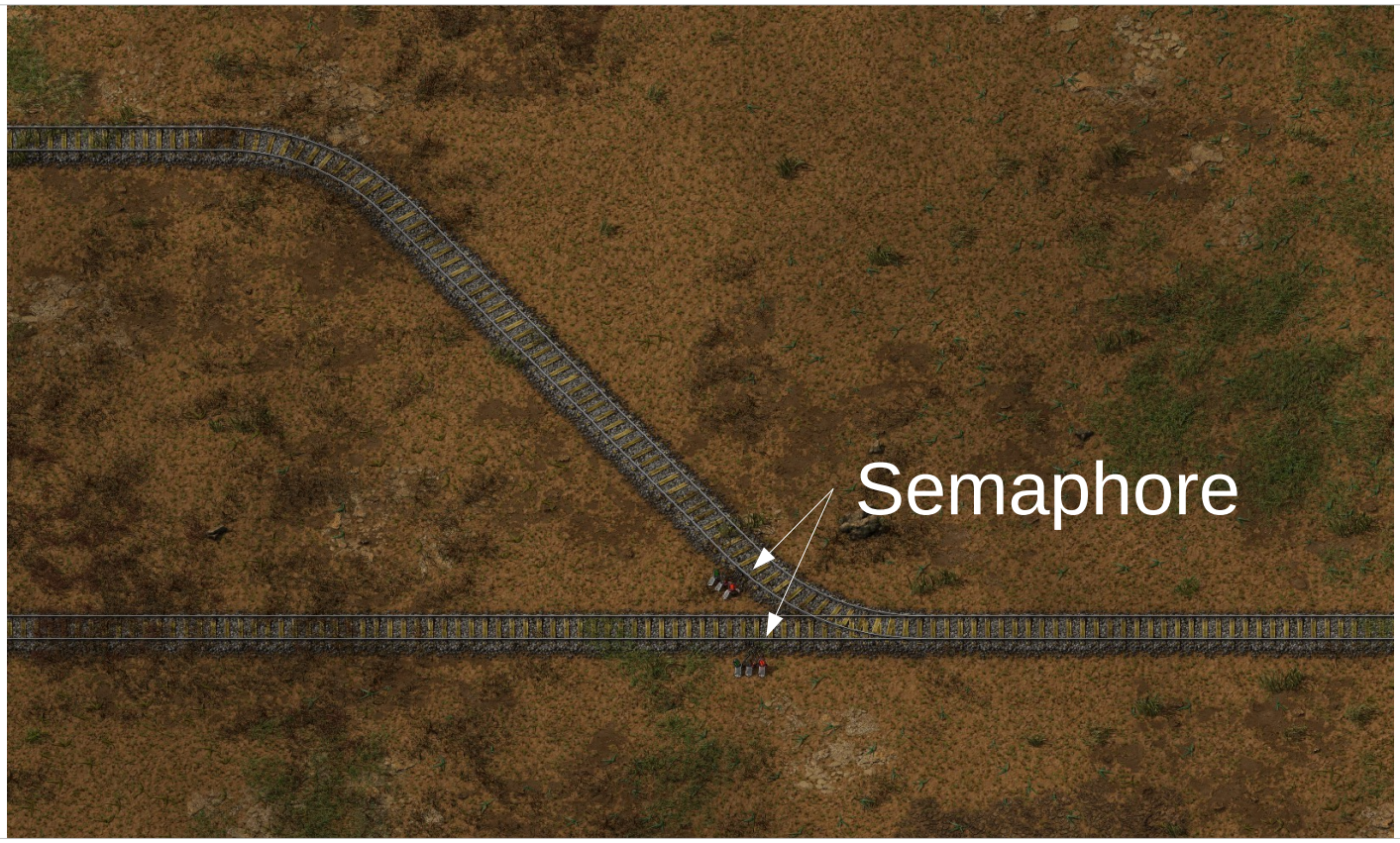
Semaphore

The classic train signal semaphore was patented in the 1840’s. It thereby predates computers by over 100 years.


Shared variable or resource


Thread




Semaphore


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


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

Two questions:

How do we decide which thread can go next?

How do we implement a semaphore?

Which thread can lock the resource next?

Which thread can lock the resource next?

Stack

Queue

How do we implement a semaphore?

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

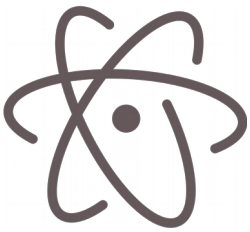
```cpp
#include <thread>
#include <iostream>

struct semaphore {
     unsigned int count = 1;
     void down() {
          while(count == 0) {}
          count--;
     }
     void up() { count++; }
};

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

int main() {
     semaphore lock;
     int sum = 0;
     std::thread threads[100];
     for(int i = 0; i < 100; i++) {
          threads[i] = std::thread(decrement, &lock, &sum);
     }
     for(int i = 0; i < 100; i++) {
          threads[i].join();
     }
     std::cout << sum << std::endl;
     return 0;
}
```

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

struct semaphore {
    unsigned int count = 1;
    void down() {
        while(count == 0) {}
        count--;
    }
    void up() { count++; }
};

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

int main() {
    semaphore lock;
    int sum = 0;
    std::thread threads[100];
```

```
};

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

int main() {
    semaphore lock;
    int sum = 0;
    std::thread threads[100];
    for(int i = 0; i < 100; i++) {
        threads[i] = std::thread(decrement, &lock, &sum);
    }
    for(int i = 0; i < 100; i++) {
        threads[i].join();
    }
    std::cout << sum << std::endl;
    return 0;
}
```

100
100
100
100
100
99
100
100

Screenshot is from the game Batman: Arkham Knight

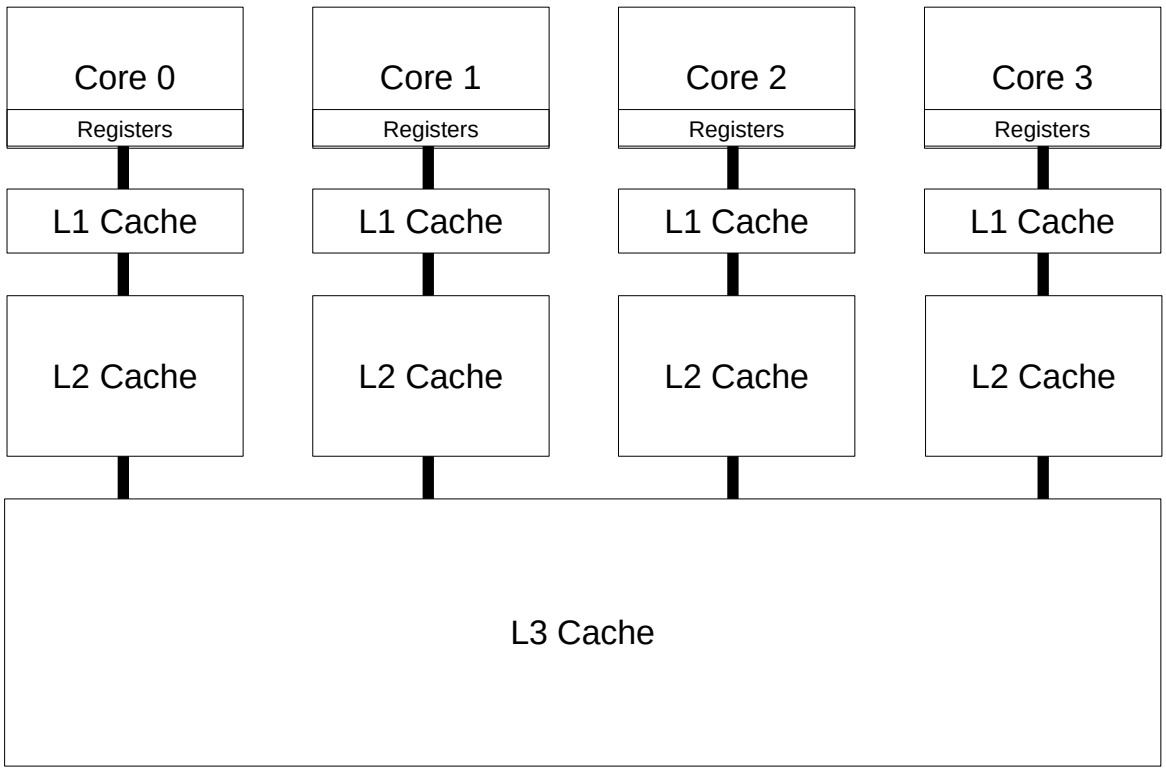


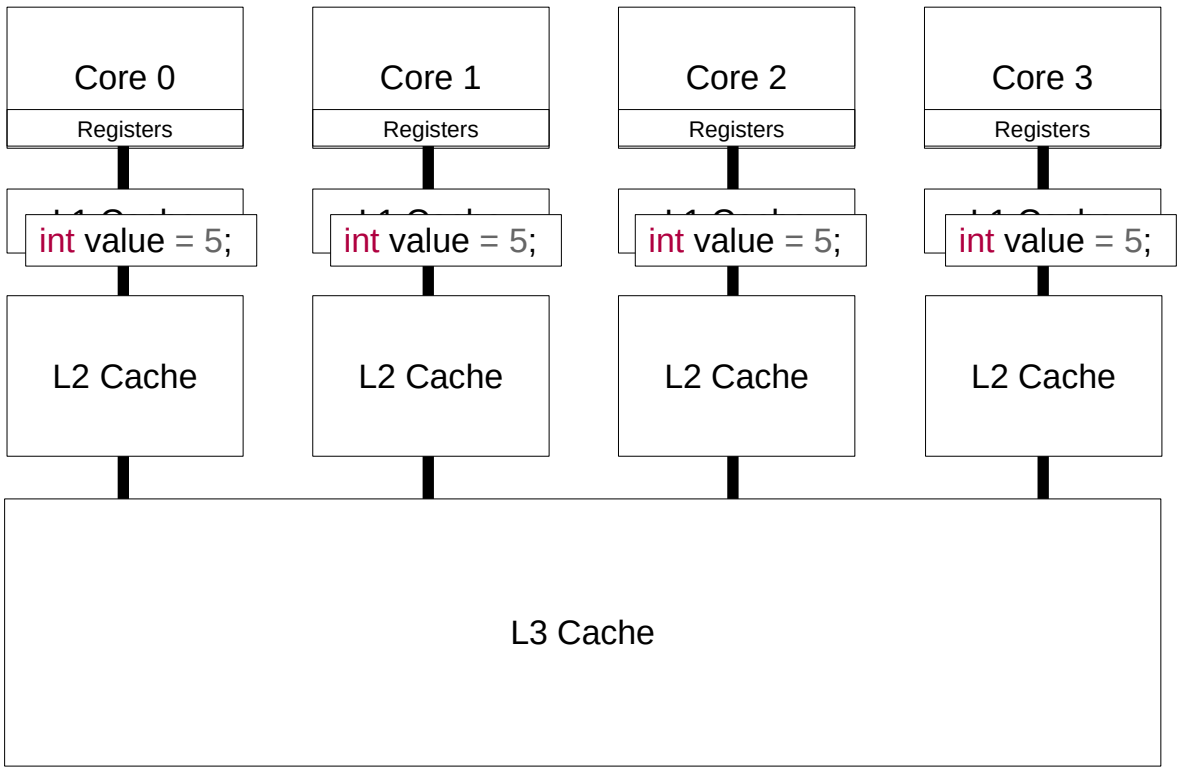
Atomic Operations

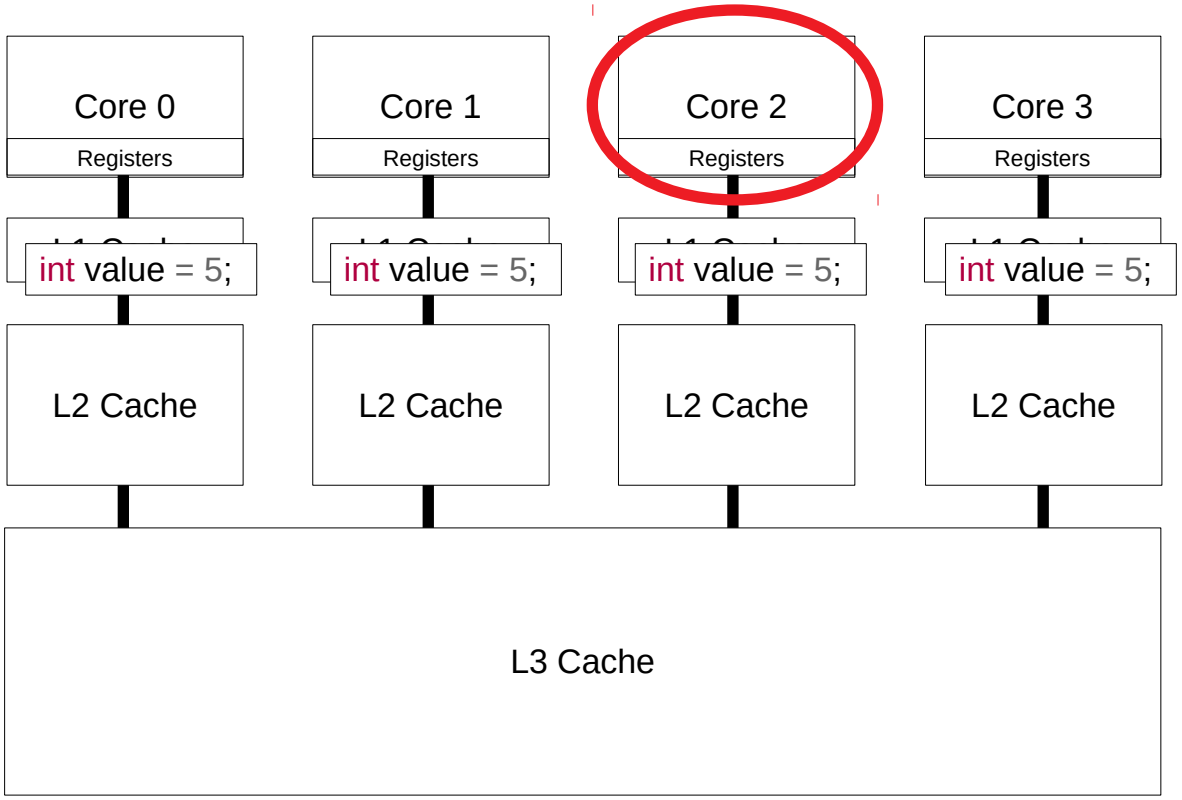
CMPXCHG

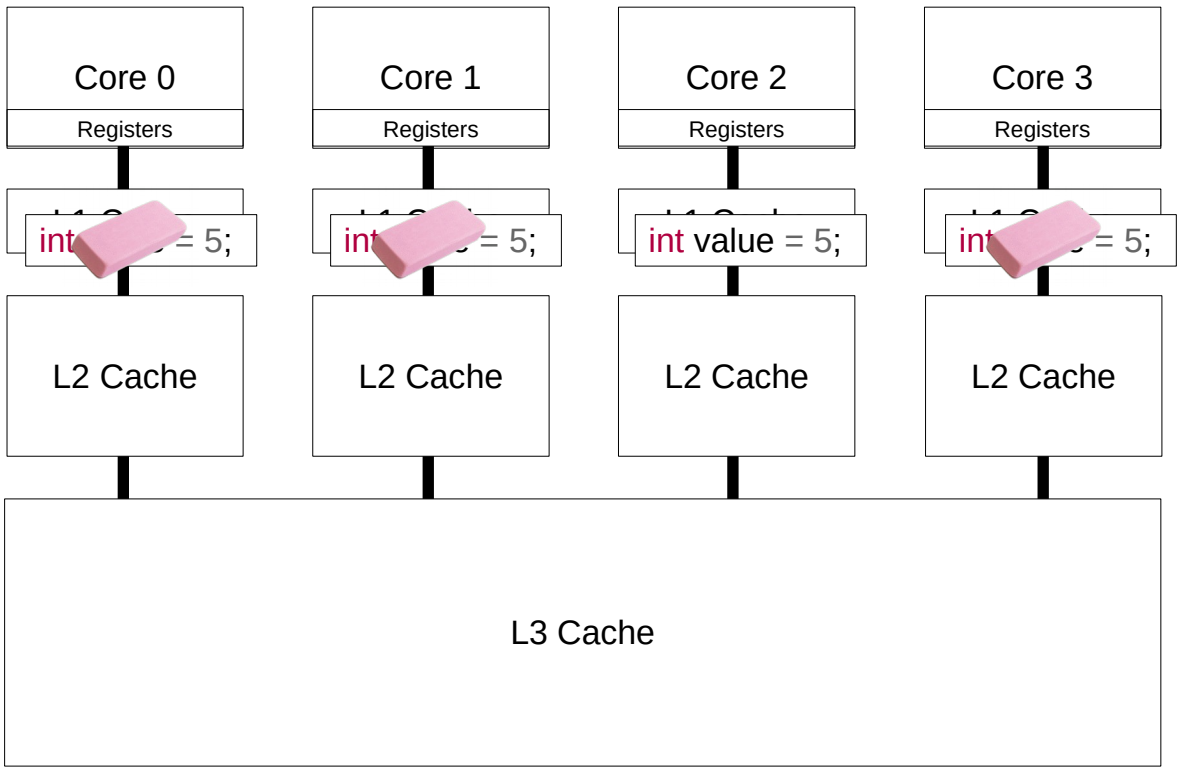
CMPXCHG

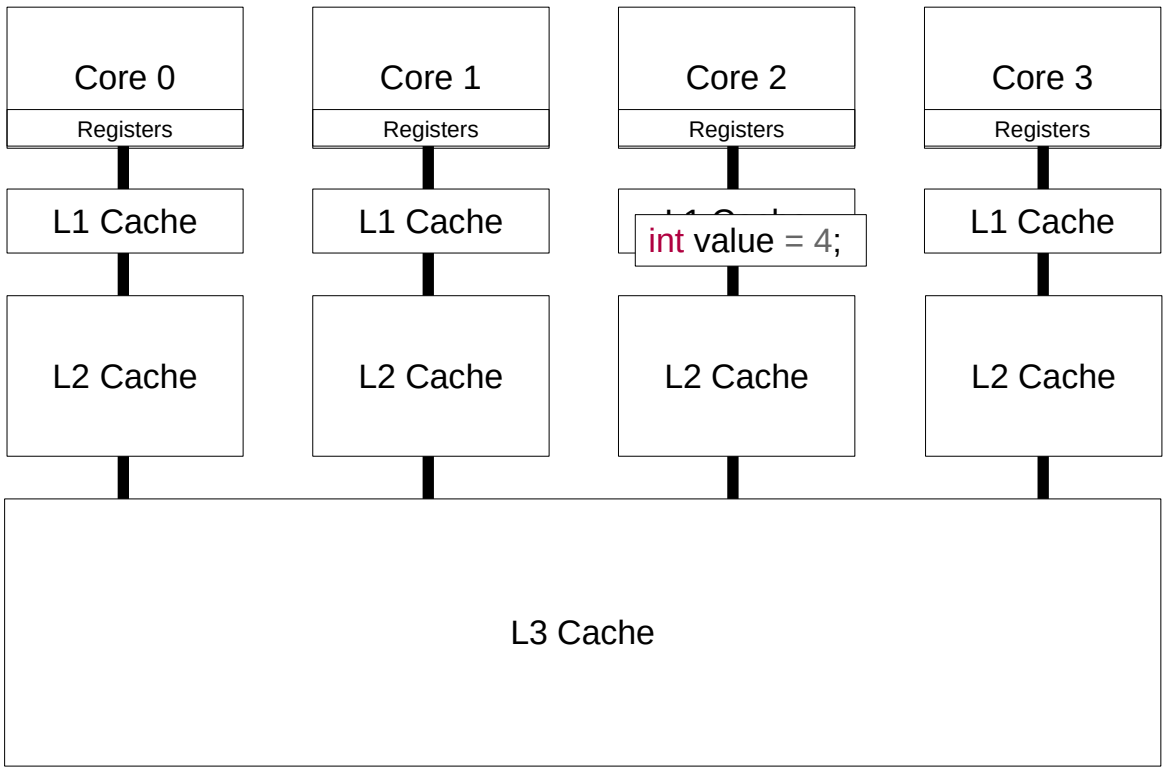
```
int compareAndSwap(int* location, int old, int new)
{
  int currentValue = *location;
  if (currentValue == old)
     *location = new;
  return currentValue;
}
```

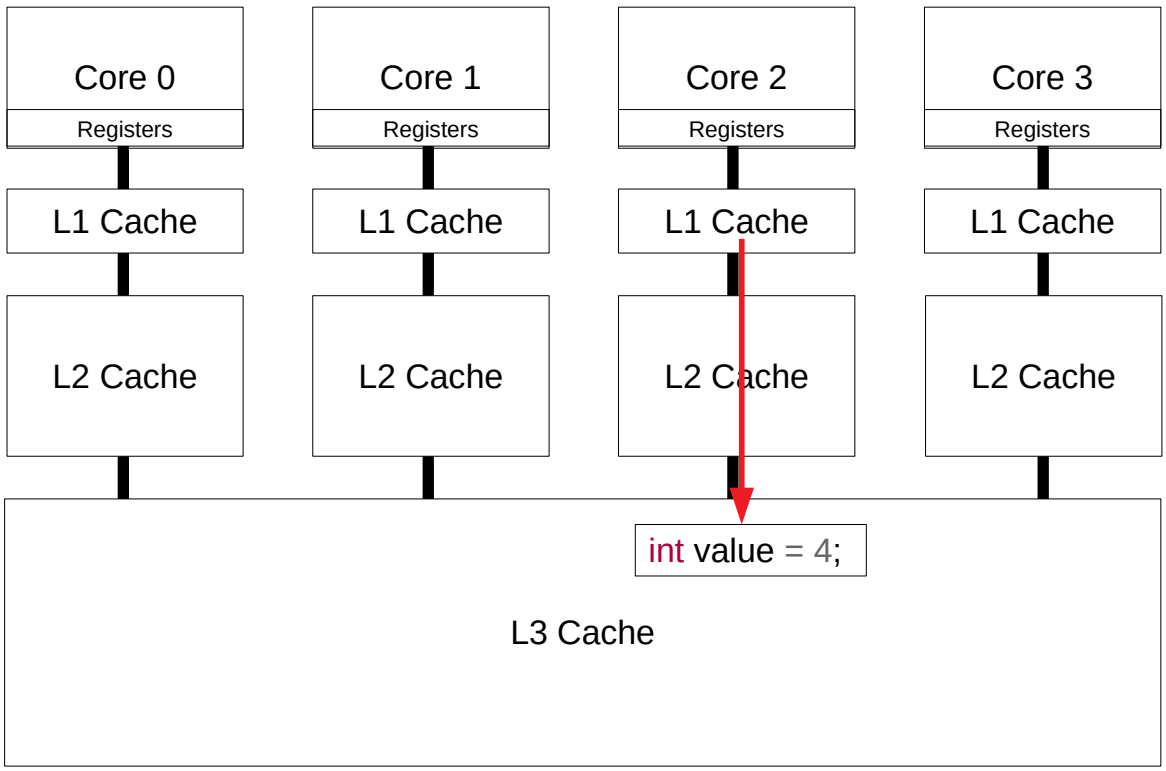

Core 0
Registers
L1 Cache
L2 Cache
Core 1
Registers
L1 Cache
L2 Cache
Core 2
Registers
L1 Cache
L2 Cache
Core 3
Registers
L1 Cache
L2 Cache
L3 Cache

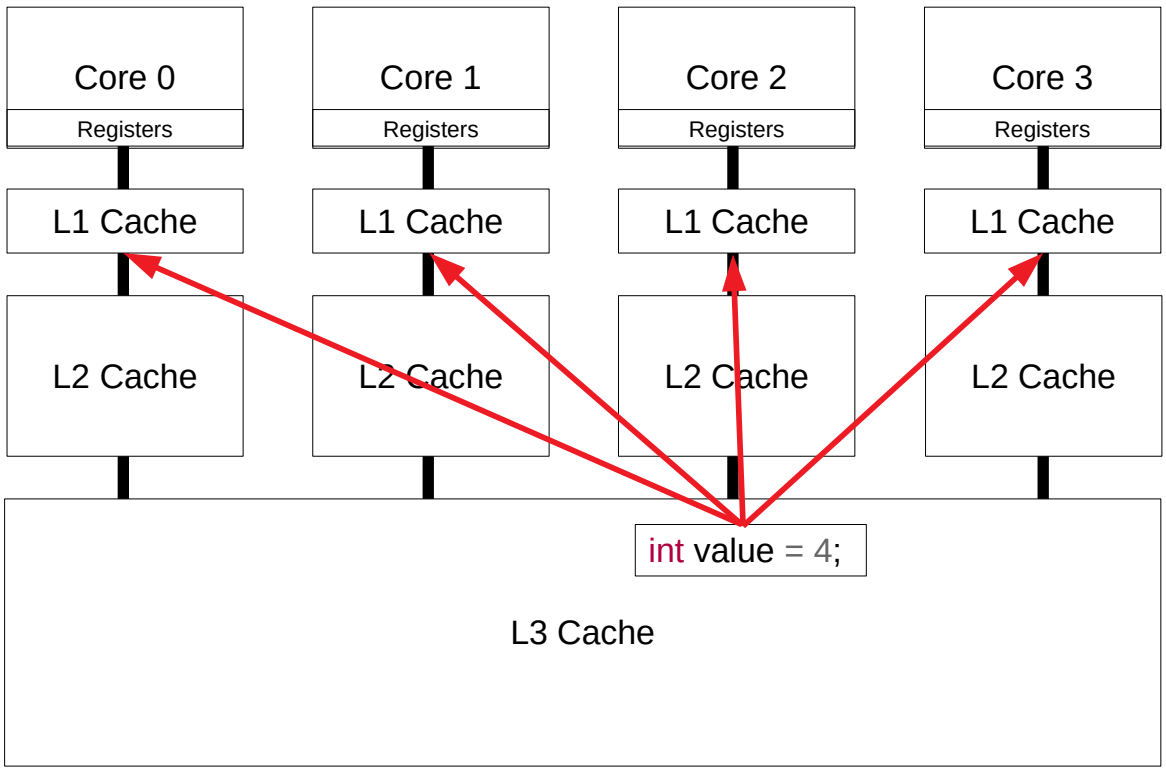

Core 0
Registers
int value = 5;
L2 Cache
Core 1
Registers
int value = 5;
L2 Cache
Core 2
Registers
int value = 5;
L2 Cache
Core 3
Registers
int value = 5;
L2 Cache
L3 Cache


Core 0
Registers
int value = 5;
L2 Cache
Core 1
Registers
int value = 5;
L2 Cache
Core 2
Registers
int value = 5;
L2 Cache
Core 3
Registers
int value = 5;
L2 Cache
L3 Cache


Core 0
Registers
Core 1
Registers
Core 2
Registers
Core 3
Registers
int value = 5;
L2 Cache
L2 Cache
L2 Cache
L2 Cache
L3 Cache


Core 0
Registers
L1 Cache
L2 Cache
Core 1
Registers
L1 Cache
L2 Cache
Core 2
Registers
int value = 4;
L2 Cache
Core 3
Registers
L1 Cache
L2 Cache
L3 Cache


Core 0
Registers
L1 Cache
L2 Cache
Core 1
Registers
L1 Cache
L2 Cache
Core 2
Registers
L1 Cache
L2 Cache
Core 3
Registers
L1 Cache
L2 Cache
int value = 4;
L3 Cache


Core 0
Registers
L1 Cache
L2 Cache
Core 1
Registers
L1 Cache
L2 Cache
Core 2
Registers
L1 Cache
L2 Cache
Core 3
Registers
L1 Cache
L2 Cache
int value = 4;
L3 Cache

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

The point of this slide is not to for me to be able to nitpick on whether you know which instructions x86 is able to do atomically on the exam. Rather, notice how the supported instructions are extremely simple. Arithmetic only has variations of additions and subtractions (negation being the only odd one out), and simple bitwise operations and value exchanges complete the set.

```
struct semaphore {
  atomic_t count;
  int sleepers;
  wait_queue_head_t wait;
};
```

Semaphore flavours

Semaphore

Mutex

Spinlock

Semaphore:

Allow n threads through

May use a combination of
busy waiting and sleeping

Semaphore:

Allow n threads through

May use a combination of busy waiting and sleeping

```
while(!unlocked(lock)) {

}
```

Waking is handled by the operating system. Threads can request sleeping threads to be awoken.

Mutex:

MUTual EXclusion

Special case of a Semaphore

Only allows one thread through

Most common locking mechanism

Spinlock:

Only allows one thread through

Uses busy waiting (and usually atomic operations)

Best used for short wait periods

Spinlock:

Only allows one thread through

Uses busy waiting (and usually atomic operations)

Best used for short wait periods

In practice:

Mutexes in C++

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

void increment(std::mutex* lock, int* value) {
    lock->lock();
    (*value)++;
    lock->unlock();
}

int main() {
    std::mutex lock;
    int sum = 0;
    std::thread threads[100];
    for(int i = 0; i < 100; i++) {
        threads[i] = std::thread(increment, &lock, &sum);
    }
    for(int i = 0; i < 100; i++) {
        threads[i].join();
    }
    std::cout << sum << std::endl;
    return 0;
}
```

In practice:

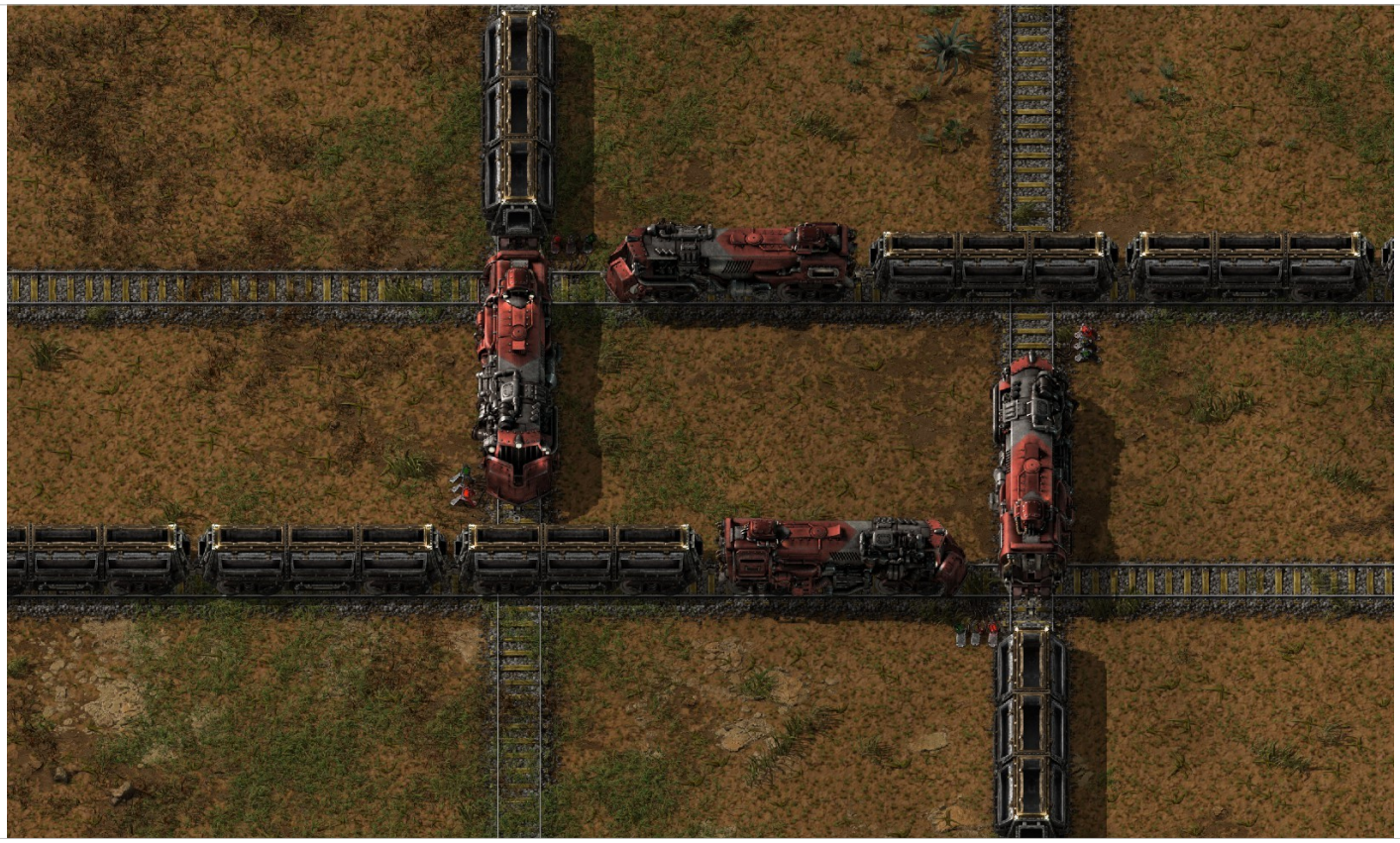
Atomic operations in C++

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

void increment(std::atomic<int>* value) {
    (*value)++;
}

int main() {
    std::atomic<int> value;
    std::thread threads[100];
    for(int i = 0; i < 100; i++) {
        threads[i] = std::thread(increment, &value);
    }
    for(int i = 0; i < 100; i++) {
        threads[i].join();
    }
    std::cout << value << std::endl;
    return 0;
}
```

```
std::atomic<int> spinlock;
spinlock = 0;
int currentValue = 0;

// lock
while(!spinlock.compare_exchange_weak(currentValue, 1))
{
   currentValue = 0;
}

// unlock
spinlock = 0;
```

Locks and Atomics are awesome!

All our problems are solved!



Deadlocks

The Dining Philosophers Problem

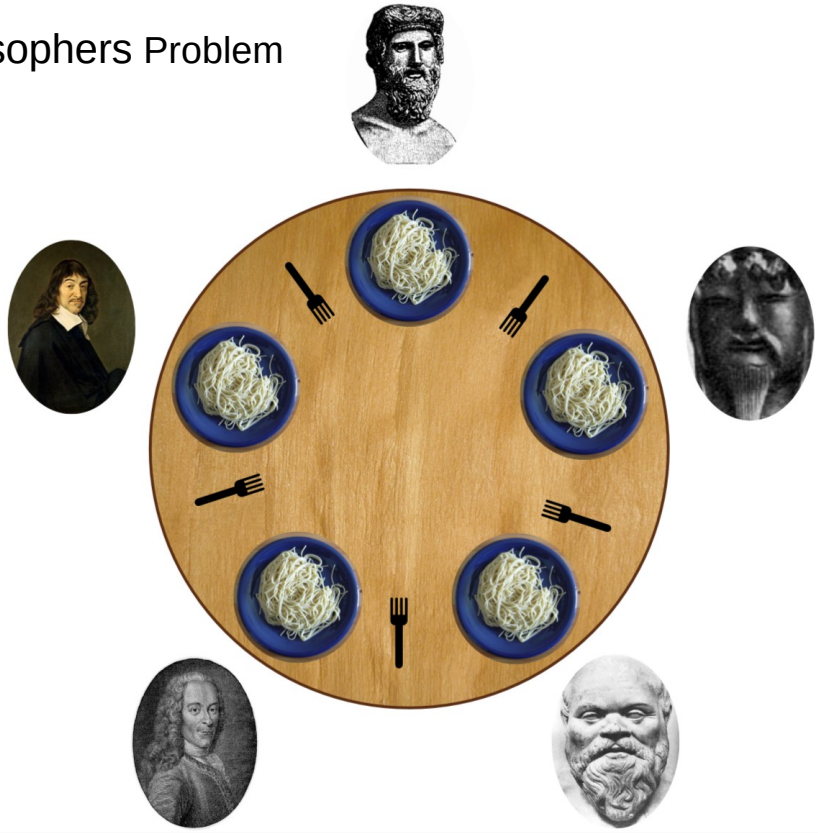
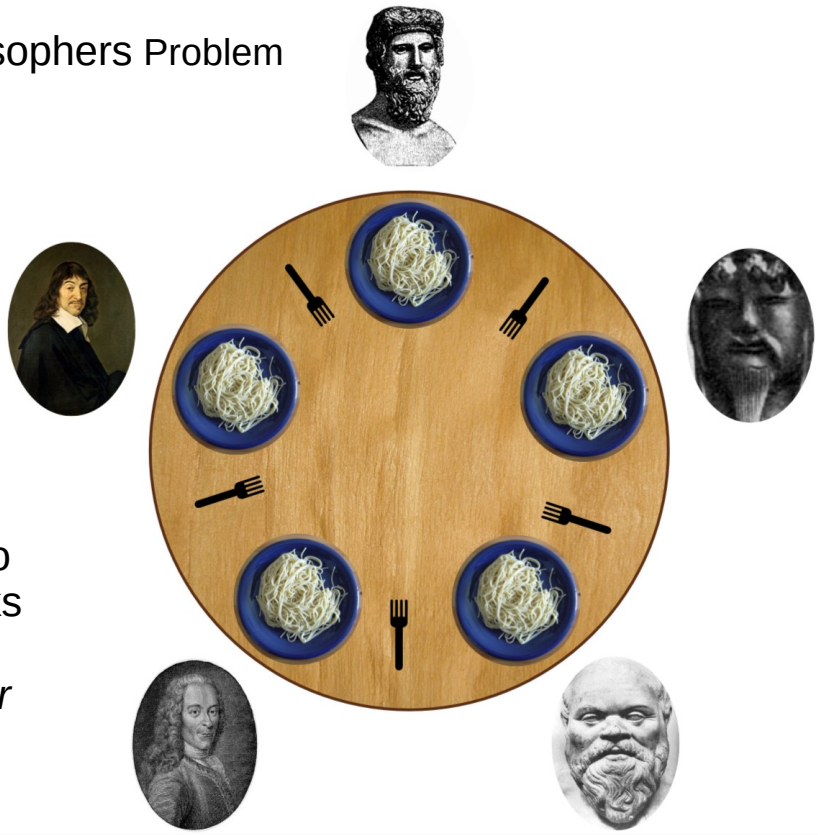


Illustration by Benjamin D. Esham

The Dining Philosophers Problem



Rules:

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

Illustration by Benjamin D. Esham

Locking does not necessarily mean
no race conditions exist!

```cpp
#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

Critical Sections run in serial.

Only do the bare minimum inside of them!

Minimise resources shared between threads.

Thread pools avoid creating and destroying threads many times

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.

Next time:

OpenMP