Multithreaded programs
The Good, the Bad and the Buggy

Mamy Ratsimbazafy
mamy@numforge.co

Weave
https://github.com/mratsim/weave

Hello!

I am Mamy Ratsimbazafy

During the day Blockchain/Ethereum 2 developer (in Nim)

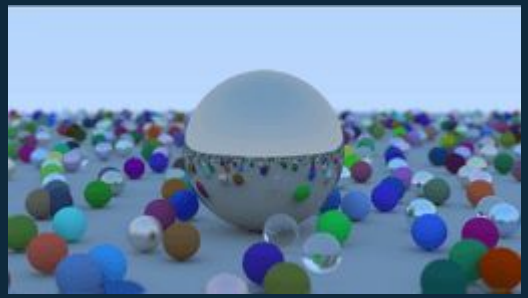
During the night, multithreading, deep learning, cryptography (in Nim) and data scientist (in Python)

You can contact me at mamy@numforge.co

Github: mratsim

Twitter: m_ratsim

Why and why not multithreading?



- Why: Time is precious (~3 hours to generate on 18 cores)
  - 10800s * 18 = 194400s
- But: Use logarithmic data structures: log2(194400) = 17.56s

My goal today

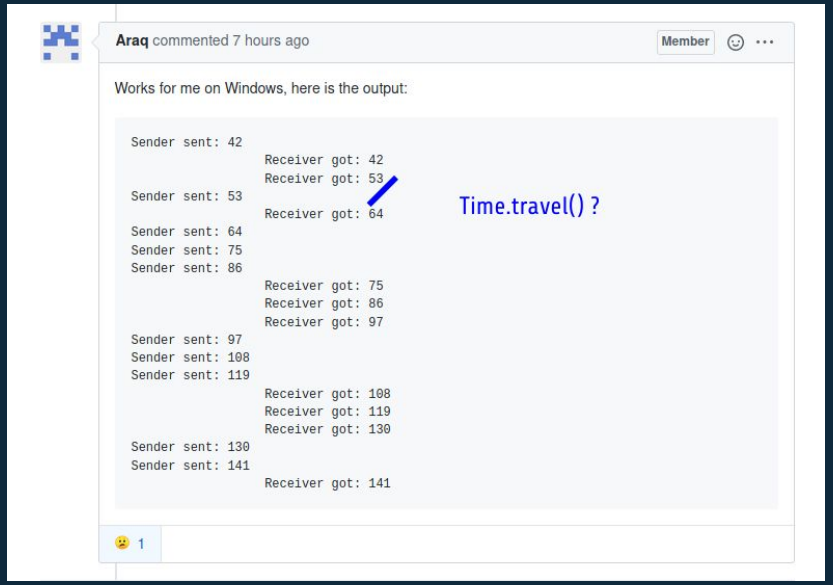
Help your inner cat deal with threads

Motivating examples (1/3)

A working channel output:

What if I told you you can't reliably debug echo?

https://github.com/nim-lang/Nim/issues/13936



Motivating examples (2/3)

Unintuitive low-level behavior

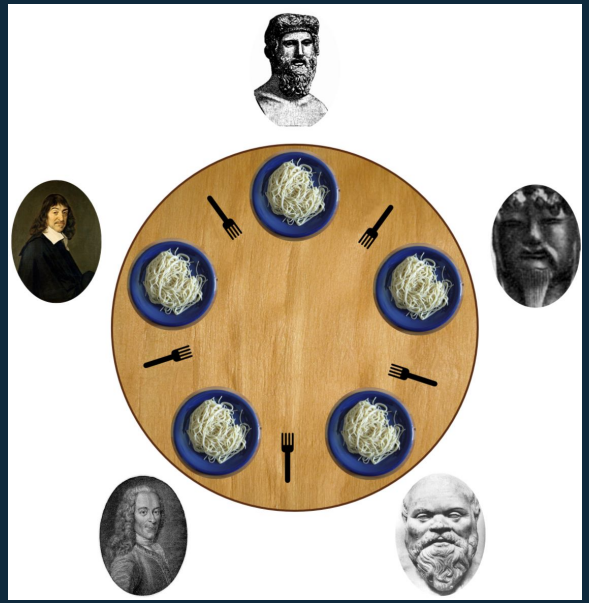
```
1   var A, B = 0
2
3   proc thread1() =
4     A = 1
5     echo B
6
7   proc thread2() =
8     B = 1
9     echo A
10
```

This can print 11, 10, 01 but unintuitively it can also display 00

Motivating examples (3/3)

Dining Philosophers

Deadlock and Starvation



Agenda

- Target audience and example threaded programs
- Debugging multithreaded programs
- Designing to reduce the potential multithreaded bug surface
- Correct-by-constructions programs

1 Target audience

And example threaded programs

Concurrent Data Structures

- A (hash)-table that allow access from multiple threads.
- An event bus (i.e. multithreaded publish/subscribe)
- An object pool for game objects, for example particles.
- A tree-like data structure, on a GPU, to accelerate raytracing (and make ray collisions logarithmic).

Concurrent systems

- Transactional code: “commit-rollback”
- Collecting metrics and/or logs from multiple threads
- Building a service architecture
  - User Interface
  - Sound
  - Graphics
  - Networking, File, IO
  - … (physics for a game, JIT for a web browser, logs)

2 Debugging

Multithreaded programs

Tests are not enough!

- This can be tested for all inputs on a single-core

```
1  proc foo(a: int16) =
2    discard
3
```

- But on 4-core, it's (2^16)^4 = 2^64 states

$2^{64} = 18\,446\,744\,073\,709\,551\,616$

- Way to many states to check exhaustively

- Also threads are not deterministic so bugs are hard to reproduce

Introducing AOP

Assertion-oriented programming

```
# Worker - Tasks handling
# ------------------------------------------------------------------------

proc restartWork*() =
  preCondition: myThefts().outstanding == WV_MaxConcurrentStealPerWorker
  preCondition: myTodoBoxes().len == WV_MaxConcurrentStealPerWorker

  # Adjust value of outstanding by MaxSteal-1, the number of steal
  # requests that have been dropped:
  # outstanding = outstanding - (MaxSteal-1) =
  #           = MaxSteal - MaxSteal + 1 = 1

  myThefts().outstanding = 1 # The current steal request is not fully fulfilled yet
  myWorker().isWaiting = false
  myThefts().dropped = 0
```

```
17  # Signals
18  # ------------------------------------------------------------------------
19
20  proc detectTermination*() {.inline.} =
21    preCondition: myID() == RootID
22    preCondition: myWorker().leftIsWaiting and myWorker().rightIsWaiting
23    preCondition: not workerContext.runtimeIsQuiescent
24
25    debugTermination:
26      log(">>> Worker %2d detects termination <<<\n", myID())
27
28    workerContext.runtimeIsQuiescent = true
29
30  proc asyncSignal(fn: proc (_: pointer) {.nimcall, gcsafe.}, chan: var ChannelSpscSinglePtr[T
31    ## Send an asynchronous signal `fn` to channel `chan`
32
```

Assertion-Oriented Programming

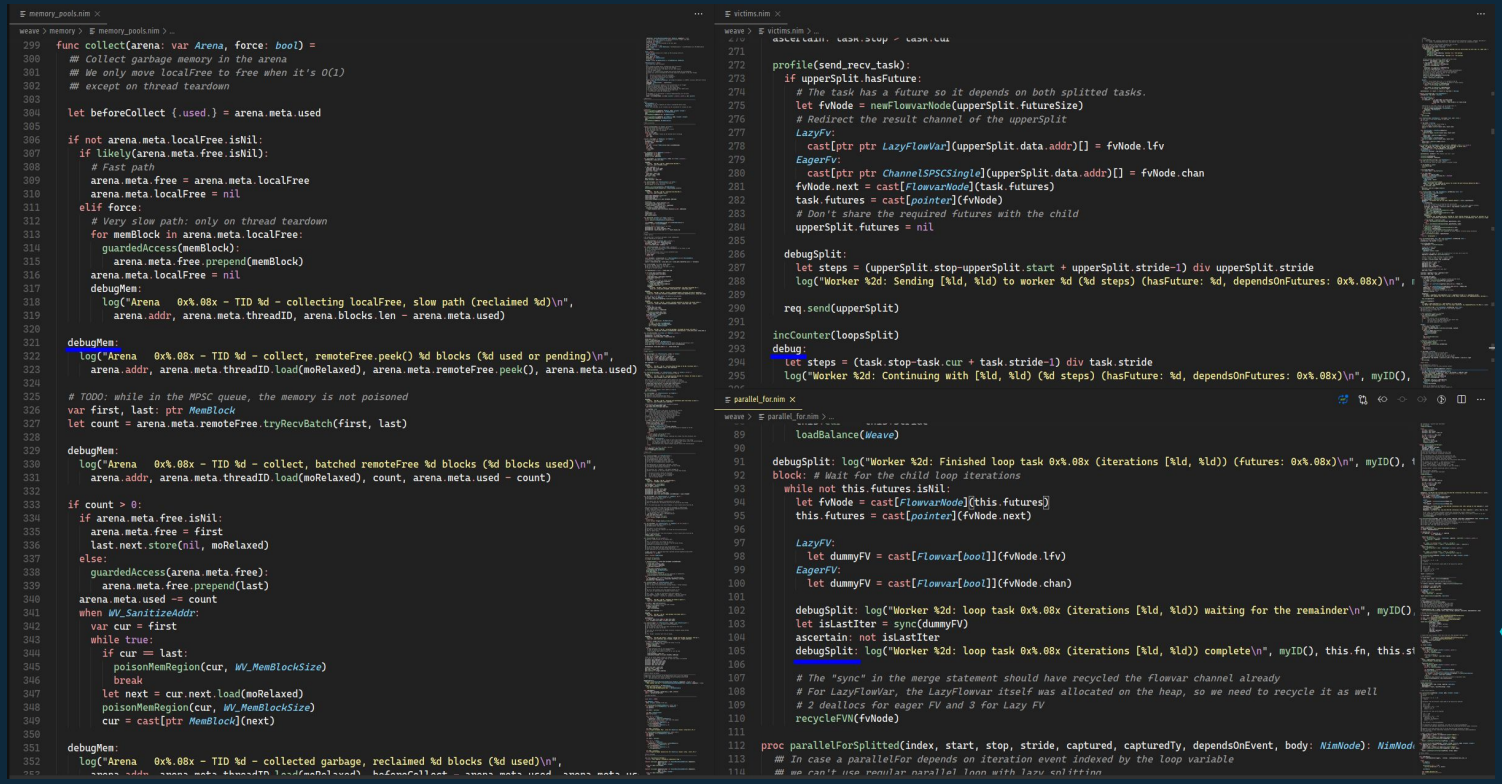
- Gate them behind “defined” or {.strdefine.} and display as much context as possible.
- Remember, log order between 2 threads is random
- but logs produced by 1 thread is (sequentially consistent)

Rabbit hole: Serializability - https://en.wikipedia.org/wiki/Serializability

```
83    result = quote do:
84      {.noSideEffect.}:
85        when compileOption("assertions"):
86          assert(`predicate`, `debug` & $`values` & "  [Worker " & `myID` & "]\n")
87        elif defined(WV_Asserts):
88          if unlikely(not(`predicate`)):
89            raise newException(AssertionError, `debug` & $`values` & '\n')
90
91  # A way way to get the caller function would be nice.
92
93  template preCondition*(require: untyped) =
94    ## Optional runtime check before returning from a function
95    assertContract("pre-condition", require)
96
97  template postCondition*(ensure: untyped) =
98    ## Optional runtime check at the start of a function
99    assertContract("post-condition", ensure)
100
101 template ascertain*(check: untyped) =
102   ## Optional runtime check in the middle of processing
103   assertContract("transient condition", check)
```

Introducing LOP

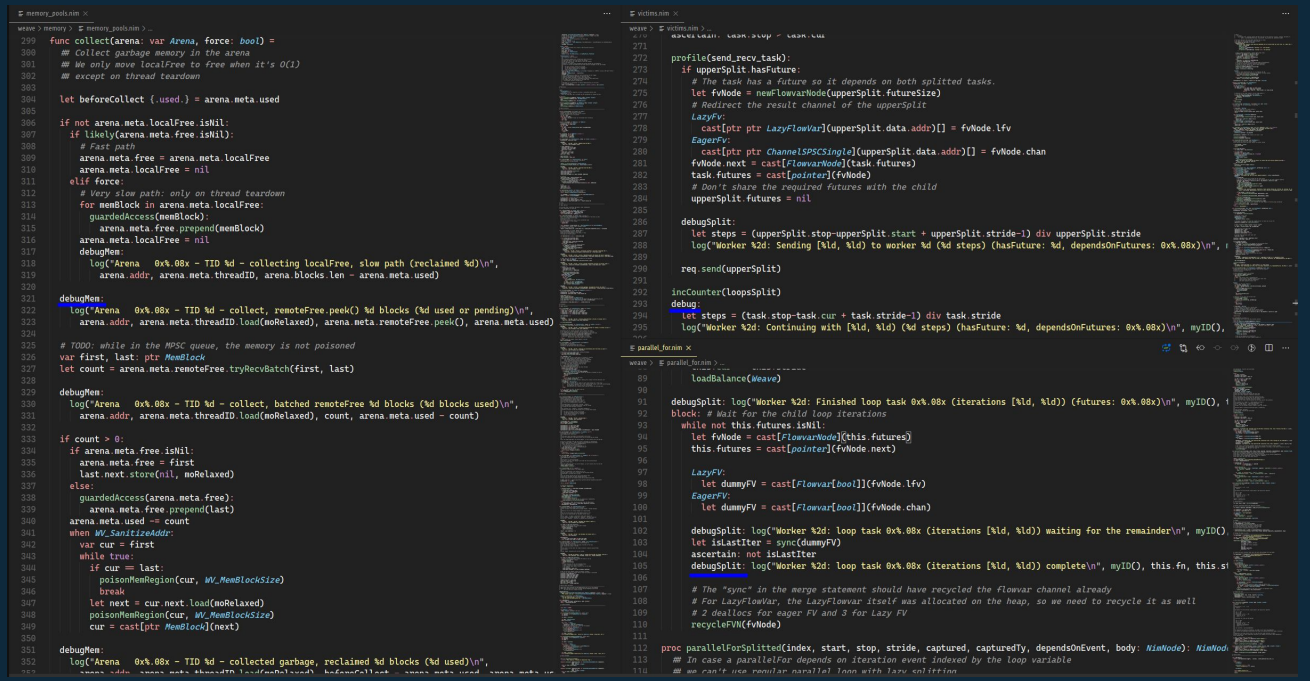
Log-oriented programming



Log-oriented programming

Semantic logs

- debug, debugSplit, debugMem



Log-oriented programming

Analyzing philosophers on parallel matrix transpose

```
$ nim c -r -d:release -d:WV_debugSplit --outdir:build benchmarks/matrix_transposition/weave_transposes.nim
```

```
93
94  proc weave2DTiledNestedTranspose(M, N: int, bufIn, bufOut: ptr UncheckedArray[float32]) =
95    ## Transpose with 2D tiling and nested
96
97    const blck = 64 # const do not need to be captured
98
99    parallelForStrided j in 0 ..< N, stride = blck:
100     captures: {M, N, bufIn, bufOut}
101     parallelForStrided i in 0 ..< M, stride = blck:
102       captures: {j, M, N, bufIn, bufOut}
103       for jj in j ..< min(j+blck, N):
104         for ii in i ..< min(i+blck, M):
105           bufOut[jj*M+ii] = bufIn[ii*N+jj]
```

```
Worker 25: has 0 steal requests queued
Worker 25: 23 steps left (start: 0, current: 2560, stop: 4000, stride: 64, 3 thieves)
Worker 25: Sending [3648, 4000) to worker 2 (6 steps) (hasFuture: 0, dependsOnFutures: 0x00000000)
Worker  8: Sending [2496, 4000) to worker 31 (24 steps) (hasFuture: 0, dependsOnFutures: 0x00000000)
Worker  8: has 5 steal requests queued
Worker  8: 22 steps left (start: 0, current: 1088, stop: 2496, stride: 64, 6 thieves)
Worker  8: Sending [2304, 2496) to worker 20 (3 steps) (hasFuture: 0, dependsOnFutures: 0x00000000)
Worker  8: has 4 steal requests queued
Worker  1: has 0 steal requests queued
Worker  1: 22 steps left (start: 2048, current: 2624, stop: 4000, stride: 64, 1 thieves)
Worker  5: has 0 steal requests queued
Worker  5: 52 steps left (start: 0, current: 704, stop: 4000, stride: 64, 1 thieves)
Worker  5: Sending [2304, 4000) to worker 2 (27 steps) (hasFuture: 0, dependsOnFutures: 0x00000000)
Worker  8: 19 steps left (start: 0, current: 1088, stop: 2304, stride: 64, 5 thieves)
Worker  8: Sending [2112, 2304) to worker 34 (3 steps) (hasFuture: 0, dependsOnFutures: 0x00000000)
Worker  8: has 3 steal requests queued
Worker  8: 16 steps left (start: 0, current: 1088, stop: 2112, stride: 64, 4 thieves)
Worker  8: Sending [1920, 2112) to worker 22 (3 steps) (hasFuture: 0, dependsOnFutures: 0x00000000)
Worker 30: has 0 steal requests queued
Worker  7: has 0 steal requests queued
Worker  7: 13 steps left (start: 0, current: 3200, stop: 4000, stride: 64, 1 thieves)
Worker  7: Sending [3584, 4000) to worker 10 (7 steps) (hasFuture: 0, dependsOnFutures: 0x00000000)
Worker  7: has 2 steal requests queued
Worker  7: 6 steps left (start: 0, current: 3200, stop: 3584, stride: 64, 3 thieves)
Worker  7: Sending [3520, 3584) to worker 18 (1 steps) (hasFuture: 0, dependsOnFutures: 0x00000000)
Worker  7: has 1 steal requests queued
Worker  7: 5 steps left (start: 0, current: 3200, stop: 3520, stride: 64, 2 thieves)
Worker  7: Sending [3456, 3520) to worker 26 (1 steps) (hasFuture: 0, dependsOnFutures: 0x00000000)
```

Log-oriented programming

Buffer logs and send at once via flushFile(stdout)

```
1  import ../weave
2
3  init(Weave)
4
5  parallelFor i in 0 ..< 100:
6    echo "Enjoy (", i, ")"
7    echo "event (", i, ")"
8    echo "ordering (", i, ")"
9
10 exit(Weave)
11
```

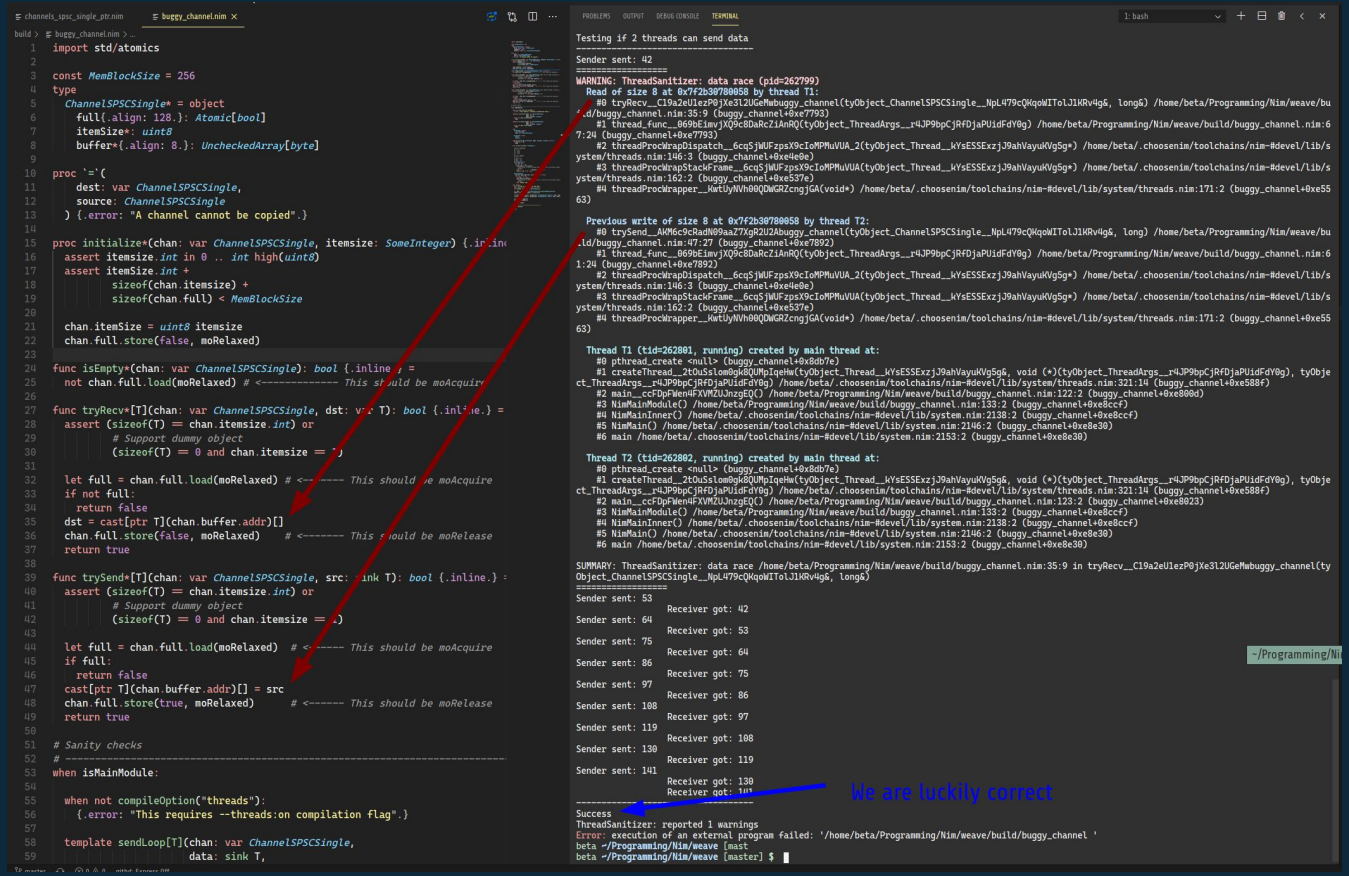
weave > instrumentation > loggers.nim

You, 6 months ago | 1 author (You)

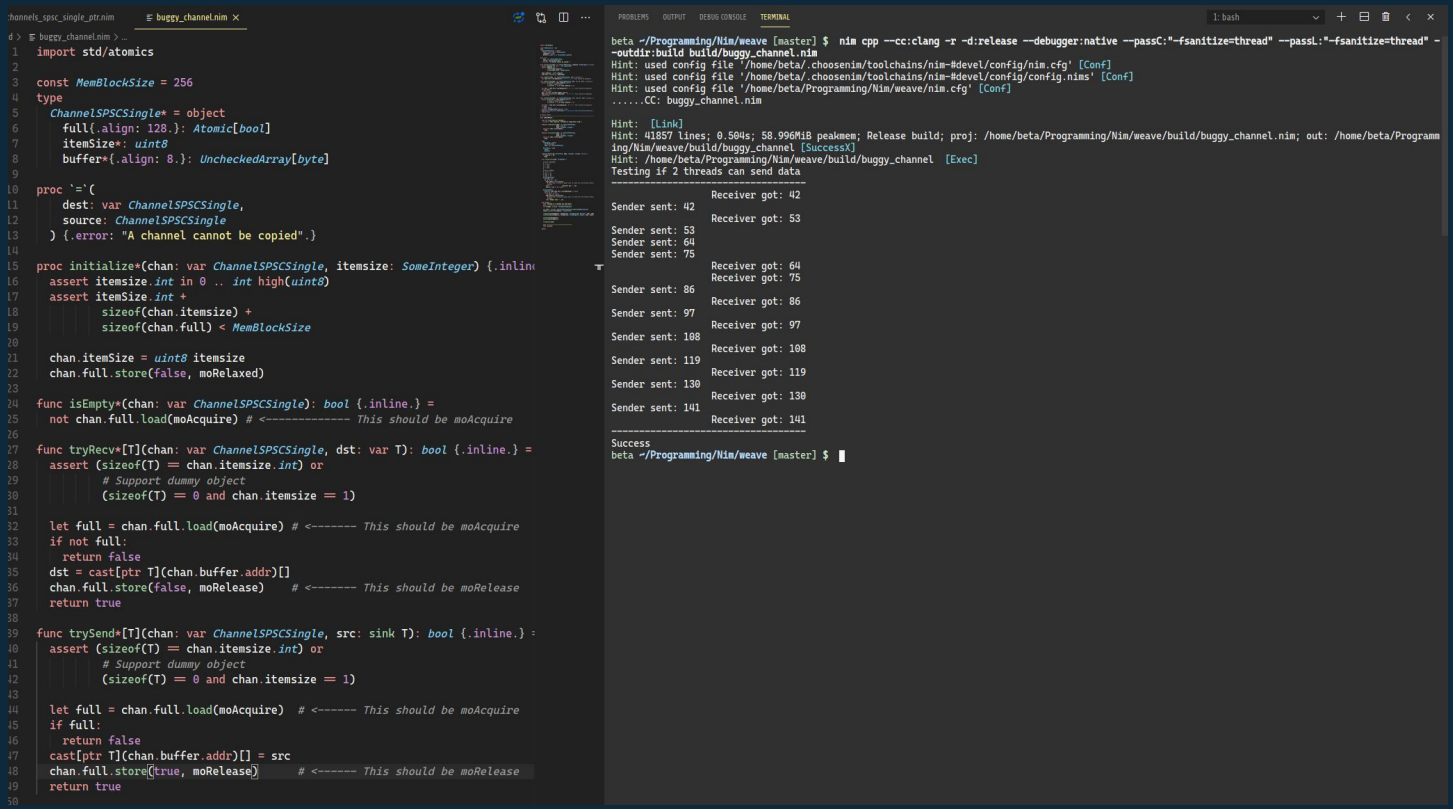
```
1  # Weave          Mamy André-Ratsimbazafy [7 months ago
2  # Copyright (c) 2019 Mamy André-Ratsimbazafy
3  # Licensed and distributed under either of
4  #   * MIT license (license terms in the root director
5  #   * Apache v2 license (license terms in the root di
6  # at your option. This file may not be copied, modifi
7
8  import system/ansi_c
9
10 {.used.}
11
12 template log*(args: varargs[untyped]): untyped =
13   c_printf(args)
14   flushFile(stdout)
15
```

```
Enjoy (0)
event (0)
ordering (0)
Enjoy (82)
Enjoy (52)
event (52)
Enjoy (16)
event (16)
ordering (16)
event (82)
Enjoy (17)
Enjoy (76)
event (76)
ordering (76)
Enjoy (77)
event (77)
Enjoy (64)
event (64)
ordering (64)
Enjoy (88)
event (88)
Enjoy (40)
event (40)
ordering (40)
Enjoy (94)
event (94)
ordering (94)
ordering (82)
Enjoy (83)
```

Sanitizers – Thread Sanitizer



Sanitizers – Thread Sanitizer



Sanitizers – Thread Sanitizer

- Always use thread sanitizer it only requires

```
nim cpp --cc:clang -r -d:release --debugger:native
--passC:"-fsanitize=thread" --passL:"-fsanitize=thread"
--outdir:build build/buggy_channel.nim
```

- Use the C++ target if you use atomics
- Easy way to detect data races for concurrent data structure, for example a concurrent hash table

Sanitizers – Address Sanitizer

- Detects incorrect memory addresses (uninitialized memory, buffer overflows, use after free, leaks).
- Can be made aware of custom memory management scheme (memory pools, ...)

Sanitizers – Address Sanitizer

weave > instrumentation > sanitizers.nim > ...

Mamy Ratsimbazafy, 6 months ago | 1 author (Mamy Ratsimbazafy)

```nim
# Weave     Mamy Ratsimbazafy [6 months ago] • Add AddressSanitizer support for the memory pool (#78)
# Copyright (c) 2019 Mamy André-Ratsimbazafy
# Licensed and distributed under either of
#   * MIT license (license terms in the root directory or at http://opensource.org/licenses/MIT).
#   * Apache v2 license (license terms in the root directory or at http://www.apache.org/licenses/LICENSE-2.0).
# at your option. This file may not be copied, modified, or distributed except according to those terms.

# Address sanitizer
# ----------------------------------------------------------------------------------

proc asan_poison_memory_region(region: pointer, size: int){.nodecl, importc:"__asan_poison_memory_region".}
proc asan_unpoison_memory_region(region: pointer, size: int){.nodecl, importc:"__asan_unpoison_memory_region".}

const WV_SanitizeAddr*{.booldefine.} = false

when WV_SanitizeAddr:
  when not(defined(gcc) or defined(clang) or defined(llvm_gcc)):
    {.error: "Address Sanitizer requires GCC or Clang".}
  else:
    {.passC:"-fsanitize=address".}
    {.passL:"-fsanitize=address".}
    when defined(clang) and defined(amd64):
      {.passL:"-lclang_rt.asan_dynamic-x86_64".}
    else:
      {.error: "Compiler + CPU arch configuration missing".}

template poisonMemRegion*(region: pointer, size: int) =
  when WV_SanitizeAddr:
    asan_poison_memory_region(region, size)
  else:
    discard

template unpoisonMemRegion*(region: pointer, size: int) =
  when WV_SanitizeAddr:
    asan_unpoison_memory_region(region, size)
  else:
    discard
```

weave > memory > memory_pools.nim > ...

```nim
  MostlyUsedRatio = 8
    ## Beyond 7/8 of its capacity an arena is considered mostly used.
  MaxSlowFrees = 8'i8
    ## In the slow path, up to 8 arenas can be considered for release at

# Sanitizer
template guardedAccess(memBlock: MemBlock, body: untyped): untyped =
  unpoisonMemRegion(memBlock.addr, WV_MemBlockSize)
  body
  poisonMemRegion(memBlock.addr, WV_MemBlockSize)

template guardedAccess(memBlock: ptr MemBlock, body: untyped): untyped =
  unpoisonMemRegion(memBlock, WV_MemBlockSize)
  body
  poisonMemRegion(memBlock, WV_MemBlockSize)

# Data structures
# ----------------------------------------------------------------------------------

iterator backward(tail: ptr Arena): ptr Arena =
  ## Doubly-linked list backward iterator
  ## Not: we assume that the list is not circular
  ## and terminates via a nil pointer
```

sanitizers.nim | memory_pools.nim ×

weave > memory > memory_pools.nim > ...

```nim
    if arena.meta.free.isNil:
      arena.meta.free = first
      last.next.store(nil, moRelaxed)
    else:
      guardedAccess(arena.meta.free):
        arena.meta.free.prepend(last)
    arena.meta.used -= count
    when WV_SanitizeAddr:
      var cur = first
      while true:
        if cur == last:
          poisonMemRegion(cur, WV_MemBlockSize)
          break
        let next = cur.next.load(moRelaxed)
        poisonMemRegion(cur, WV_MemBlockSize)
        cur = cast[ptr MemBlock](next)

  debugMem:
    log("Arena   0x%.08x - TID %d - collected garbage, reclaimed %d block
      arena.addr, arena.meta.threadID.load(moRelaxed), beforeCollect - ar

func allocBlock(arena: var Arena): ptr MemBlock {.inline.} =
  ## Allocate from an arena
  preCondition: not arena.meta.free.isNil
  preCondition: arena.meta.used < arena.blocks.len

  arena.meta.used += 1
  result = arena.meta.free
  unpoisonMemRegion(result, WV_MemBlockSize)
  # The following acts as prefetching for the block that we are returning
  arena.meta.free = cast[ptr MemBlock](result.next.load(moRelaxed))

  postCondition: arena.meta.used in 0 .. arena.blocks.len
```

master ⊘ 0 ⚠ 0 github: Express Off

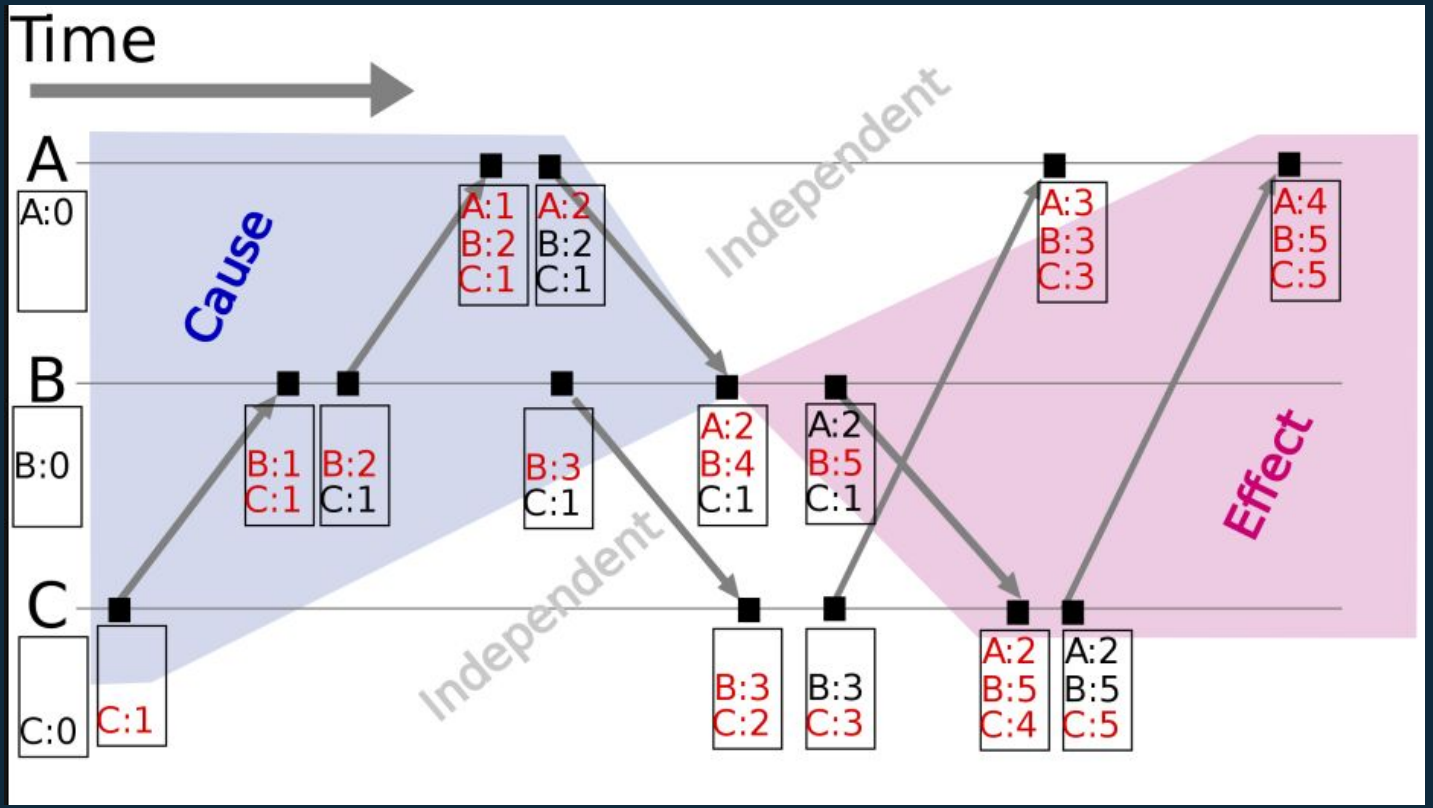
Sanitizers

- *nim c --cc:clang -r -d:release --debugger:native --passC:"-fsanitize=address" --passL:"-fsanitize=address" --outdir:build weave/memory/memory_pools.nim*
- Also *valgrind --tool=helgrind build/mybinary* for deadlocks (no need for recompilation but slow)
- And *valgrind --tool=drd build/mybinary* for data races

The limits

- At the very least we miss a notion of relative time between events that lead to the bad situation
- Ideally we have a trace or replay of the problematic sequence of events
- This does not prove the absence of synchronization bugs, what if the race is triggered once 1/1000000
- We debugged the low-level but what if we're having the wrong approach at a high-level?

Relative time via Vector Clock



Tracking causality of distributed events

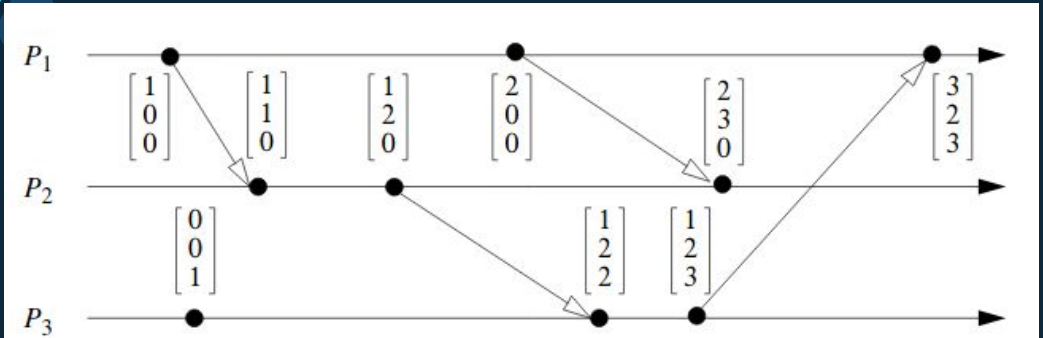
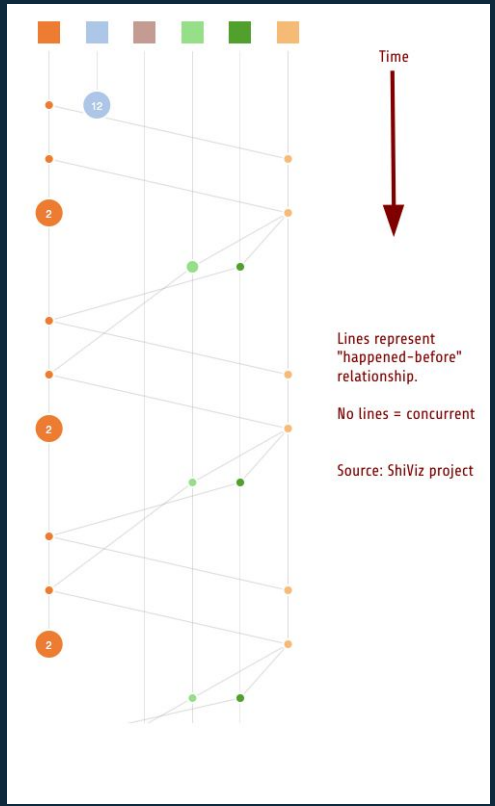


Figure 2: Events with their associated vector timestamps



- Detecting causal relationships in distributed computations: In Search of the Holy Grail Schwarz, Mattern, 1994

http://www.vs.inf.ethz.ch/publ/papers/holygrail.pdf

- https://bestchai.bitbucket.io/shiviz/

Implementing Vector Clock in ~70 lines

```nim
type
  Timestamp* = distinct int64

  CausalOrdering* = enum
    ## The ordering relation of 2 distributed events
    ## according to their Vector Clock.
    ##
    ## 2 events are concurrent when their observed timestamps are in "conflict"
    ## And we can't deduce a total ordering.
    Identical
    HappenedBefore
    HappenedAfter
    Concurrent

  VectorClock* = object
    ## A vector clock tracks the partial ordering of event in a distributed system.
    ## 2 clocks can be compared to check if
    ## - 1 event must have happened before the other
    ## - They could happen concurrently
    ## - They violate the laws of causality
    ##       You, a month ago • Add a vector clock for causal ordering of events
    ## The length is the number of processes/threads tracked
    ## The array of known local timestamps is indexed by the threadID (< len)
    localTS: array[MaxThreads, Timestamp]
    len: int8

proc inc(time: var Timestamp){.borrow.}
proc `$`*(time: Timestamp): string {.borrow.}
proc `<`*(a, b: Timestamp): bool {.borrow.}
proc `==`*(a, b: Timestamp): bool {.borrow.}

proc init*(clock: var VectorClock) {.inline.} =
  ## Initialize a vector clock
  zeroMem(clock.addr, sizeof(clock))

func tick*(clock: var VectorClock, tid: ThreadID) {.inline.}=
  ## Local tick in thread 'tid'
  if int8(tid) > clock.len:
    clock.len = int8(tid) + 1
  clock.localTS[tid.int].inc()

func synchronize*(a: var VectorClock, b: VectorClock) {.inline.}=
  ## Synchronize clock `a` with clock `b`
  if b.len > a.len:
    a.len = b.len
  for i in 0 ..< a.len:
    if a.localTS[i] < b.localTS[i]:
      a.localTS[i] = b.localTS[i]

func causality*(a, b: VectorClock): CausalOrdering =
  ## Returns the causal order between 2 vector clocks

  result = Identical
  for i in 0 ..< MaxThreads:
    let tsA = a.localTS[i]
    let tsB = b.localTS[i]

    if tsA == tsB:
      continue
    elif tsA < tsB:
      if result == HappenedAfter:
        return Concurrent
      result = HappenedBefore
    else:
      if result == HappenedBefore:
        return Concurrent
      result = HappenedAfter
```

```nim
# ----------------------------------------------------------------------------------
# Sanity checks

when isMainModule:
  import random

  proc clock[N: static int](a: array[N, int]): VectorClock =
    static: doAssert N ≤ MaxThreads
    result.len = N.int8
    for i in 0 ..< N:
      result.localTS[i] = TimeStamp(a[i])

  proc randomClock(rng: var Rand, N: static int): VectorClock =
    static: doAssert N ≤ MaxThreads
    result.len = N.int8
    for i in 0 ..< N:
      result.localTS[i] = TimeStamp(rng.rand(high(int)))

  proc causality[N: static int](a, b: array[N, int]): CausalOrdering =
    clock(a).causality(clock(b))

  proc test_causality() =
    # From https://en.wikipedia.org/wiki/Vector_clock#/media/File:Vector_Clock.svg

    doAssert causality([0, 0, 0], [0, 0, 1]) == HappenedBefore
    doAssert causality([0, 0, 1], [0, 1, 1]) == HappenedBefore
    doAssert causality([0, 1, 1], [0, 2, 1]) == HappenedBefore
    doAssert causality([0, 2, 1], [1, 2, 1]) == HappenedBefore
    doAssert causality([1, 2, 1], [2, 2, 1]) == HappenedBefore

    doAssert causality([0, 2, 1], [0, 3, 1]) == HappenedBefore
    doAssert causality([0, 3, 1], [0, 3, 2]) == HappenedBefore
    doAssert causality([2, 2, 1], [2, 4, 1]) == HappenedBefore
    doAssert causality([0, 3, 1], [2, 4, 1]) == HappenedBefore

    doAssert causality([0, 3, 2], [0, 3, 3]) == HappenedBefore
    doAssert causality([2, 4, 1], [2, 5, 1]) == HappenedBefore
    doAssert causality([0, 3, 3], [3, 3, 3]) == HappenedBefore
    doAssert causality([2, 5, 1], [2, 5, 4]) == HappenedBefore
    doAssert causality([0, 3, 3], [2, 5, 4]) == HappenedBefore

    doAssert causality([2, 5, 4], [2, 5, 5]) == HappenedBefore
    doAssert causality([3, 3, 3], [4, 5, 5]) == HappenedBefore
    doAssert causality([2, 5, 5], [4, 5, 5]) == HappenedBefore

    echo "Causality tests - Happened before - SUCCESS"

    # ----------------------------------------------------------

    doAssert causality([0, 3, 1], [1, 2, 1]) == Concurrent
    doAssert causality([0, 3, 1], [2, 2, 1]) == Concurrent
    doAssert causality([1, 2, 1], [0, 3, 1]) == Concurrent
    doAssert causality([2, 2, 1], [0, 3, 1]) == Concurrent

    doAssert causality([2, 4, 1], [0, 3, 2]) == Concurrent
    doAssert causality([2, 5, 1], [0, 3, 2]) == Concurrent
    doAssert causality([2, 4, 1], [0, 3, 3]) == Concurrent
    doAssert causality([2, 5, 1], [0, 3, 3]) == Concurrent

    doAssert causality([3, 3, 3], [2, 5, 4]) == Concurrent
    doAssert causality([3, 3, 3], [2, 5, 5]) == Concurrent

    doAssert causality([3, 3, 3], [2, 4, 1]) == Concurrent
    doAssert causality([3, 3, 3], [2, 5, 1]) == Concurrent

    doAssert causality([0, 3, 2], [1, 2, 1]) == Concurrent
    doAssert causality([0, 3, 1], [2, 2, 1]) == Concurrent
    doAssert causality([2, 2, 1], [0, 3, 3]) == Concurrent

    echo "Causality tests - Concurrency - SUCCESS"
```

Vector Clock – The gist

- In multithreaded programs or distributed systems or micro-services, classic timestamps are unreliable to know the order of events.

- Vector clocks give you partial ordering (Happened Before, Happened After, Concurrent)

- If you have 4 services, you only need to add

  - “*clock(0, 0, 0, 0)*” at time 0

  - “*clock(0, 2, 1, 0)*” when the logging thread knows that thread 1 advanced twice and thread 3 advanced once.

3 Designing

Multithreaded programs

Engineering

- Civil engineering
- Mechanical engineering
- Material engineering
- Industrial engineering
- Electrical engineering
- Nuclear engineering
- Chemical engineering
- Aerospace engineering
- Marine engineering

Years of study, design, checks, and audits go before building any of the other “engineering” projects.

The higher the risks and the stakes, the longer the design phase.

Software engineering

- Multithreaded programs must be designed
  - I.e. We want a **blueprint** that we can check
- Many of their properties are emergent at a high-level

Example behaviour:

- “Always open the door and let the other pass before you”

What if 2 programs exhibits this behavior?

- Contention to open the door (“Always open”).
- Deadlock as no one wants to go through.
- Deadlock if the door is already opened.

Blueprints

- Code changes fast:
  - Comments, documentation go stale
- In many cases the devil is in the implementation details (try reproducing an academic paper, for example on garbage collection)
- A blueprint of what we have implemented may be more useful than a planned design, in particular to find corner cases

State Machines

- The hardest part of multithreading is handling state
- Visualizing the possible states and events your components react to is invaluable to detect “design bugs”
- State machines (but also Petri Nets)
  lend themselves to formal verification

Hardware engineering

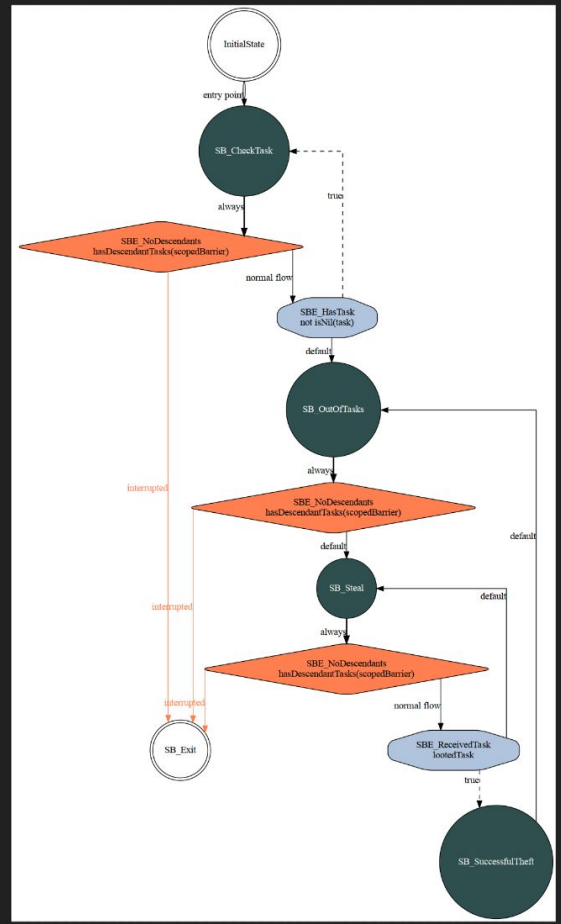
- Designing a chip, router, memory model, cache coherence protocol, embedded device requires modelling.
- They use state machines to validate the design before thousands are poured into a doomed to fail prototypes

Synthesis - A compile-time state machine generator

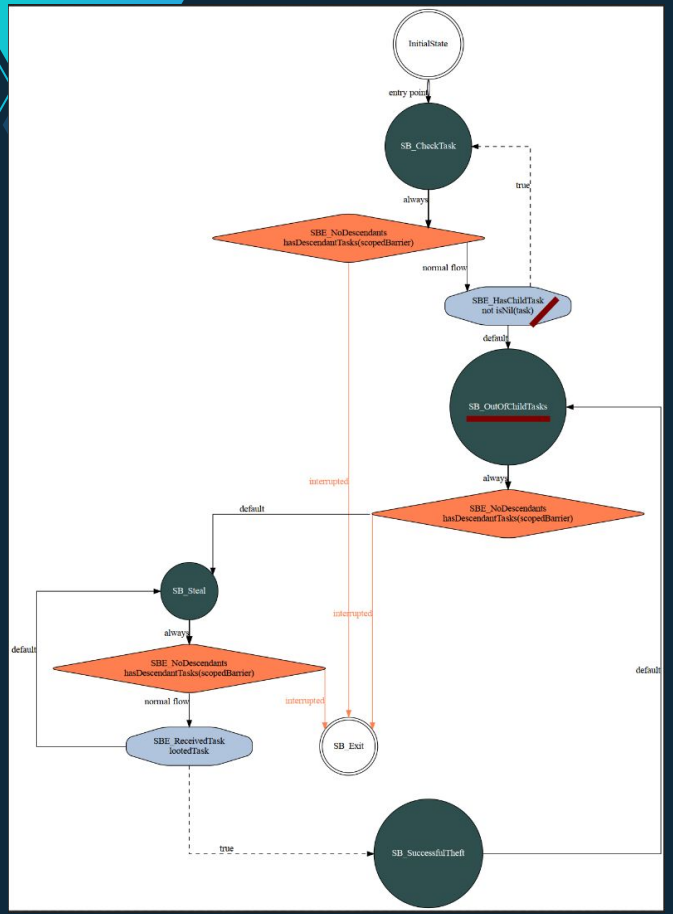
sync_scope.nim

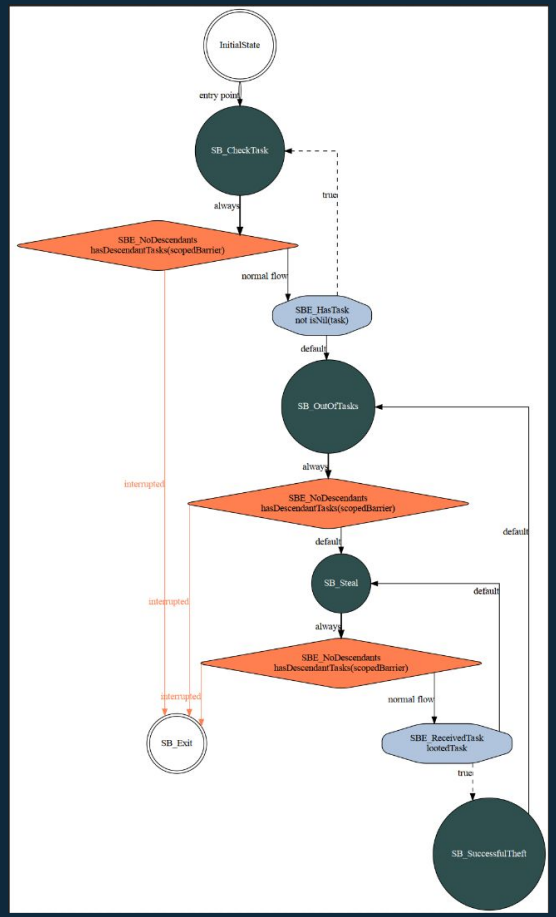
```
weave > state_machines > sync_scope.nim > ...
    fin: SB_OutOfTasks

  # ------------------------------------------------
  # These states are interrupted when the scope has no more descendant

  behavior(syncScopeFSA):
    ini: SB_OutOfTasks
    transition:
      # Steal and hope to advance towards the child tasks in other workers' queues.
      trySteal(isOutOfTasks = false) # Don't sleep here or we might stall the runtime
      profile_start(idle)
    fin: SB_Steal

  onEntry(syncScopeFSA, SB_Steal):
    let lootedTask = recvElseSteal(task, isOutOfTasks = false) # Don't sleep here or we might stall the runtim

  implEvent(syncScopeFSA, SBE_ReceivedTask):
    lootedTask

  behavior(syncScopeFSA):
    ini: SB_Steal
    event: SBE_ReceivedTask
    transition: profile_stop(idle)
    fin: SB_SuccessfulTheft

  behavior(syncScopeFSA):
    steady: SB_Steal
    transition:
      profile_stop(idle)
      dispatchToChildrenAndThieves()
      profile_start(idle)

  # ------------------------------------------------
  # This is not interrupted when scoped barrier has no descendant tasks

  behavior(syncScopeFSA):
    ini: SB_SuccessfulTheft
    transition:
      ascertain: not task.fn.isNil
      debug: log("Worker %2d: syncScope 3 - stoled tasks\n", myID())
      TargetLastVictim:
        if task.victim ≠ Not_a_worker:
          myThefts().lastVictim = task.victim
          ascertain: myThefts().lastVictim ≠ myID()

      if not task.next.isNil:
        profile(enq_deq_task):
          # Add everything
          myWorker().deque.addListFirst(task)
          # And then only use the last
          task = myWorker().deque.popFirst()

      StealAdaptative:
        myThefts().recentThefts += 1

      # Share loot with children workers
      debug: log("Worker %2d: syncScope 4 - sharing work\n", myID())
      shareWork()

      # Run the rest
      profile(run_task):
        execute(task)
      profile(enq_deq_task):
        # The memory is re-used but not zero-ed
        localCtx.taskCache.add(task)
    fin: SB_OutOfTasks

  # ------------------------------------------------

  synthesize(syncScopeFSA):
    proc wait(scopedBarrier: var ScopedBarrier)
```

sync_scope.png



Visual bug hunting with state machines





This is a barrier where a worker helps other until all descendants tasks spawned are done.

On the left, the old implementation, the worker was sometimes trying to “help” while all the other workers actually finished, i.e. deadlock.

The design bug was that the worker only drained direct child tasks and may have grandchildren in its queue

Share nothing or be SLOW

Starvation

Latencies

Overheads

Waiting for contention

(Credit: The HPX multithreading runtime)

Functions & messages over mutable objects

- Prefer side-effect free functions
- Avoid mutable objects or at least don't share mutable fields across threads (one branch of a tree data structure per-thread for example)
- Avoid OOP unless you are ready to split your "Person" and "Dog" instances across threads

  and deal with the contention.

  Prefer plain objects like object variants.

  Copy is cheaper than contention and synchronization

https://github.com/mratsim/trace-of-radiance/blob/99f7d85d/trace_of_radiance/support/emulate_classes_with_ADTs.nim#L251-L268

Functions & messages over mutable objects

Isolate synchronization to few data structures that will be under higher scrutiny, like channels.

Functions & messages over mutable objects

Functions & messages over mutable objects

- Function and messages allows you to do black-box testing and ease fuzzing.
- For example you can write a runner that has different scenarios, provided in a convenient format (JSON, YAML) and produces certain sequences of messages and verifying the output messages.
- Randomize the message order and verify that output is still consistent.

3

Correct-by-construction

Multithreaded programs

Correct-by-construction multithreading

- By design, your software is free of multithreading bugs:
  - Deadlocks
  - Livelocks
  - Data races
  - Object lifetime bugs (use before init, use-after-free)
- More importantly it does what you specified

Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth.

- The story of how I found a deadlock bug in glibc and musl.

- I was implementing a “2-phase commit” protocol to put a thread to sleep if there is no work.
  - The events:
    - I want to sleep
    - Sleeping
    - Wakeup!
- A race between sleep and wakeup will deadlock your thread

80 lines of code, what could go wrong?

```nim
type
  EventNotifier* = object
    ## Multi Producers, Single Consumer event notification
    ## This is can be seen as a wait-free condition variable for producers
    ## that avoids them spending time in expensive kernel land due to mutexes.
    ##
    ## This data structure should be associated with a MPSC channel
    ## to notify that an "event" happened in the channel.
    ## It avoid spurious polling of empty channels,
    ## and allow parking of threads to save on CPU power.
    ##
    ## See also: binary semaphores, eventcounts
    ## On Windows: ManuallyResetEvent and AutoResetEvent
    when supportsFutex:
      # ---- Consumer specific ----
      ticket{.align: WV_CacheLinePadding.}: uint8 # A ticket for the consumer to sleep in a phase
      # ---- Contention ---- no real need for padding as cache line should be reloaded in case of contention anyway
      futex: Futex                        # A Futex (atomic int32 that can put thread to sleep)
      phase: Atomic[uint8]                # A binary timestamp, toggles between 0 and 1 (but there is no atomic "not")
      signaled: Atomic[bool]              # Signaling condition
    else:
      # ---- Consumer specific ----
      lock{.align: WV_CacheLinePadding.}: Lock # The lock is never used, it's just there for the condition variable
      ticket: uint8                       # A ticket for the consumer to sleep in a phase
      # ---- Contention ---- no real need for padding as cache line should be reloaded in case of contention anyway
      cond: Cond                          # Allow parking of threads
      phase: Atomic[uint8]                # A binary timestamp, toggles between 0 and 1 (but there is no atomic "not")
      signaled: Atomic[bool]              # Signaling condition

func initialize*(en: var EventNotifier) {.inline.} =
  when supportsFutex:
    en.futex.initialize()
  else:
    en.cond.initCond()
    en.lock.initLock()
  en.ticket = 0
  en.phase.store(0, moRelaxed)
  en.signaled.store(false, moRelaxed)

func `=destroy`*(en: var EventNotifier) {.inline.} =
  when not supportsFutex:
    en.cond.deinitCond()
    en.lock.deinitLock()

func `=`*(dst: var EventNotifier, src: EventNotifier) {.error: "An event notifier cannot be copied".}
func `=sink`*(dst: var EventNotifier, src: EventNotifier) {.error: "An event notifier cannot be moved".}
```

```nim
func prepareToPark*(en: var EventNotifier) {.inline.} =
  ## The consumer intends to sleep soon.
  ## This must be called before the formal notification
  ## via a channel.
  if not en.signaled.load(moRelaxed):
    en.ticket = en.phase.load(moRelaxed)

proc park*(en: var EventNotifier) {.inline.} =
  ## Wait until we are signaled of an event
  ## Thread is parked and does not consume CPU resources
  ## This may wakeup spuriously.
  if not en.signaled.load(moRelaxed):
    if en.ticket == en.phase.load(moRelaxed):
      when supportsFutex:
        en.futex.wait(0)
      else:
        en.cond.wait(en.lock) # Spurious wakeup are not a problem
  en.signaled.store(false, moRelaxed)
  when supportsFutex:
    en.futex.initialize()
  # We still hold the lock but it's not used anyway.

proc notify*(en: var EventNotifier) {.inline.} =
  ## Signal a thread that it can be unparked

  if en.signaled.load(moRelaxed):
    # Another producer is signaling
    return
  en.signaled.store(true, moRelease)
  discard en.phase.fetchXor(1, moRelaxed)
  when supportsFutex:
    en.futex.store(1, moRelease)
    en.futex.wake()
  else:
    en.cond.signal()
```

The version with lock + condition variable deadlocks on Linux but not Mac or Windows.

Futex is the raw kernel primitive that glibc and Windows synchronization use to implement locks and condition variables.

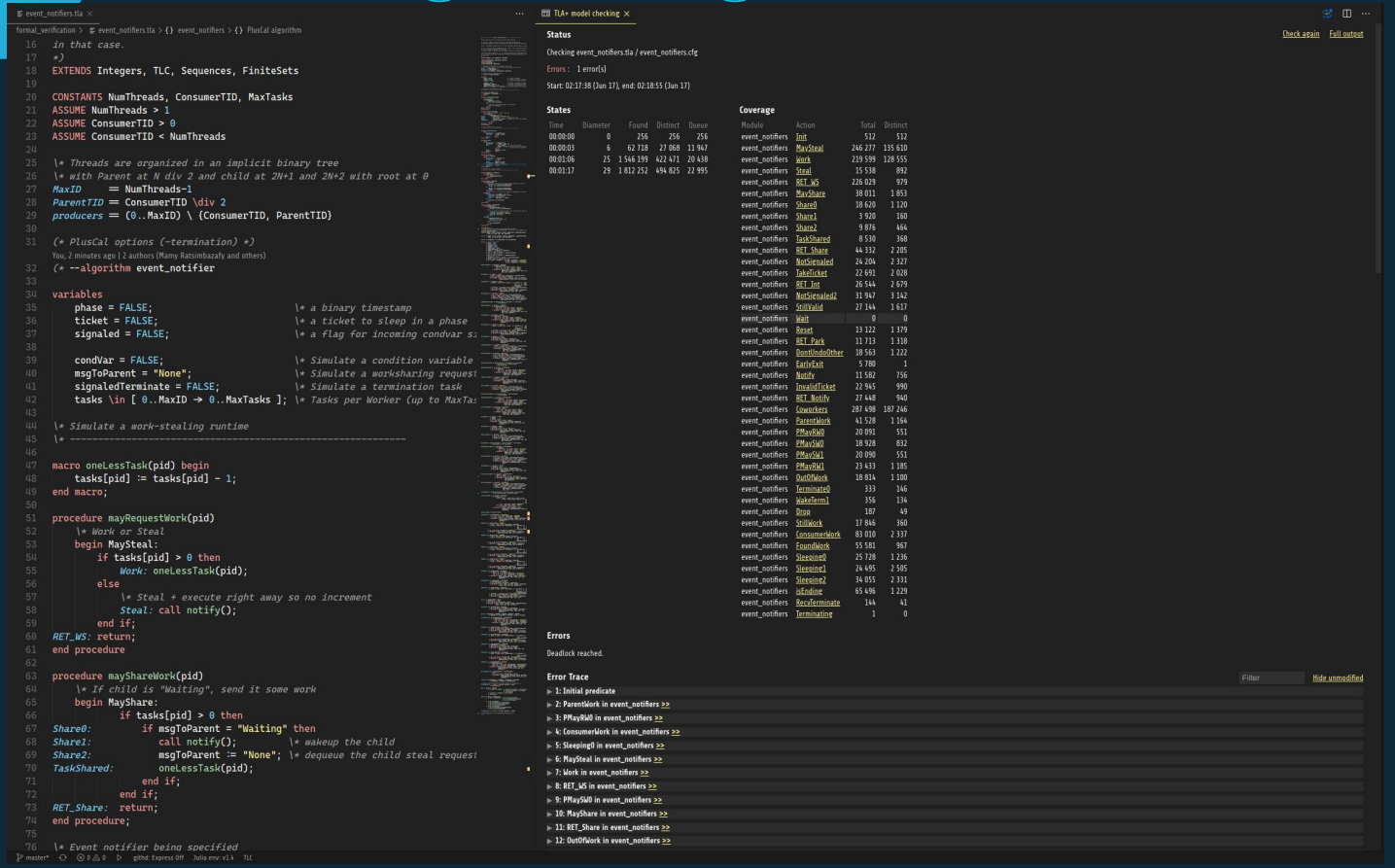
Futex don’t deadlock.

Proving my code bug-free

- Formal verification via model checking: verifying all possible state my threads can be (i.e. preparing to sleep, sleeping or waking another thread) and trying all possible threads interleaving

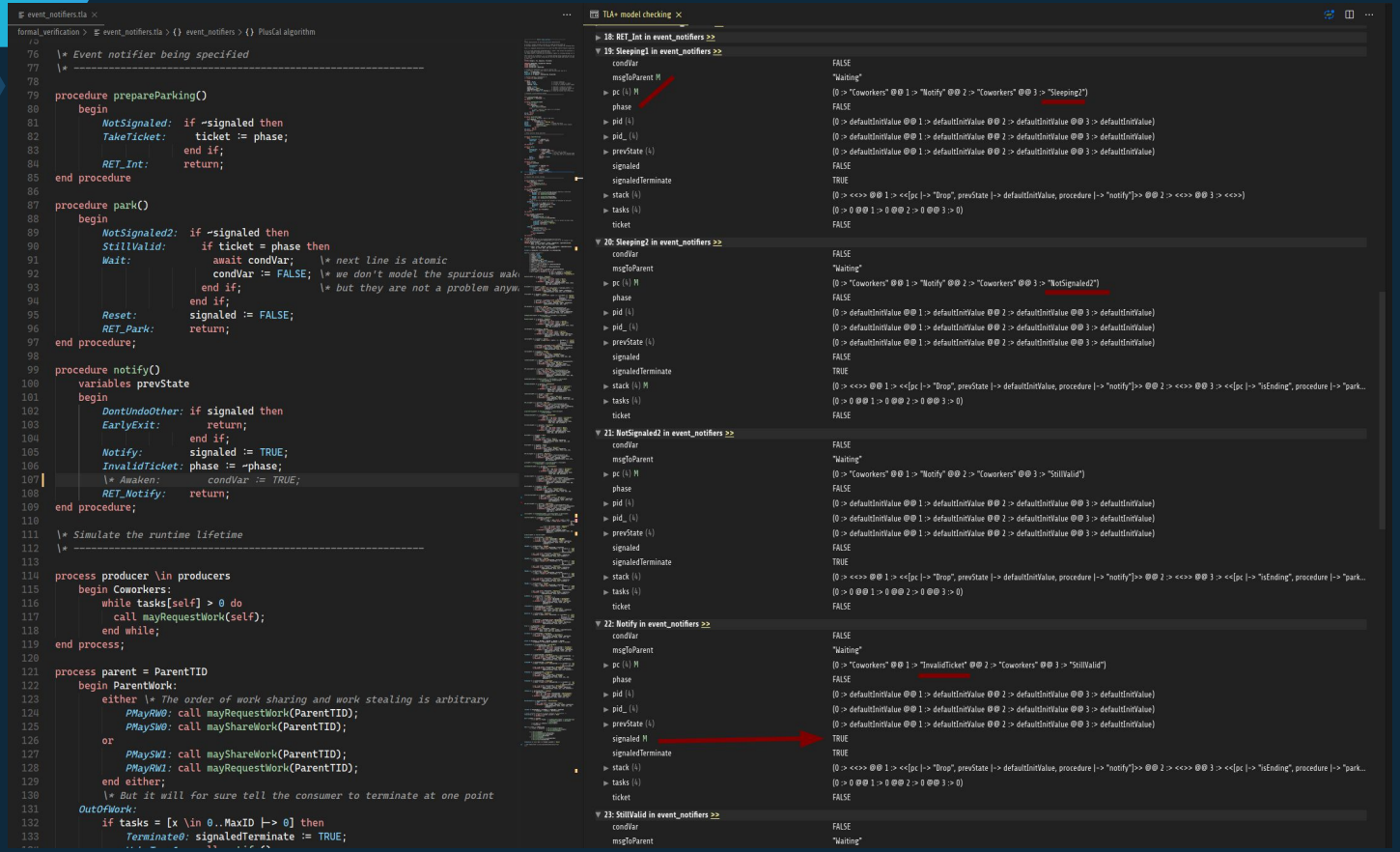
- Using TLA+

Proving code bug-free



Notice how this simple data structure has millions of states

Proving code bug-free



Proving my code bug-free

- This helped me debug 2 things I probably wouldn't be able to debug otherwise:
  - Livelock sequences of 20 to 40 steps that repeated themselves when I tried to properly order my wakeups.
  - Proving that my code was bug-free and so that even if it seemed improbable, it was glibc that was buggy.

Future goal

- Bring model checking to Nim

  - ThreadCollider: Formal Verification of Nim concurrent programs
  - Test a program correctness by testing all possible thread interleaving without having to rewrite it in another language.
  - https://github.com/nim-lang/RFCs/issues/222
  - https://github.com/mratsim/weave/pull/127

Multithreaded programs
The Good, the Bad and the Buggy

Mamy Ratsimbazafy
mamy@numforge.co

Weave
https://github.com/mratsim/weave