P99 CONF

A ScyllaDB Community

Low Latency Gal presents Low Latency Stuff

Sonia Sadhbh Kolasinska

IT Consultant | MSE Computer Science | Poland | Ireland

Low Latency Gal

presents

Low Latency *Stuff*

Who Am I?

Low Latency Gal

?

Thread Synchronisation 101

Thread Synchronisation 101

The basics: **Mutex, Monitor & Condition Variable**

Monitor is like a dental clinic:

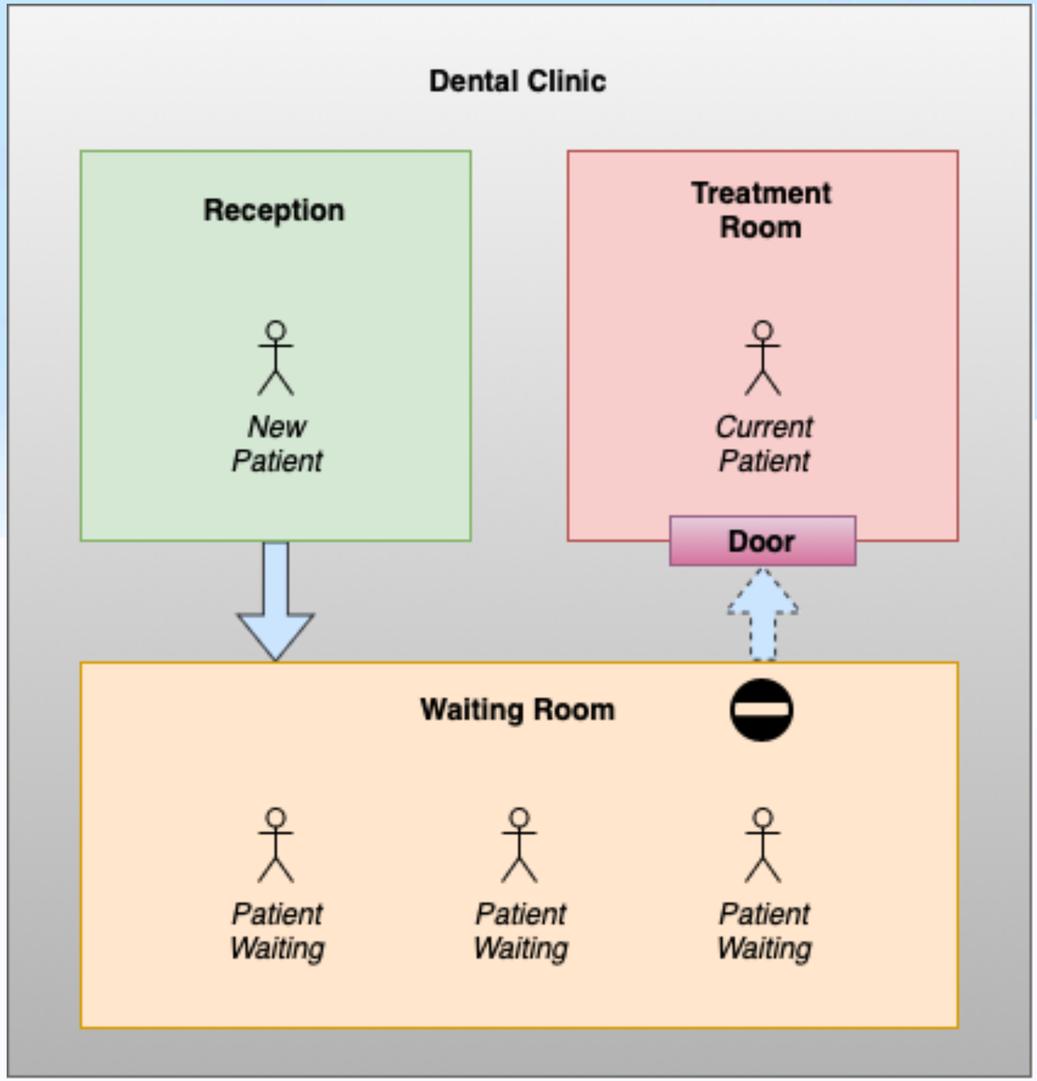
- Thread registers at reception (**Entry Set Joined**)
- Checks their registration ticket (**Lock Acquired**)
- Awaits in waiting room until notified (**Lock Released, Wait Set Joined**)
- Once notified checks their ticket (**Condition Signalled**)
- Enters the treatment room (**Lock Acquired**)
- Finishes and leaves the treatment room (**Lock Released**)

Mutex is like a door of the dental treatment room:

- Closes immediately after one thread enters the room
- Opens back again when thread leaves the room

Condition Variable is like a callout in the clinic:

- Thread in waiting room receives notification to enter treatment room



Thread Synchronisation 101

What is **Lock** and what is **Wait**?

Lock is a barrier that allows only one thread to enter critical section, the code of which may:

- Modify at least one piece of shared data
- Read multiple-pieces of shared data

Wait is an inactive state of the thread, where thread execution is suspended until e.g.:

- Lock becomes available
- Condition variable is signalled
- Blocking operation has completed, e.g. read file

Thread Synchronisation 101

Problems of **Lock** and **Wait**

Lock is a barrier that:

- Stops other threads from making a progress until thread holding a lock releases the lock, e.g. leaves critical section of the code that modifies shared data
- Deadlock is possible when two threads cross-lock each other, i.e. each thread holds a lock that the other thread tries to acquire

Wait is an inactive state that:

- Causes expensive thread context switch, and requires another expensive context switch to resume progress

Lock-Free / Wait-Free 101

Lock-Free / Wait-Free 101

Cons & Pros of **Lock-Free** and **Wait-Free**

Lock-Free and **Wait-Free** programming:

Pros:

- At least one thread always makes a progress
- No thread will enter inactive suspended state
- Ultra low nanosecond range latency

Cons:

- CPU intensive

Lock-Free / Wait-Free 101

What is **Lock-Free** and what is **Wait-Free**?

Lock-Free is a method of programming where:

- Atomic instructions guarantee that:
  - Only one thread will modify shared data at a time, and data will stay consistent

Wait-Free is a method of programming where:

- Atomic instructions guarantee that:
  - A thread can wait for a condition to be true

Cache Coherency 101

Cache Coherency 101

What is **Cache Coherency** and **Memory Barrier**?

Cache Coherency is an algorithm implemented by CPU to:

- Maintain consistent view of data at given memory address

Memory Barrier is an internal mechanism in CPU to enforce **Cache Coherency** when we:

- Perform Reads & Writes to **volatile** data

Cache Coherency 101

Example: Intel x86 Assembly code with **Memory Barrier**

Read after Read on Intel x86

```
_TryAgain:
    mfence                          ; wait for load and stores
    mov  eax, dword ptr [edx]       ; read 32-bit value pointed by edx into eax
    test eax, eax                   ; test eax & eax
    jnz  _TryAgain                  ; jump back to the beginning of the loop
```

Above assembly code is an example tight loop:

- places memory barrier
- reads value from memory
- tests if value is zero and jumps back to continue the loop, or otherwise loop ends

Cache Coherency 101

Example: ARM64 Assembly code with **Memory Barrier**

Read after Read on ARM64

```
_TryAgain:
    dmb   ish              ; wait for loads and stores
    ldr   x0, [x8]         ; read value pointed by x8 into x0
    cbnz  x0, _TryAgain    ; if not zero, then repeat whole block
```

Above assembly code is an example tight loop:

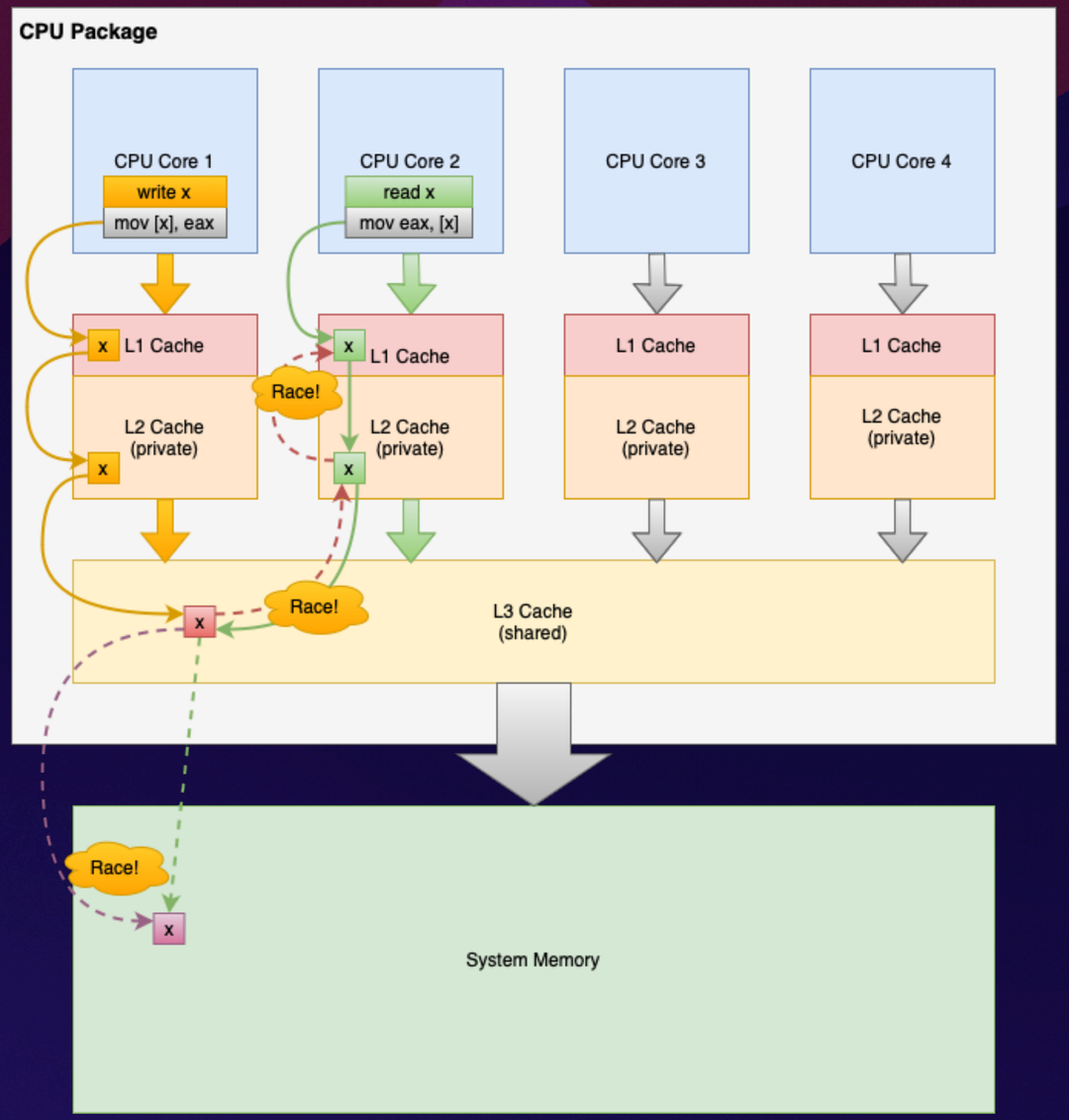
- places memory barrier
- reads value from memory
- tests if value is zero and jumps back to continue the loop, or otherwise loop ends

Cache Coherency 101

What is **Cache Coherency** and **Memory Barrier**?

The **Cache Coherency** is an algorithm, which distributes changes to the data at given memory address across all L1 and L2 caches that store that data. It is important to note that Cache Coherency does not prevent data races, and if one CPU Core has written some data at given address, and now it has read different data from that address, then there was another CPU Core writing to that address after this CPU Core. The ordering of concurrent memory access events is undefined, however it remains consistent across whole CPU, and L3 cache is the ultimate arbiter.

The **Memory Barrier** is a mechanism, which ensures that memory access instructions located before the barrier happen before the instructions that are located after the barrier. Note that without Memory Barrier two instructions accessing the same memory address in the same direction, could be reordered and executed together. This would be very undesirable in Lock-Free / Wait-Free programming, where we often read value from the same address in a tight loop, and as a result of such optimisation we wouldn't be reading fresh values. That would delay exit from the loop, and to prevent that from happening, and to gain better performance, we insert Memory Barriers.



Compare & Swap 101

Compare & Swap 101

Example: Intel x86 Assembly code with **Race Condition**

Read & Update on Intel x86

```
lea edx, [ebp + 8 * eax] ; calculate effective address
mov eax, dword ptr [edx] ; read 32-bit value pointed by edx into eax
add eax, ebx             ; add value of ebx to eax and store result in eax
mov dword ptr [edx], eax ; write 32-bit value from eax into location pointed by edx
```

Above assembly code:

- calculates memory address
- reads value from memory
- changes it using ALU
- writes result back into memory

Compare & Swap 101

Example: Intel x86 Assembly code with **Compare & Swap**

Atomic Read & Update on Intel x86

```
_TryAgain:
    mov eax, dword ptr [edx] ; read value pointed by edx into eax
    add ecx, eax             ; add value of eax to ecx and store result in ecx
    lock cmpxchg [edx], ecx  ; atomic write value of ecx into location pointed by edx
    jnz _TryAgain            ; if atomic write failed, then repeat whole block
```

Above assembly code:

- reads value from memory
- changes it using ALU
- writes result back into memory only if value in memory haven't changed
- repeats the whole process from the beginning until successful write

Compare & Swap 101

Example: ARM64 Assembly code with **Race Condition**

Read & Update on ARM64

```
adrp x8, .x           ; calculate upper bits of memory address
add x8, :lo12:.x      ; calculate lower bits of memory address
ldr x0, [x8]          ; read 64-bit value pointed by x8 into x0
add x0, x0, x1        ; add value of x1 to x0 and store result in x0
str x0, [x8]          ; write 64-bit value into location pointed by x8
```

Above assembly code:

- calculates memory address
- reads value from memory
- changes it using ALU
- writes result back into memory

Compare & Swap 101

Example: ARM64 Assembly code with **Compare & Swap**

Atomic Read & Update on ARM64

```
_TryAgain:
    ldaxr   x0, [x8]          ; atomic read 64-bit value pointed by x8 into x0
    add     x0, x0, x1        ; add value of x1 to x0 and store result in x0
    stlxr   x9, x0, [x8]      ; atomic write 64-bit value into location pointed by x8
    cbnz    x9, _TryAgain     ; if atomic write failed, then repeat whole block
```

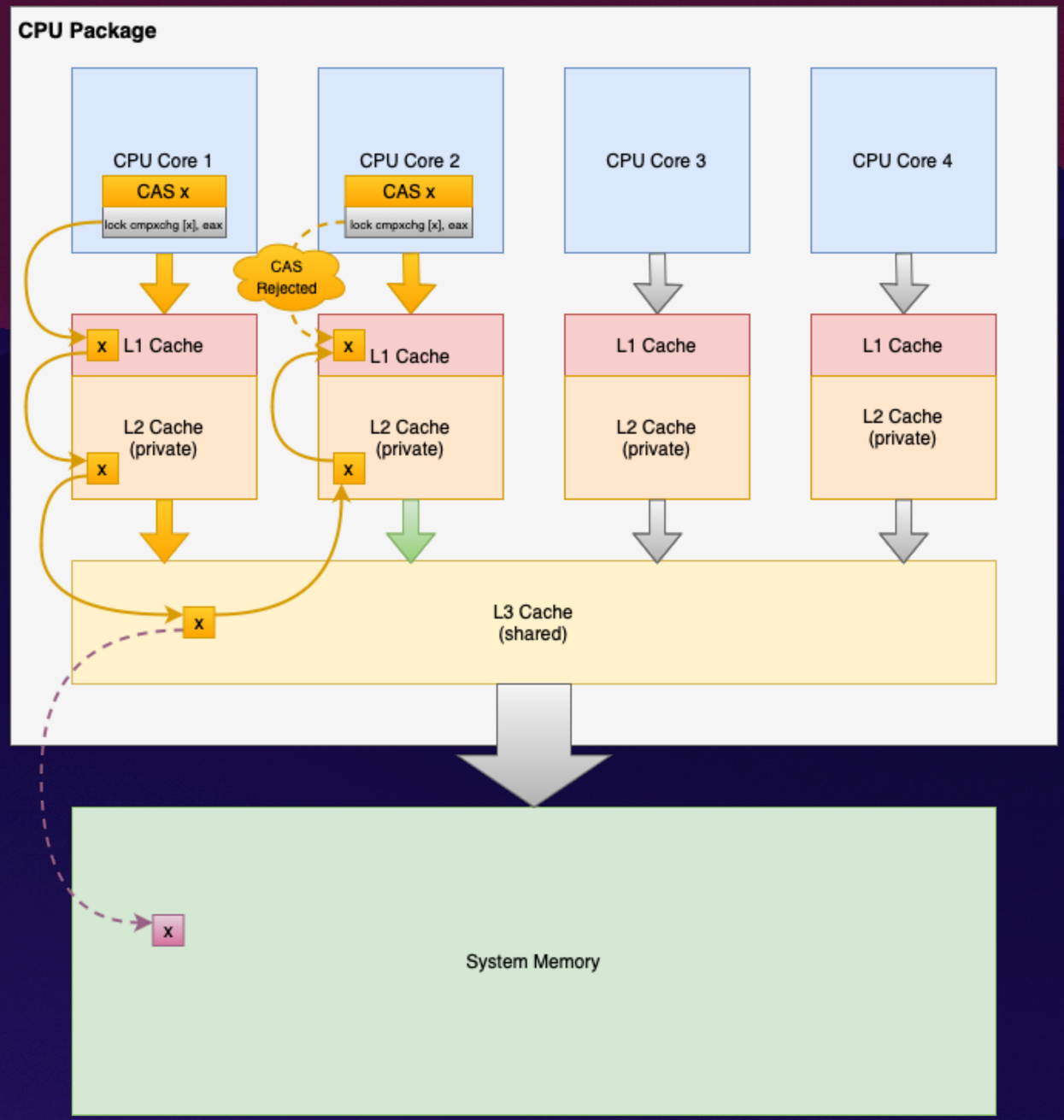
Above assembly code:

- reads value from memory acquiring exclusive access
- changes it using ALU
- writes result back into memory only if value in memory haven't changed
- repeats the whole process from the beginning until successful write

Compare & Swap 101

What is **Compare & Swap**?

Compare & Swap (**CAS**) is an atomic operation, which enforces sequential dependency between atomic operations performed by otherwise independent threads executing on different CPU cores. Threads of the program that run on multi-core CPU execute independently. This means that instructions of one thread are executed completely independent from instructions of the other thread, and there is no correlation of time between those threads. This means that first thread might have written new value at given memory address, and other thread has read old value from that address, and there is no notion of before and after. The only way to introduce the notion of before and after is by introducing some synchronisation mechanism that would enforce that.



Synchronous Concurrency

Synchronous Concurrency

Example: **Single-Threaded Code**

```
class BasicStats
{
public:
    void add_value(double value) {
       m_totalSum += value;
       ++m_totalCount;
    }

    double get_average() const {
       return (m_totalSum / m_totalCount);
    }

private:
    double m_totalSum{};
    double m_totalCount{};
};
```

Synchronous Concurrency

Example: **Multi-Threaded Code**

```
class BasicStatsMT
{
public:
    void add_value(double value) {
        std::scoped_lock lk{m_mutex};
        m_totalSum += value;
        ++m_totalCount;
    }

    double get_average() const {
        std::scoped_lock lk{m_mutex};
        return (m_totalSum / m_totalCount);
    }

private:
    std::mutex m_mutex{};
    double m_totalSum{};
    double m_totalCount{};
};
```

Synchronous Concurrency

Example: **Wait-Free Multi-Threaded Code**

```
class BasicStatsSL
{
public:
    void add_value(double value) {
        std::scoped_lock lk{m_spinner};
        m_totalSum += value;
        ++m_totalCount;
    }

    double get_average() const {
        std::scoped_lock lk{m_spinner};
        return (m_totalSum / m_totalCount);
    }

private:
    spinlock m_spinner{}; // spin-lock implementation (e.g. https://rigtorp.se/spinlock/)
    double m_totalSum{};
    double m_totalCount{};
};
```

Synchronous Concurrency

Example: **Lock-Free & Wait-Free Multi-Threaded Code (Bad)**

```cpp
#include <winnt.h>

class NTAtomicStats_Bad
{
public:
    void add_value(double value) {
        BasicStats const *pOld{};
        BasicStats *pNew{};
        do {
            if (nullptr != pNew) { delete pNew; }
            // <<< memory barrier >>>
            pOld = m_pData;                          // 1. Take recent value.
            pNew = new BasicStats{*pOld};            // 2. Make a copy of that value // operator new!
            pNew->add_value(value);                  // 3. Update that copy
                                                     // 4. Try to publish that copy or try again
        } while (pOld == InterlockedCompareExchangePointer((PVOID volatile*)&m_pData, (PVOID)pNew, (PVOID)pOld));
        // <<< memory barrier >>>                    // 5. Copy was published delete old value
        delete pOld; // Cause use after free!
    }

    double get_average() const {
        // <<< memory barrier >>>
        return m_pData->get_average(); // Use after free!
    }

private:
    BasicStats const * volatile m_pData{new BasicStats{}}; // volatile adds memory barriers
};
```

Synchronous Concurrency

Example: **Lock-Free(ish) & Wait-Free Multi-Threaded Code**

```
class StdAtomicStats
{
public:
    void add_value(double value) {
        std::shared_ptr<BasicStats const> old{};
        std::shared_ptr<BasicStats> new_{};
        do {
            // <<< memory barrier >>>
            old = std::atomic_load(&m_data); // spin lock
            new_ = std::move(std::make_shared<BasicStats>(*old)); // memory allocation
            new_->add_value(value);

        } while (std::atomic_compare_exchange_strong(&m_data, &old,
                    std::const_pointer_cast<BasicStats const>new_)); // spin lock
        // <<< memory barrier >>>
    }

    double get_average() const {
        // <<< memory barrier >>>
        return std::atomic_load(&m_data)->get_average(); // spin lock
    }

private:
    std::shared_ptr<BasicStats const> m_data{std::make_shared<BasicStats>()};
};
```

Synchronous Concurrency

NTARC: **Lock-Free & Wait-Free** Shared Pointer in C

```c
#include <NTARC.H> // NTARC library (https://github.com/sadhbh-c0d3/lock-free)

volatile NTARC g_foo = { 0, 0 }; // Allocate atomic shared pointer

void make_foo() {
    NTARC foo; // Allocate local reference to foo

    if (foo_new(&foo, 1, 2) == FALSE) // Create new instance of foo and store local reference to it
    {
        printf("Cannot create Foo!");
        return;
    }

    ntarc_atomic_store(&g_foo, &foo); // Atomically store newly created shared object into shared variable
    // <<< memory barrier >>>
    foo_created(&foo); // Do some work with local reference to foo
    ntarc_drop(&foo);  // Drop local reference to shared object
}

void read_foo() {
    NTARC foo;         // Allocate local reference to foo
    ntarc_atomic_load(&g_foo, &foo); // Load shared object from shared variable into local shared pointer
    // <<< memory barrier >>>
    process_foo(&foo); // Do some work with local reference to foo
    ntarc_drop(&foo);  // Drop local reference to shared object
}
```

Synchronous Concurrency

Example: **Lock-Free & Wait-Free Multi-Threaded Code**

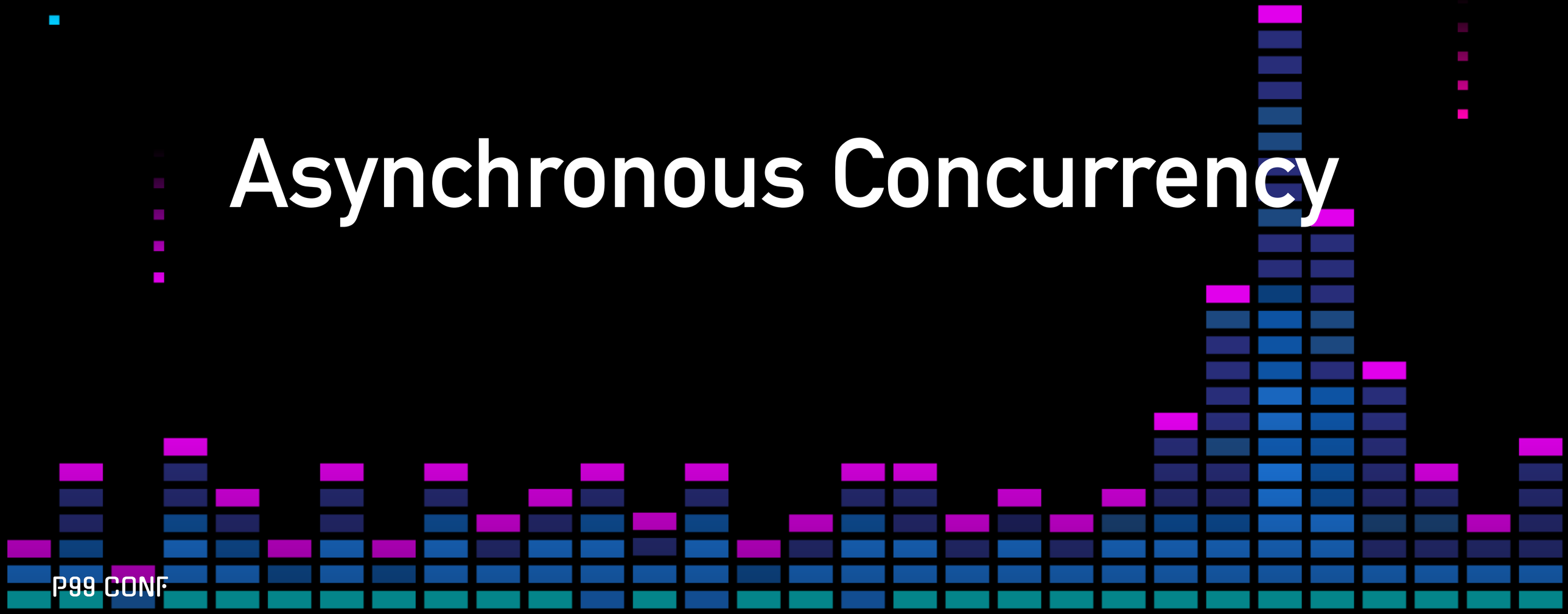
```
class AtomicStats
{
public:
    void add_value(double value) {
        lock_free::arc<BasicStats const> old{}; // e.g. C++ version of my NTARC
        lock_free::arc<BasicStats> new_{};      // (https://github.com/sadhbh-c0d3/lock-free)
        do {
            // <<< memory barrier >>>
            old = m_data.atomic_clone();                          // 1. begin atomic transaction
            new_ = lock_free::free_list<BasicStats>::copy(*old); // 2. lock-free allocation
            new_->add_value(value);                               // 3. thread local update

        } while (m_data.atomic_compare_store(old, lock_free::const_arc_cast<
                                BasicStats const>::new_));        // 4. commit atomic transaction
        // <<< memory barrier >>>
    }

    double get_average() const {
        // <<< memory barrier >>>
        return m_data.atomic_clone()->get_average();   // 1. atomic load - OK
    }                                                  // 2. return thread local computation

private:
    lock_free::arc<BasicStats const> m_data{lock_free::free_list<BasicStats>::alloc()};
};
```

Asynchronous Concurrency



Asynchronous Concurrency

Basics: **Queue, Producer and Consumer**

```
#include <thread>
#include “ProducerConsumer.h”

// queue [TBD]
struct Queue {
    using ElementType = int;
    void enqueue(int&&);
    int dequeue();
};

// produce new item
int make_int();

// consume received item
void process_int(int &&);
```

```
int main()
{
    Queue q{};

    std::thread t1{[&q]{
        producer(q, make_int);
    }}; // producer thread
    std::thread t2{[&q]{
        consumer(q, process_int);
    }}; // consumer thread

    t1.join();
    t2.join();

    return 0;
}
```

Asynchronous Concurrency

Basics: **Queue, Producer and Consumer**

```
#include “QueueConcept.h”

void producer(QueueConcept auto &queue, auto makeItem) {
    for (;;) {
        QueueTraitsElementT<decltype(queue)> item{makeItem()};
        queue.enqueue(std::move(item));
    }
}

void consumer(QueueConcept auto &queue, auto processItem) {
    for (;;) {
        QueueTraitsElementT<decltype(queue)> item{queue.dequeue()};
        processItem(std::move(item));
    }
}
```

Asynchronous Concurrency

Basics: **Queue, Producer and Consumer**

```cpp
#include <type_traits>
#include <utility>

template<class T> concept QueueConcept = requires(T &q)
{
    typename std::remove_cvref_t<T>::ElementType;

    { q.enqueue(std::declval<typename std::remove_cvref_t<T>::ElementType&&>()) };
    { q.dequeue() } -> std::convertible_to<typename std::remove_cvref_t<T>::ElementType>;
};

template<QueueConcept T> struct QueueTraits {
    using QueueType = std::remove_cvref_t<T>;
    using ElementType = typename QueueType::ElementType;
};

template<QueueConcept T> using QueueTraitsQueueT = typename QueueTraits<T>::QueueType;
template<QueueConcept T> using QueueTraitsElementT = typename QueueTraits<T>::ElementType;
```

Asynchronous Concurrency

Basics: **Mutex and Condition Variable**

```
#include <thread>
#include <mutex>
#include <condition_variable>
#include <deque>

template<class T,
    template<class, class> class ContainerT =
std::deque,
    template<class> class AllocatorT = std::allocator>
class UnboundedQueueMT
{
public:
    using ElementType = T;

    void enqueue(T &&value)
    {
        {
            std::unique_lock lk{m_mutex};
            // ^ context switch

            m_data.emplace_back(
                std::move(value));
        }

        m_cond.notify_one(); //< wakeup call
    }
```

```
    T dequeue()
    {
        std::unique_lock lk{m_mutex};
        // ^ context switch

        m_cond.wait(lk, [this]{
            return !m_data.empty();
        }); //< context switch

        T value = std::move(m_data.front());
        m_data.pop_front();

        return std::move(value);
    }

private:
    using ContainerType = ContainerT<T, AllocatorT<T>>;
    ContainerType m_data{};
    std::mutex m_mutex{};
    std::condition_variable m_cond{};
};
```

Asynchronous Concurrency

Basics: **Semaphore and Atomic**

```cpp
#include <array>
#include <atomic>
#include <semaphore>

template<size_t N> struct IsPowerOfTwo { constexpr static const bool Value = (0 == ((N - 1) & N)); };
template<const size_t N> constexpr static const bool IsPowerOfTwoValue = IsPowerOfTwo<N>::Value;
```

```cpp
template<class T, const size_t N>
requires IsPowerOfTwoValue<N>
class BoundedQueueMT {
public:
    using ElementType = T;
    constexpr static const size_t Size = N;

    void enqueue(T &&value)
    {
        m_freeCount.acquire();  // context switch
        size_t pos = m_writePos++;   // atomic increment
        m_data[pos % N] = std::move(value); // write data
        m_readyCount.release(); // context switch
    }
```

```cpp
    T dequeue()
    {
        m_readyCount.acquire(); // context switch
        size_t pos = m_readPos++; // atomic increment
        T value{std::move(m_data[pos % N])}; // read data
        m_freeCount.release();  // context switch
        return std::move(value);
    }

private:
    std::array<T, N> m_data{};
    std::atomic<size_t> m_writePos{};
    std::atomic<size_t> m_readPos{};
    std::counting_semaphore<N> m_freeCount{N};
    std::counting_semaphore<N> m_readyCount{0};
};
```

Asynchronous Concurrency

NTRINGB: **Lock & Wait-Free** Ring-Buffer in C

```c
#include <NTRINGB.H>                            // NTRINGB library (https://github.com/sadhbh-c0d3/lock-free)

volatile FOO g_buffer[FOO_COUNT];               // Allocate buffer
NTRINGB g_ringb;                                // Allocate ring-buffer control structure

void keep_producing() {
    NTRINGB_POS ring_pos;                       // Ring-buffer stream position
    FOO local_data;                             // Local copy of the next value
    ntringb_pos_init(&g_ringb, &ringb_pos);     // Initialise ring-buffer stream position

    for (;;) {                                  // Loop until (forever)
        produce_next_value(&local_data);        // Produce next value into local variable
        pos = ntringb_begin_write(&ringb_pos); // Begin WRITE transaction - returns position
        // <<< memory barrier >>>
        memcpy(&g_buffer[pos], &local_data, sizeof(FOO)); // Copy local value into buffer at position
        ntringb_commit_write(&ringb_pos);       // Commit WRITE transaction
        // <<< memory barrier >>>
    }
}

void keep_consuming() {
    NTRINGB_POS ring_pos;                       // Ring-buffer stream position
    FOO local_data;                             // Local copy of the next value
    ntringb_pos_init(&g_ringb, &ringb_pos);     // Initialise ring-buffer stream position

    for (;;) {                                  // Loop until (forever)
        pos = ntringb_begin_read(&ringb_pos); // Begin READ transaction - returns position
        // <<< memory barrier >>>
        memcpy(&local_data, &g_buffer[pos], sizeof(FOO)); // Copy value at position in buffer into local variable
        ntringb_commit_read(&ringb_pos);        // Commit READ transaction
        // <<< memory barrier >>>
        consume_next_value(&local_data);        // Consume next value from local variable
    }
}
```

Asynchronous Concurrency

Basics: Lock-Free & Wait-Free Multi-Threaded Code

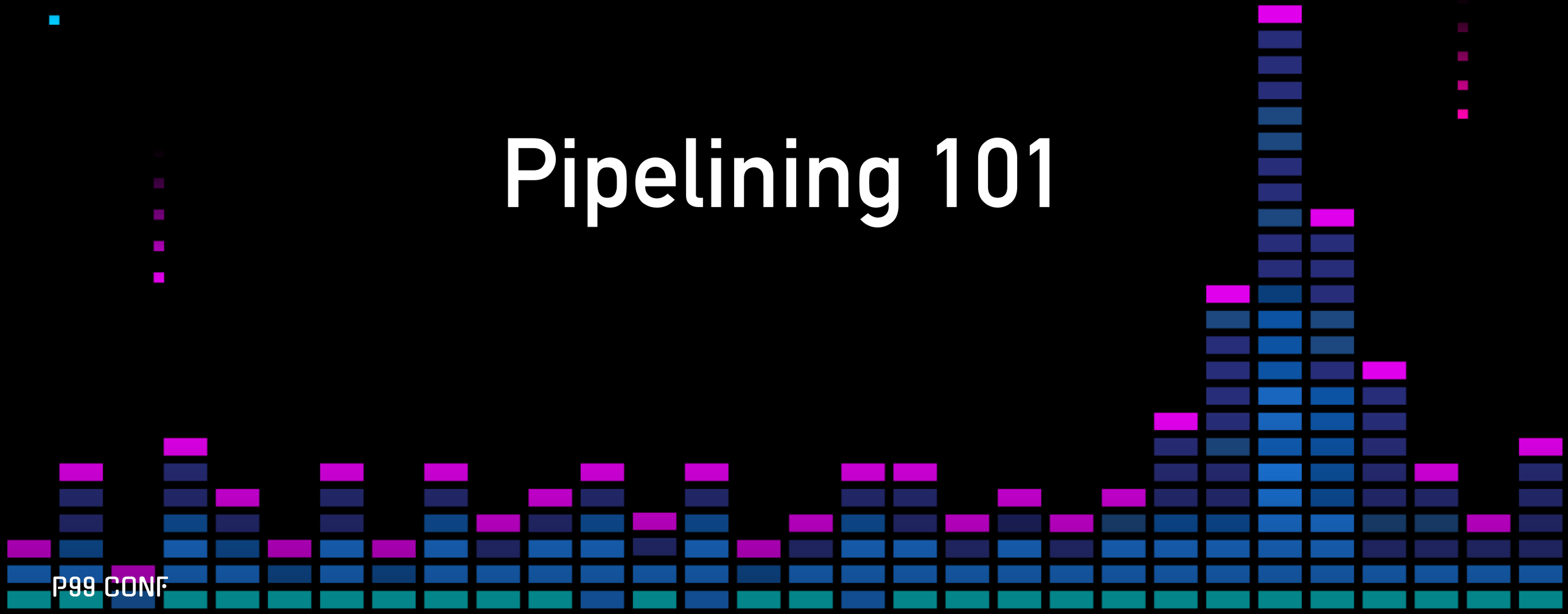
```cpp
template<class T, const size_t N> requires IsPowerOfTwoValue<N>
class BoundedQueue {
public:
    using ElementType = T;

    void enqueue(T &&value)
    {
        lock_free::ringbuf_write<N> transact{m_ringbuf}; // e.g. C++ version of my NTRINGB_POS
        size_t pos = transact.begin_write(); //< lock-free & wait-free acquire
        // <<< memory barrier >>>
        m_data[pos % N] = std::move(value);
        transact.commit_write();             //< lock-free & wait-free release
        // <<< memory barrier >>>
    }

    T dequeue()
    {
        lock_free::ringbuf_read<N> transact{m_ringbuf}; // e.g. C++ version of my NTRINGB_POS
        size_t pos = transact.begin_read();  //< lock-free & wait-free acquire
        // <<< memory barrier >>>
        T value{std::move(m_data[pos % N])};
        transact.commit_read();              //< lock-free & wait-free release
        // <<< memory barrier >>>
        return std::move(value);
    }

private:
    std::array<T, N> m_data{};
    lock_free::ringbuf<N> m_ringbuf{}; // e.g. C++ version of my NTRINGB (https://github.com/sadhbh-c0d3/lock-free)
}
```

Pipelining 101



Pipelining 101

What is **Pipelining**?

Pipelining is a method of organising multiple threads so that:

- The result of first thread is an input for second thread
- While first thread processes first portion of an item, the second thread processes second portion of the previous item

Thread Pinning is a method of explicitly:

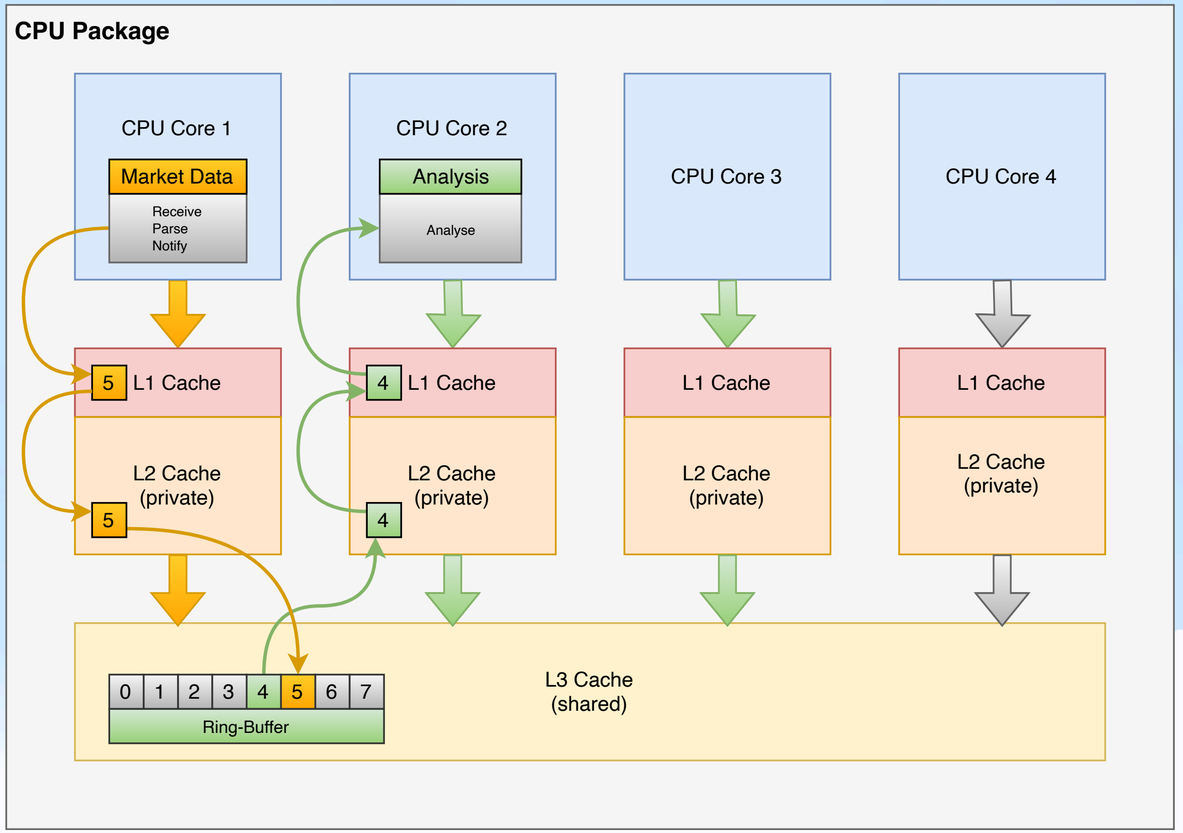
- Assigning exact CPU core to each thread in the pipeline
- And it is called **Setting Affinity**

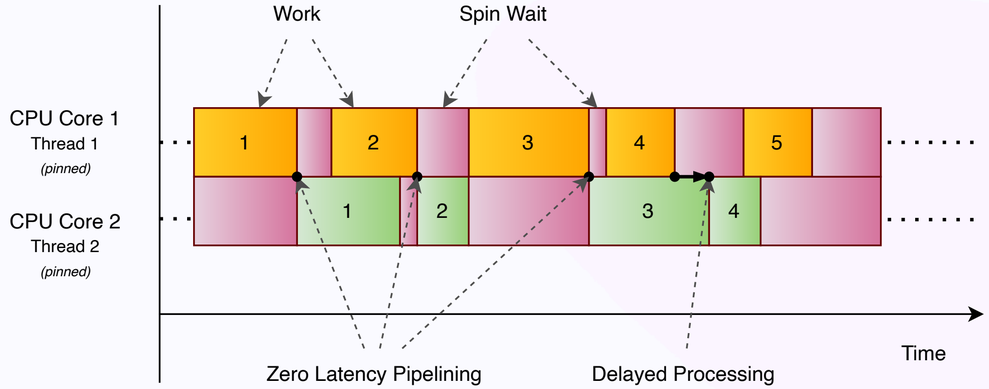
Pipelining 101

In Order Processing

On this example CPU Core 1 runs thread that receives and parses Market Data, and CPU Core 2 runs thread that analyses parsed data.

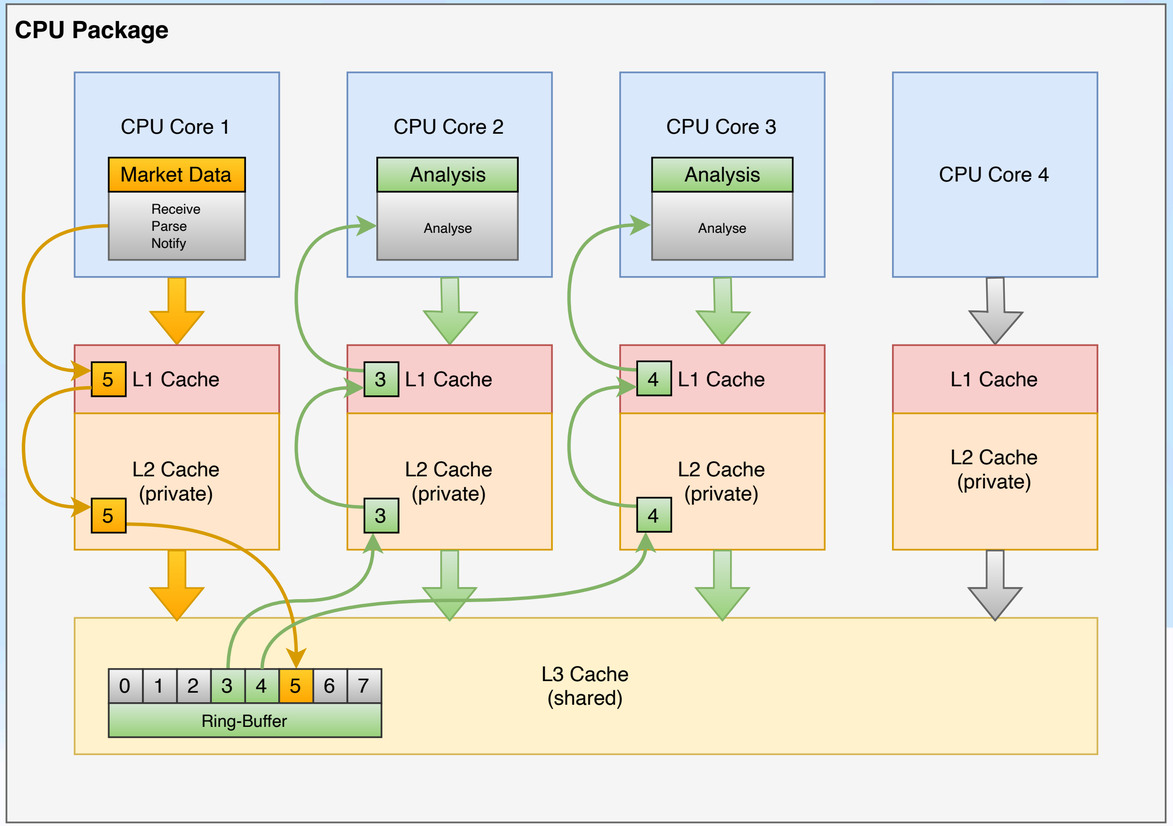
On the time diagram we can see jitter happening on both threads, which is caused by processing taking different amounts of time depending on data. We can see how 2nd item starts being processed by Thread 1 in parallel with Thread 2 performing second portion of the worn on 1st item. If we were to process items without pipelining on a single-thread, then 2nd item would need to wait until 1st item is fully complete. We can see how processing times benefit from pipelining.





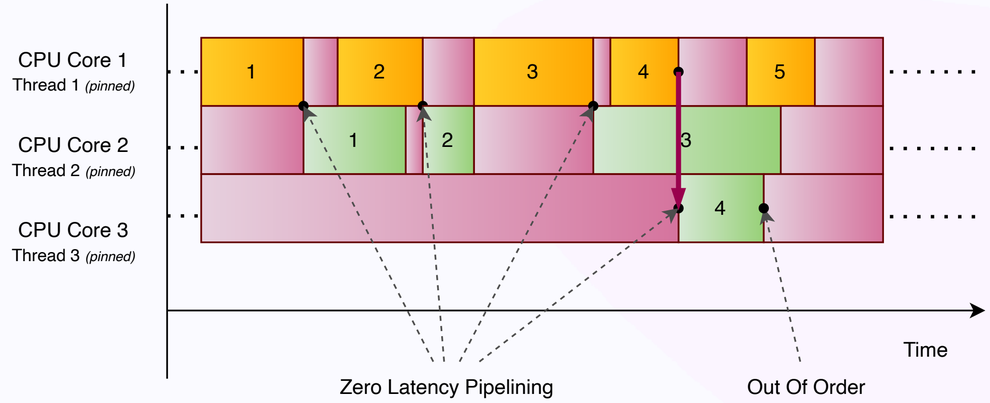
Pipelining 101

Out Of Order Processing



On this example CPU Core 1 runs thread that receives and parses Market Data, and then two CPU Cores 2 and 3 run threads that analyse parsed data.

On the time diagram we can see that thread running on CPU Core 3 processed packet 4 before thread running on CPU Core 2 finished packet 3.



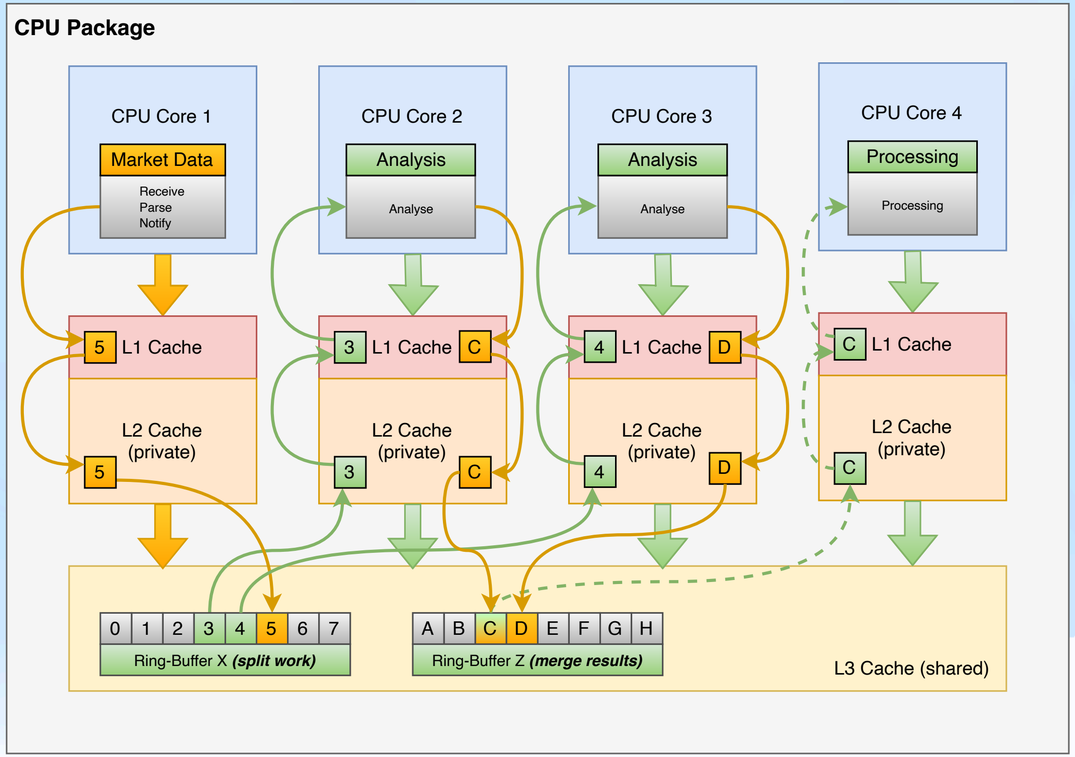
Pipelining 101

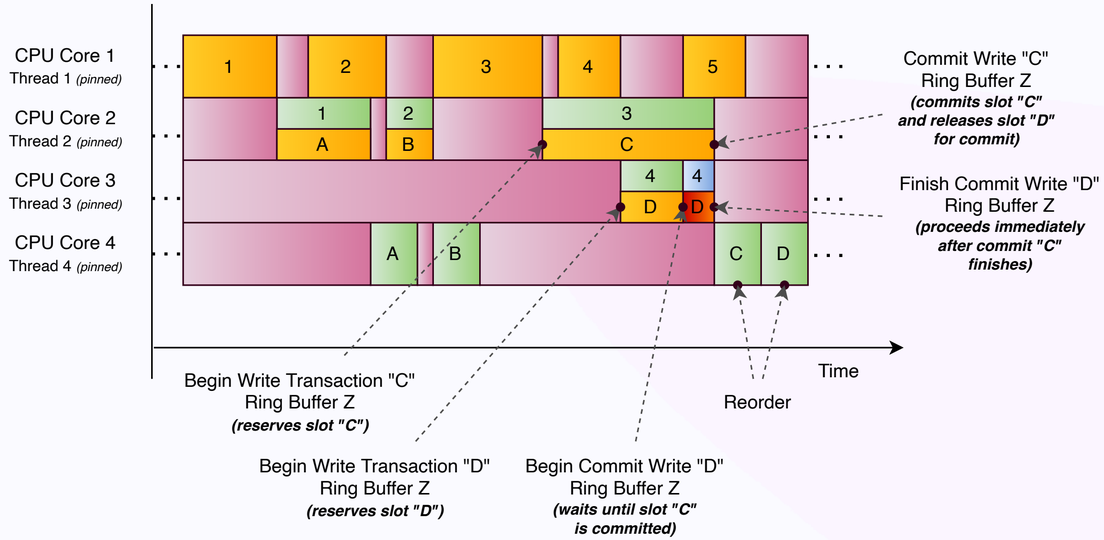
Split & Merge

On this example CPU Core 1 runs thread that receives and parses Market Data, and then two CPU Cores 2 and 3 run threads that analyse parsed data, and then CPU Core 4 runs thread that merges results.

On the time diagram we can see that while thread running on CPU Core 3 processed packet 4 before thread running on CPU Core 2 finished packet 3, it has to wait for thread running on CPU Core 2 to commit to slot “C” first, before committing to slot “D”.

We used here second ring-buffer here to merge results in-order.

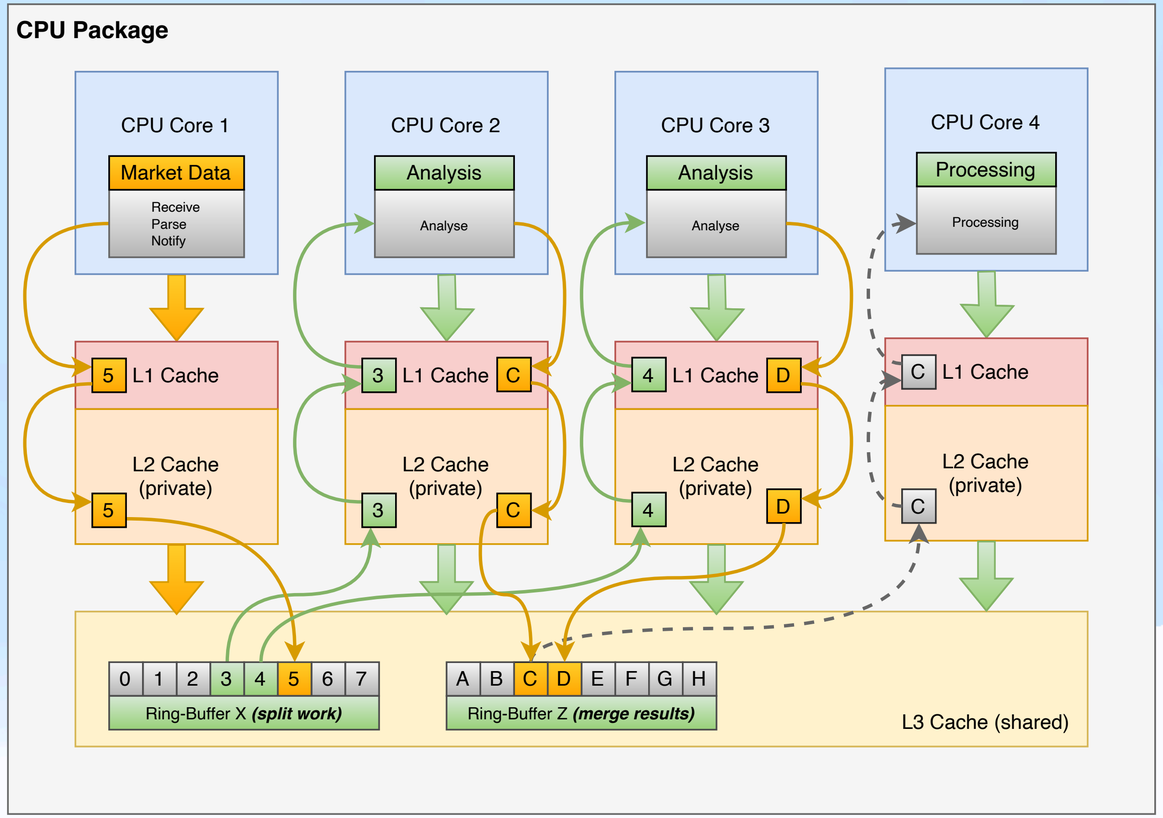


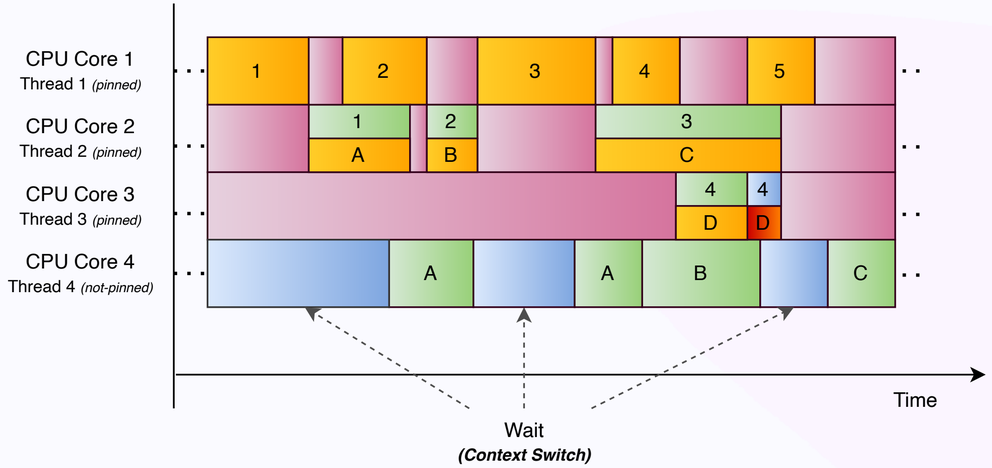


Pipelining 101

Mixed Low and High Latency

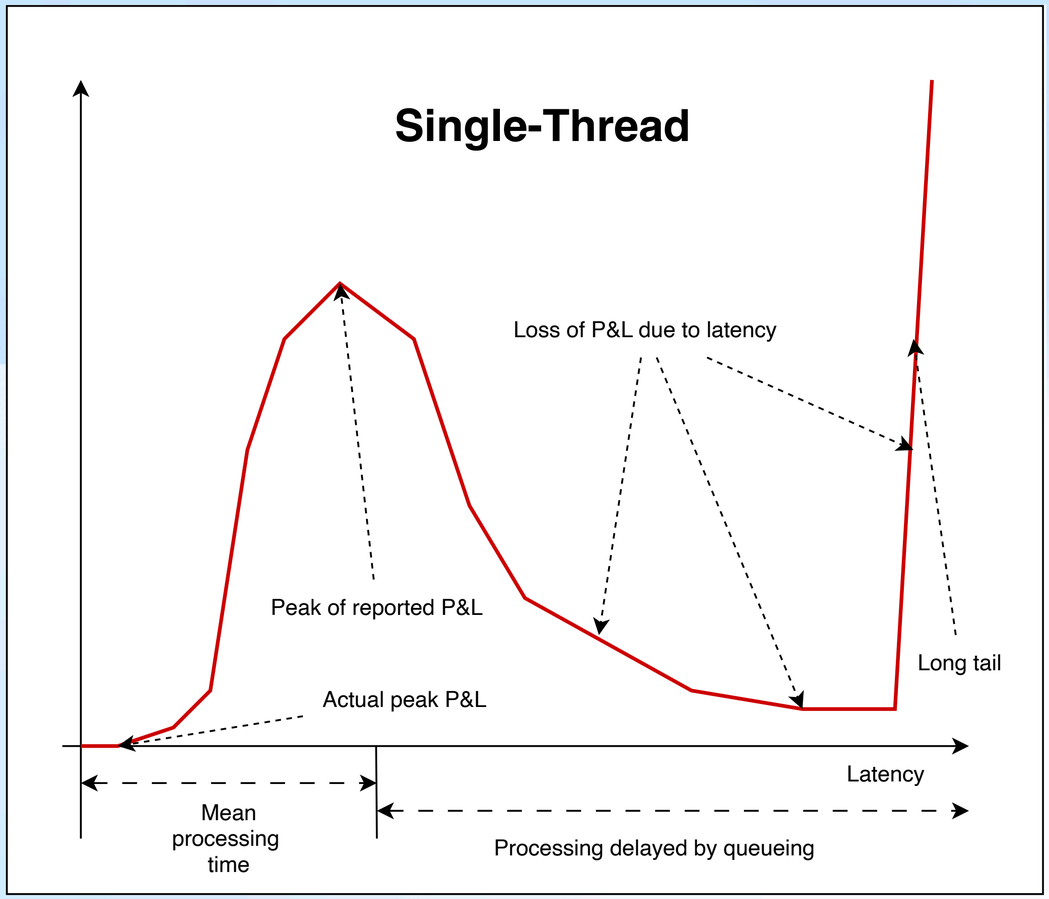
This is example is very similar to Split & Merge except that Thread 4 is not pinned to CPU Core 4, and also it run high latency mode.

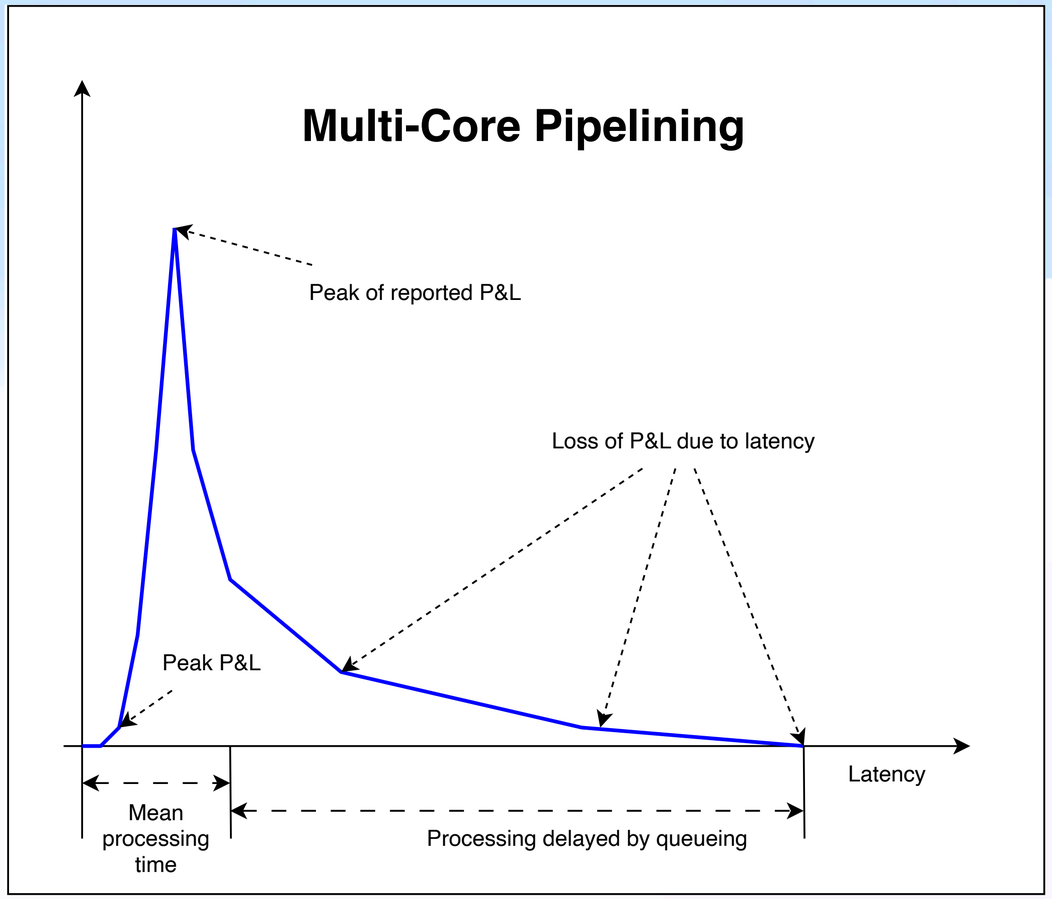




Pipelining 101

Profit & Loss (P&L) vs Latency





Pipelining 101

Cons & **Pros** of Pipelining?

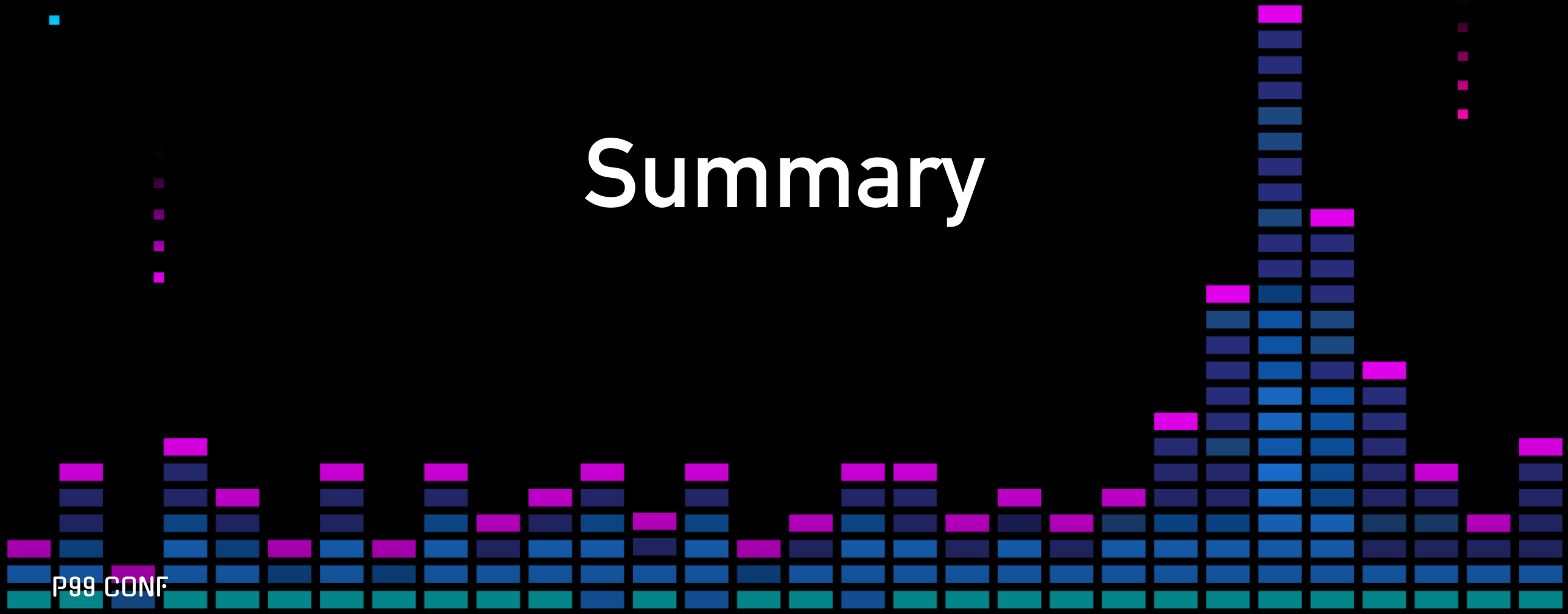
Pros:

- We get lower amortised latency, because we start processing next item, before we finish processing previous item, and thus next item does not need to wait to start
- We amortise processing jitter by performing jittery operations in second stage of the pipeline without affecting uniformity of the first stage
- We get better use of cache, as each CPU core runs smaller portion of the code, and accesses smaller number of variables

Cons:

- We introduce latency in communication between CPU cores:
  - Notification overhead: When code running on one CPU core needs to notify other CPU core of new data available
  - Wake-up overhead: When code running on the other CPU core awaits new data

Summary



Summary

Lock-Free and Wait-Free programming:

- Reduces thread synchronisation latency to very small nanosecond value range
- Avoids context switching and putting thread into inactive suspended state
- Avoids deadlocks as we know that one thread will always progress
- However it consumes CPU power when awaiting change of shared state

Pipelining takes advantage of multi-core CPU architecture and:

- Allows us to process higher amounts of packets in given unit of time
- Reduces jitter by amortising processing time
- Adds only small amount of latency within range of few nanoseconds

Monitor, Mutex and Condition Variable should be used on non-critical execution paths:

- Where ultra low latency is not a factor
- To save energy and allow other tasks to run smoothly

Low Latency Gal — presents — **Low Latency *Stuff***

Project version: `0.1`

Date: 2024-09-15

Project location: https://github.com/sadhbh-c0d3/cc-docs

Tool used for diagrams: https://draw.io

Thank you!

P99 CONF

Let's connect!

Sonia Sadhbh Kolasinska

GitHub:
https://github.com/sadhbh-c0d3

YouTube:
https://www.youtube.com/@soniakolasinska3850