The Design of A High-performance Tracing Library in Rust

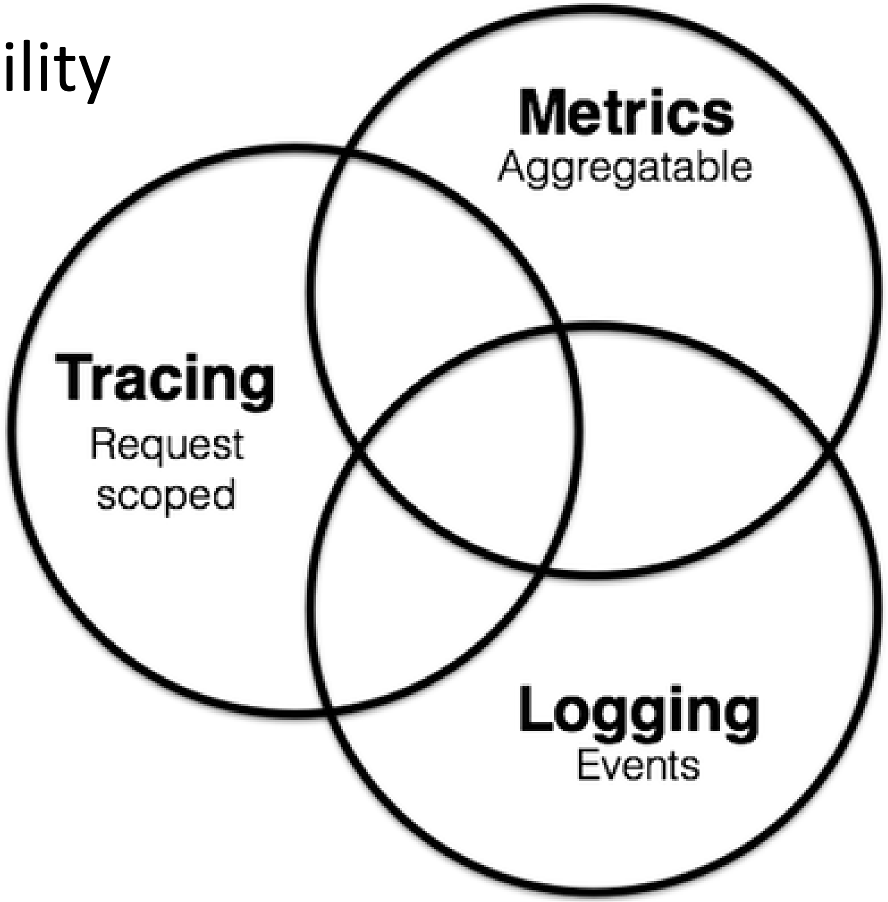
Zhenchi Zhong (钟镇炽) @ PingCAP

Who am I

- Infrastructure Engineer in PingCAP
  - PingCAP
    - TiDB
      - Distributed transactional relational database
      - HTAP, MySQL Compatibility…
    - TiKV
      - Distributed transactional key-value database
      - Support TiDB as storage engine
      - Developed in Rust
      - Graduated from Cloud Native Computing Foundation (CNCF)
- Mainly work on tracing for TiDB & TiKV

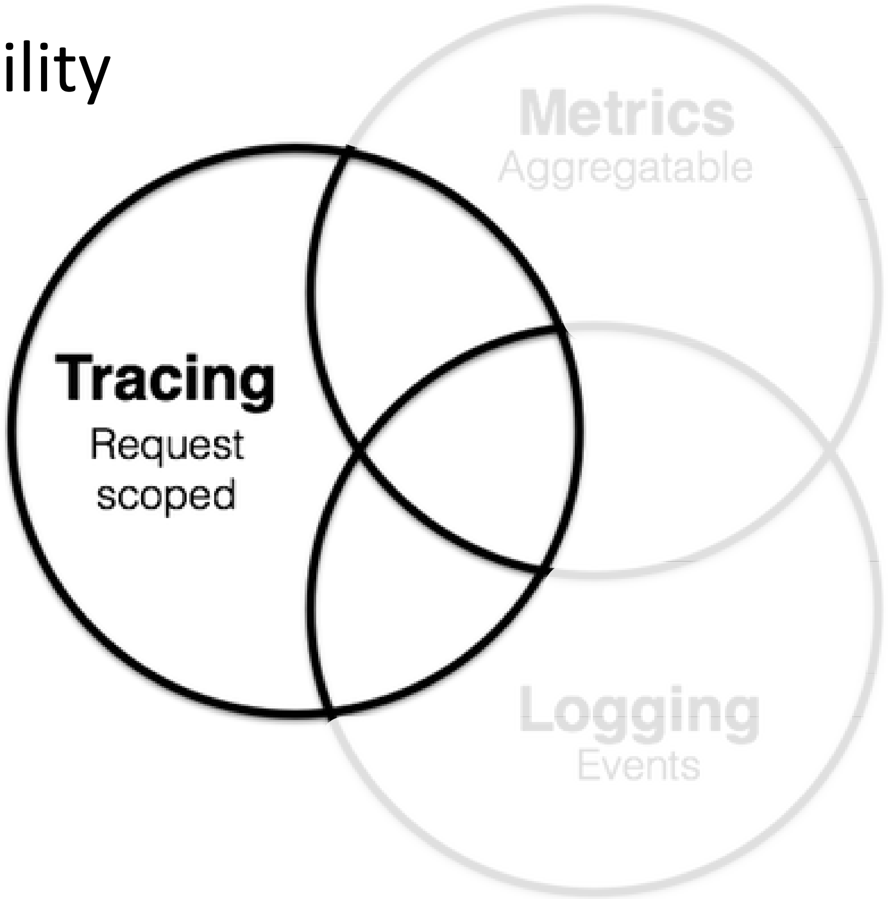
About Tracing

- One of the three pillars of observability



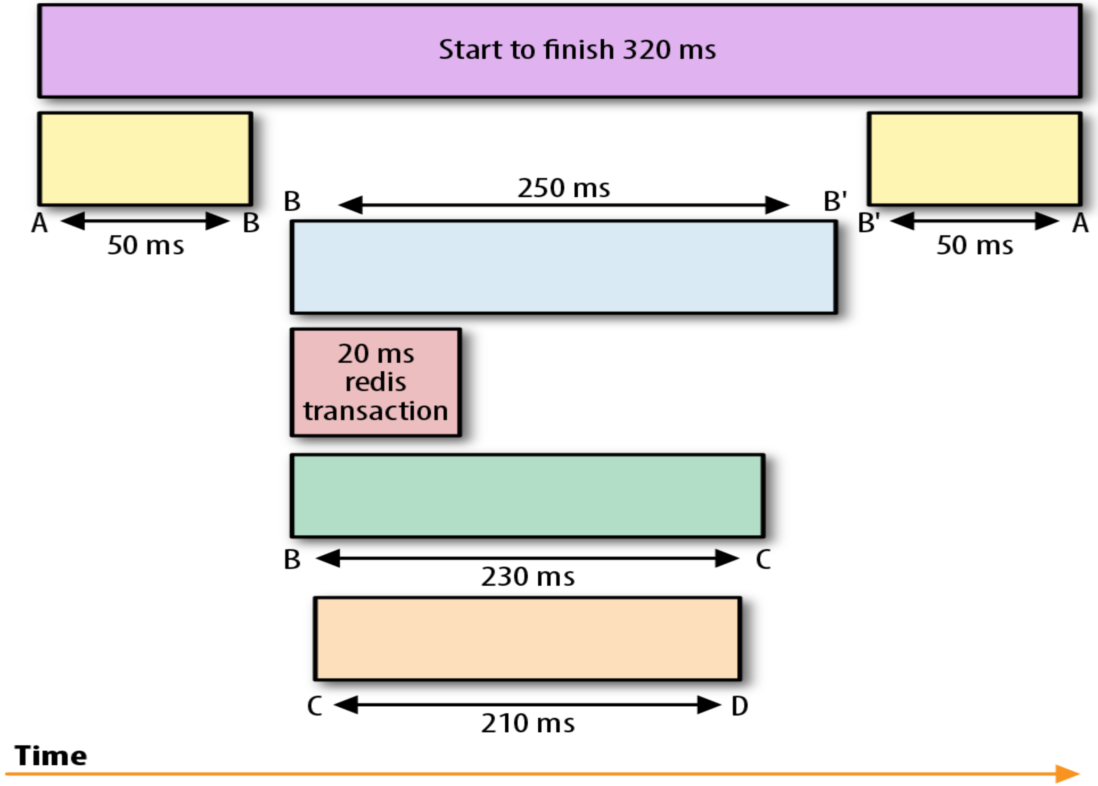
About Tracing

- One of the three pillars of observability
- Request scoped
- Use cases
  - Profiling
  - Debugging
  - Latency jitter analysis



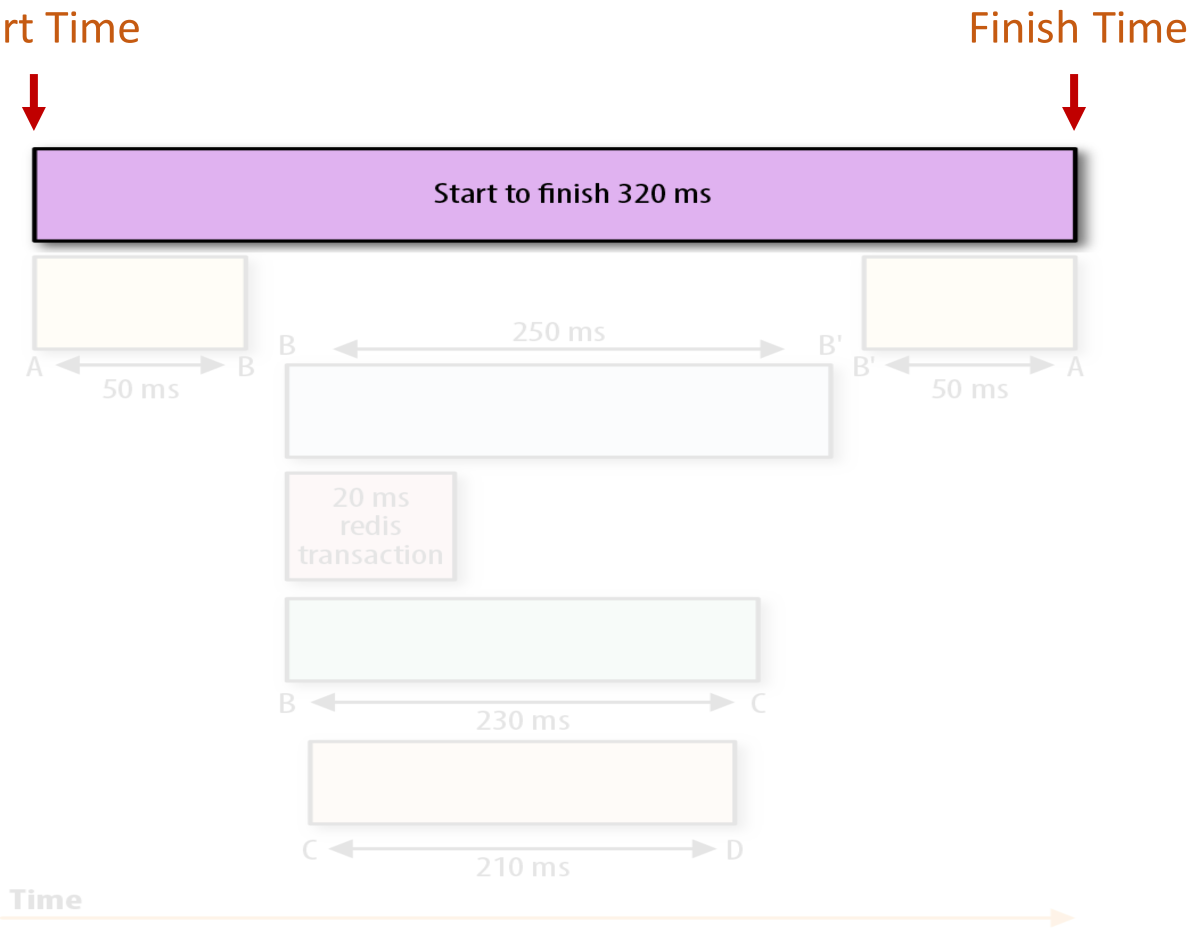
About Tracing

- Trace requests: HTTP, RPC…
- Across components
  - Redis
  - RDBMS
  - …



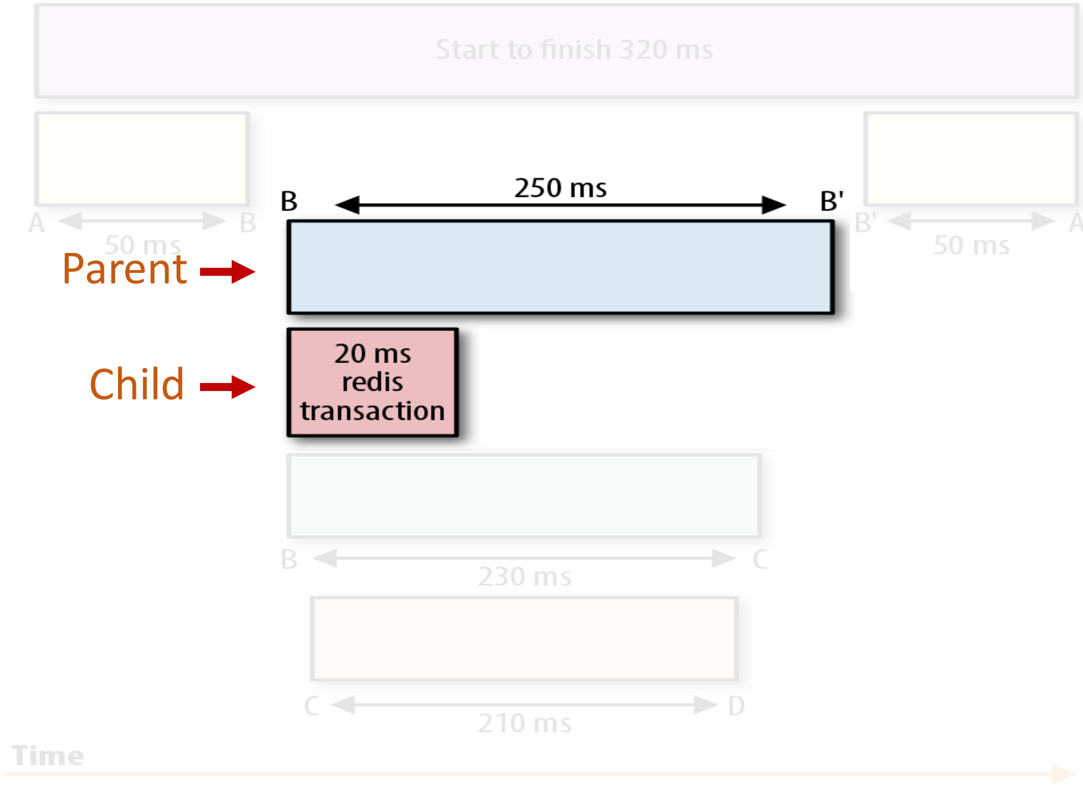
About Tracing

- Span
  - Start time
  - Finish time
  - Operation name



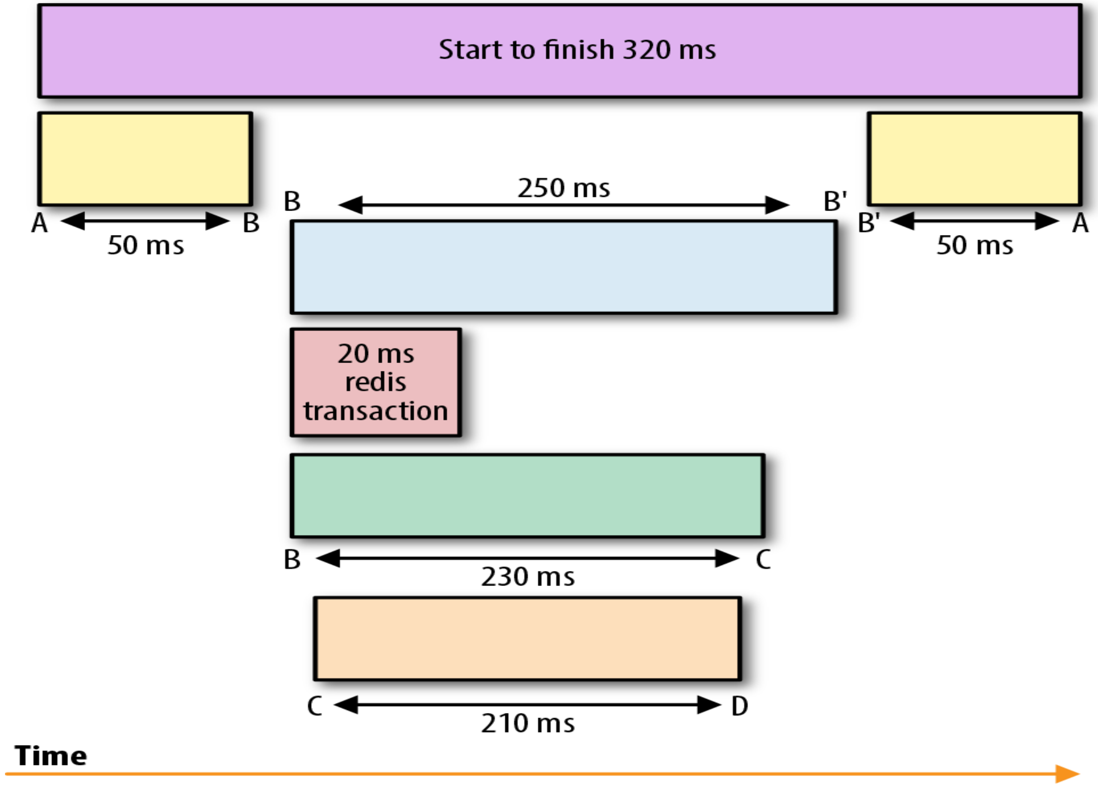
About Tracing

- Reference
  - Dependency
  - Parent-Child



About Tracing

- Reference
  - Dependency
  - Parent-Child



Early Try

- Plan
  - Choose a library from open-source community
    - Tokio tracing: https://github.com/tokio-rs/tracing.git
    - Rustracing: https://github.com/sile/rustracing.git
  - Apply it
  - Happy ending 🤔

Early Try

- Plan
  - Choose a library from open-source community
    - Tokio tracing: https://github.com/tokio-rs/tracing.git
    - Rustracing: https://github.com/sile/rustracing.git
  - Apply it
  - Happy ending 🤔

- Reality
  - QPS is reduced by 50% if enabled 😥

Expected Library

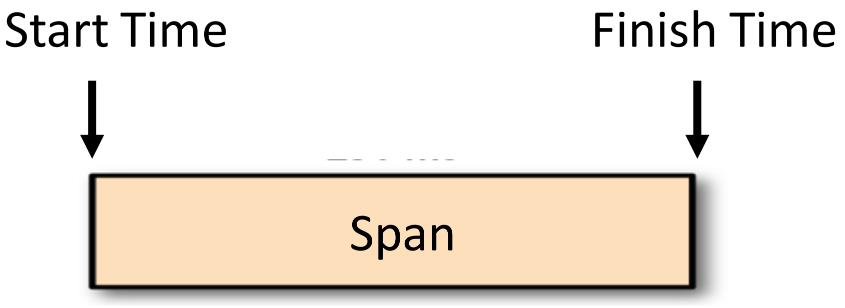
- High performance (Minimal overhead)
- Compatibility with OpenTracing
  - Jaeger, Datadog…

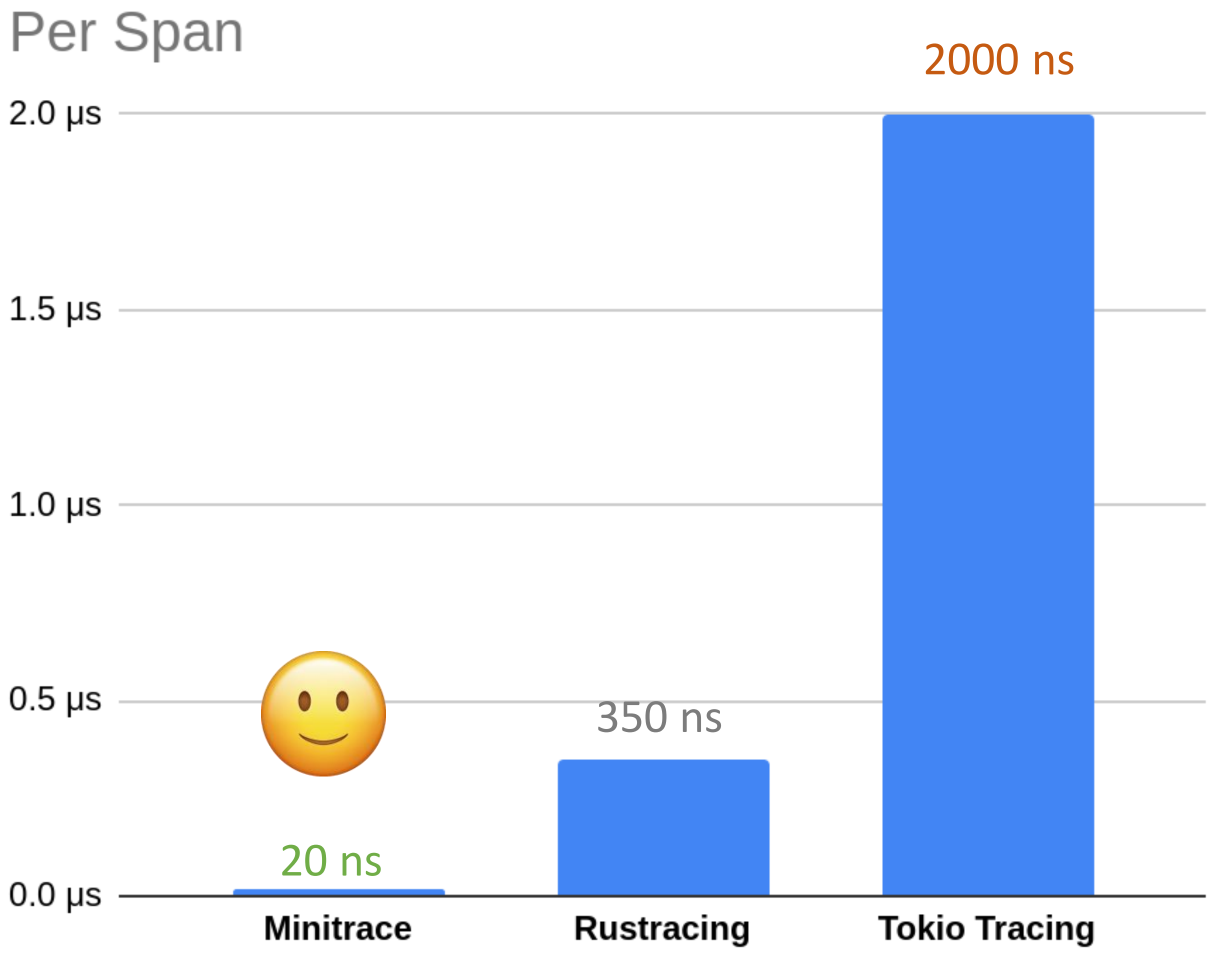
Expected Library

- High performance (Minimal overhead)
- Compatibility with OpenTracing
  - Jaeger, Datadog…

- Minitrace
  - https://github.com/tikv/minitrace-rust.git

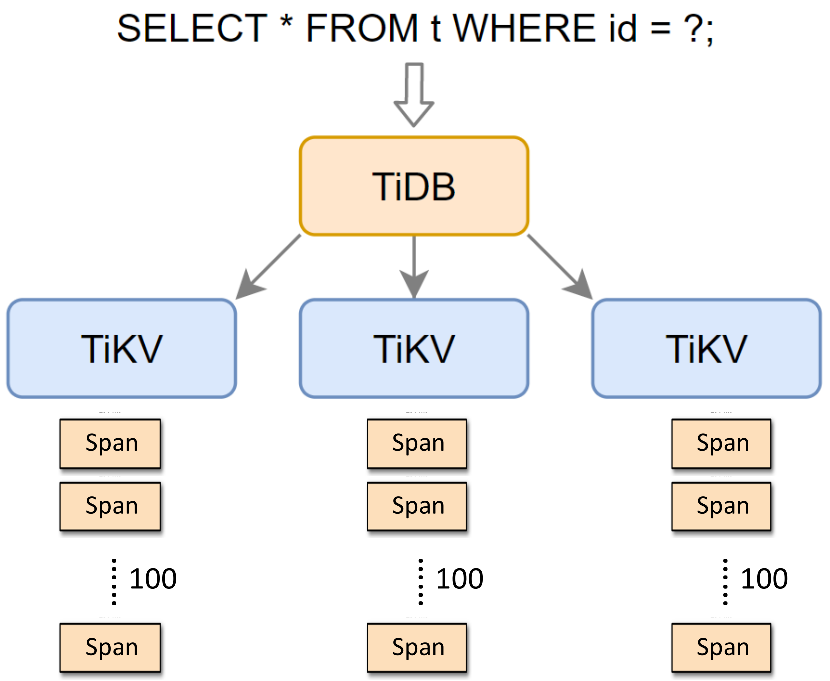
20ns!

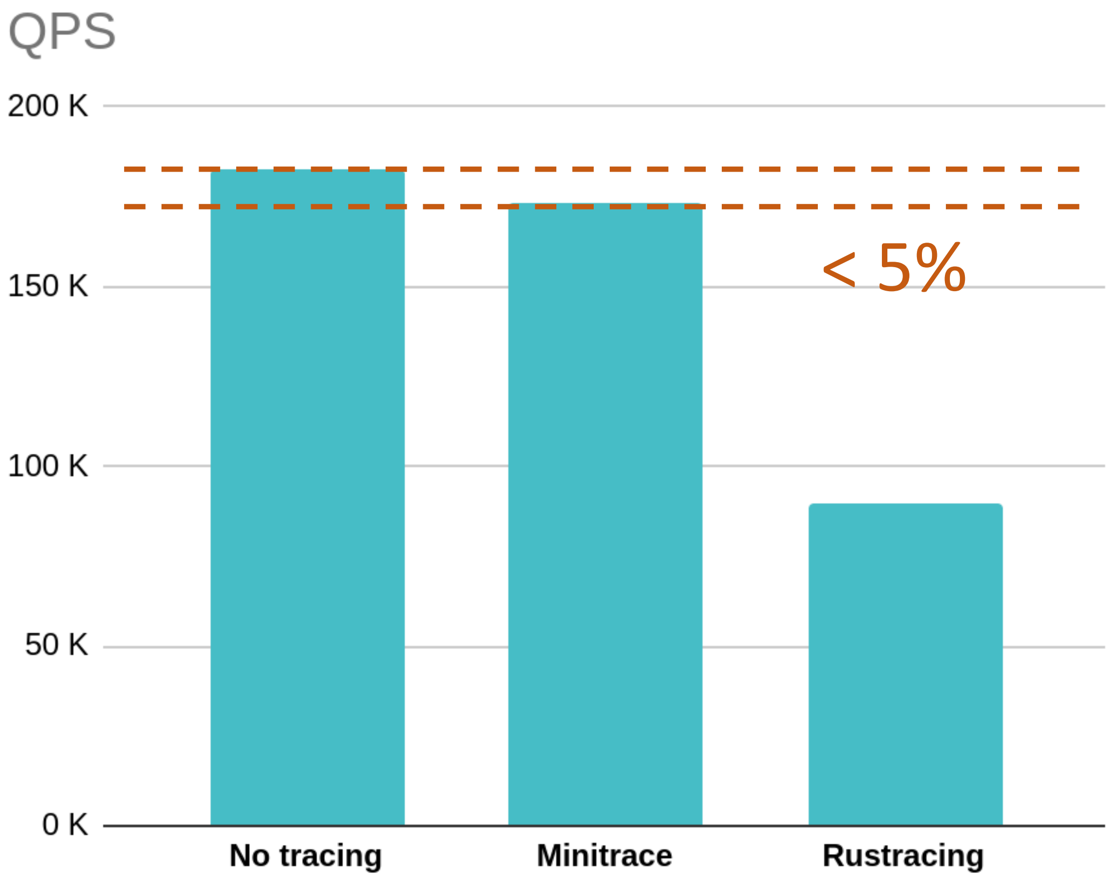




Micro benchmark (lower is better)

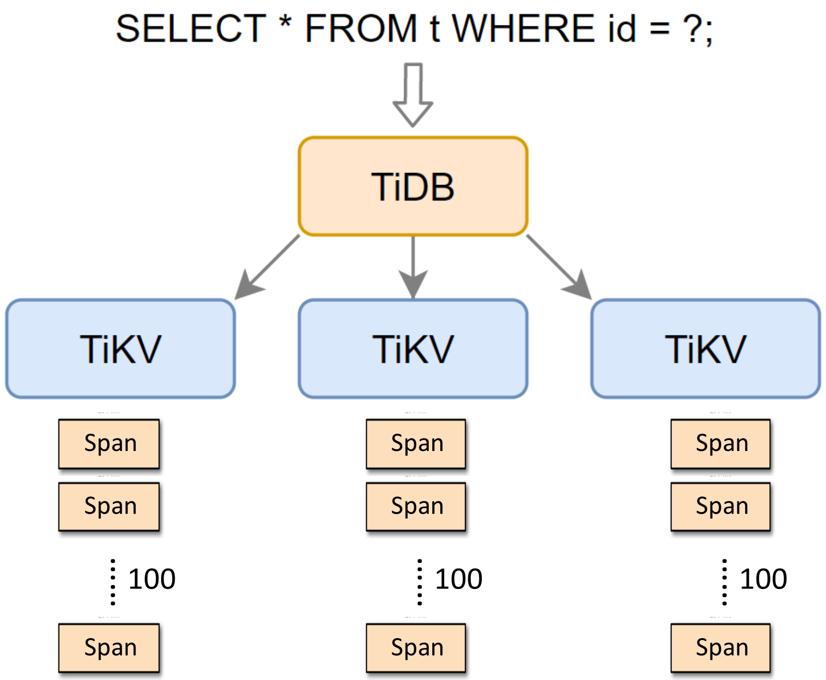
Overhead < 5%!

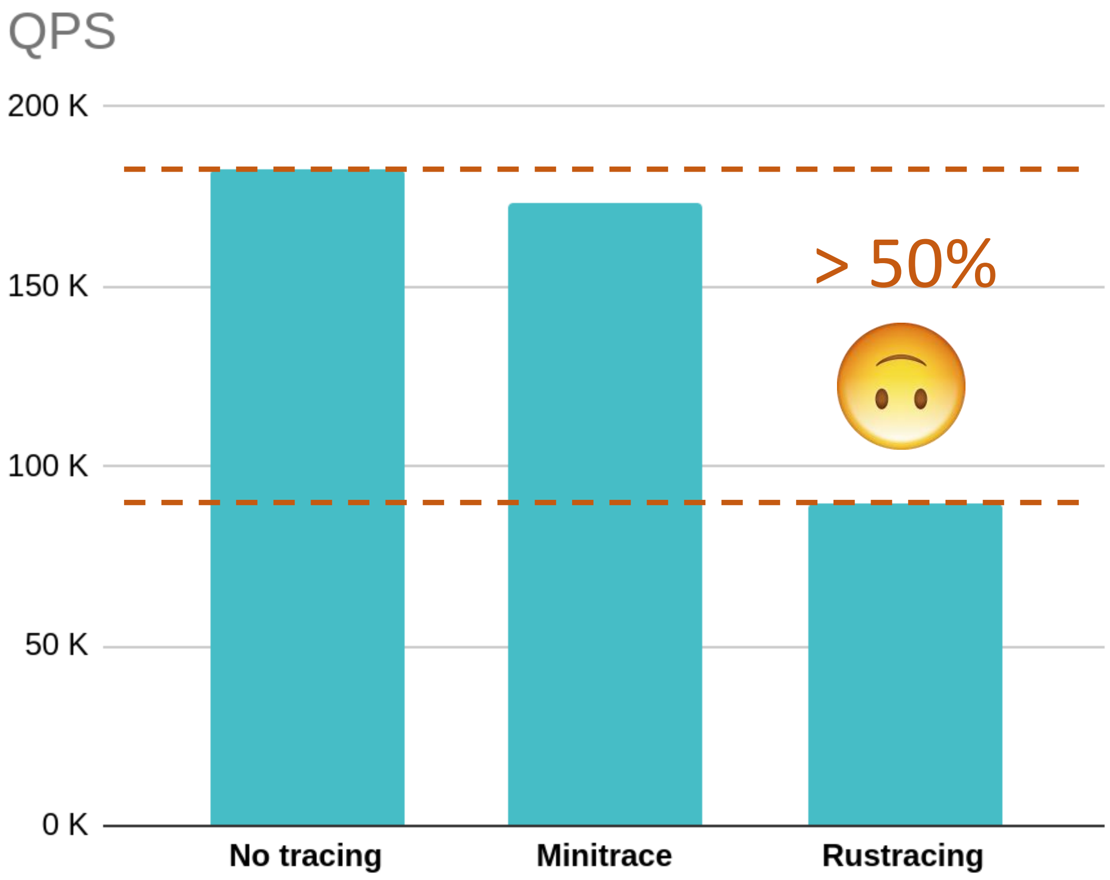




Integration benchmark (higher is better)

Overhead < 5%!

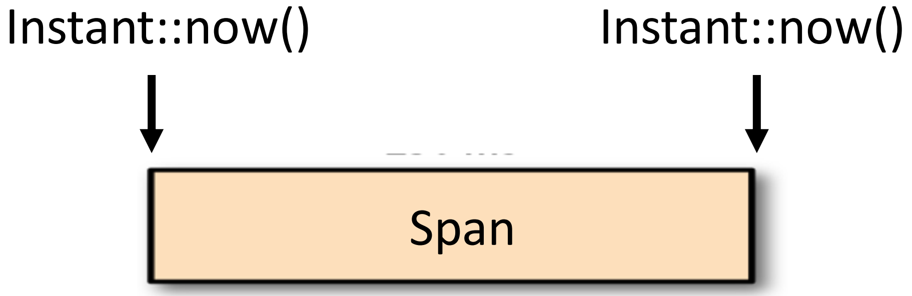




Integration benchmark (higher is better)

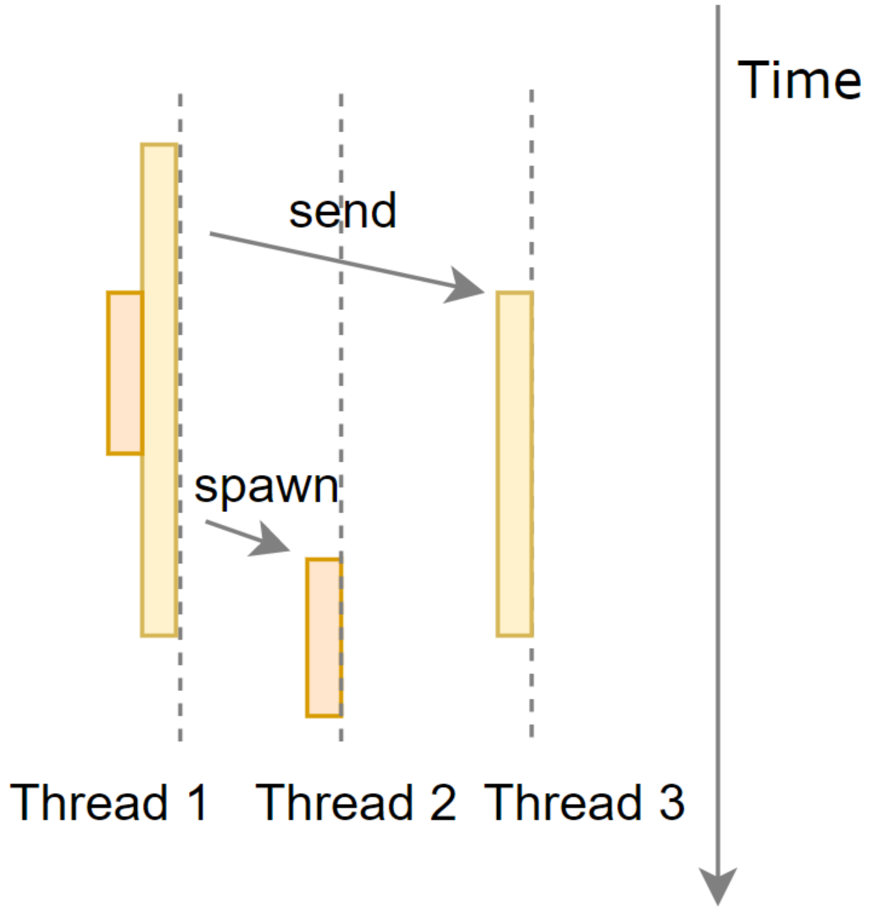
Naïve Implementation

- Timing approach
  - std::time::SystemTime::now()?
  - std::time::Instant::now()
  - One at started, one at finishing



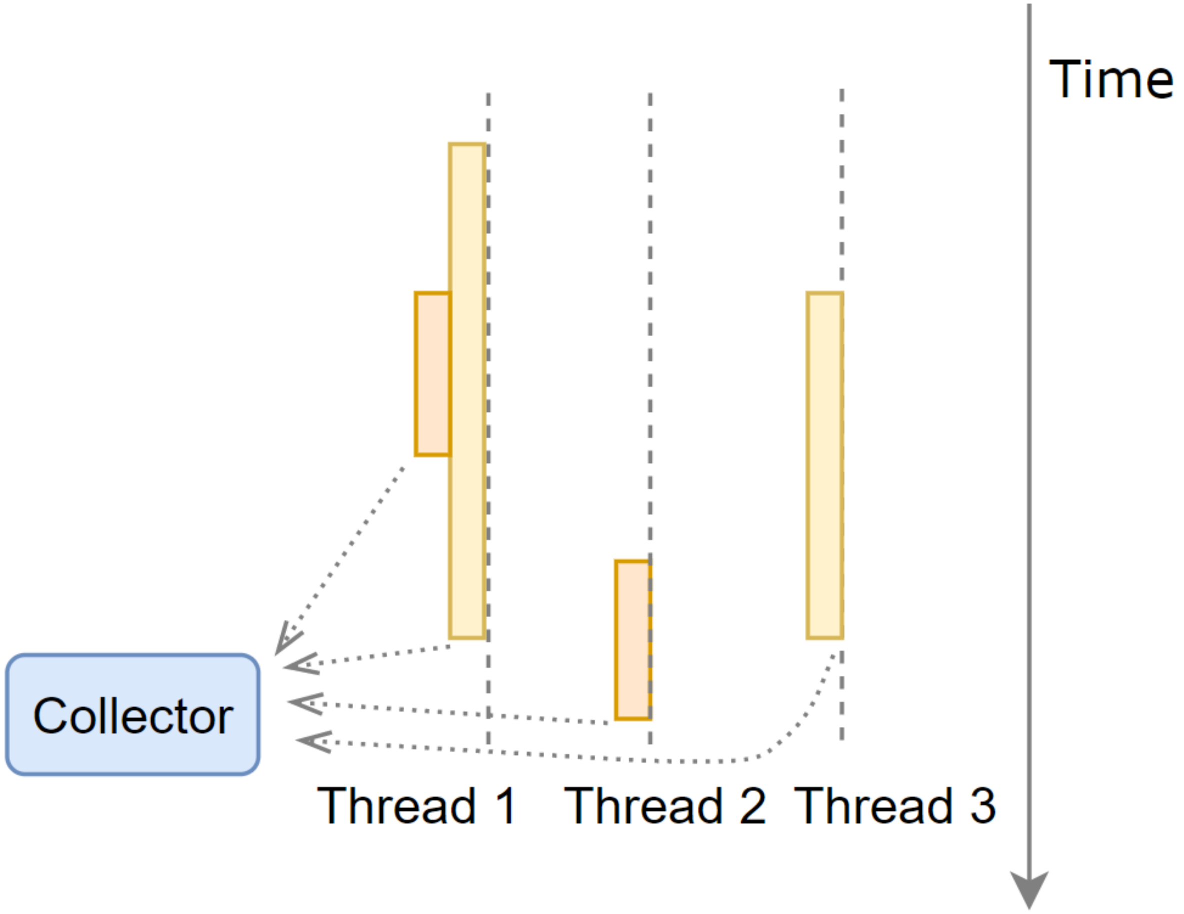
Naïve Implementation

- Timing approach
  - std::time::Instant::now()
  - One at started, one at finishing
- A shared collector
  - From multiple threads



Naïve Implementation

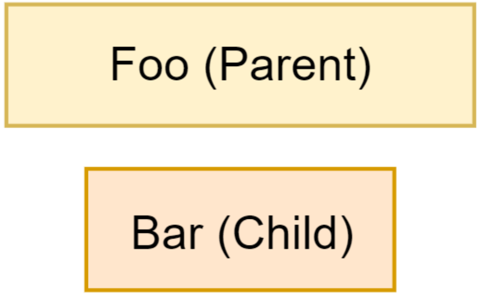
- Timing approach
  - std::time::Instant::now()
  - One at started, one at finishing
- A shared collector
  - From multiple threads
  - std::sync::mpsc::channel



Naïve Implementation

- Timing approach
  - std::time::Instant::now()
  - One at started, one at finishing
- A shared collector
  - From multiple threads
  - std::sync::mpsc::channel
- An explicit context
  - Build references

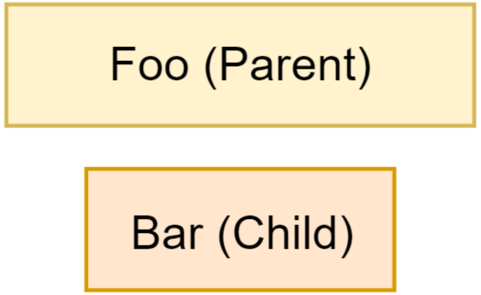
```
fn foo(           ) {

    bar(   );
}

fn bar(           ) {


}
```



Naïve Implementation

- Timing approach
  - std::time::Instant::now()
  - One at started, one at finishing
- A shared collector
  - From multiple threads
  - std::sync::mpsc::channel
- An explicit context
  - Build references
  - Update context when creating spans
  - Retrieve parent span from context

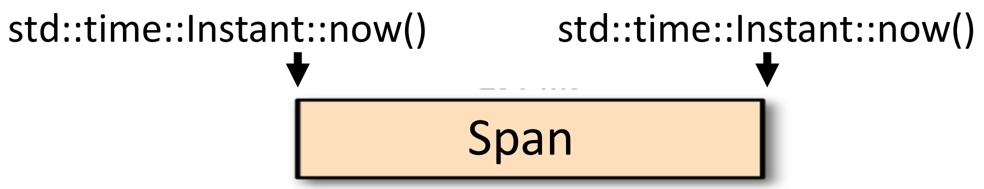
```rust
fn foo(ctx: &mut Context) {
    let span = ctx.create_span();
    bar(ctx);
}

fn bar(ctx: &mut Context) {
    let span = ctx.create_span();
}
```



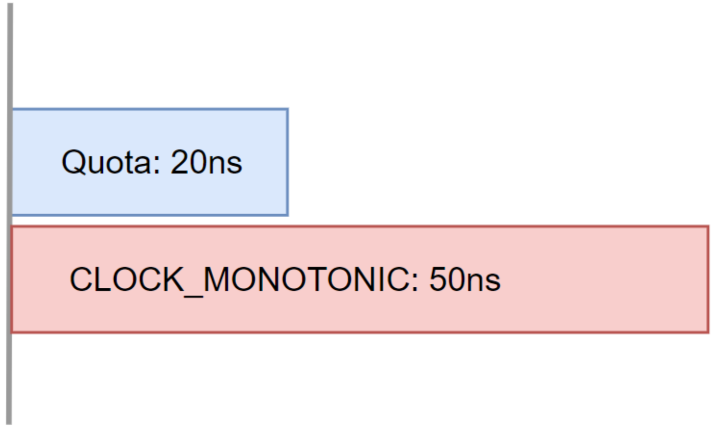
How to Improve

- Expected impact < 5%
  - < 20ns/span

How to Improve: Timing

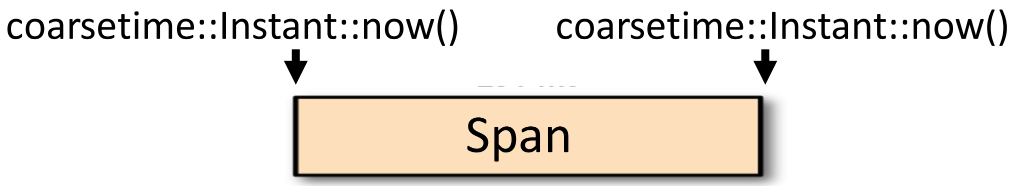


- Overhead of Timing
  - std::time::Instant::now()
  - Identical to clock_gettime(CLOCK_MONOTONIC, …) in Linux
  - 25 ns × 2

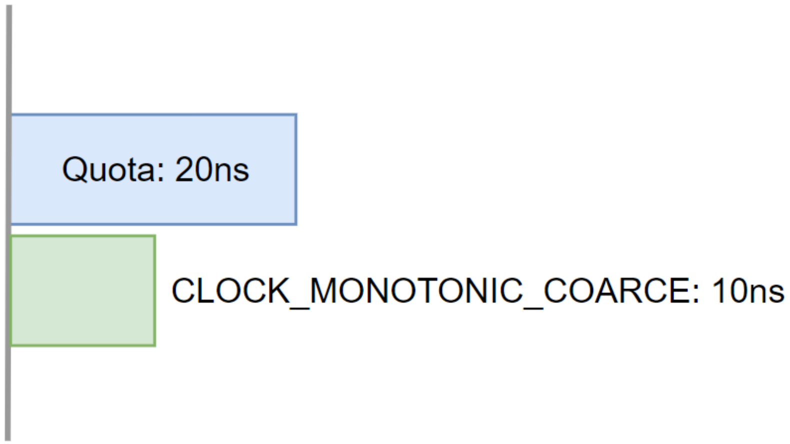


😥

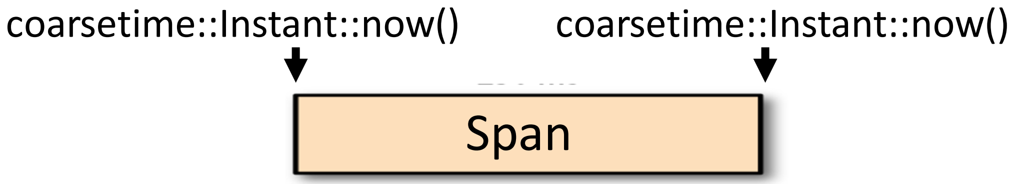
How to Improve: Timing



- An Option of Low Precision
  - coarsetime::Instant::now()
  - Identical to clock_gettime(CLOCK_MONOTONIC_COARCE, ...) in Linux
  - 5 ns × 2

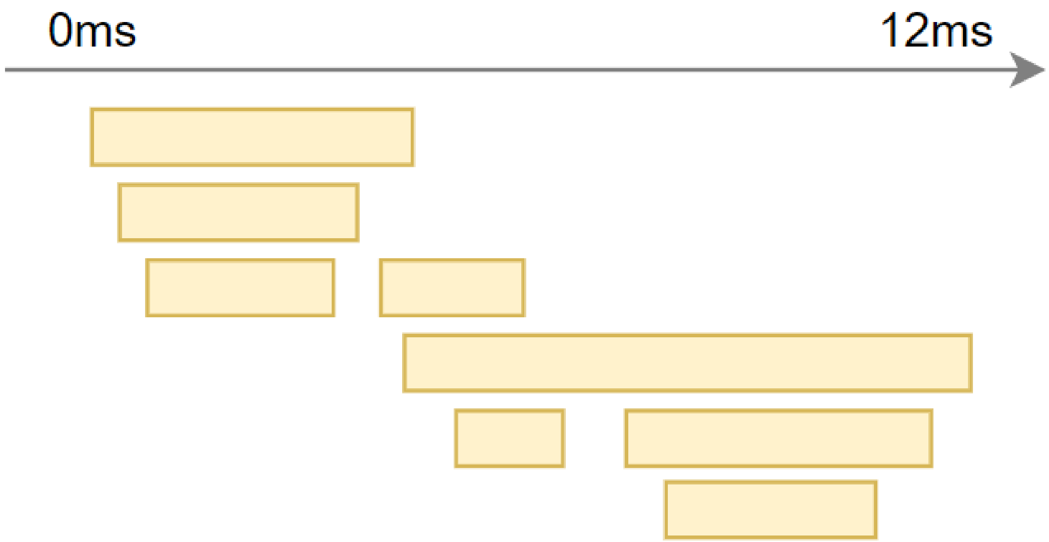


How to Improve: Timing

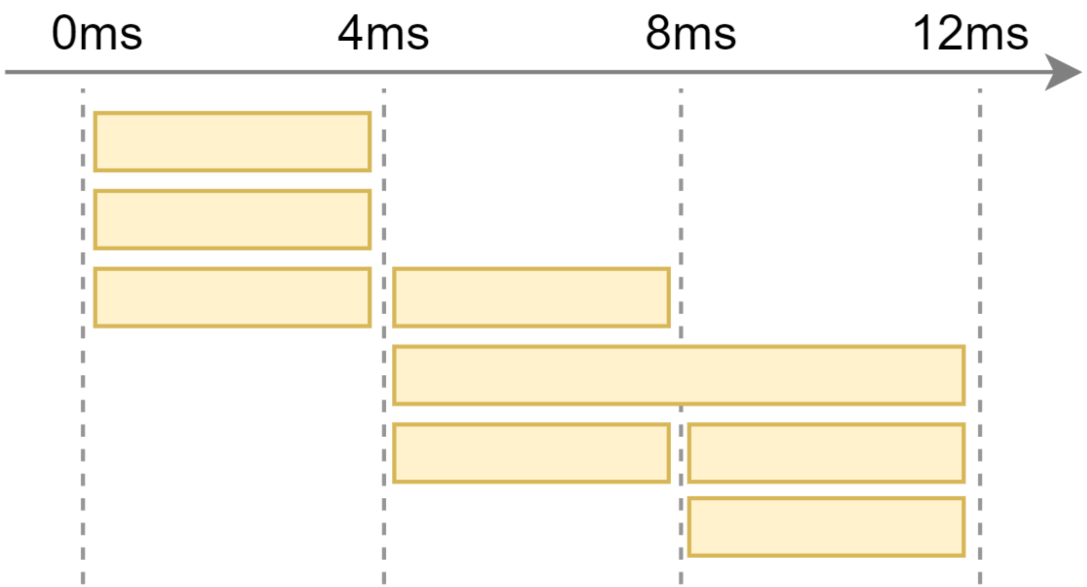


- An Option of Low Precision
  - coarsetime::Instant::now()
  - Identical to clock_gettime(CLOCK_MONOTONIC_COARCE, …) in Linux
  - 5 ns × 2
  - Kernel jiffy precision (4ms by default) 🤔

How to Improve: Timing



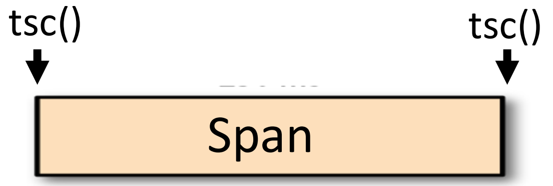
High Precision

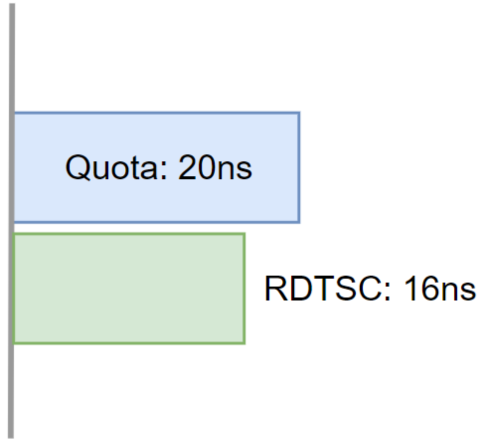


Precision of 4ms

How to Improve: Timing

- Better Choice on x86/x64
  - TimeStampCounter register
    - Increase per tick
  - 8 ns × 2
  - Nanoseconds precision

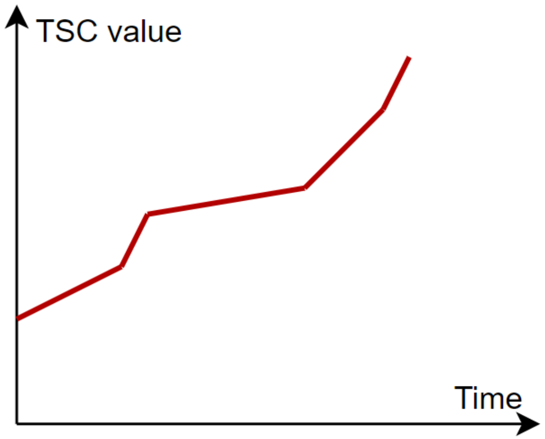




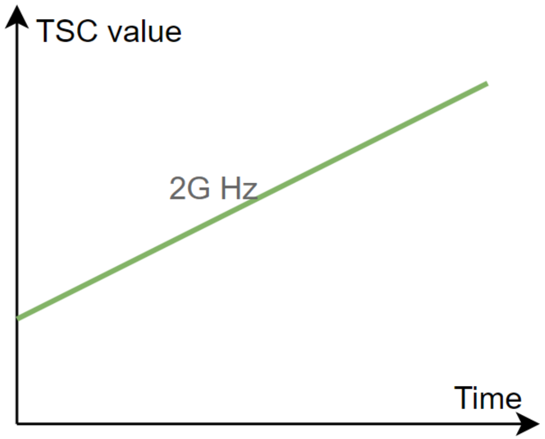
```rust
fn tsc() -> u64 {
    #[cfg(target_arch = "x86")]
    use core::arch::x86::_rdtsc;
    #[cfg(target_arch = "x86_64")]
    use core::arch::x86_64::_rdtsc;

    unsafe { _rdtsc() }
}
```

How to Improve: Timing

- TimeStamp Counter
  - constant_tsc



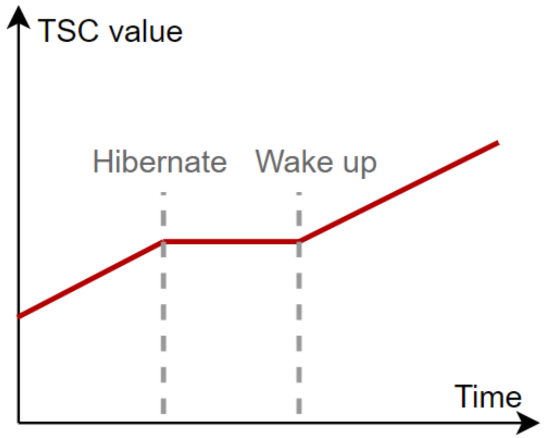
Without constant_tsc



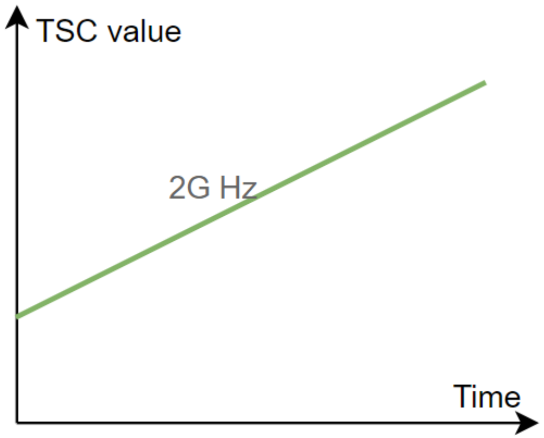
With constant_tsc

How to Improve: Timing

- TimeStamp Counter
  - constant_tsc
  - nonstop_tsc



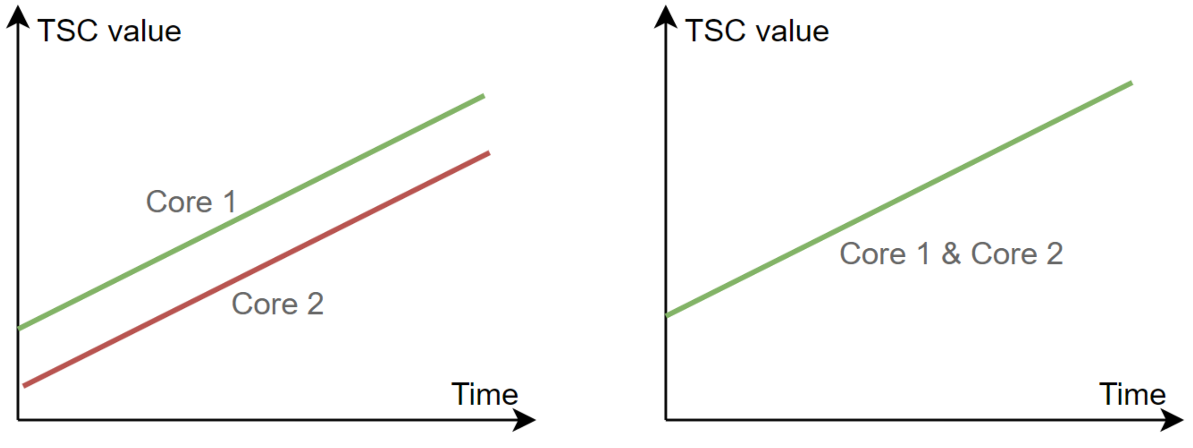
Without nonstop_tsc



With nonstop_tsc

How to Improve: Timing

- TimeStamp Counter
  - constant_tsc
  - nonstop_tsc
  - Unsynchronized among cores

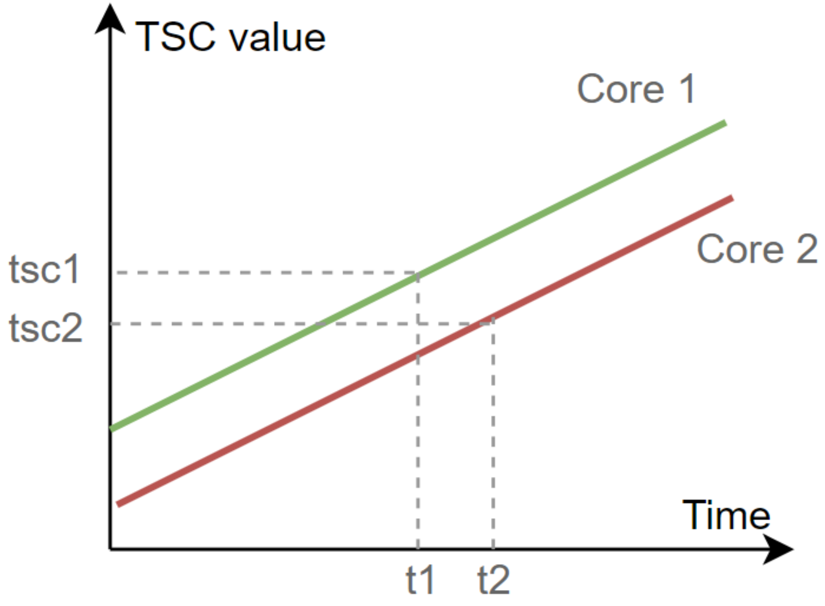


Unsynchronized

Synchronized

How to Improve: Timing

- TimeStamp Counter
  - constant_tsc
  - nonstop_tsc
  - Unsynchronized among cores
    - Threads scheduling

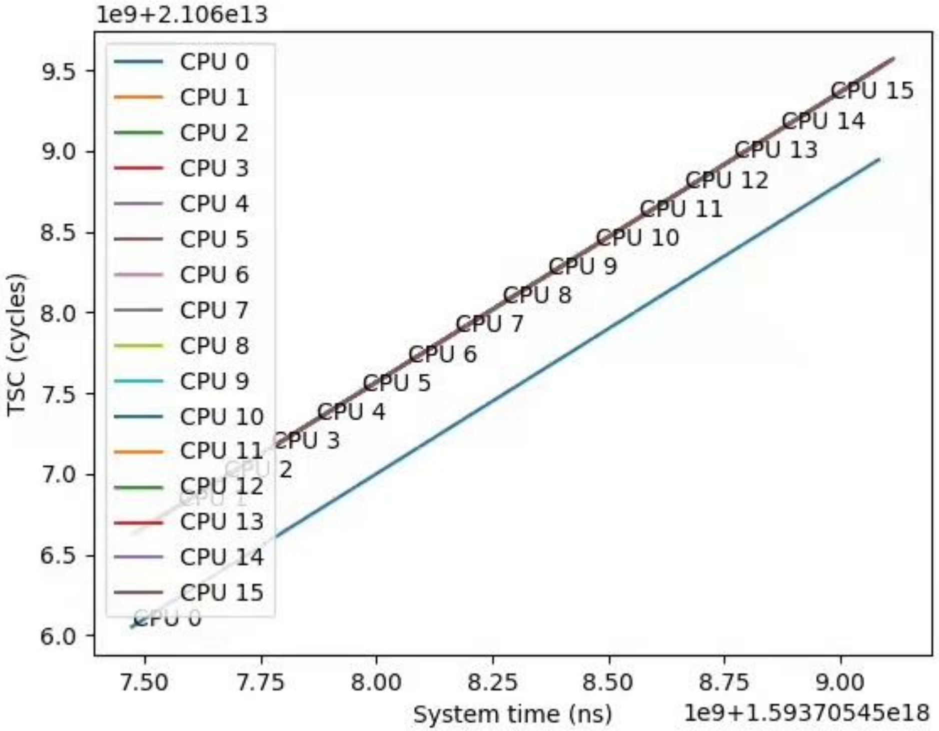


t1 < t2

tsc1 > tsc2

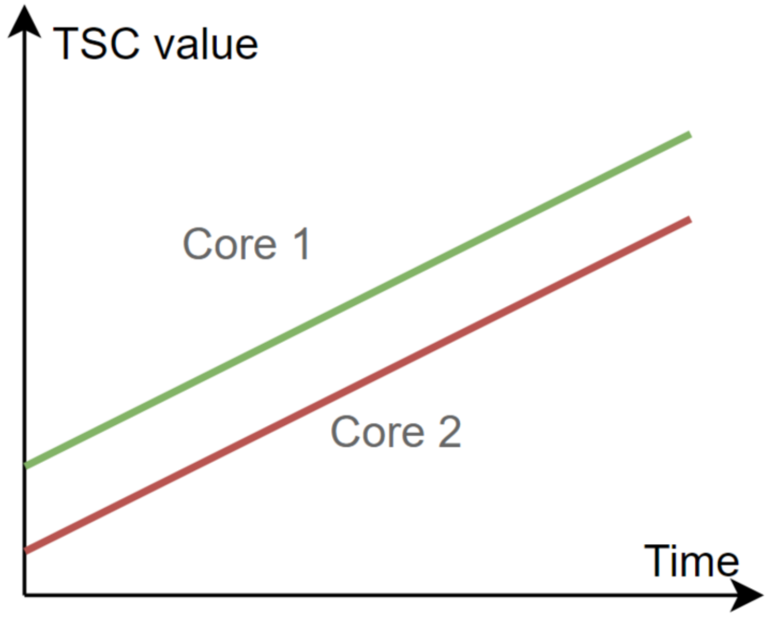
How to Improve: Timing

- TimeStamp Counter
  - constant_tsc
  - nonstop_tsc
  - Unsynchronized among cores
    - Threads scheduling



How to Improve: Timing

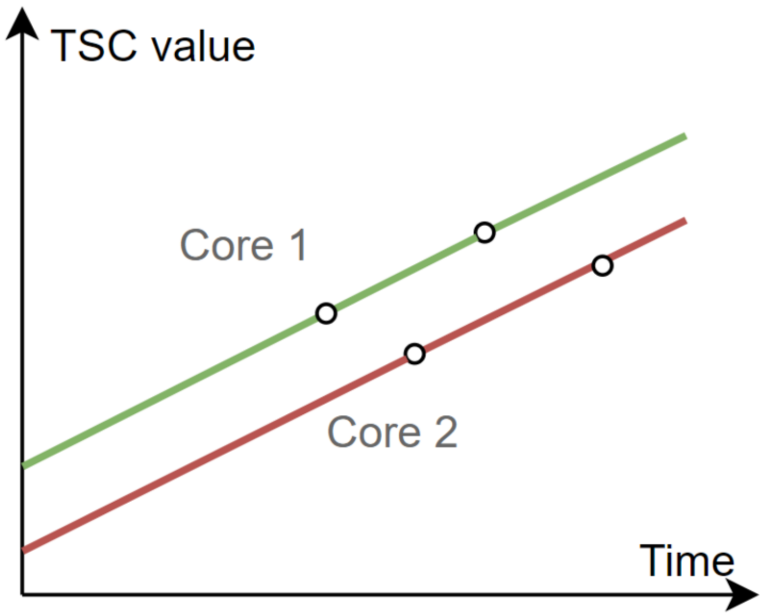
- Synchronize TSCs



Unsynchronized

How to Improve: Timing

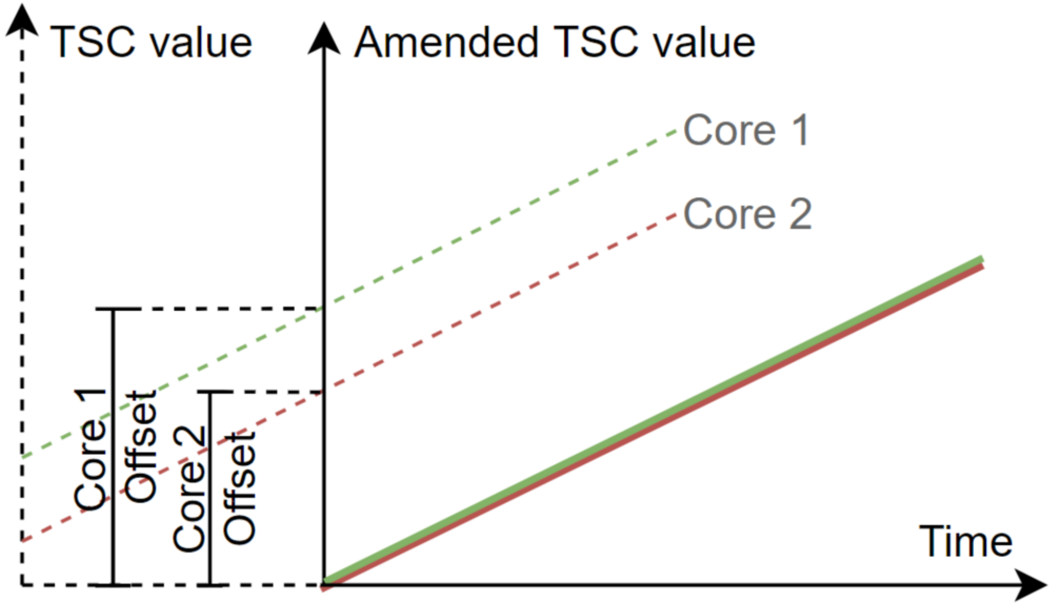
- Synchronize TSCs
  - Offset & Rate
    - libc::sched_setaffinity()
    - Retrieve tsc & systime twice



Unsynchronized

How to Improve: Timing

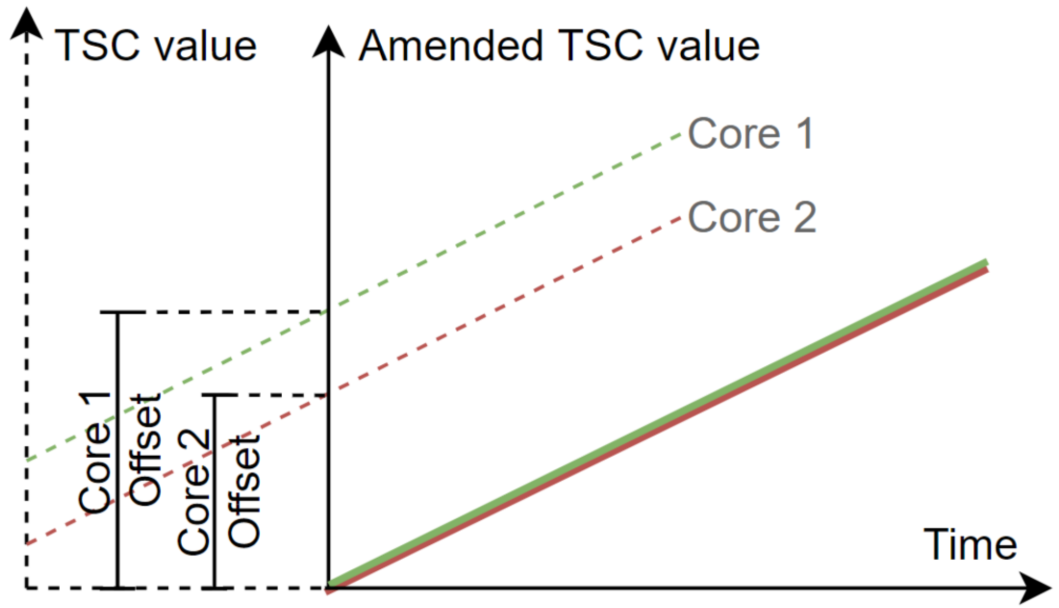
- Synchronize TSCs
  - Offset & Rate
    - libc::sched_setaffinity()
    - Retrieve tsc & systime twice
  - TSC + CPU ID
    - RDTSC + CPUID



Synchronizing

How to Improve: Timing

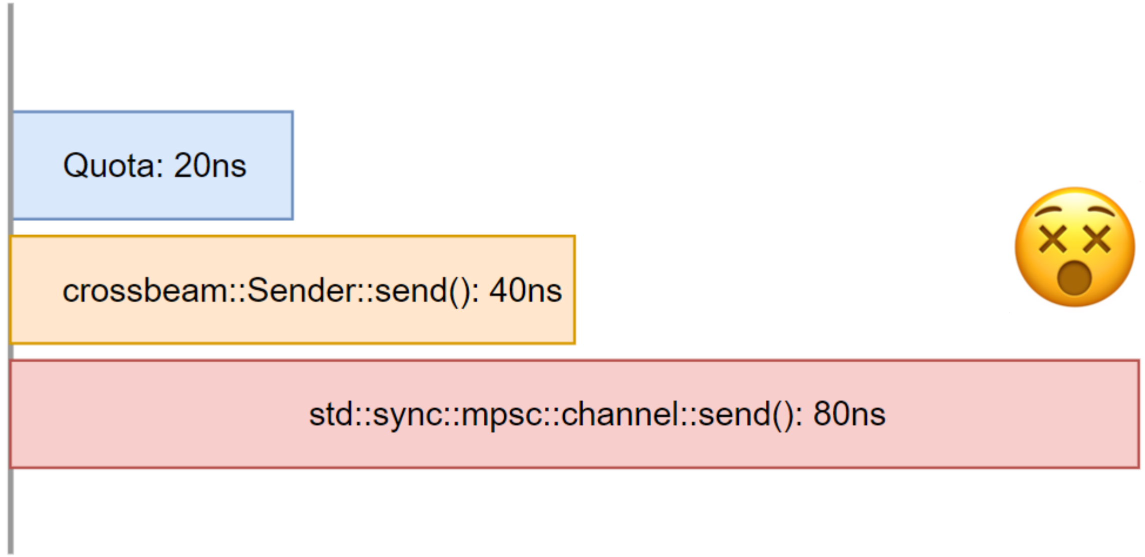
- Synchronize TSCs
  - Offset & Rate
    - libc::sched_setaffinity()
    - Retrieve tsc & systime twice
  - TSC + CPU ID
    - RDTSC + CPUID
    - RDTSCP



Synchronizing

How to Improve: Collection

- Overhead of Span Collection
  - Crossbeam channel
    - Based on atomic variables
    - Prevent compiler from optimizing
    - Unfriendly to CPU cache



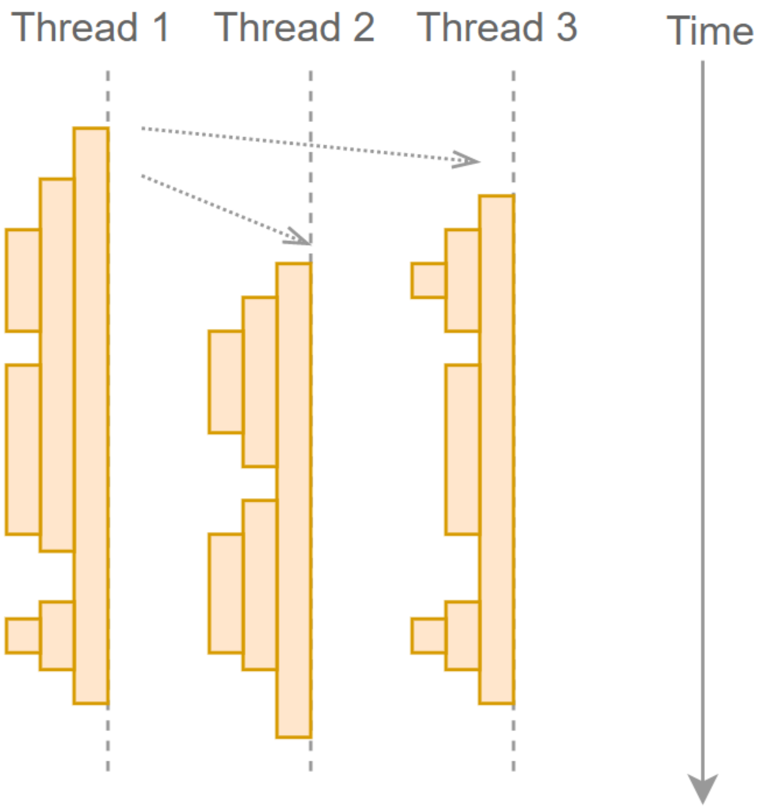
Quota: 20ns

crossbeam::Sender::send(): 40ns

std::sync::mpsc::channel::send(): 80ns

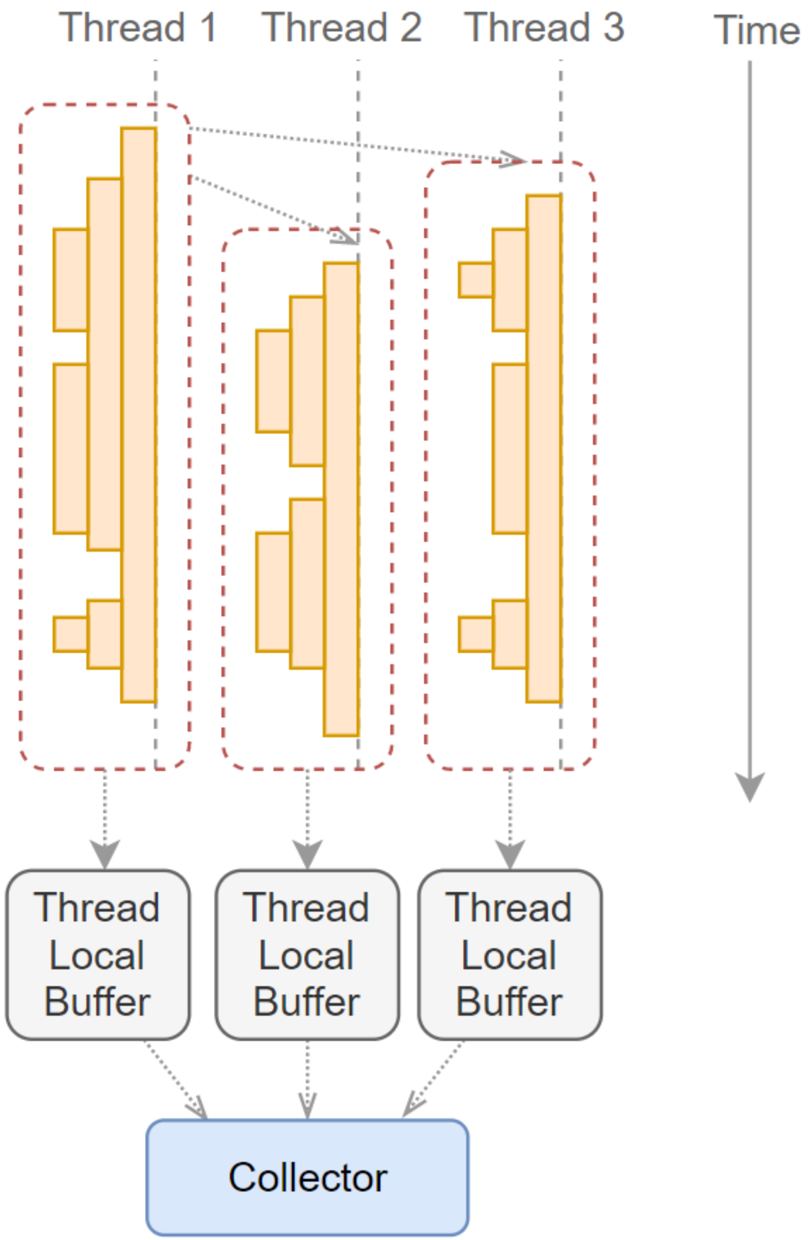
How to Improve: Collection

- Improvement: Local + Batch
  - Execution doesn’t switch threads all the time



How to Improve: Collection

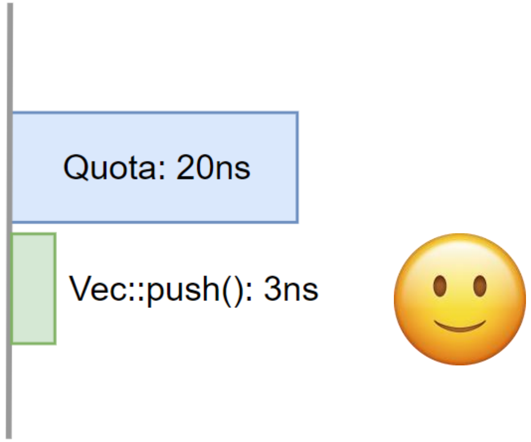
- Improvement: Local + Batch
  - Execution doesn’t switch threads all the time
  - Use thread local buffer, collect in batch



How to Improve: Collection

- Improvement: Local + Batch
  - Execution doesn’t switch threads all the time
  - Use thread local buffer, collect in batch

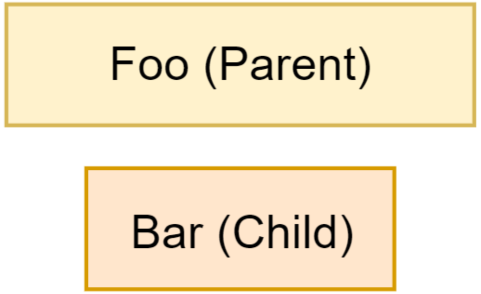
```
thread_local! {
    pub static BUFFER: RefCell<Vec<Span>> = RefCell::new(vec![]);
}
```



Minitrace Usage

- Implicit Context

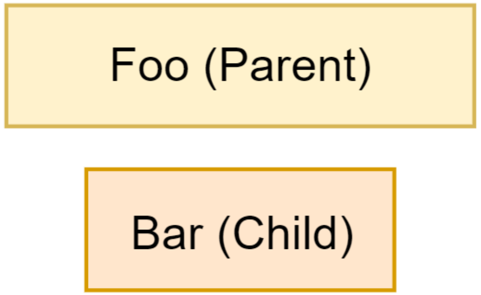
```
fn foo(~~ctx: &mut Context~~) {
    let span = ~~ctx.~~create_span();
    bar(~~ctx~~);
}

fn bar(~~ctx: &mut Context~~) {
    let span = ~~ctx.~~create_span();
}
```



Minitrace Usage

- Implicit Context

```
  use minitrace::*;

  fn foo() {
+     let _guard = start_span("Foo");
      bar();
  }

  fn bar() {
+     let _guard = start_span("Bar");
  }
```



Foo (Parent)

Bar (Child)

Minitrace Usage

- Use Macros

```
  use minitrace::*;
  use minitrace_macro::*;

+ #[trace("Foo")]
  fn foo() {
      bar();
  }

+ #[trace("Bar")]
  fn bar() { }
```

Minitrace Usage

- Async Function

```
  use minitrace::*;
  use minitrace_macro::*;

+ #[trace_async("Async Foo")]
  async fn async_foo() {
      async_bar().await;
  }

+ #[trace_async("Async Bar")]
  async fn async_bar() { }
```

Except Performance

- Safety
  - Unsafe-free (except timing)
  - Thread-safe
    - Thread-local type: !Send, !Sync
- Compatibility with OpenTracing
  - Report to Jaeger and Datadog

Thanks

https://github.com/tikv/minitrace-rust.git
https://github.com/zhongzc
zhongzhenchi@pingcap.com