Meeting C++ 2022

Keeping Track of Your Deadlines in Time-Critical Systems

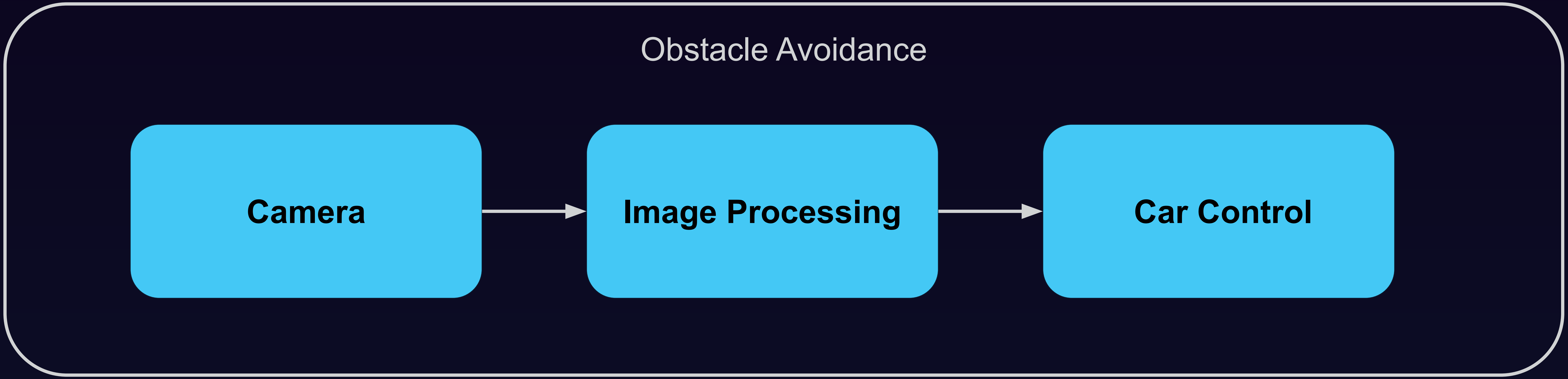
Matthias Killat

Keeping Track of Your Deadlines in Time-Critical Systems

Introduction

- Real-Time Systems
- Problem Statement
- Minimal Use Case

Time-Critical Systems



25 frames per second | Detect obstacle | Braking or steering decision

- Runs in a loop
- If we do not want to drop frames we need to process each frame in at most 40ms
- Deadlines like this are typical for many real word applications
- Modeled as real-time system

Incorrect results are useless
Correct but late results are also useless…

Real-time correctness also depends on the time at which the results are available.

Real-Time Systems

Deadline

- Time until some function must have completed
- Generally measured by wall-clock in practical applications

Real-Time System

- Impose deadlines on some functions
- Hard deadlines - must be met or it is a system failure
- Soft deadlines - failure to meet the deadline leads to degraded performance
- Often use multiple threads
- Requires OS with fair scheduling

Progress

- A function is said to make progress if it completes in finite time
- Deadlines are more strict: require progress within a concrete time span

Reasons for lack of progress

- Deadlock: programming error or partial system failure
- Starvation: scheduling or program logic error
- Priority inversion: special case of starvation

How Can We Ensure Deadlines Are Met?

Assume we want to build a Real-Time application in C++

1. Gather requirements and identify deadlines
2. Separate real-time algorithms and protocols
3. Avoid certain problems like priority inversion
    - Limit data sharing between threads
    - Limit context switches
    - Avoid locks (lock-free programming)
    - Careful thread priority assignment

Problem: How can we guarantee that deadlines are met by our implementation?

Depends on:

- Hardware
- Operating system and scheduler configuration
- Algorithm inputs (worst case)
- Other processes running in parallel

It is generally impossible to check at compile time whether deadlines are met.

Monitoring Deadlines at Runtime

Minimal version

```
#include "monitoring.hpp"
using namespace std::chrono_literals;

EXPECT_PROGRESS_IN(100ms);

time_critical_function(x, y);

CONFIRM_PROGRESS;
```

Syntax

- Simple to include in a project (header only)
- Use compact `std::chrono` literal syntax
- Should look distinct (e.g. like macros)
- Function style only if they take arguments
- Look and feel like Google test expectations

Semantics

- `EXPECT_PROGRESS_IN` starts a deadline section
- `CONFIRM_PROGRESS` marks the end of the deadline
- Acts like parentheses
- Any code in the deadline section is monitored
- If the deadline is not met (wall-clock), a handler is invoked

Naming functions is hard ...

Monitoring Deadlines at Runtime

What about multi-threading?

```
#include “monitoring.hpp”
using namespace std::chrono_literals;

// thread not yet monitored
START_THIS_THREAD_MONITORING;
// no active deadline
EXPECT_PROGRESS_IN(100ms, 73);
// deadline ID 73 is active
time_critical_function(x, y);
// more monitored code
CONFIRM_PROGRESS;
// no active deadline
STOP_THIS_THREAD_MONITORING;
```

Support for multiple threads

- Toggle monitoring for individual threads
- Interface refers to the current thread that is executing this code section

Checkpoint IDs

- Refer to a deadline by some checkpoint ID
- IDs must be managed externally (no duplicate detection)
- Multiple threads can execute the same section

Monitoring Deadlines at Runtime

How are deadline violations detected?

```
#include “monitoring.hpp”
using namespace std::chrono_literals;

// thread not yet monitored
START_THIS_THREAD_MONITORING;
// no active deadline
EXPECT_PROGRESS_IN(100ms, 73);
// deadline ID 73 is active
time_critical_function(x, y);
// more monitored code
CONFIRM_PROGRESS;
// no active deadline
STOP_THIS_THREAD_MONITORING;
```

Detection by the thread itself (Passive)

```
//pseudo code

//assume sufficiently accurate clock
auto timestamp = now();


EXPECT_PROGRESS_IN(100ms, 73)
  deadline_id = 73;
  start = now();
  deadline = start + 100ms;


CONFIRM_PROGRESS
  end = now();
  if(end > deadline)
    report_violation(deadline_id, end);
```

Monitoring Deadlines at Runtime

How are deadline violations detected?

```
#include "monitoring.hpp"
using namespace std::chrono_literals;

// thread not yet monitored
START_THIS_THREAD_MONITORING;
// no active deadline
EXPECT_PROGRESS_IN(100ms, 73);
// deadline ID 73 is active
time_critical_function(x, y);
// more monitored code
CONFIRM_PROGRESS;
// no active deadline
STOP_THIS_THREAD_MONITORING;
```

Challenges

- Where do we store the deadline for each thread?
- What type should the timestamp have?
- Timestamp overflow
- How do we compare timestamps?

What if the active thread does not make progress?

Monitoring Deadlines at Runtime

What if the thread does not make progress?

```
#include “monitoring.hpp”
using namespace std::chrono_literals;

// thread not yet monitored
START_THIS_THREAD_MONITORING;
// no active deadline
EXPECT_PROGRESS_IN(100ms, 73);
// deadline ID 73 is active
time_critical_function(x, y);
// more monitored code
CONFIRM_PROGRESS;
// no active deadline
STOP_THIS_THREAD_MONITORING;
```

Possible reasons for deadline violation

- Deadlock
- Starvation
- Deadline section takes unexpectedly long

The thread may never detect the violation or detect it too late.

Background monitoring thread

- Active monitoring (e.g. polling)
- High priority
- Low synchronization overhead

Monitoring by the thread itself is not sufficient.

Interesting problem …

How do we solve it efficiently?

Design

- Requirements and Constraints
- Performance Considerations
- System Overview

Requirements

Functionality

1. Detect deadline violations in multiple threads
2. Ensure violations are detected even in the case of deadlocks
3. Track deadline locations using checkpoint IDs and source location
4. Allow setting a custom deadline handler
5. Disable the monitoring completely or run only in passive mode
6. Optional mode to gather runtime statistics

Usage

- Easy to integrate in existing C++ code
- No additional dependencies apart from STL

```
// initialize and start active monitoring
START_ACTIVE_MONITORING
STOP_ACTIVE_MONITORING

// monitoring of the current thread
START_THIS_THREAD_MONITORING
STOP_THIS_THREAD_MONITORING

// establish a deadline
EXPECT_PROGRESS_IN(time, id)
CONFIRM_PROGRESS

//custom deadline handler
SET_MONITORING_HANDLER(handler)
UNSET_MONITORING_HANDLER
```

Performance Considerations

Performance

- Low influence on regular computation
- Minimal data sharing
- Lock-free happy path (no deadlines violated)
- Remaining lock contention should be low

Real-time safety

- Predictable response time with fair scheduling
- Avoid dynamic memory allocation
- No exceptions
- Avoid blocking unless required by design

Performance has to be considered early in design.

Restrictions

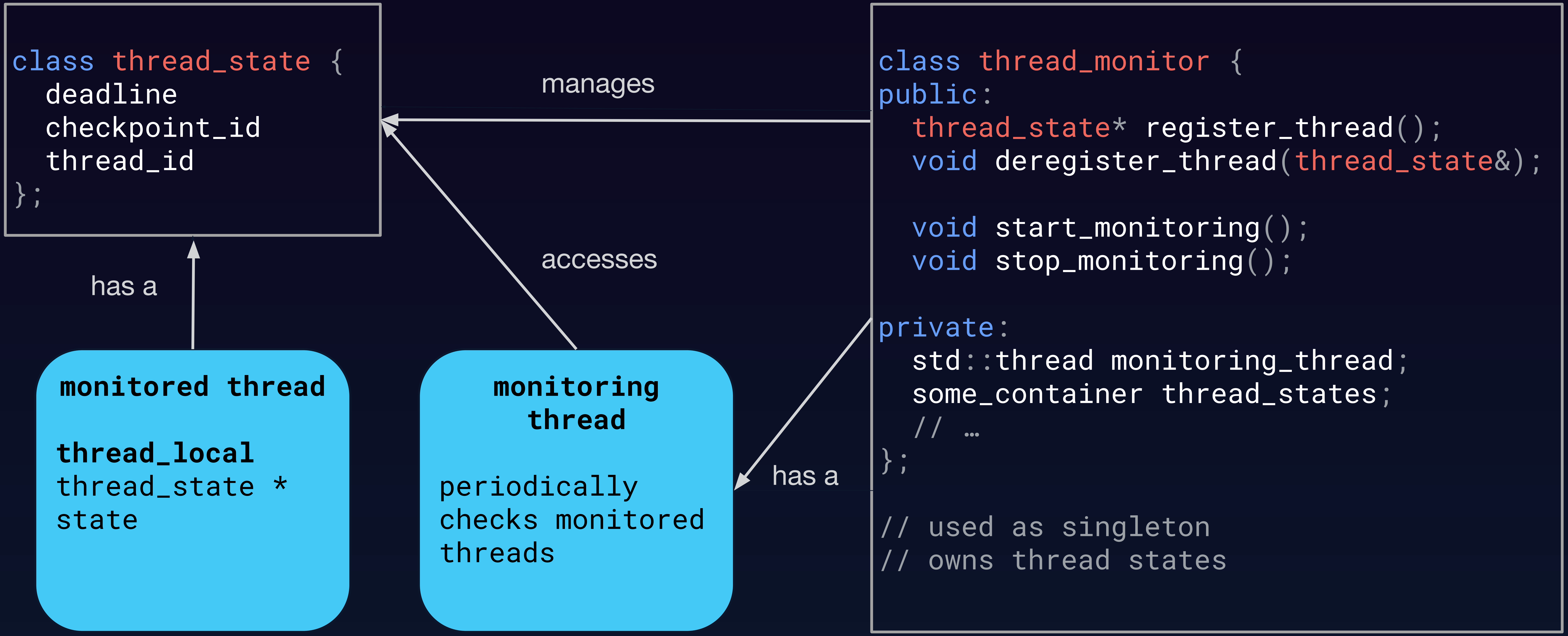
- Limit the number of monitored threads
- Deadlines have an upper limit
- Fast detection can only be ensured if the background thread gets sufficient priority and time to run
- No nested deadlines (e.g. across function calls)

```
EXPECT_PROGRESS_IN(300ms);
f(x, y);
EXPECT_PROGRESS_IN(100ms);
g(x, z);
CONFIRM_PROGRESS;
CONFIRM_PROGRESS;
```

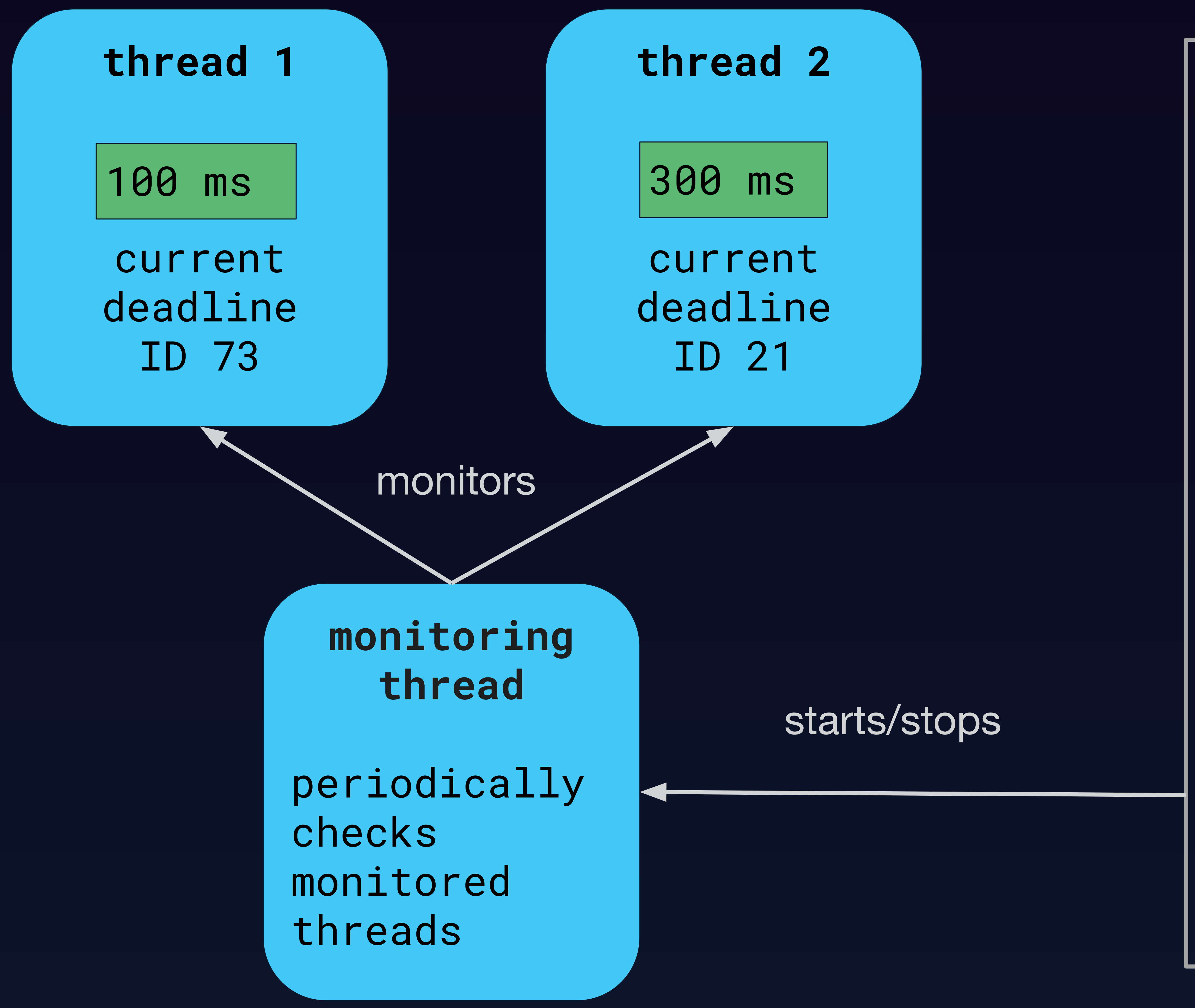
A more advanced version supports this

Advanced version supports nested deadlines.

System Design



System Design




```
class thread_monitor {
public:
  thread_state* register_thread();
  void deregister_thread(thread_state&);

  void start_monitoring();
  void stop_monitoring();

private:
  std::thread monitoring_thread;
  some_container thread_states;
};
```

Almost no lock-based synchronization between threads

That could really work!

Onward to the …

Implementation

- Checking Deadlines
- Measuring Time
- Nested Deadlines

Representing Deadlines

- Deadlines are time points
- Time points are durations measured relatively to some epoch
- Epoch is a fixed reference time like
  - 1st of January, 1970) or
  - system start
- Durations are number of ticks in some time unit (e.g. nanoseconds)

Constraint

- Representation should be small for atomic operations (e.g. 8 bytes)

Assume

- `time_t = uint64_t`
- Some function to get the current time: `time_t now();`
- Reasonable arithmetic
- Nanosecond ticks

We can represent ~293 years and do not care about time overflow for now.

Checking Deadlines

```
class thread_state {
  time_t deadline{0}
  id_t checkpoint_id{0}
  tid_t thread_id;
};

// properly initialized when thread
// is monitored
thread_local thread_state* ts;

EXPECT_PROGRESS_IN(time, id) :
ts->deadline = now() + time;
ts->deadline_id = id;
```

- Consider a deadline of `0` as invalid for now
- Store deadline in a thread local state
- Afterwards the deadline is valid

Limit thread interaction by using thread local scope.

Checking Deadlines

```
CONFIRM_PROGRESS :
// invalidate deadline
auto d = ts->deadline.exchange(0);
if(is_valid(d) && is_violated(d) {
  report_violation(ts);
}

bool is_valid(time_t d) {
  return d != 0;
}

bool is_violated(time_t d) {
  return now() > d;
}
```

- Background thread concurrently checks deadlines
- Deadline might have been invalidated by background thread
- Atomic exchange prevents multiple reports

We have to invalidate violated deadlines to avoid multiple violation reports.

Monitoring Thread

- Can access the thread state of each thread
- Reads and checks the deadlines (periodically or when needed)
- Deadline check is lock-free (atomic read)
- Deadline violation invokes a CAS operation to reset the deadline (Why?)

Interaction of two monitored thread

Thread 1 vs. Thread 2	Register Thread	Unregister Thread	Expect	Confirm	Monitoring Thread Check
Register Thread	Mutex	Mutex	None	None	Mutex
Unregister Thread	Mutex	Mutex	None	None	Mutex
Expect	None	None	None	None	Lock-free
Confirm	None	None	None	None	Lock-free

- Mutex contention is rare - only when monitoring of a thread starts or ends
- Lock-free interaction happens frequently
- Deadline reset via expensive CAS is also rare if deadline violation is rare

The happy path is lock-free.

Monitoring Thread

```
// check a thread deadline
void check_deadline(thread_state_t& ts) {
  auto d = ts->deadline.load();
  if(is_valid(d) && is_violated(d)) {
    // mark as reported
    if(ts->deadline.compare_exchange_strong
      (d,0)) {
      report_violation(ts);
    }
  }
}

// run periodically/if needed in a thread
void check_threads() {
  for(auto &ts : registered_thread_states) {
    check_deadline(ts);
  }
}
```

- Atomic deadline reset is required
- Requires CAS
  - After the thread loads d the deadline could be reset by the monitored thread
- Ensures not missing deadline violations
- Single report per violation

Monitoring thread runs periodically

- Could run on notification only but this requires notification
- Condition variables are relatively expensive
- No priority queue

Deadline Monitoring Example

Possible interleaving

Time (in ms)	Thread 1	Thread 2	Deadlines	Monitoring Thread (20ms)	Violation
0	EXPECT(15ms, 1)		d1(15)	Check nothing	
	f(x)		d1		
20		EXPECT(50ms, 2)	d1,d2(70)	Check d1, d2	d1(+5)
		g(y)	d1, d2		
40	CONFIRM		d2	Check d2	d1(+25)
			d2		
60		CONFIRM		Check d2	
70	EXPECT(15ms, 1)		d1(90)		
	f(x)		d1	Check d1	
90	CONFIRM				d1(+5)
				Check nothing	

If the monitoring thread is scheduled on time, each deadline violation will be detected with a predictable delay.

Generalizations

1. Overflow Tolerance
2. Nested Deadlines

Measuring Time

```
#include <chrono>
using time_t = uint64_t;
using clock_t = chrono::steady_clock;
using time_unit_t = chrono::nanoseconds;

time_t now() {
  auto tp = chrono::time_point_cast<time_unit_t>(clock_t::now());
  // usually signed int64_t representation
  auto ticks = tp.time_since_epoch().count();
  return static_cast<time_t>(ticks);
}
```

Bad

- Loses chrono unit safety, but is only used internally
- Generally not safe in the overflow case
  - Depends on `clock_t::now()`
  - Depends on `rep_t count();`
- Usually rep_t is signed

Good

- Monotonic clock
- Not exposed to user; API can use `chrono::literals`
- Unsigned overflow is well-defined (modular arithmetics)

Overflow Tolerance

Advantages

- Epoch does not matter
- System can run forever
- Can use smaller types than `time_t = uint64_t` (e.g. only for milliseconds)
    - Can use some arbitrary real-time counter
    - Can use the extra bits for additional information

Disadvantages

- Clock must support overflow
- Slightly more complex violation checks
- Deadline invalidation becomes more complex

Time can be considered as a cycle with power of two length.

Measuring Time Intervals

Wrong - does not handle overflow correctly

```
bool is_violated(time_t deadline) {
  time_t t = now();
  time_t delta = t - deadline;
  // t > deadline?
  // delta is always >= 0
  return delta > 0;
}
```

Correctly handles overflow

```
using stime_t = int64_t;
bool is_violated(time_t deadline) {
  time_t t = now();
  time_t delta = t - deadline;
  stime_t s = static_cast<stime_t>(delta)
  return s > 0;
}
```

Problem

Need to check whether the current time t precedes the deadline on a cycle of length $M = 2^{64}$

- Modular arithmetic wrt. M
- Time points ahead of t by at most M/2 are considered to be later
- Other half cycle is considered before

Observation

Due to two's complement these are equivalent

1. deadline is before t (violation)
2. `0 < delta < M/2`
3. The most significant bit of `delta` is `0`
4. The signed representation of `delta` is `>0`

Is Overflow only a Theoretical Problem?

That depends on

- Resolution of the clock
- Epoch of the clock (how close do we start to overflow?)
- Application runtime
- Safety requirements (proof that overflow is never a problem)

With nanosecond ticks and epoch being system start we can count for ~293 years with signed 64 bit integers.

Why do we have to be careful with `chrono::steady_clock`?

- Signed integer overflow is undefined behavior
- `It` does not define the epoch (usually system start, but we cannot rely on that)
- The representation type of the ticks can be signed
  - `count()` will return a signed or unsigned type depending on implementation (usually signed)

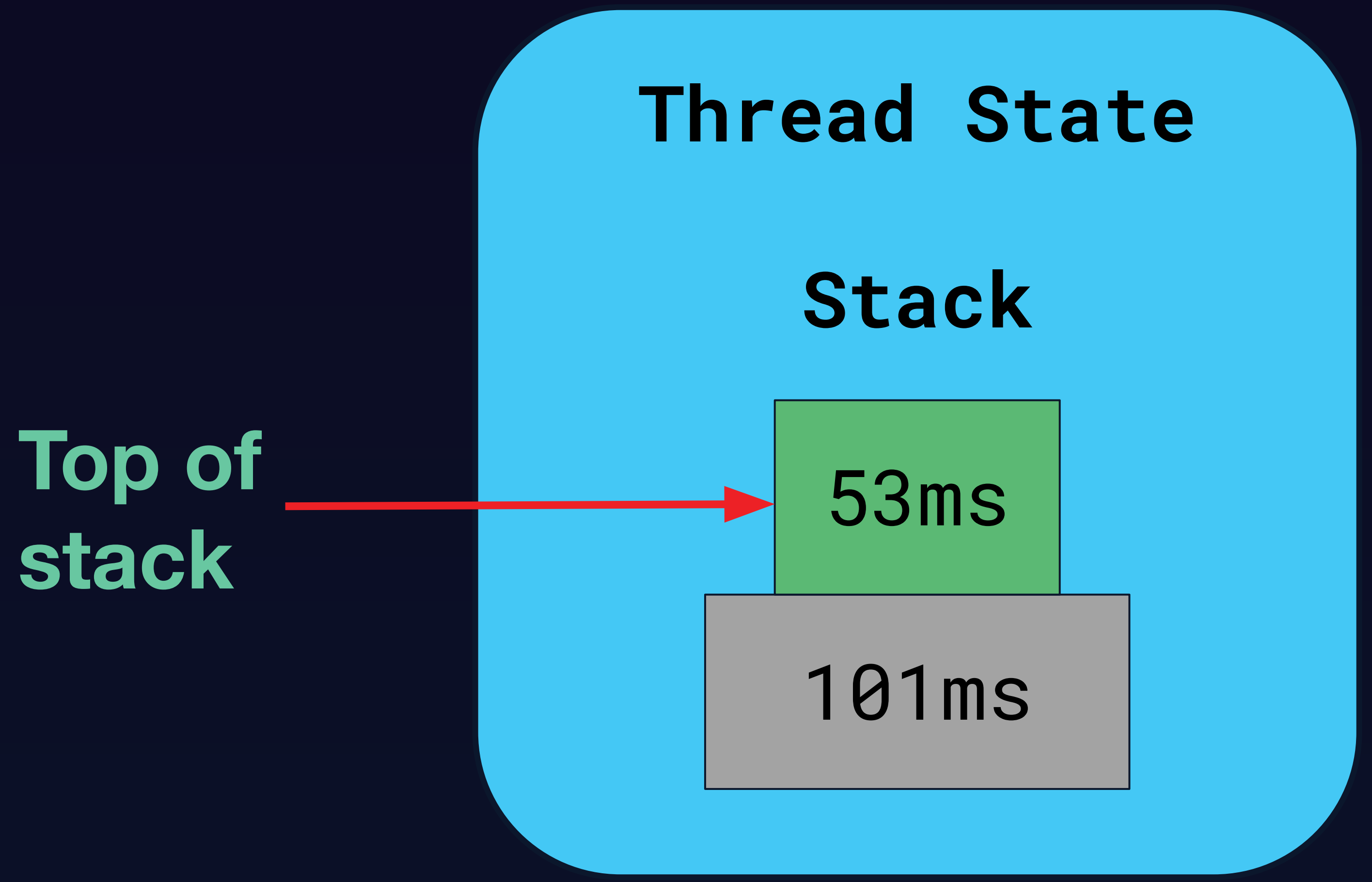
Overflow problems can be avoided with access to a monotonic clock with unsigned time ticks.

Nested Deadlines

- Important since functions declaring the deadlines may call each other
- Same idea, but now each thread keeps a stack of deadlines
- Nested deadlines are assumed to be monotonic for optimal detection (but do not need to be)

```
EXPECT_PROGRESS_IN(100ms, 1);
// time critical code
// maybe in another function
EXPECT_PROGRESS_IN(50ms, 2);
// more time critical code

CONFIRM_PROGRESS;
CONFIRM_PROGRESS;
```



Nested Deadlines - Stack

Properties

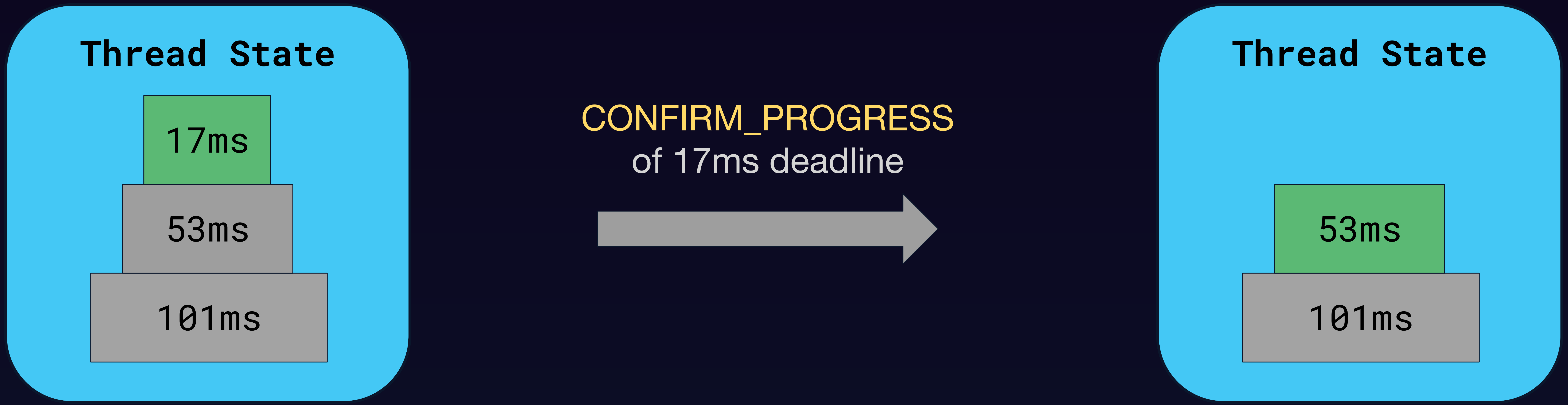
1. Lock-free
2. Single producer/consumer - multiple readers
3. Monitored thread is producer and consumer
   - push/pop elements (deadlines)
4. Monitoring thread needs to
   - Read elements
   - Write elements (only to invalidate deadlines)
5. Stack elements must be trivially copyable

Stack memory management

- Intrusive stack elements, i.e. they expose internal node structure (next pointer)
- No or limited dynamic allocation

Skip implementation details

Nested Deadlines - Confirm



Monitoring thread will only check top of stack

- Sufficient for monotonic deadlines
- Delayed detection if not monotonic

No violation or violation detected by thread itself

1. Pop deadline from stack
2. If the deadline was not marked as invalidated by the monitoring thread check the deadline
3. Report violation if any

Nested Deadlines - Violation

Monitoring thread detected violation at top of stack

1. Invalidate top of stack deadline (no pop!)
2. Report violation
3. Check deadlines further down the stack from now on
    - Violated deadlines are invalidated

Time for some field tests.

Application

- Application Monitoring
- Timing Tests
- Runtime Statistics
- Performance

Monitoring Application Progress at Runtime

Cyclic time-critical application

```
#include "monitoring_api.hpp"
mutex g_mutex;
atomic<bool> g_run{true};

void thread_main() {
  // can reduce this boilerplate
  START_THIS_THREAD_MONITORING;
  SET_MONITORING_HANDLER(handler);
  while(g_run) {
    EXPECT_PROGRESS_IN(10ms, SOME_ID);
    lock_guard<mutex> guard(g_mutex);
    time_critical_function();
    CONFIRM_PROGRESS;
  }
  STOP_THIS_THREAD_MONITORING;
}
```

Macro API advantages

- Source location
- Distinct from regular functions by convention
- Look and feel like e.g. Google Test
- Easy to disable

Possible API additions

Progress scope guard

- Reduces the risk of forgetting the closing `CONFIRM_PROGRESS`
- Restricts deadline end to scope end

Monitored thread

- Reduces thread monitoring boilerplate

Timing Tests

```
#include "monitoring_api.hpp"

atomic<bool> g_deadline_violation{false};
// assumed to be set in general setup code
void handler(checkpoint &) {
  g_deadline_violation = true;
}

TEST_F(SomeFixture, deadline) {
  // test specific setup

  EXPECT_PROGRESS_IN(10ms, TEST_ID);
  int result = sut.critical_function();
  CONFIRM_PROGRESS;

  EXPECT_FALSE(g_deadline_violation);
  EXPECT_EQ(result, 73);
}
```

Timing tests are problematic

- Threads may not be scheduled
- Timing expectation may be unreasonable for slower hardware
- Running under test conditions may be different from real conditions

Complements Google Test Framework

- Easy to add by just including a header
- Code to start monitoring in setup code
- Similar syntax for expected behavior

Combine to reduce boilerplate

```
CONFIRM_PROGRESS
EXPECT_FALSE(g_deadline_violation);
```

Runtime Statistics

For each time-critical section we can incrementally update

- Number of executions
- Number of violations
- Minimum and maximum runtime
- Mean runtime
- Standard deviation
- …

Overhead

1. Manage the statistics in e.g. (thread local) `map<id_t,statistics>`
2. Store the start time of a section
3. Compute statistics incrementally at the end of a section
4. Update statistics

Disabled by default due to overhead
Can be further optimized!

Example Output

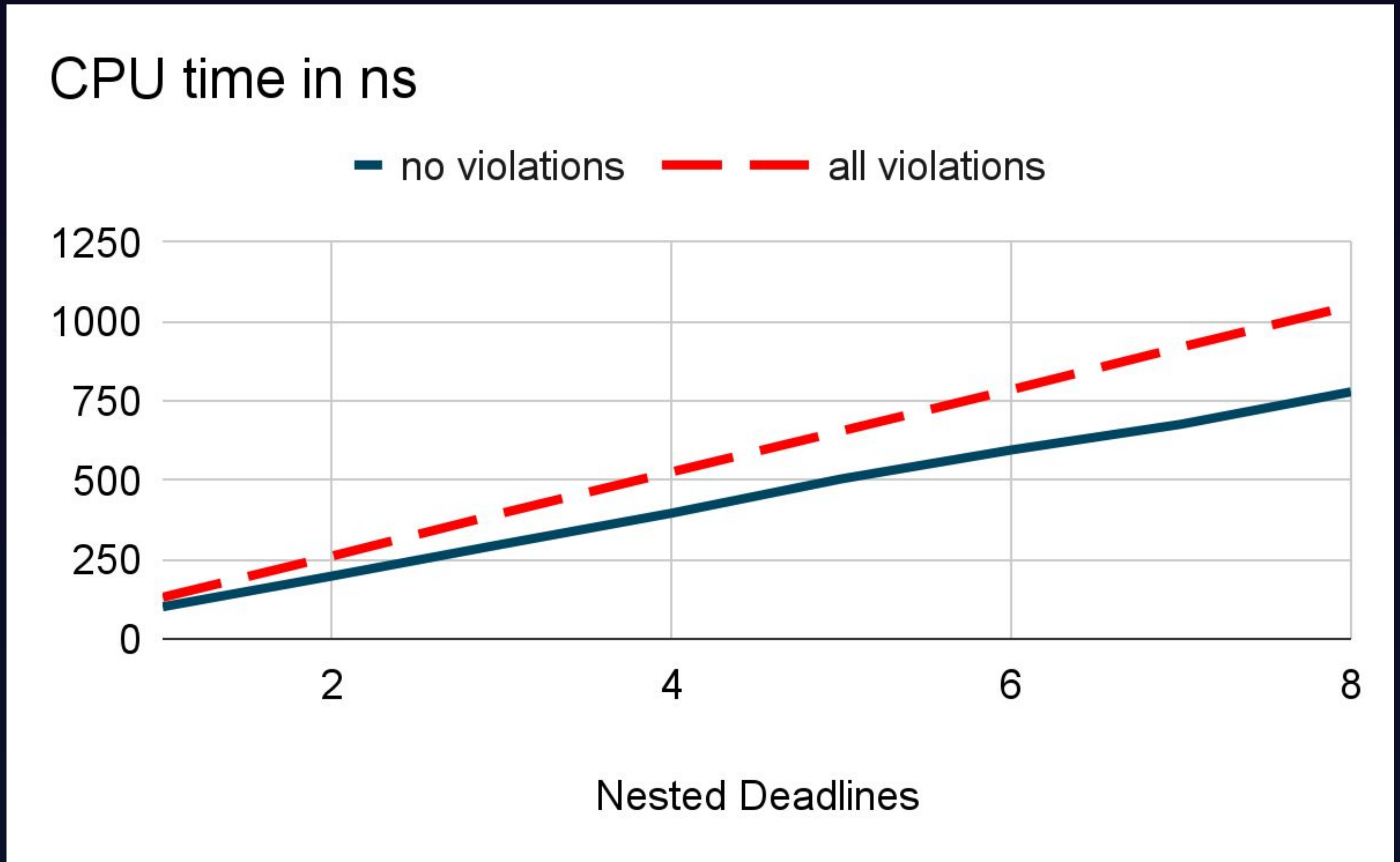
deadline id 1 (10000us)
count : 2000
violations : 0
min : 1635
max : 8427
mean : 5097.02
standard deviation : 988.378

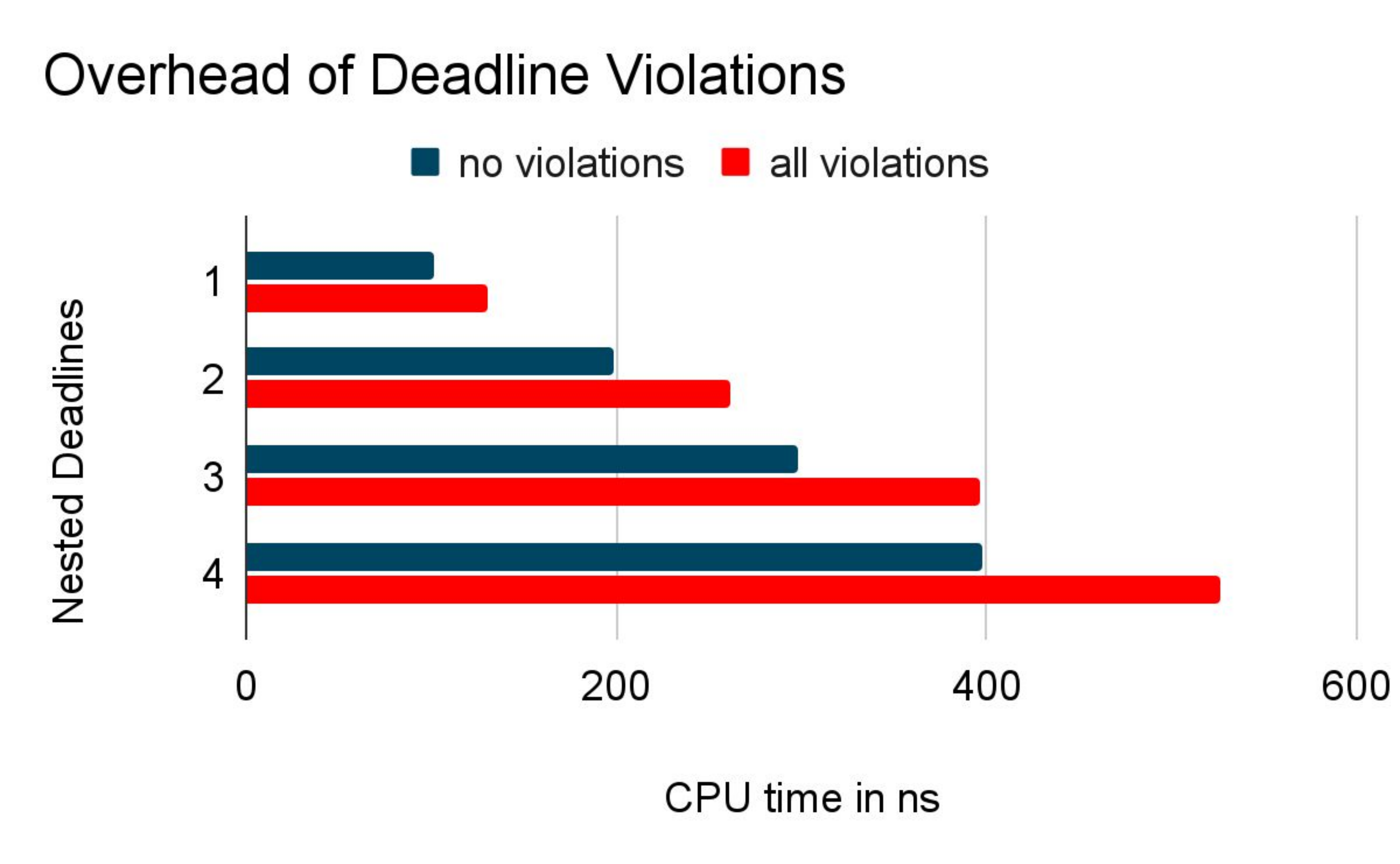
deadline id 2 (10000us)
count : 1000
violations : 11
min : 1155
max : 10165
mean : 5646.78
standard deviation : 2587.96

Performance

Google benchmark on Intel Core i7 x86-64 - 12 cores with gcc -O3

Measure CPU time of `EXPECT_PROGRESS` - `CONFIRM_PROGRESS` cycles without extra work

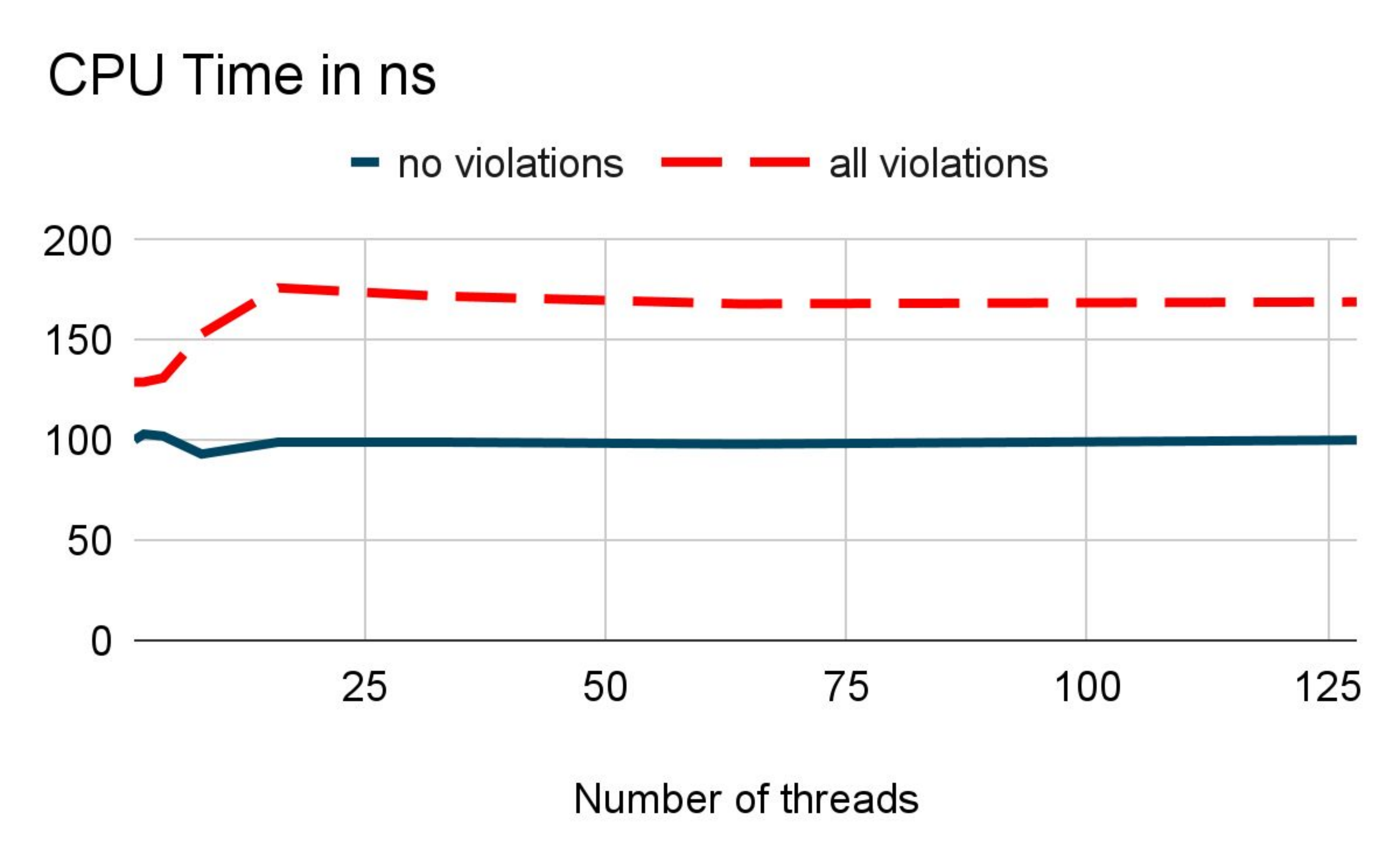




- Linear scaling wrt. number of nested deadlines
- Overhead per deadline is about 100ns (uncontended mutex lock - unlock takes about 12ns)
- Deadline violations require ~1.3 times the CPU time (trivial handler that only sets a flag, no output)

Performance

Measure CPU time of `EXPECT_PROGRESS - CONFIRM_PROGRESS` cycles in multiple concurrent threads



- Monitored threads are independent from each other
- Limited lock-free data sharing has the intended effect
- Deadline violations appear to cause some minimal contention

Caution - Microbenchmarks often run with favorable cache conditions.

Conclusion

We have implemented Framework to monitor deadlines in C++ that can easily be integrated into applications.

Features

- Visibly specify deadlines in code
  - The code is the source of truth
  - Can be matched with timing requirements in design
- Deadline violation will be detected eventually with fair scheduling
- Low detection delay (depends on configuration)
- Low overhead
- Monitoring can be disabled at compile time (no overhead)
- Measures statistics to e.g. fine tune deadlines on a specific system

Key Takeaways

1. **Data sharing**
   - Avoid if possible (e.g. thread local)
   - If it cannot be avoided, keep the critical sections small
   - Lock-free code can boost performance
   - Blocking is ok for rare operations
   - Measure the scaling with multiple threads

2. **Time measurement**
   - Measuring time correctly is tricky
   - Use a monotonic clock since we measure time intervals (stopwatch)
   - With unsigned time stamps, the system can tolerate overflow and run forever
   - Can use a real-time counter if available

3. **Use Custom Data Structures and Memory allocation**
   - Reduce allocation at runtime (fragmentation, time overhead)
   - Prefer preallocation
   - Consider special purpose intrusive data structures

Limitations

1. **No interprocess monitoring**
   - Monitoring thread per process
   - Can be extended by using efficient shared memory data transfer
     - Monitoring process instead of thread
     - Could use e.g. iceoryx zero-copy middleware to share deadline data

2. **Inefficient monitoring thread**
   - Intentional time-based checking (performance)
   - No mutex for synchronization of e.g. a priority queue
   - No wake-up notification by other threads (for performance)
   - Does not affect other threads until the system is overloaded

3. **Inefficient statistics mode**
   - Simple test implementation uses a mutex and avoidable data sharing
   - Only supposed to be used for development purpose

4. **Not completely real-time safe yet**
   - Some data structures still use dynamic memory and exceptions
   - Can be replaced with fixed size alternatives

References

- Current Implementation: https://github.com/MatthiasKillat/progress_monitoring/
- Real-time counter: https://luckyresistor.me/2019/07/10/real-time-counter-and-integer-overflow/
- Lock-free buffer: Meeting C++ 2021, Lock-free Programming for Real-Time Systems
- Concurrency benchmarks: CppCon 2016, The Speed of Concurrency (is lock-free faster?), Fedor Pikus
- C++ Concurrency in Action, Anthony Williams
- Google Test: https://github.com/google/googletest
- iceoryx zero-copy middleware: https://github.com/eclipse-iceoryx/iceoryx
  - Real-time safe fixed size data structures
  - Lock-free data structures
  - Potentially useful for efficient monitoring across process boundaries

The implementation is work in progress.

Suggestions for improvements are welcome!

There is a subtle problem in the implementation related to invalid deadline representation.

Could you sp0t it?

Thanks for your Attention

Deadline Invalidation

- Generally `0` can be a valid deadline (especially if overflow is possible)
- Cannot invalidate by setting to `0` (by exchange or CAS)

Solution 1: Use least significant bit

- `1` becomes the invalid value
- Unsigned range is reduced by half
- Caution with monotonic counters, times are now represented by even values

Solution 2: Dual Counter - Use additional deadline validation value v

- Deadline d is valid if and only if d equals v
- Invalidation sets these to unequal (`+1` or `-1`)
- Keep full unsigned range
- Caution with ABA problem
  - Requires correct invalidation scheme
  - Monotonic time is effectively an ABA counter

Both solutions increase the overhead slightly

Deadline Stack

Lock-free deadline stack

```
struct stack_entry {
    atomic<time_t> deadline; // payload
    stack_entry* next; // intrusive data
};

class deadline_stack {
public:
  void push(stack_entry&);
  stack_entry* pop();
  stack_entry* top();

  uint64_t generation();

private:
  atomic<stack_entry*> m_top{nullptr};
  // generation count
  atomic<uint64_t> m_generation{0};
};
```

- Allocation of entries happens externally
- In practice can be a simple thread local allocator
- `stack_entry` can be copied by memcpy

1. Lock-free push and pop are simple since there is only one thread modifying the stack
2. Each operation increases the generation counter
3. Counter has to synchronize data properly

Deadline Stack

Lock-free deadline stack

```
struct stack_entry {
    atomic<time_t> deadline; // payload
    stack_entry* next; // intrusive data
};

class deadline_stack {
public:
  void push(stack_entry&);
  stack_entry* pop();
  stack_entry* top();

  uint64_t generation();

private:
  atomic<stack_entry*> m_top{nullptr};
  // generation count
  atomic<uint64_t> m_generation{0};
};
```

Reading the stack concurrently

```
deadline_stack stack;

auto gen = stack.generation();

while(gen % 2 == 1) {
  // odd generation, write in progress
  // retry or give up
}

// sync with fence (gen read happens before)
auto *top = stack.top();

if(top) {
  // read from top and traverse stack
  // cannot crash(!), even if stack changes
  if(gen == stack.generation()) {
    // read is valid (modulo ABA problem)
  }
}
```

Measuring Time Intervals - Example

Why does this work?

```
using stime_t = int64_t;
bool is_violated(time_t deadline) {
  time_t t = now();
  time_t delta = t - deadline;
  stime_t s = static_cast<stime_t>(delta)
  return s > 0;
}
```

Assume we only use 8 bit counters, i.e.
`time_t = uint8_t, stime_t = int8_t`

`EXPECT_PROGRESS_IN(10ns, 1)`

- `t0 = 250`
- `deadline = 250 + 10 = 4 (modulo 256)`

No overflow, not expired

```
t = 253
delta = 253 - 4 = 249
s = -7 is < 0
```

Overflow, not expired

```
t = 2
delta = 2 - 4 = 254
s = -2 is < 0
```

Overflow, expired

```
t = 5
delta = 5 - 4 = 1
s = 1 is > 0
```

What about t = 249?
Expired, also OK!

Measuring Time Intervals - Correctness

Why does this work?

```cpp
using stime_t = int64_t;
bool is_violated(time_t deadline) {
  time_t t = now();
  time_t delta = t - deadline;
  stime_t s = static_cast<stime_t>(delta)
  return s > 0;
}
```

- Modular arithmetic wrt. $M = 2^k$
- `EXPECT_PROGRESS_IN(r, 1)`
- `r < M/2 = m`
- Start time t0
- Deadline d = t0 + r

Case 1: deadline not yet violated

- `m < delta <= M` (since `r<m`)
- MSB of delta is 1
- `s <= 0`
- return false

Case 2: deadline violated by at most m

- `0 < delta < m`
- MSB of delta is 0
- s > 0
- return true

Case 3: deadline violated by more than m

- `0 <= delta <= M` (potential wraparound)
- MSB of delta is `0` or 1
- Cannot distinguish by using s
- Possibly incorrect result

Must check sufficiently often to avoid case 3
(with nanoseconds we have ~293 years)