CS3211 Tutorial 4

Lock free programming in C++

Learning objectives

Make fine-grained queue in lecture lock-free

- CAS
- ABA
- UAF (with a different method:))
- Data race (again)

PollEv.com/travistoh125

Synchronization in progress

Queue is FIFO (default).

Singly Linked-List

Pop <-> Dequeue

Push <-> Queue

For diagrams, first element (front()) is on the left.

PollEv.com/travistoh125

Warm-up & recap: What are the possible races?

```cpp
std::shared_ptr<T> try_pop()
{
    if(!front)
    {
        return std::shared_ptr<T>();
    }
    std::shared_ptr<T> const res(
        std::make_shared<T>(std::move(front->data)));
    std::unique_ptr<node> const old_front=std::move(front);
    front=std::move(old_front->next);
    if(!front)
        back=nullptr;
    return res;
}
void push(T new_value)
{
    std::unique_ptr<node> p(new node(std::move(new_value)));
    node* const new_back=p.get();
    if(back)
    {
        back->next=std::move(p);
    }
    else
    {
        front=std::move(p);
    }
    back=new_back;
}
```

Warm-up & recap: What are the possible races?

Both front (line 28) and back (line 40)

point to the same node.

Hence, old_front->next and back->next point to same location.

Line 29 reads, line 40 writes

```
20  std::shared_ptr<T> try_pop()
21  {
22      if(!front)
23      {
24          return std::shared_ptr<T>();
25      }
26      std::shared_ptr<T> const res(
27          std::make_shared<T>(std::move(front->data)));
28      std::unique_ptr<node> const old_front=std::move(front);
29      front=std::move(old_front->next);
30      if(!front)
31          back=nullptr;
32      return res;
33  }
34  void push(T new_value)
35  {
36      std::unique_ptr<node> p(new node(std::move(new_value)));
37      node* const new_back=p.get();
38      if(back)
39      {
40          back->next=std::move(p);
41      }
42      else
43      {
44          front=std::move(p);
45      }
46      back=new_back;
47  }
```

Alternative “solution”: Just lock it

When the linked list is empty and we want to insert?

1. Lock **back** pointer
2. Lock **front** pointer (why?)
3. Insert node, update pointers
4. Unlock both

When the linked list has one element and we want to remove it?

1. Lock **front** pointer, node
2. Check if node->next is null
3. Lock **back** pointer
4. Remove node, update pointers
5. Unlock

See anything wrong?

Recap: Why we need a dummy node?

Push (with dummy)

1. Create new dummy node
2. Lock **back**, node
3. Update **back** pointer to new dummy node
4. Update old dummy node with data and new next pointer
5. Unlock all

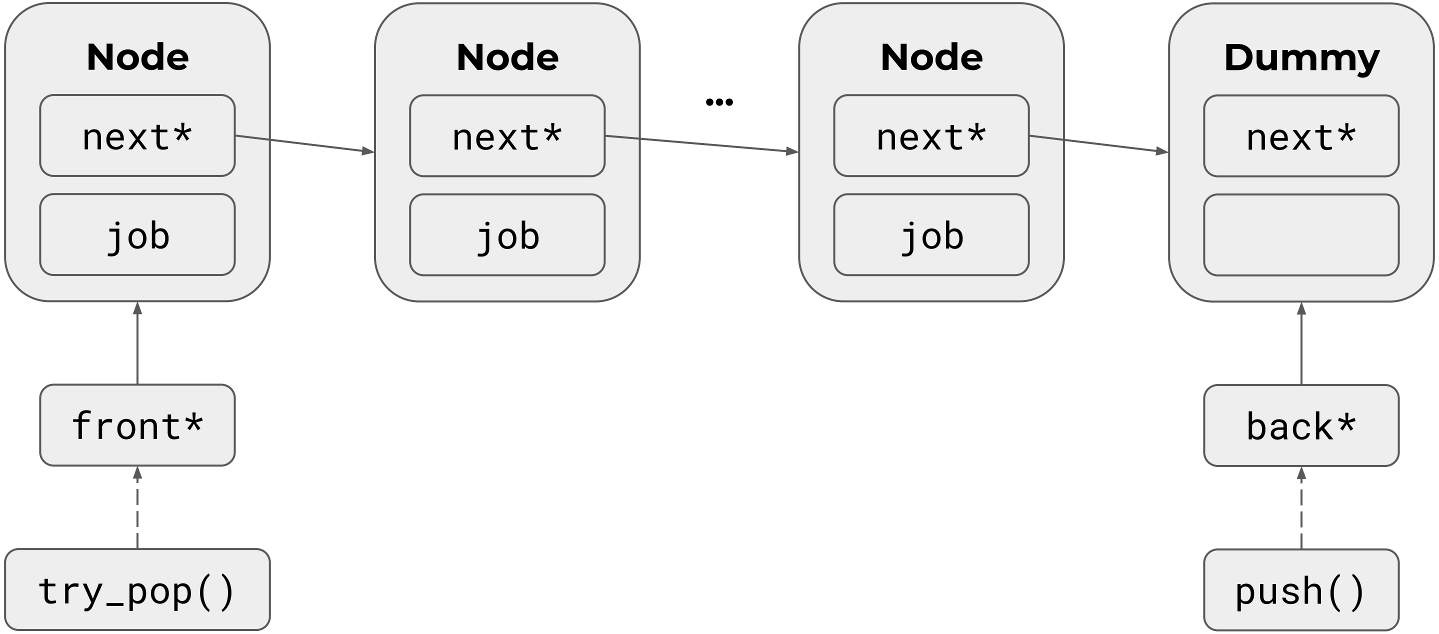
Pop (with dummy)

1. Lock **front**, node
2. Check if next is nullptr, if yes, return
3. Update next pointer of node
4. Unlock all

No more inverse locking sequence! yay?

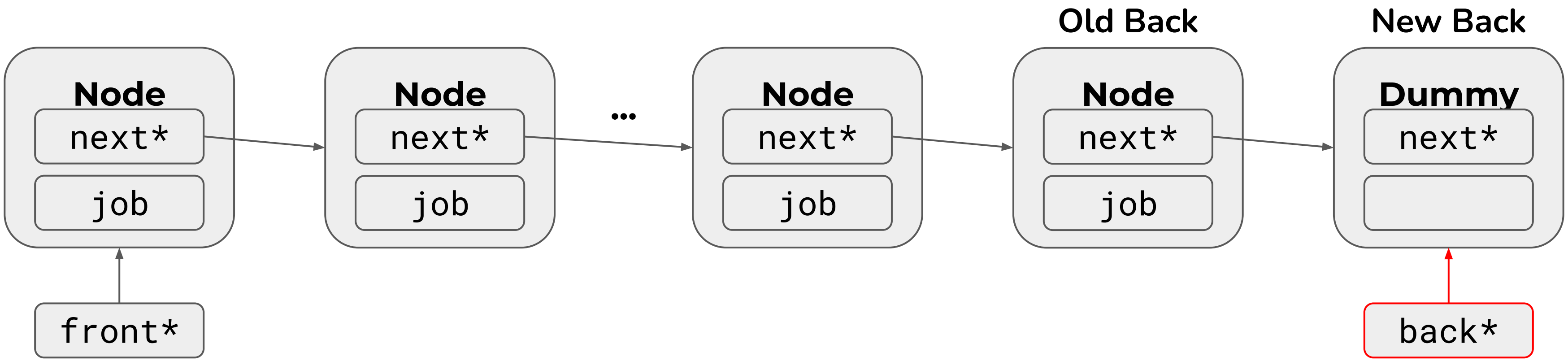
Lock-free Queue

Lock Free Queue



Producers push()

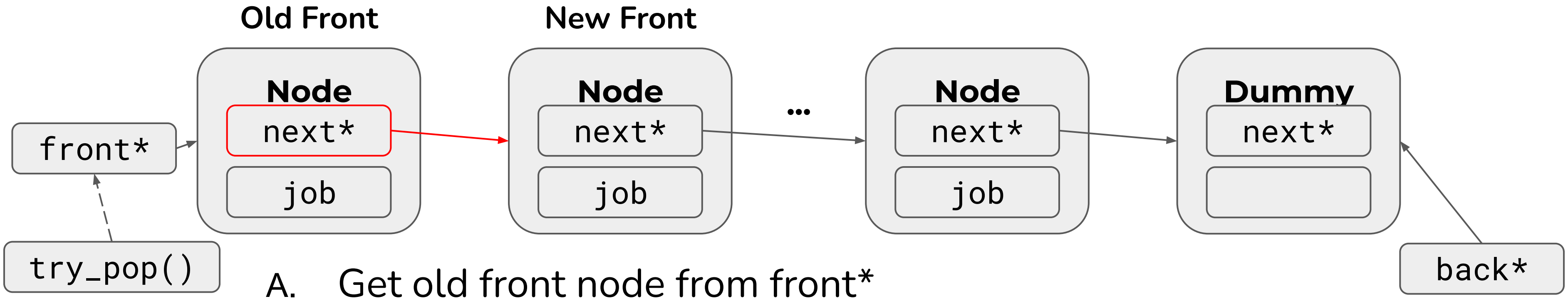
Producers push() - Naive Attempt



A. Create a new back node
B. Get old back node from <u>back*</u>
C. Set the new job in the old back node.
D. Point old back node at new back node
E. Also update <u>back*</u> so other producers know where the new end of the queue is.

Consumers try_pop()

Consumers try_pop() - Naive Attempt



A. Get old front node from front*
B. Check whether old front node is a dummy or not, by reading its next pointer.
   a. If it's a dummy, queue is empty so return empty
   b. Otherwise, continue – there are some jobs in the queue.
   c. Update front* to point at the next node
   d. Return the job in old front node

Making it lock free

Race conditions vs. data races

Race condition → not always bad, might not always happen

- The outcome depends on the relative ordering of execution of operations on two or more threads
- The threads race to perform their respective operations
- Usually, *race condition* is a flaw that occurs when the timing or ordering of events affects a program's correctness.

• Might be same or diff mem location

vs

Data race

- A *data race* happens when there are two memory accesses in a program where both:
  - target the same location
  - are performed concurrently by two threads
  - are not reads
  - are not synchronization operations
- Causes undefined behavior

↳ actually undefined or multiple possible behaviours (probabilistic)

Race conditions

P-P

A producer may get in another producer’s way, eg. by overwriting their job.

C-C

A consumer may get in another consumer’s way, eg. by consuming a node meant for another consumer

P-C

A producer may not be correctly synchronised with consumers, causing them to read an invalid state.

Solving P-C (Producer release-write)

Push()

A. Create a new back node
B. Get old back node from back*.
C. Set the new job in the old back node.
D. Point old back node at **new back node**
E. Also update back* so other producers know where the new end of the queue is.

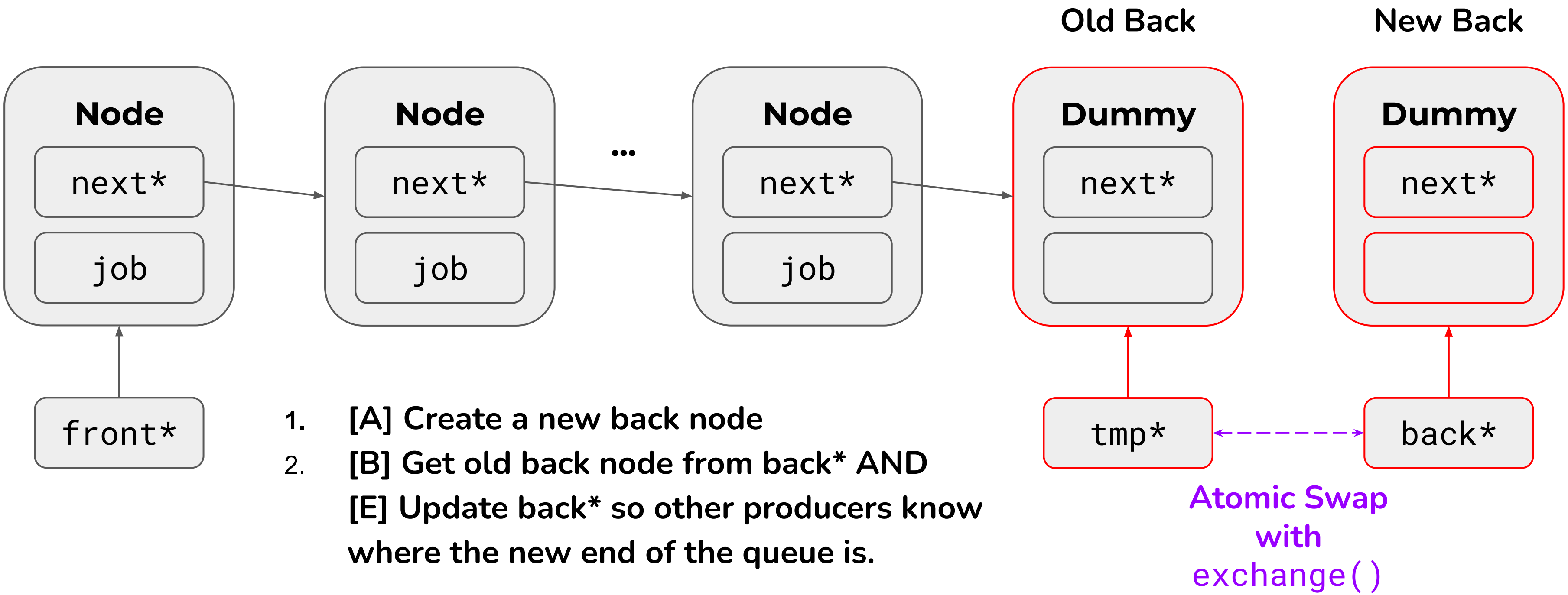
Pop()

A. Get old front node from front*
B. Check whether old front node is a dummy or not, by reading its **next pointer.**
   a. If it's a dummy, queue is empty so return empty
   b. Otherwise, continue – there are some jobs in the queue.
   c. Update front* to point at the next node
   d. Return the job in old front node

Issue: producer racing with consumer

We need to form a **Synchronises-With** relationship from the construction of the Node, to the access and destruction of the node. To do that, we use **`acquire-release`** semantics and the **`next`** pointer as the shared memory location

Solving P-P (m_queue_back.exchange())

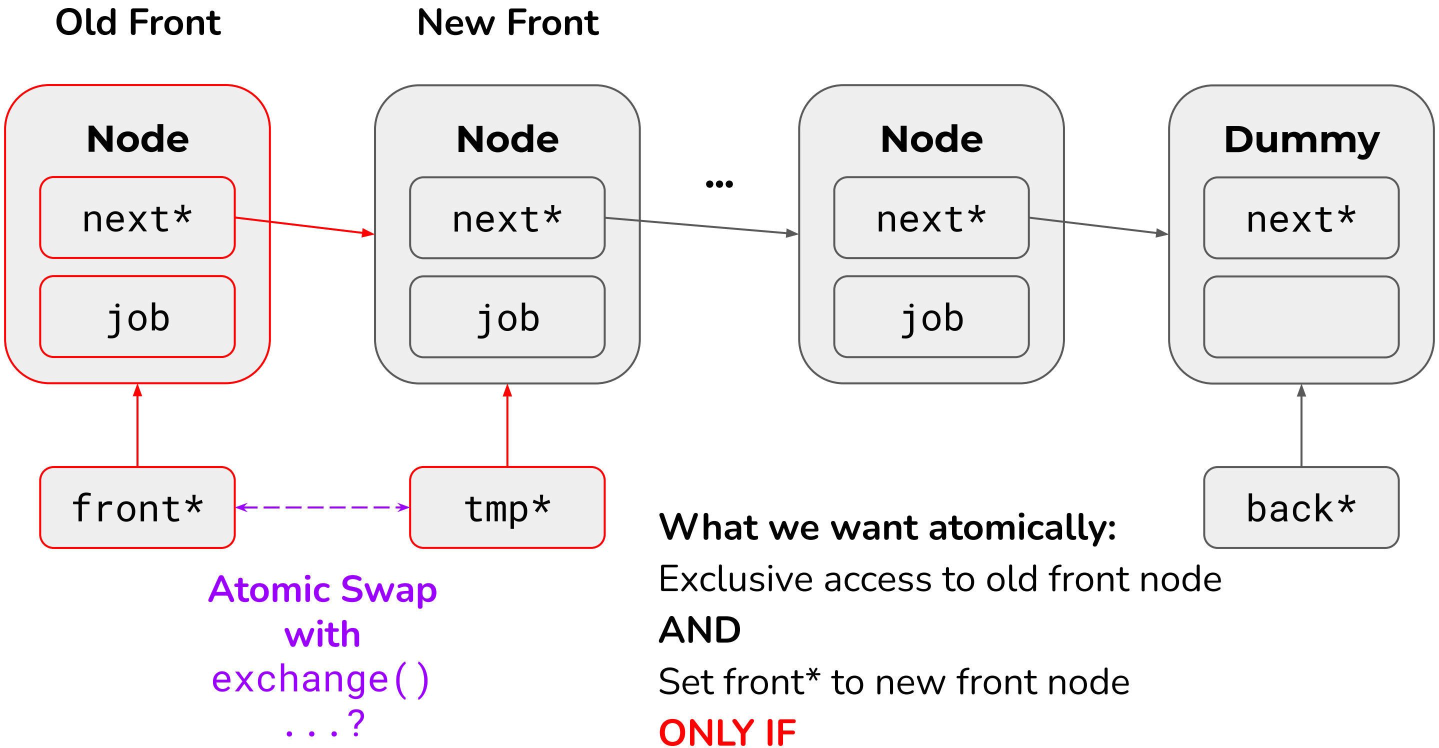


1. **[A] Create a new back node**
2. **[B] Get old back node from back* AND [E] Update back* so other producers know where the new end of the queue is.**

Solving C-C

Can I just do this?

Node* node_to_consume = m_queue_front.exchange(m_queue_front->next);



What we want atomically:

Exclusive access to old front node

AND

Set front* to new front node

ONLY IF

Old front node is not dummy (next* is not nullptr)

Compare-And-Swap (CAS) Pattern

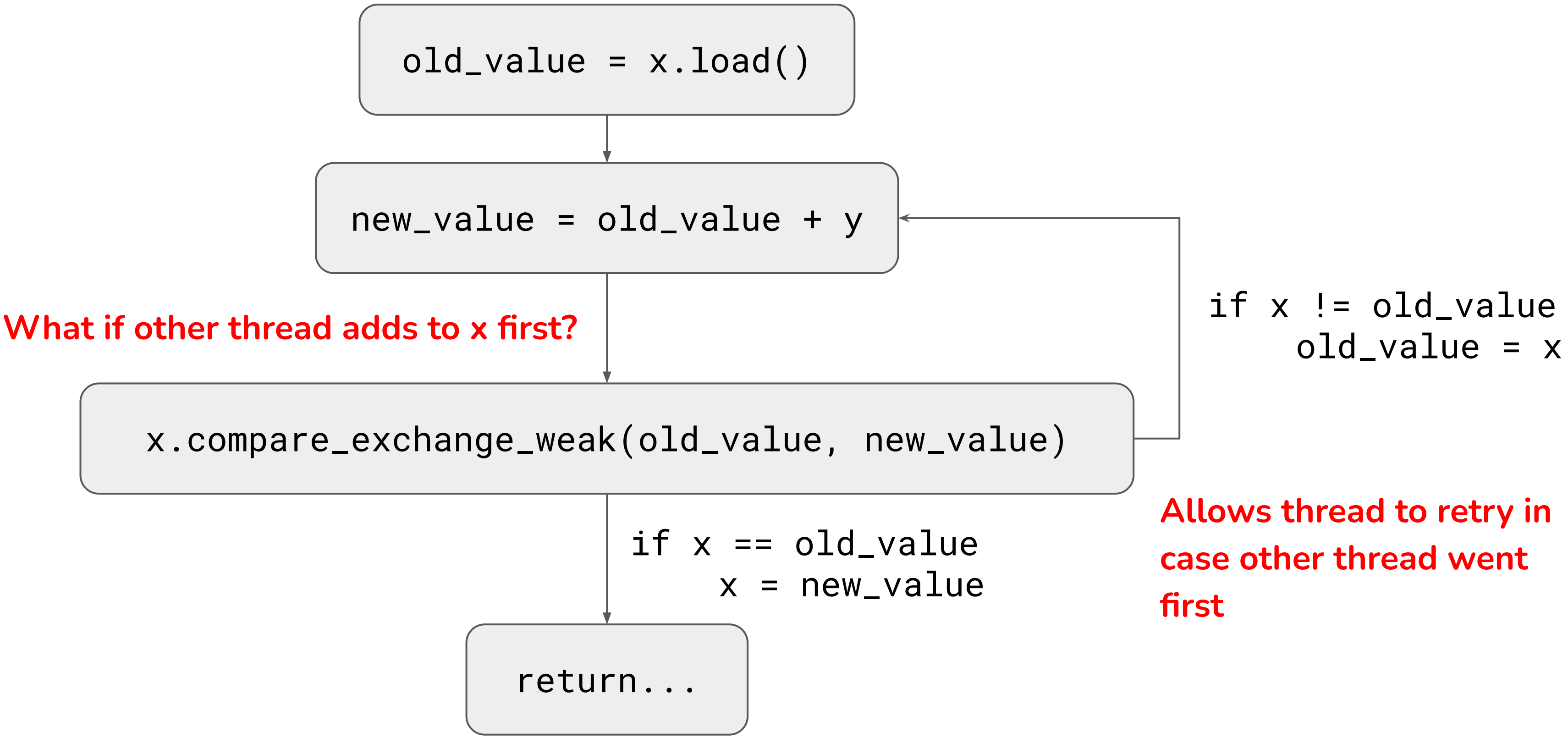
```
bool compare_exchange_weak

( T& expected, T desired, std::memory_order success)
```

```
Usage:
```

```
value_to_be_swapped_wait.compare_exchange_weak(expected_value, new_value)
```

Compare-And-Swap (CAS) Pattern



Compare-and-swap

Strong vs Weak

weak can spuriously fail, **strong** can't

tl;dr: Use **`_weak`** versions in loops, **`_strong`** if not a loop or failure requires heavy recomputation

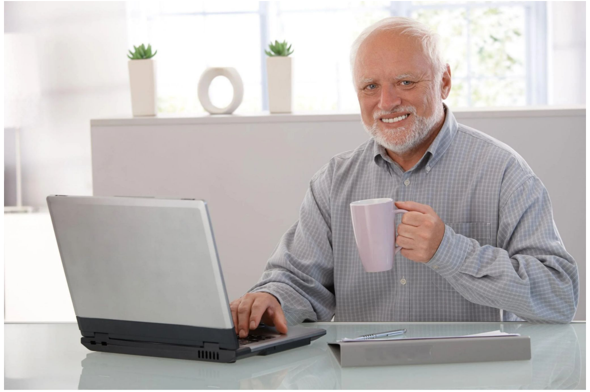
Making it lock free (Consumers)

```
std::optional<Job> try_pop() {
/* A */ Node *old_node = jobs_front;
/* B */ if (old_node->next == QUEUE_END)
  /* Bi */ return std::nullopt;
/* C */ jobs_front = jobs_front->next;
        Job job = old_node->job;
        delete old_node;
/* D */ return job;
}
```

```
std::optional<Job> try_pop() {
/* a */ Node *old_front = jobs_front.load(stdmo::relaxed);
/* b */ while (true) {
  /* a */ Node *new_front =
              old_front->next.load(stdmo::acquire);
          if (new_front == QUEUE_END) {
    /* i */ return std::nullopt;
          }
  /* b */ if (jobs_front.compare_exchange_weak(
                  old_front, new_front, stdmo::relaxed)) {
  /* c */   break;
          }
        }
        Job job = old_front->job;
        delete old_front;
        return job;
}
```

Keep trying to remove the first node if it is not a dummy

Problem #1: The ABA problem

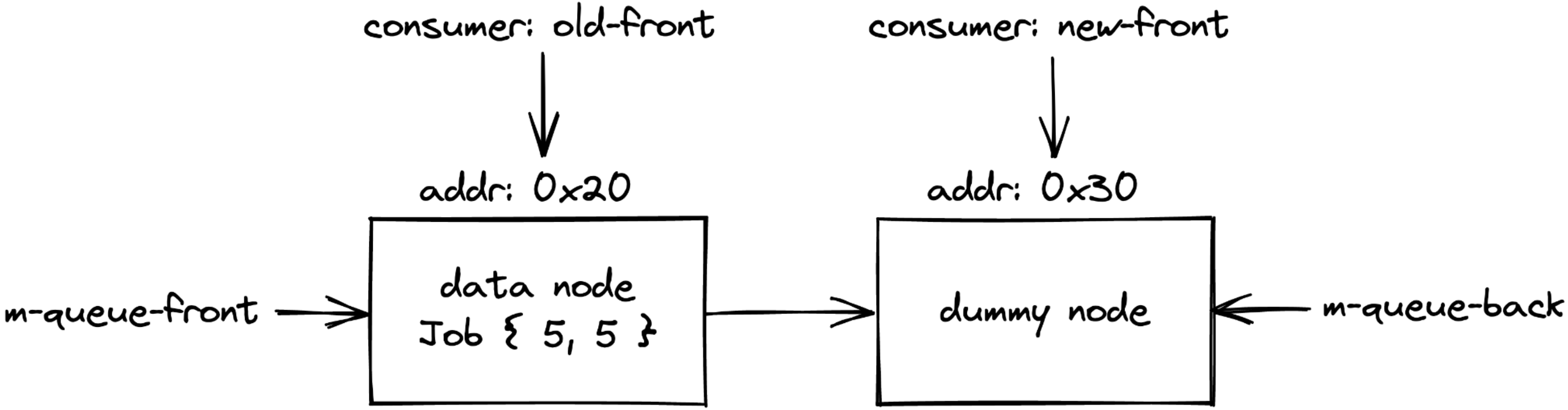


Lock free woes #1 – ABA problem

Given `std::atomic<Node*> front;`
we want to perform a CAS on `front`, to the next node with
`std::atomic<Node*> next = 0x30`;

1) Suppose `front` has a value of `0x20`. – A
2) We get the value of `next` and will perform the swap if front is still `0x20`
3) 30,000 years passed, nodes have come and gone – B
    a) The `front` is `0x20` again. But now `0x30` is long buried in the grave
4) We see that the current value in `front` is equal to `0x20`, so we swap.
5) The front now points to the grave, the rest of the nodes are not accessible

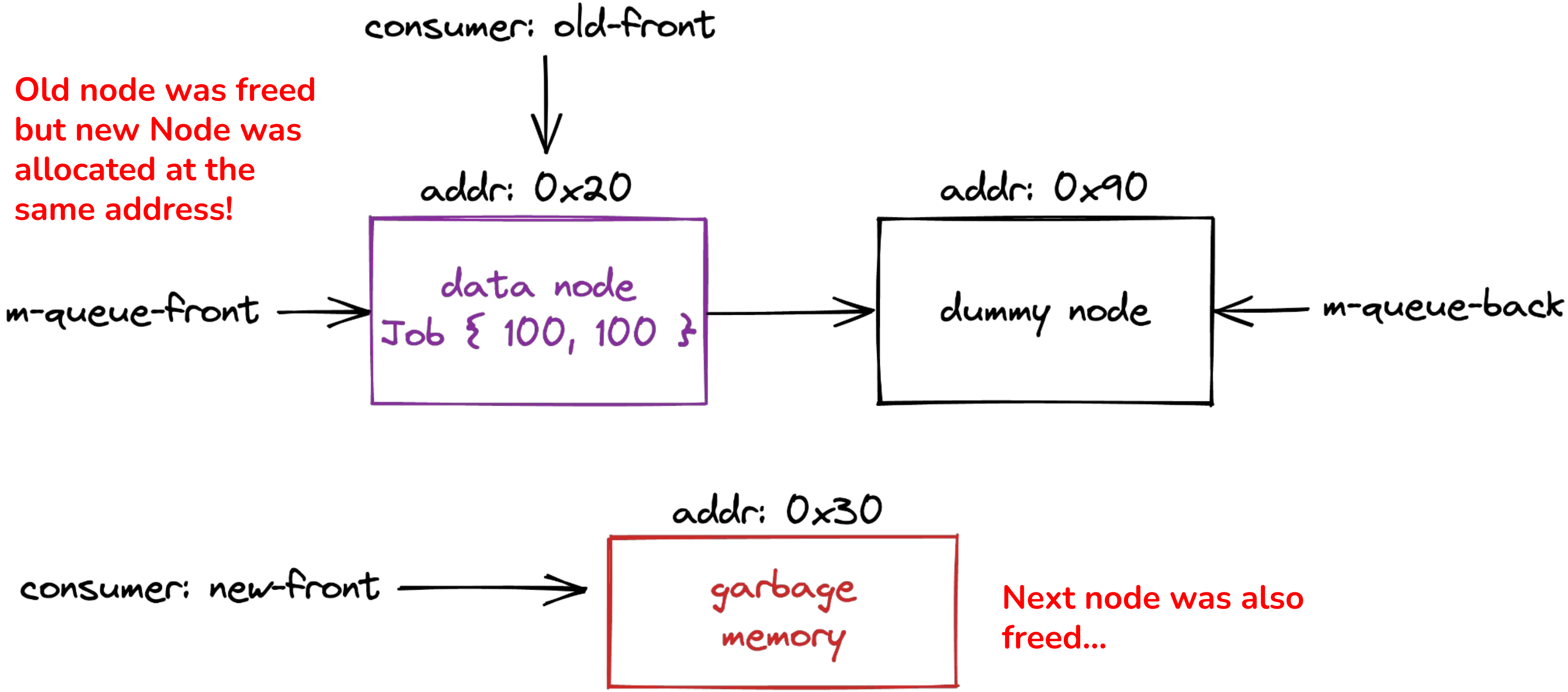
Problem #1: the ABA problem




A FEW
MOMENTS LATER

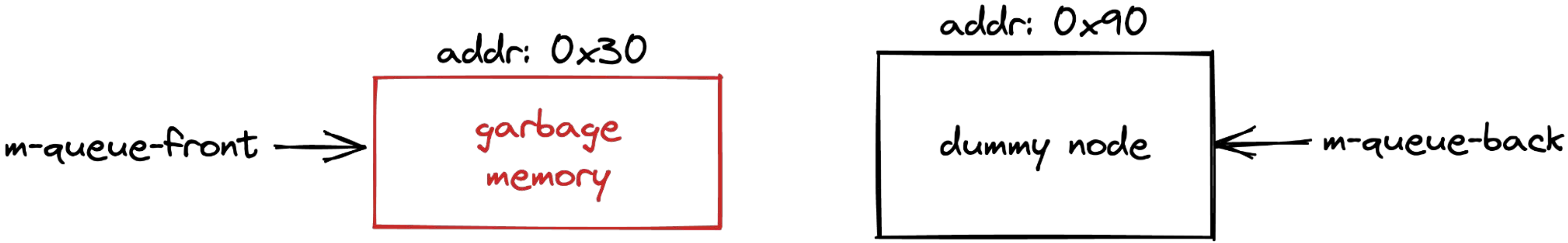
Problem #1: the ABA problem

Old node was freed but new Node was allocated at the same address!



Next node was also freed...

Problem #1: the ABA problem



CAS Succeeds!
Everything has gone wrong

How to eradicate ABA (generation-counted pointers)

Problem: Memory addresses are re-used when memory is allocated and deallocated.

`CAS` is susceptible to ABA, but not **`exchange`**. Why?

Intuition: Make the value to be compared unique.
Like how we have NRICs since our names aren't unique, give the pointers a "UUID"!

How to eradicate ABA (generation-counted pointers)

```cpp
struct alignas(16) GenNodePtr
{
    Node* node;
    uintptr_t gen;
};

static_assert(std::atomic<GenNodePtr>::is_always_lock_free);

alignas(64) std::atomic<Node*> m_queue_back;       // producer end
alignas(64) std::atomic<GenNodePtr> m_queue_front; // consumer end
```

2^64 is a big number so it's **unlikely** to suffer from ABA

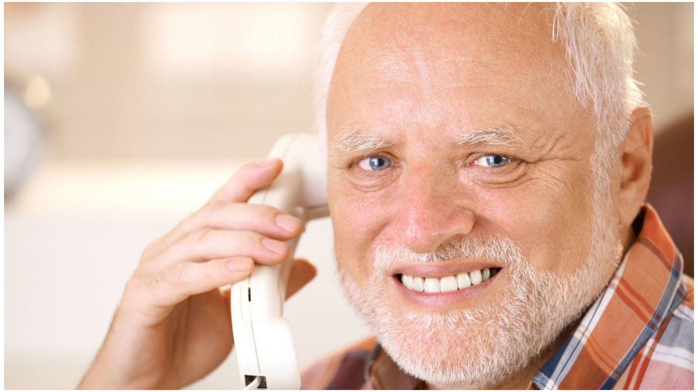
And well there's a squiggly line cause DWCAS

How to eradicate ABA (generation-counted pointers)

```
GenNodePtr old_front = jobs_front.load(stdmo::relaxed);
while (true) {
    Node *old_front_next = old_front.node->next.load(stdmo::acquire);
    if (old_front_next == QUEUE_END) {
        return std::nullopt;
    }

    GenNodePtr new_front{old_front_next, old_front.gen + 1};
    if (jobs_front.compare_exchange_weak(old_front, new_front,
                                         stdmo::relaxed)) {
        break;
    }
```

Notice how the **generation count** is monotonically increasing



Problem #2: use-after-free (UAF)

Lock free woes #2 – use after free (UAF)

```
GenNodePtr old_front = jobs_front.load(stdmo::relaxed); // A
while (true) {
  Node *old_front_next = old_front.node->next.load(stdmo::acquire); // B
```

If old_front was freed, we will be accessing junk :(

Lock free woes #2 – use after free (UAF)

Solutions:

1. Never free anything (just download more RAM)

2. Procrastinate deleting the node – like when there are no one accessing the nodes. For example, use a freelist to store the nodes.

3. Use reference counting (or even split reference counting)

4. Use hazard pointers to track which threads have references to objects

How to be eco-friendly

Since we are storing freed nodes in a freelist, it would be wasteful for us to not use them

Idea: Have a lock-free (for obvious reason) stack inside the queue to keep track of available free nodes

How to be eco-friendly

```
static inline Node *const QUEUE_END = nullptr;
static inline Node *const STACK_END = QUEUE_END + 1;

Cristina Carbunaru, last month | 1 author (Cristina Carbunaru)
struct GenNodePtr {
  Node *node;
  std::uintptr_t gen;
};

alignas(hardware_destructive_interference_size)
    std::atomic<Node *> jobs_back;  // producer end

alignas(hardware_destructive_interference_size)
    std::atomic<GenNodePtr> jobs_front;  // consumer end

alignas(hardware_destructive_interference_size)
    std::atomic<GenNodePtr> stack_top;  // consumer end
```

How to be eco-friendly

Things don’t clean up on their own… (unless told?)

```
~JobQueue11() {

  // ... queue cleanup

  Node *cur_stack = stack_top.load(stdmo::relaxed).node;
  while (cur_stack != STACK_END) {
    Node *next = cur_stack->next;
    delete cur_stack;
    cur_stack = next;
  }
}
```

How to be eco-friendly

Instead of calling **`new Node{}`** in enqueue, we use

```
Node *allocate_node() {
    // Standard CAS loop with generation counter to avoid ABA.
    GenNodePtr cur_stack = stack_top.load(stdmo::relaxed);
    while (true) {
        if (cur_stack.node == STACK_END) {
            // If the recycling centre is empty, we'll allocate a new node
            return new Node{};
        }
        Node *cur_stack_next = cur_stack.node->next.load(stdmo::acquire);
        GenNodePtr new_stack{cur_stack_next, cur_stack.gen + 1};
        if (stack_top.compare_exchange_weak(cur_stack, new_stack,
                                            stdmo::relaxed)) {
            // Successfully spliced out a node from recycling centre
            return cur_stack.node;
        }
    }
}
```

How to be eco-friendly

Rather than just **delete**, we recycle

```
void deallocate_node(Node *node) {
  // Standard CAS loop with generation counter to avoid ABA.
  GenNodePtr cur_stack = stack_top.load(stdmo::relaxed);
  while (true) {
    node->next.store(cur_stack.node, stdmo::release);
    GenNodePtr new_stack{node, cur_stack.gen + 1};
    if (stack_top.compare_exchange_weak(cur_stack, new_stack,
                                        stdmo::relaxed)) {
      break;
    }
  }
}
```

Problem #3: Data race in recycling stack

The endless onslaught of races #3

Race between **read** in **consumer** and **write** in **producer**
we need 2 **producers** as the first one will use the node as a dummy node

T1: consumer

take node X
`m_queue_front.cmpxchg( … )`

↓ SB + HB

read Job from node X
`Job j = old_front.node->job`
note: the race

↓ SB + HB

prepare node X for recycling
`node->next.store(cur_stack_top.node)`

↓ SB + HB

add node X to recycling stack
`m_recycling_stack_top.cmpxchg( … )`

T2: producer

prepare to take node X
`old_stack_top.node->next.load( … )`

↓ SB + HB

take node X from recycling stack
`m_recycling_stack_top.cmpxchg( … )`

↓ SB + HB

add node X as new dummy node in queue
`work_node = m_queue_back.exchange( … )`
note: acq-rel

sync-with
happens-before

T3: producer

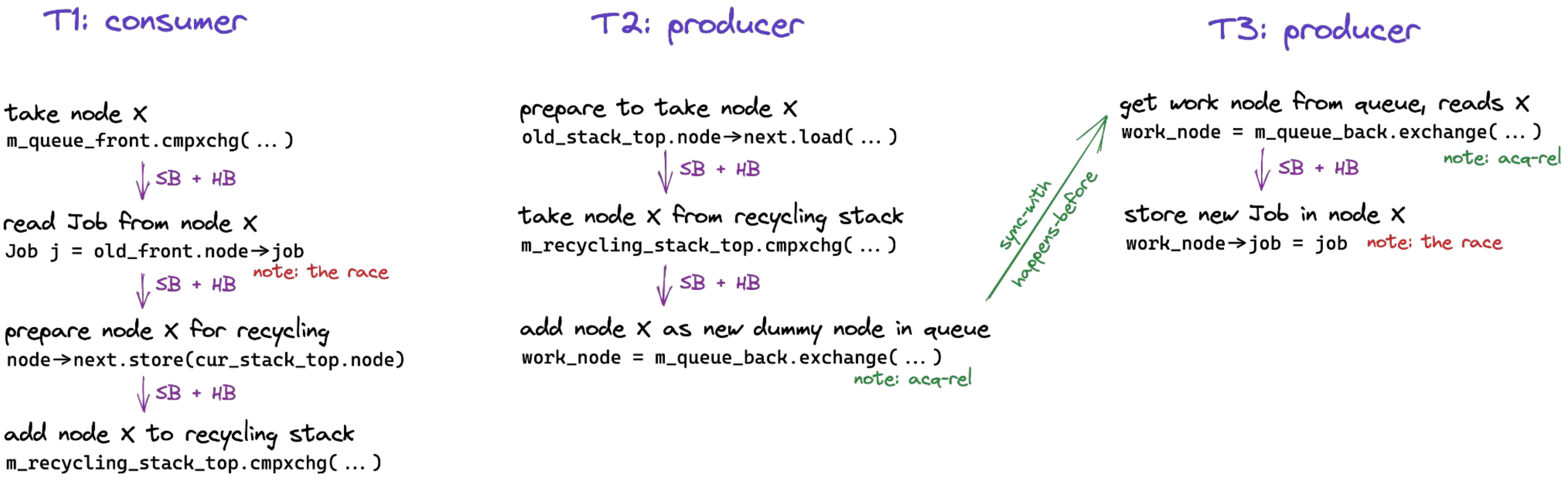
get work node from queue, reads X
`work_node = m_queue_back.exchange( … )`
note: acq-rel

↓ SB + HB

store new Job in node X
`work_node->job = job`
note: the race

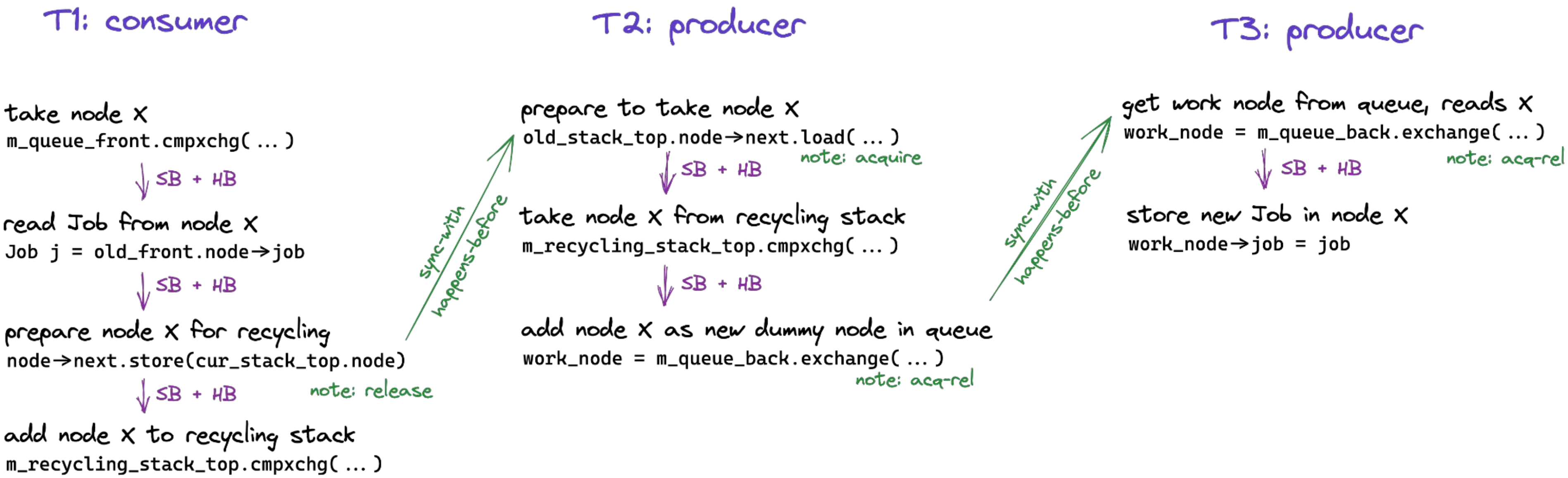
The endless onslaught of races #3

Looks like time travelling but since it is not synchronised, there are no guarantees



The endless onslaught of races #3 – fix

By using transitive acquire-release between threads, we achieve a synchronises-with relationship and fixed the race!



50 races in 50 days #4

Race between **read** in **consumer** and **constructor (write)** in **producer**

We need 2 **consumers** cause the first consumer will consume the old_dummy

T1: producer

create new dummy

```
new_dummy = get_or_allocate_node()
```

SB + HB

create new Node

```
return new Node();
```

note: the race

SB + HB

add new dummy node to queue

```
m_queue_back.exchange(new_dummy)
```

SB + HB

release work node

```
work_node→next.store(new_dummy)
```

note: release

sync-with

happens-before

T2: consumer

load current front

```
old_front = m_queue_front.load( … )
```

SB + HB

load old-front's next

```
old_front_next = old_front.next→load( … )
```

note: acquire

SB + HB

compare-exchange queue front

```
m_queue_front.compare_exchange( … )
```

note: relaxed

T3: consumer

load current front

```
old_front = m_queue_front.load( … )
```

note: relaxed

SB + HB

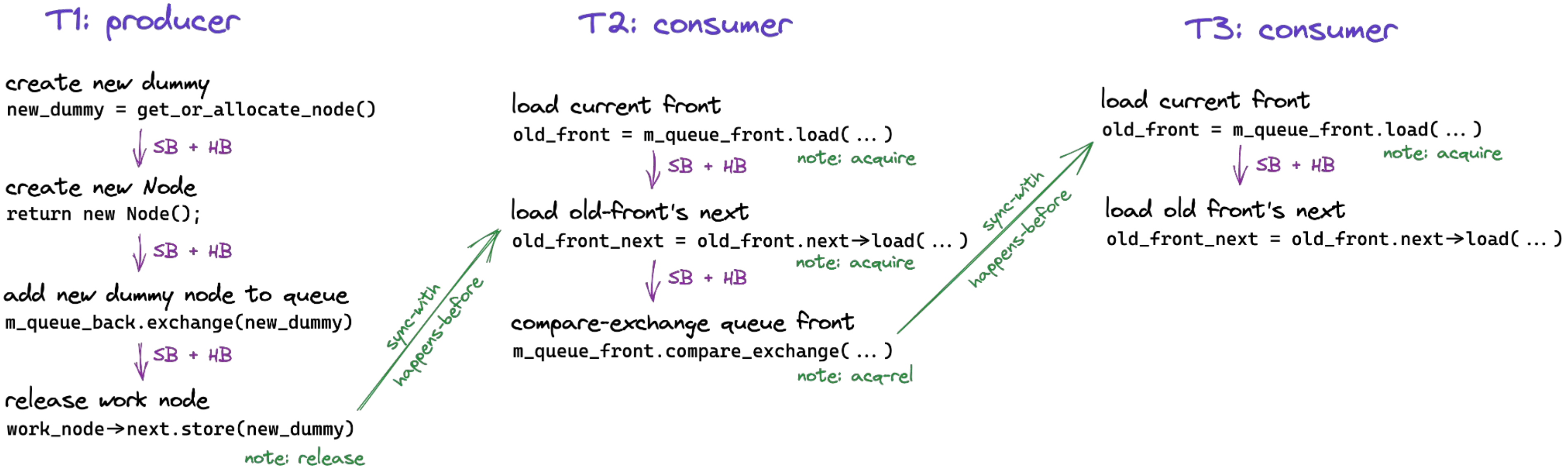
load old front's next

```
old_front_next = old_front.next→load( … )
```

note: the race

50 races in 50 days #4 – fix

Similarly, we ensure that the racing threads have their vehicles impounded



Almost there, how fast did we go?

Microbenchmarks: track the performance of a single well-defined task, and is most useful for CPU work that is run many times (also known as hot code paths)

Top contenders:

1. a basic coarse-grained-lock queue
2. the fine-grained-lock queue from Lecture 5
3. the lock-free queue we just wrote

Almost there, how fast did we go?

For this benchmark, we will be using **real time** as our metric.

Setups:

- Single Producer, Single Consumer (SPSC)
- Single Producer, Multiple Consumers (SPMC)
- Multiple Producers, Single Consumer (MPSC)
- Multiple Producers, Multiple Consumers (MPMC)

code

See you next week!