CS3211 Tutorial 4

Lock free programming in C++
Simon

Credit to Kingsley

Overview (2 hours + 10-15’ break in the middle)

1. Lock Free Queue
2. Producers `push()`
3. Consumers `try_pop()`
4. Problem #1: The ABA problem
5. Problem #2: use-after-free (UAF)
6. Problem #3: Data race in recycling stack
7. Problem #4: Internal data race
8. Problem #5: Lack of “Linearizability”

WARNING:

This Tutorial Has Been Identified to Cause Academic Trauma and Fried Brain


Are you ready Kids?

What is “Lock-Free” Data Structure?

Lock Free Data Structure

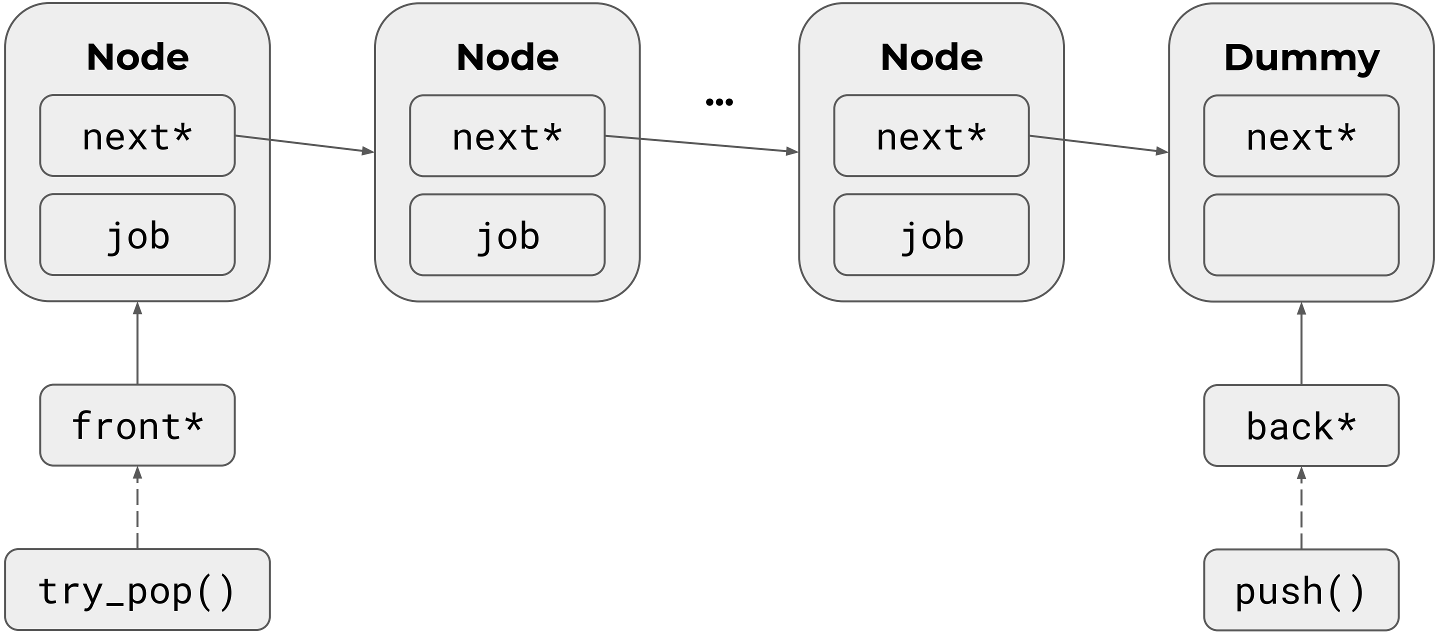
1. C++ Concurrency in Action
    a. Able to Access the DS concurrently (but doesn’t need to be the same DS)
    b. If >= 1 thread(s) is suspended by the scheduler, the other threads must be able to complete their operations without waiting for the suspended thread
    c. TLDR; No Deadlock in the System

Lock Free Data Structure

1. C++ Concurrency in Action
    a. Able to Access the DS concurrently (but doesn’t need to be the same DS)
    b. If >= 1 thread(s) is suspended by the scheduler, the other threads must be able to complete their operations without waiting for the suspended thread
    c. TLDR; No Deadlock in the System
2. Follow-up Question: Why using Mutex is not lock free then?

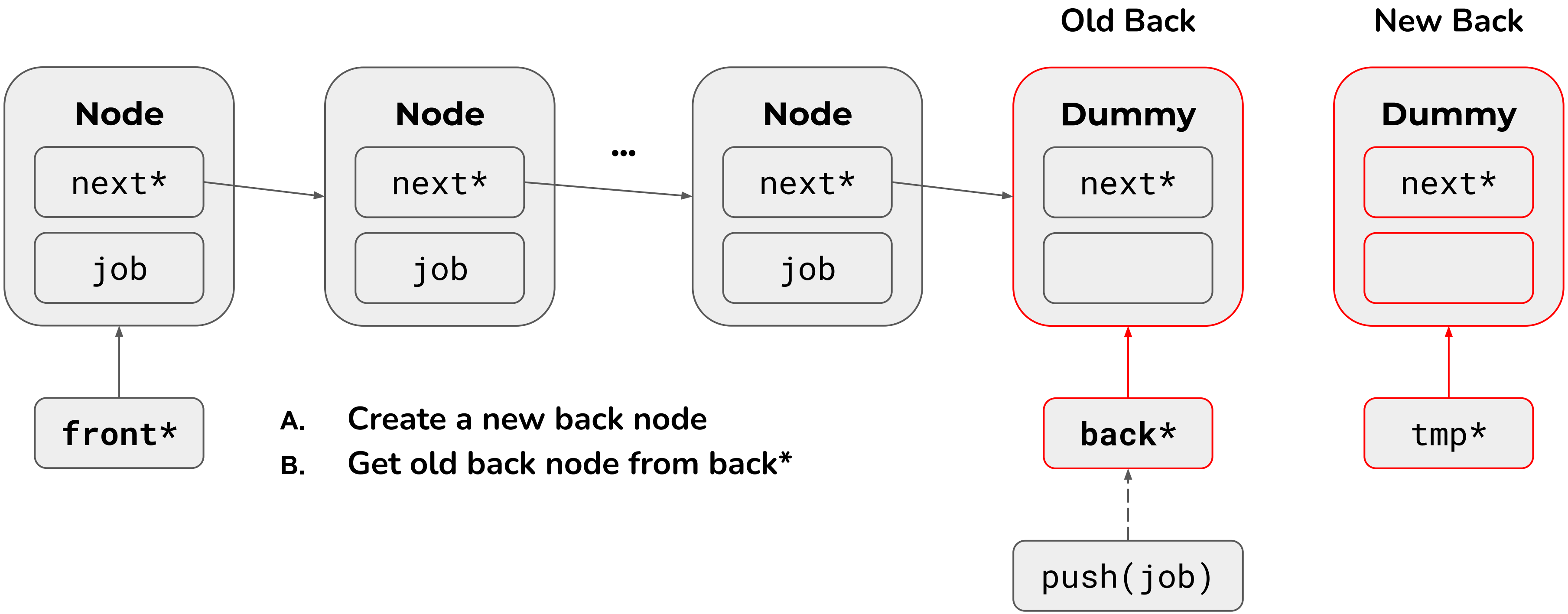
Lock Free Queue

Lock Free Queue - Producer + Consumer



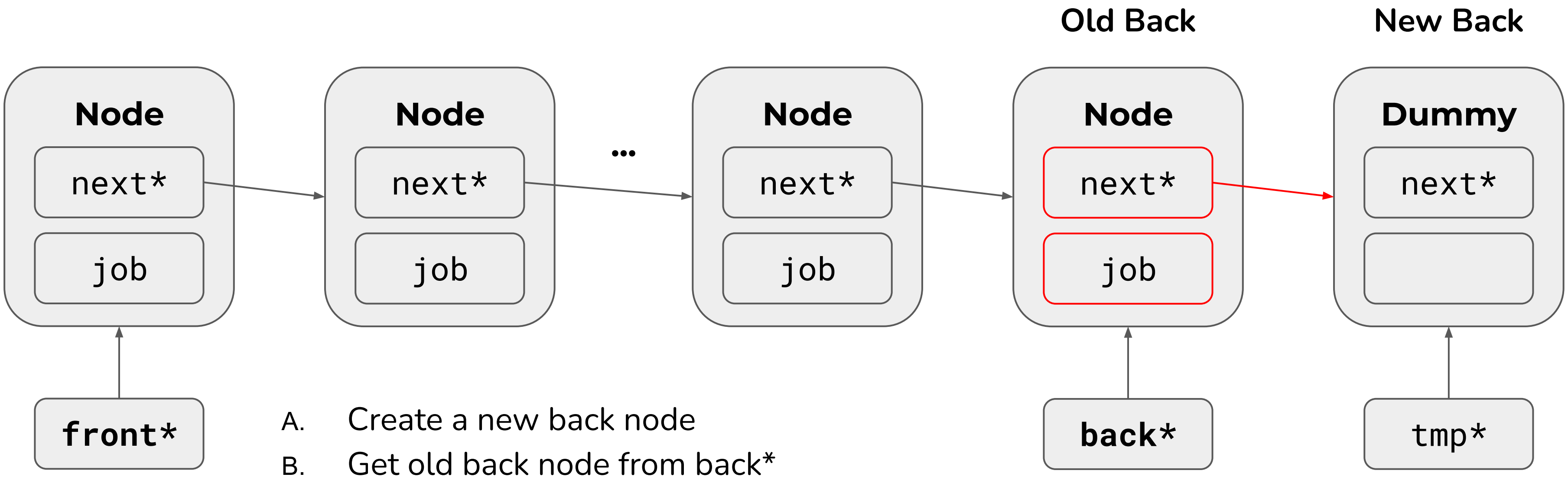
Producers push()

Producers push() - Naive Attempt



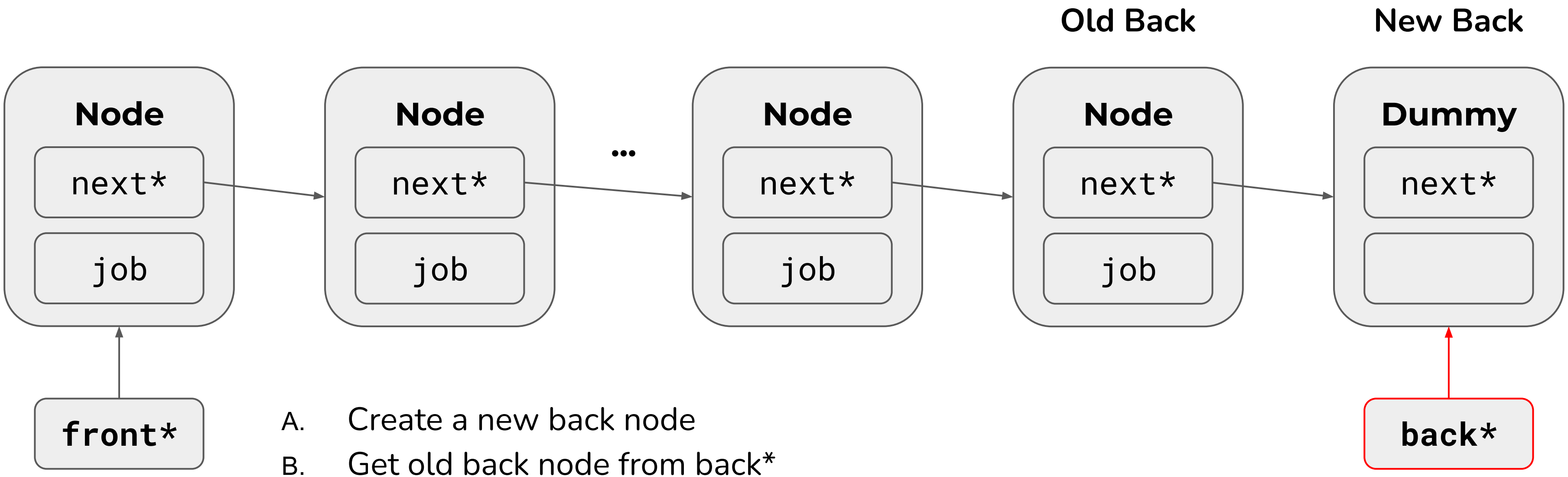
A. Create a new back node
B. Get old back node from back*

Producers push() - Naive Attempt



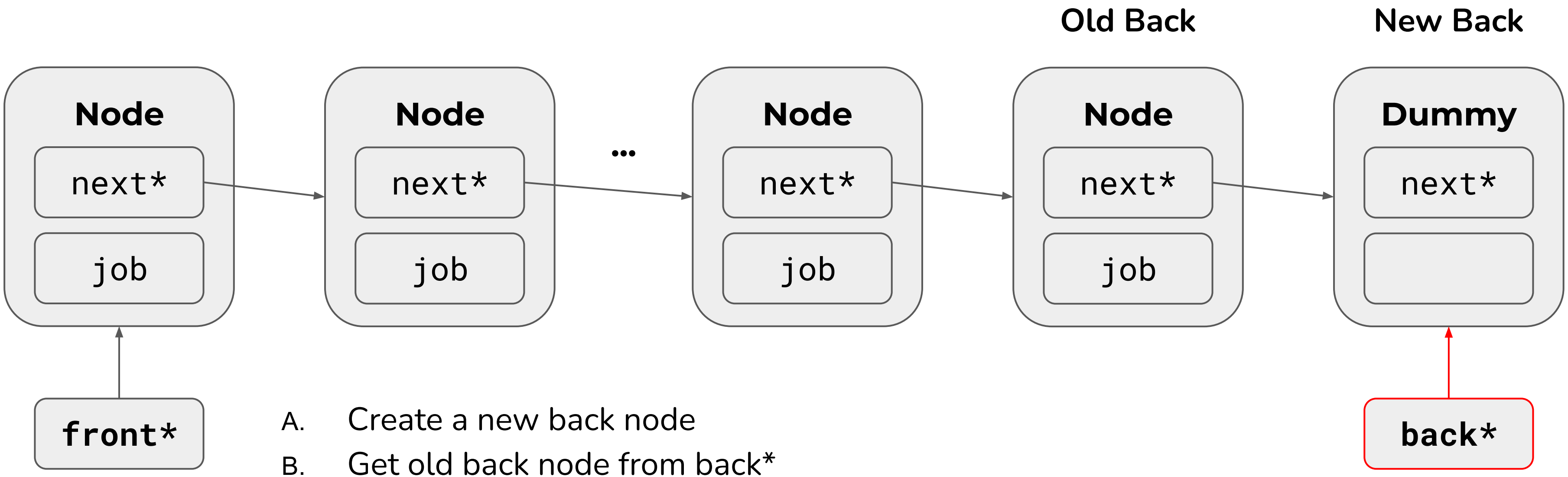
A. Create a new back node
B. Get old back node from back*
C. Set the new job in the old back node.
D. Point old back node at new back node.

Producers push() - Naive Attempt



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.**

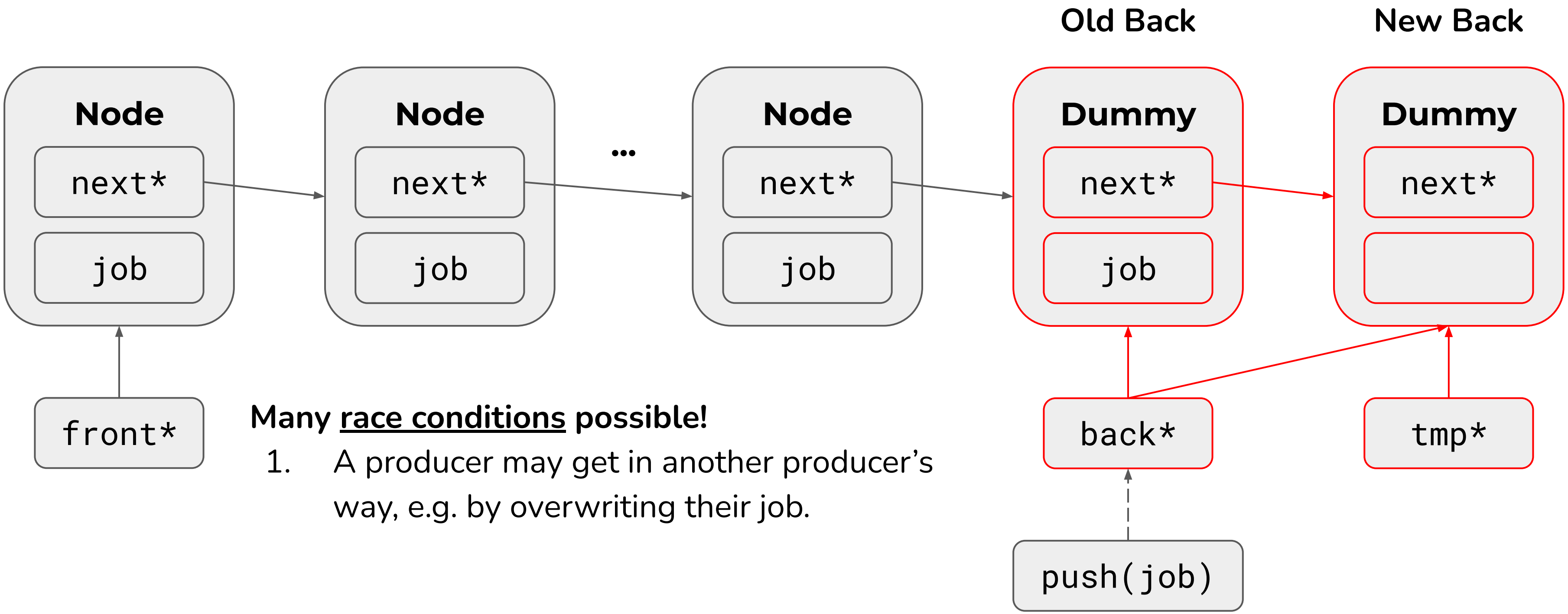
Producers push() - Naive Attempt



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.

What's the problem? (Naive = cannot be correct in CS3211)

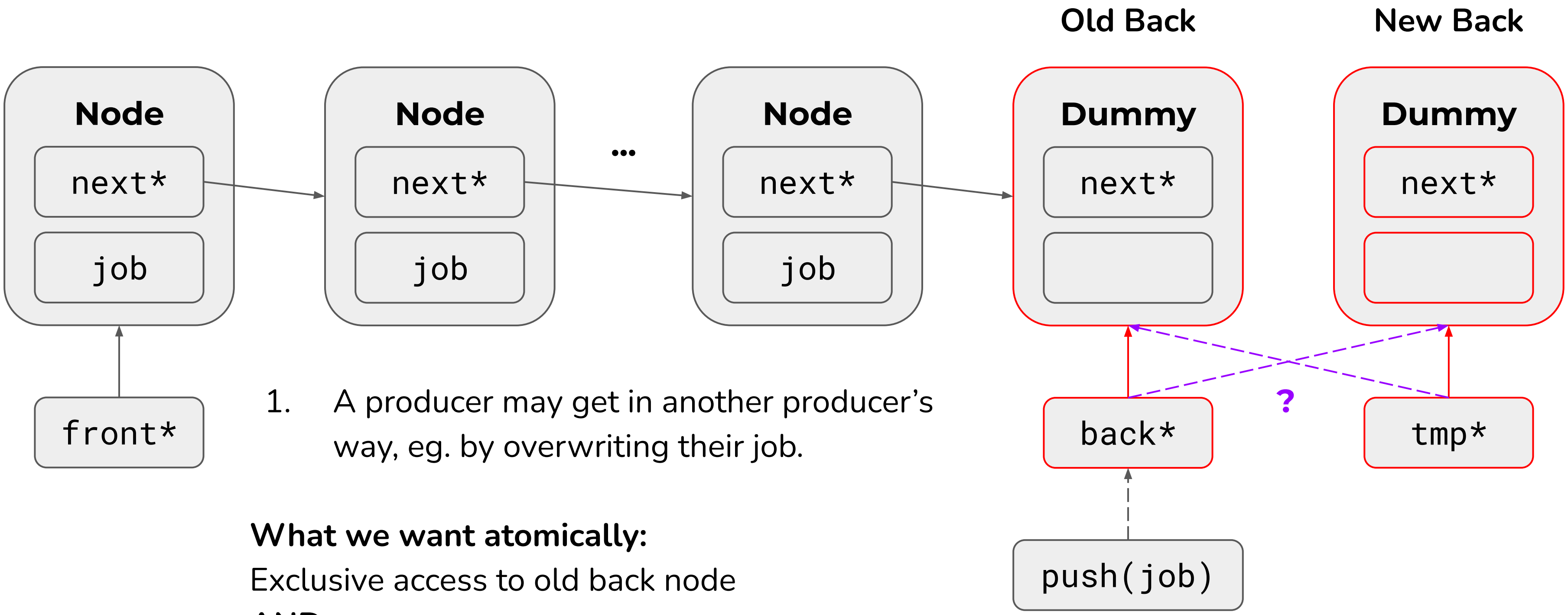
Producers push() - Naive Attempt



Many <u>race conditions</u> possible!

1. A producer may get in another producer's way, e.g. by overwriting their job.

Producers push()



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

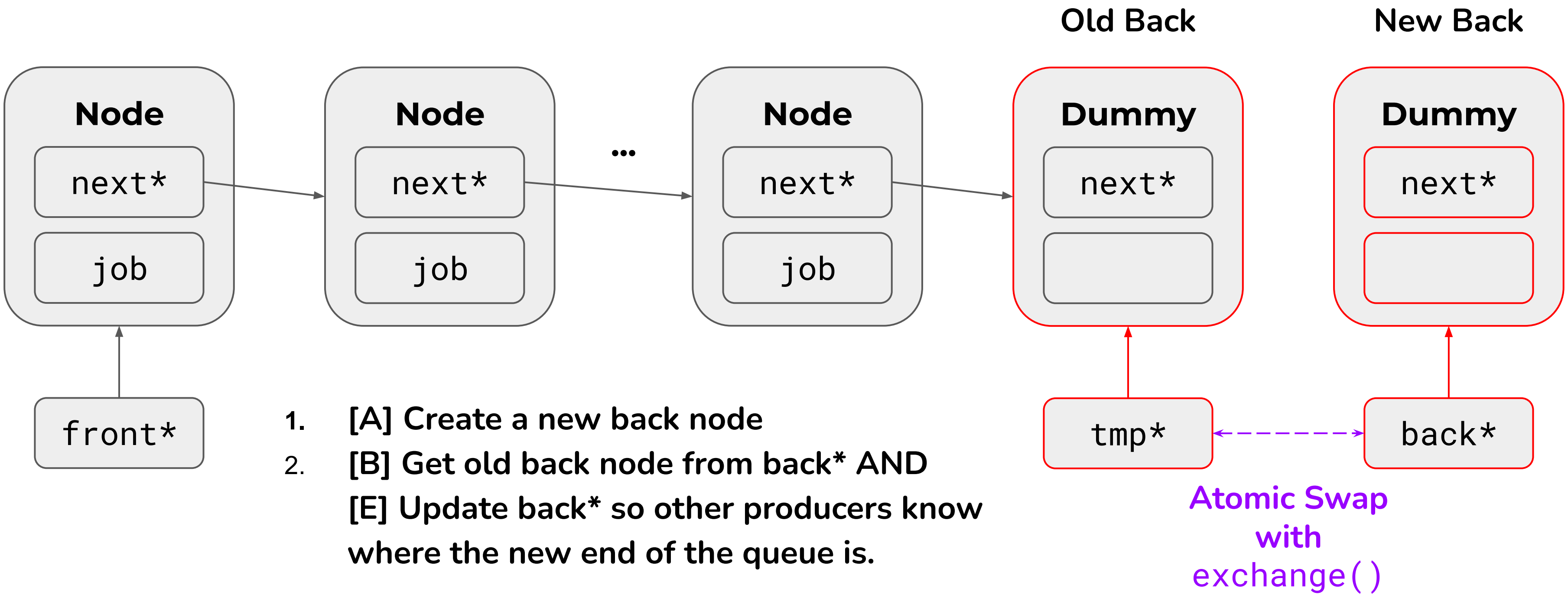
What we want atomically:

Exclusive access to old back node

AND

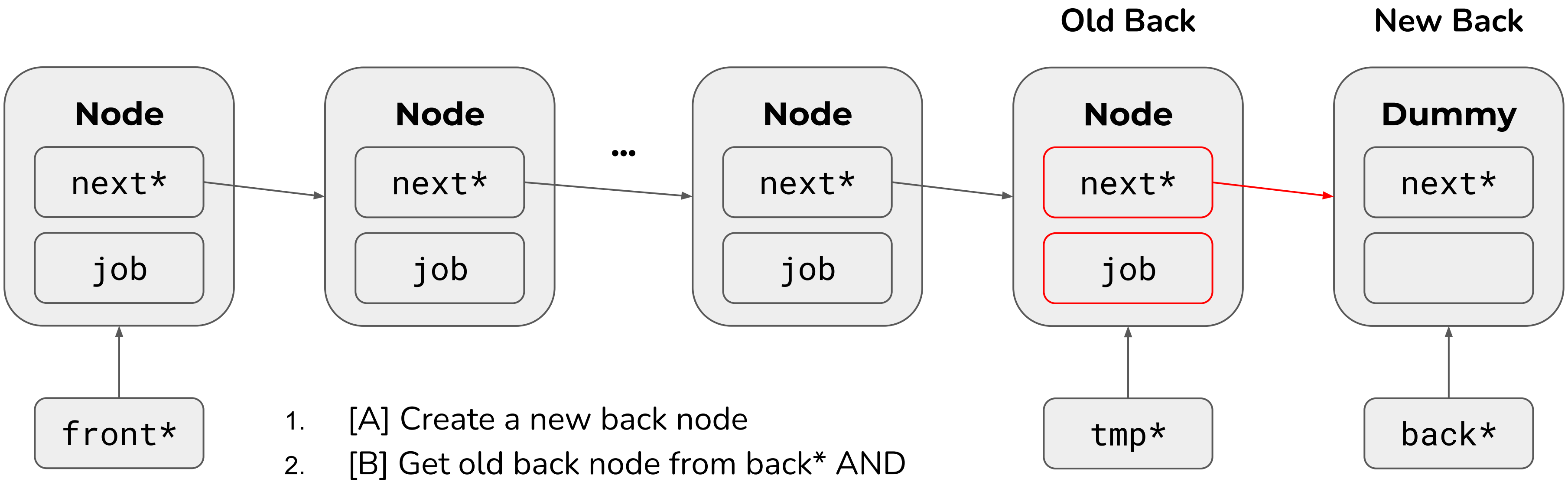
Set back* to new back node

Producers push()



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.

Producers push()

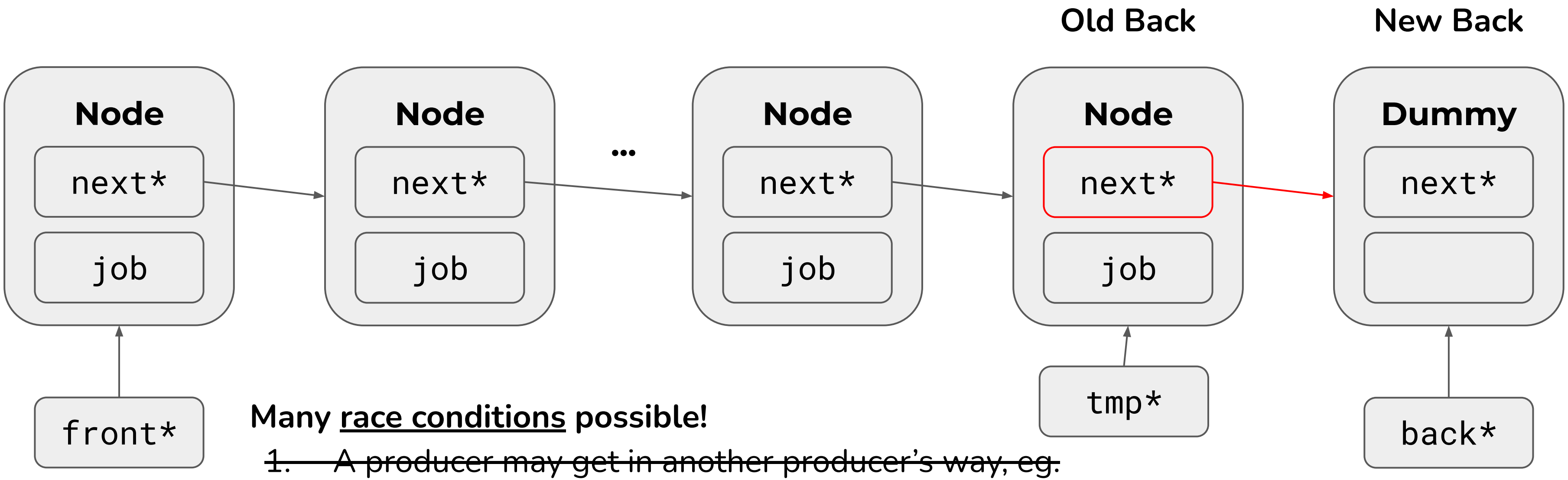


1. [A] Create a new back node
2. [B] Get old back node from back* AND
   [E] Update back* so other producers know where the new end of the queue is.
3. **[C] Set the new job in the old back node.**
4. **[D] Point old back node at new back node**

Producer has exclusive access (wrt other producers)

Other producers can access freely

Producers push()



Many <u>race conditions</u> possible!

1. ~~A producer may get in another producer's way, eg. by overwriting their job.~~
2. A producer may not be correctly synchronised with consumers, causing them to read an **invalid state**
   → What happen if we use **release-release?**

What’s wrong with this?

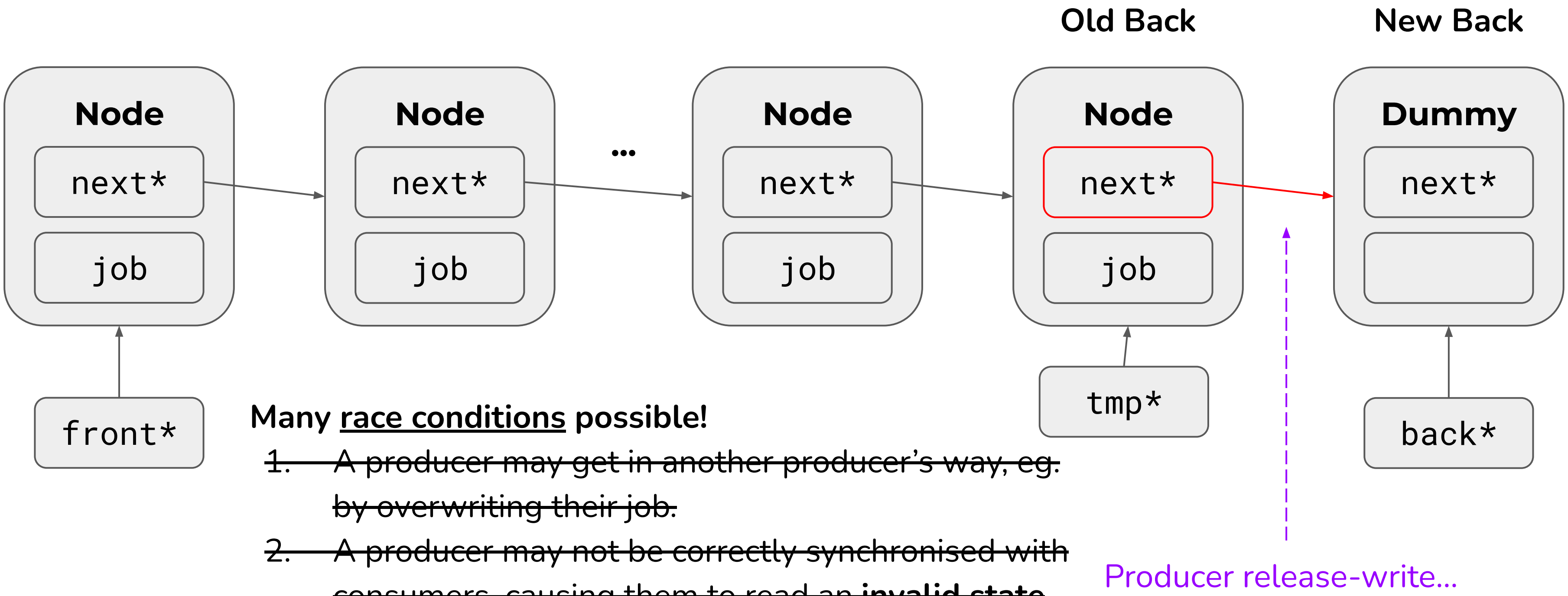
```
void push(Job job)
{
      Node* new_dummy = new Node();
      Node* work_node = m_queue_back.exchange(new_dummy, stdmo::acq_rel);
      work_node->job = job;
      work_node->next.store(new_dummy, stdmo::relaxed);
}

std::optional<Job> try_pop()
{
      ...
      Node* new_front = old_front->next.load(stdmo::relaxed);
      if(new_front == QUEUE_END)
      {

            return std::nullopt;
      }

   ...
}
```

What’s wrong with this?

```
void push(Job job)
{
	Node* new_dummy = new Node();
	Node* work_node = m_queue_back.exchange(new_dummy, stdmo::acq_rel);
	work_node->job = job;
	work_node->next.store(new_dummy, stdmo::relaxed);
}

std::optional<Job> try_pop()
{
	...
	Node* new_front = old_front->next.load(stdmo::relaxed);
	if(new_front == QUEUE_END)
	{

		return std::nullopt;
	}

   ...
}
```

It’s possible for new_front to read the memory location of `next` but not the job (i.e. `work_node->job = job;`)

Producers push()



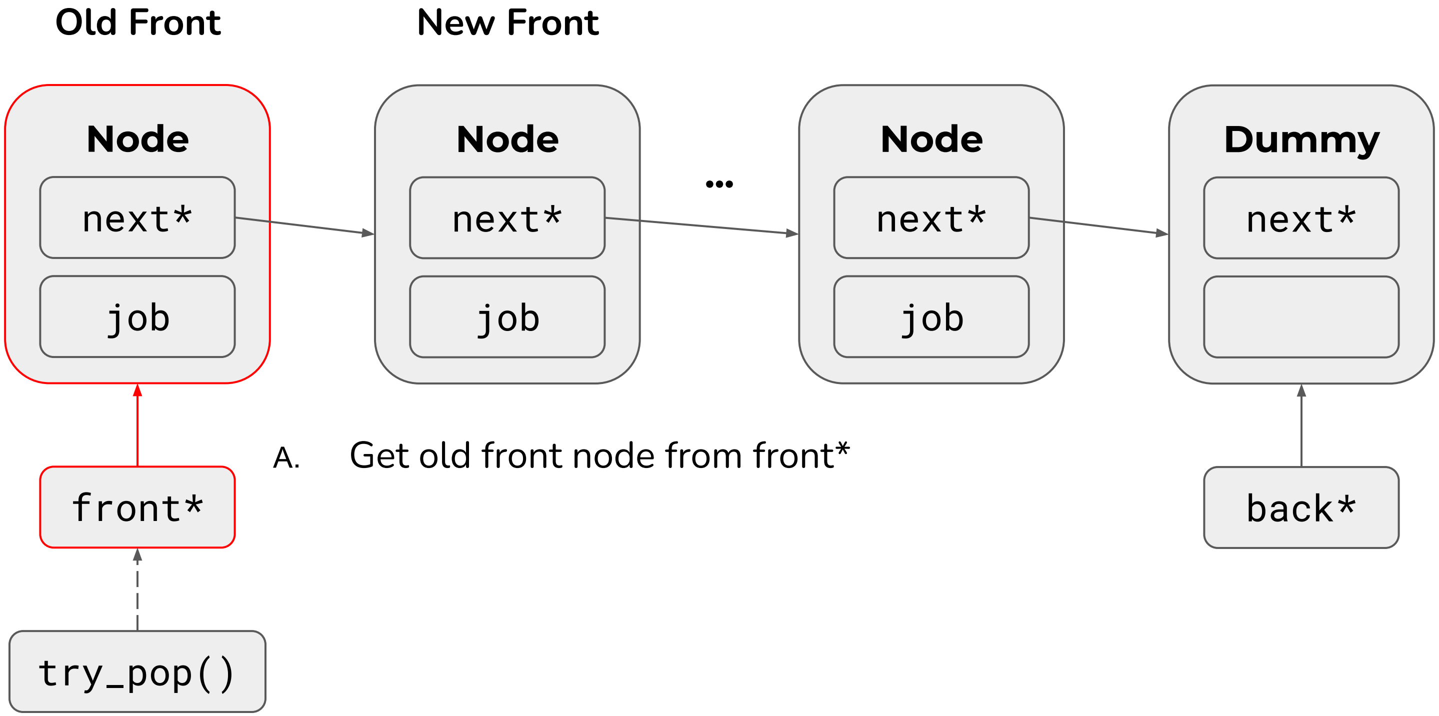
Many <u>race conditions</u> possible!

1. ~~A producer may get in another producer's way, eg. by overwriting their job.~~
2. ~~A producer may not be correctly synchronised with consumers, causing them to read an **invalid state**~~

→ We use release-acquire

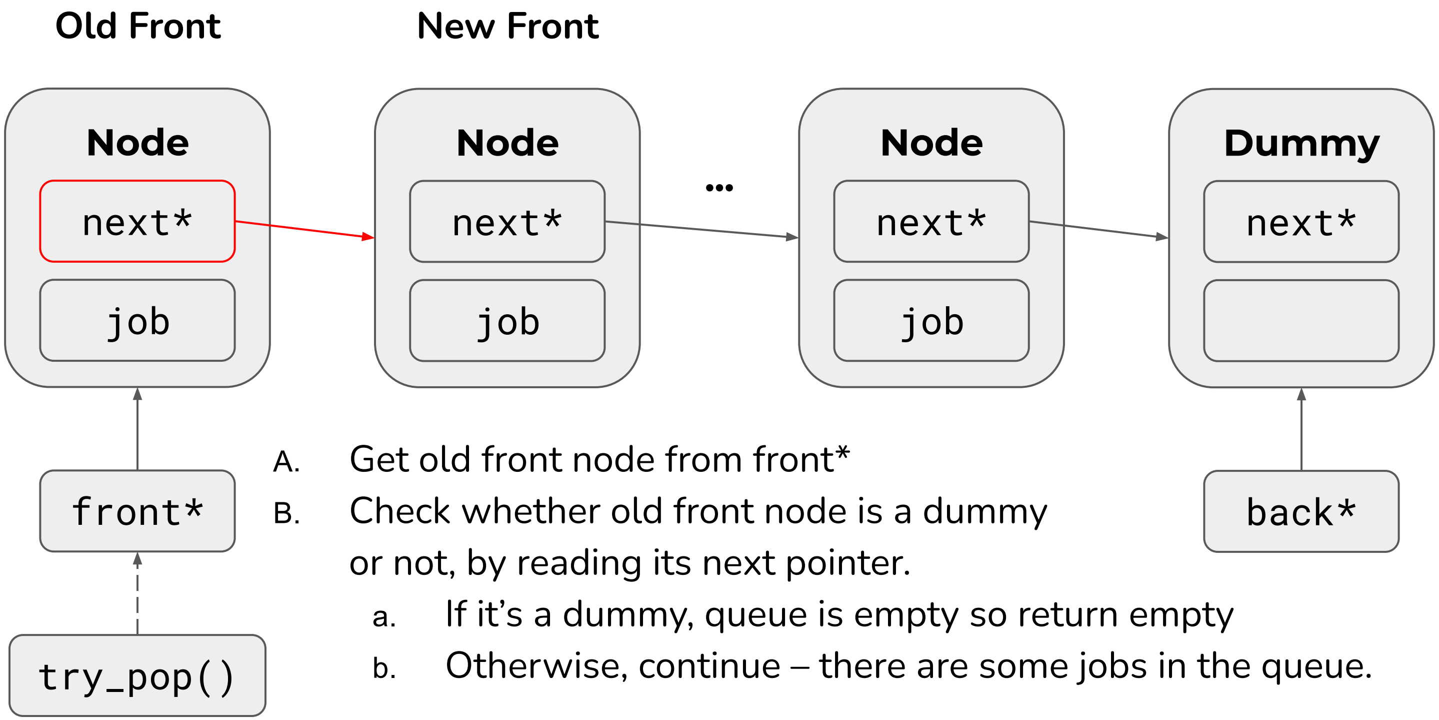
Consumers try_pop()

Consumers try_pop() - Naive Attempt



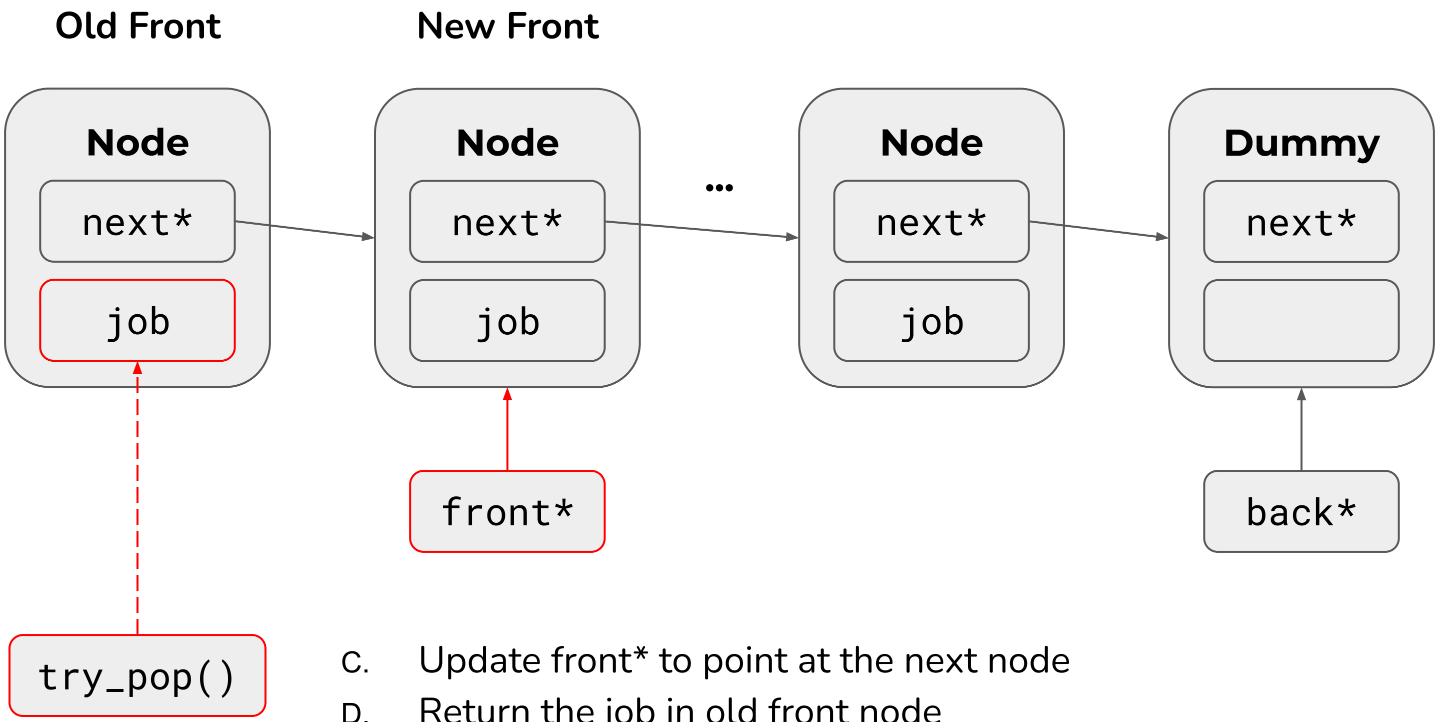
A. Get old front node from front*

Consumers try_pop() - Naive Attempt



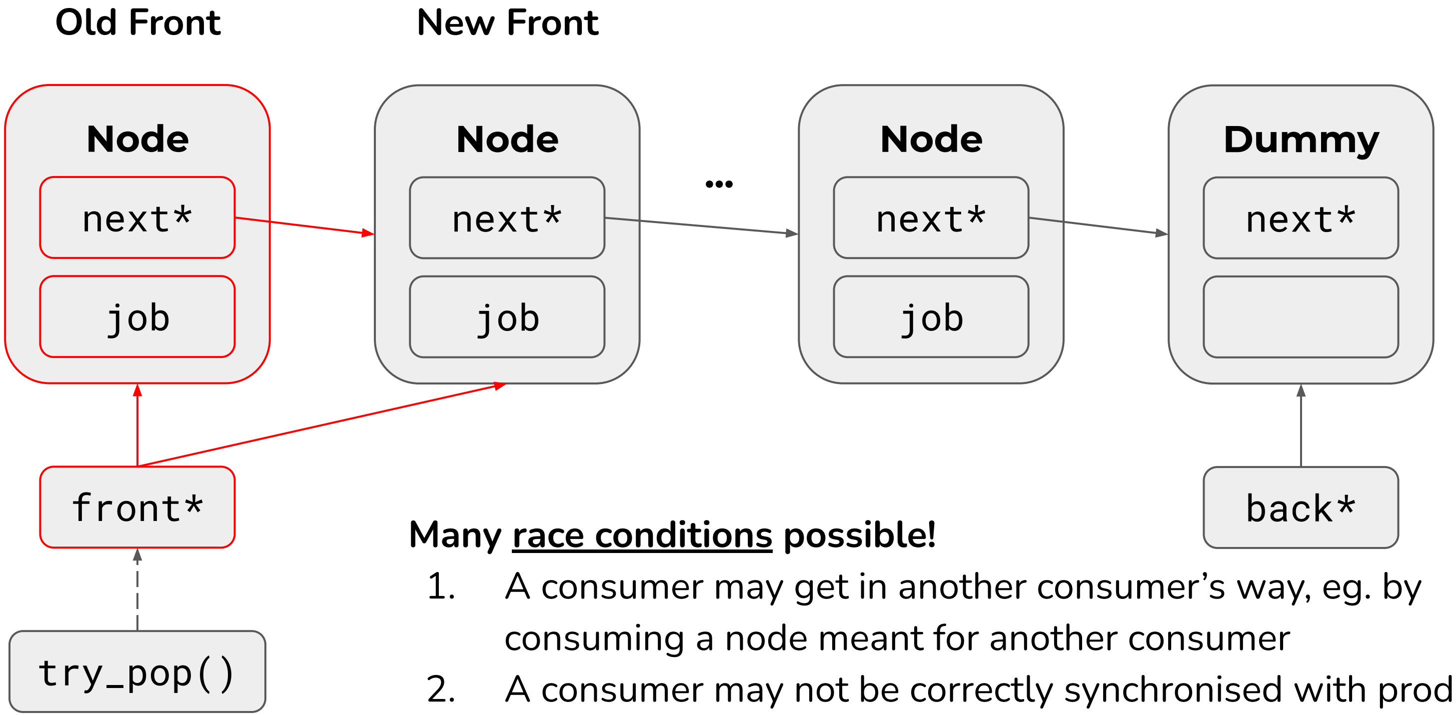
A. Get old front node from front*
B. Check whether old front node is a dummy or not, by reading its next pointer.
   a. If it's a dummy, queue is empty so return empty
   b. Otherwise, continue – there are some jobs in the queue.

Consumers try_pop() - Naive Attempt



C. Update front* to point at the next node
D. Return the job in old front node

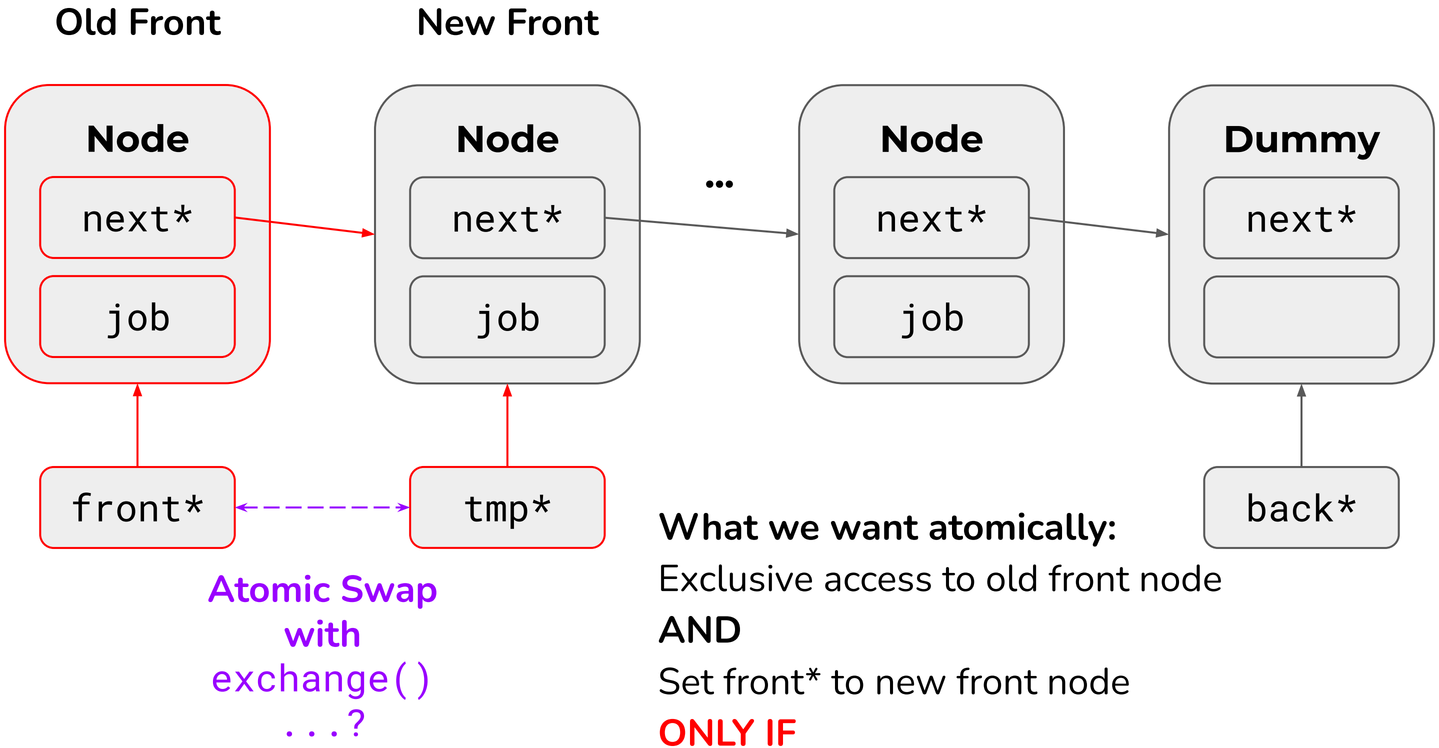
Consumers try_pop() - Naive Attempt



Many <u>race conditions</u> possible!

1. A consumer may get in another consumer's way, eg. by consuming a node meant for another consumer
2. A consumer may not be correctly synchronised with producers, causing them to read an invalid state (same as before)

Consumers try_pop()



What we want atomically:

Exclusive access to old front node

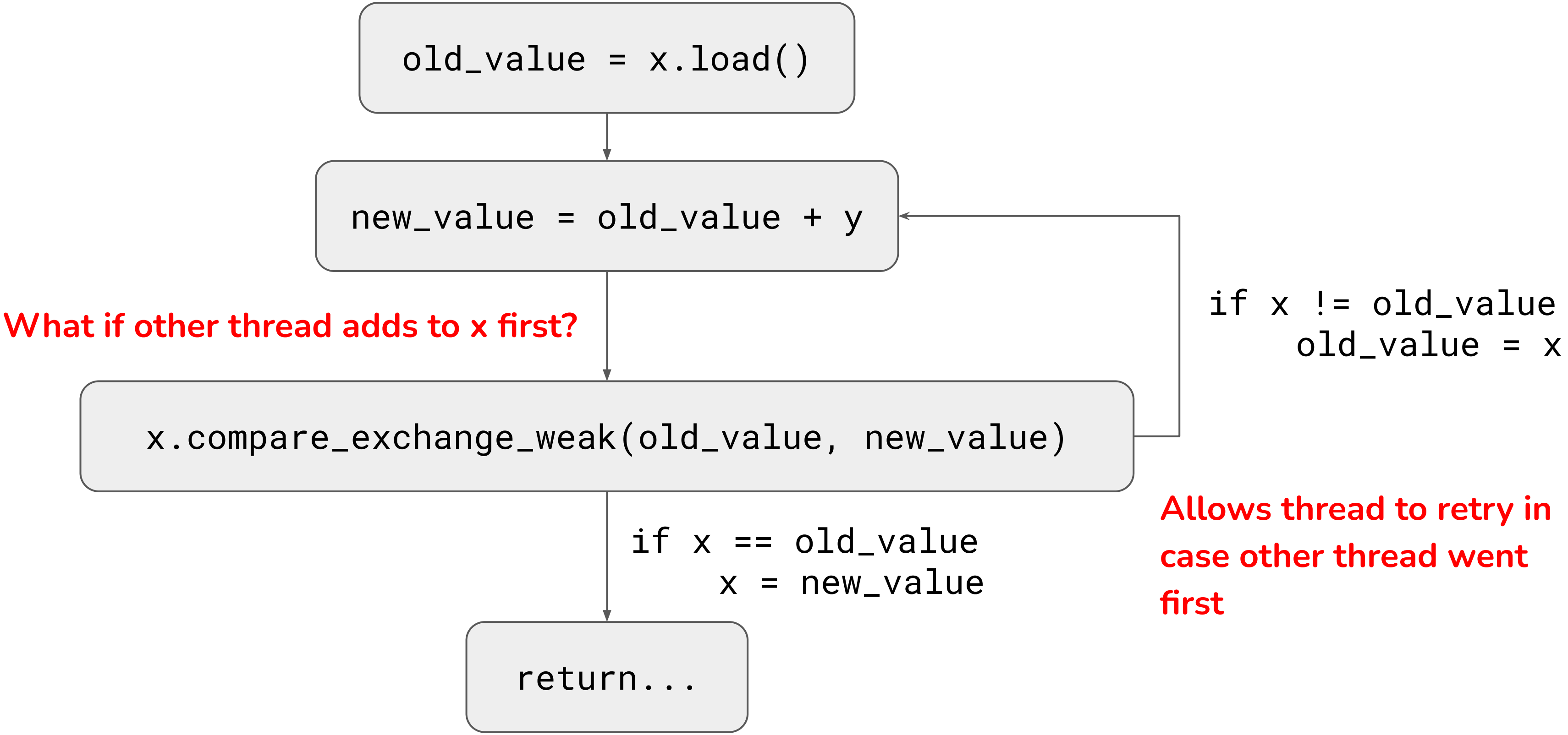
AND

Set front* to new front node

ONLY IF

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

Compare-And-Swap (CAS) Pattern



Follow up: Compare And Swap

`bool compare_exchange_weak(T& expected, T desired, std::memory_order order = std::memory_order_seq_cst) volatile noexcept;`

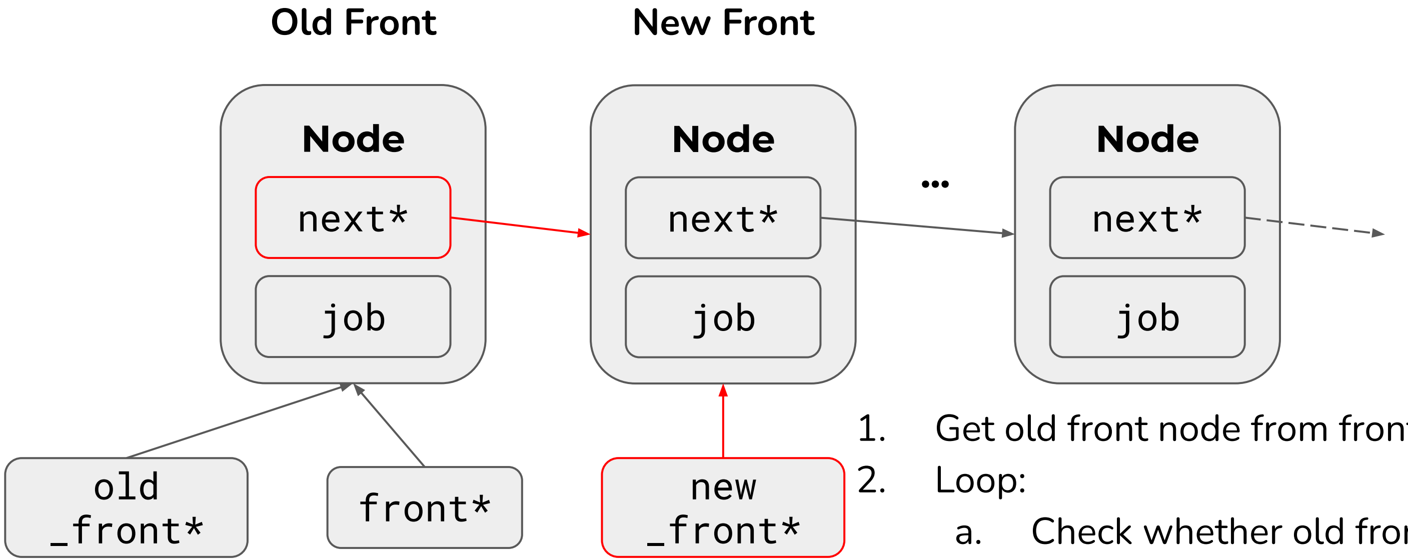
1. Given what you have learned in the previous slide and also the function signature of `compare_exchange_weak`, What is the semantic of `compare_exchange_weak`?
   a. What does `x.compare_exchange_weak(old_value, new_value)`do?

Follow up: Compare And Swap

`bool compare_exchange_weak(T& expected, T desired, std::memory_order order = std::memory_order_seq_cst) volatile noexcept;`

1. Given what you have learned in the previous slide and also the function signature of `compare_exchange_weak`, What is the semantic of `compare_exchange_weak`?
   a. What does `x.compare_exchange_weak(old_value, new_value)` do?
2. If x == old_value, then x = new_value!
   a. why do we need to check x == old_value?
3. if x != old_value, then **old_value = x!**
   a. Note that old_value is passed by reference, and the old_value is updated to the "newer" value

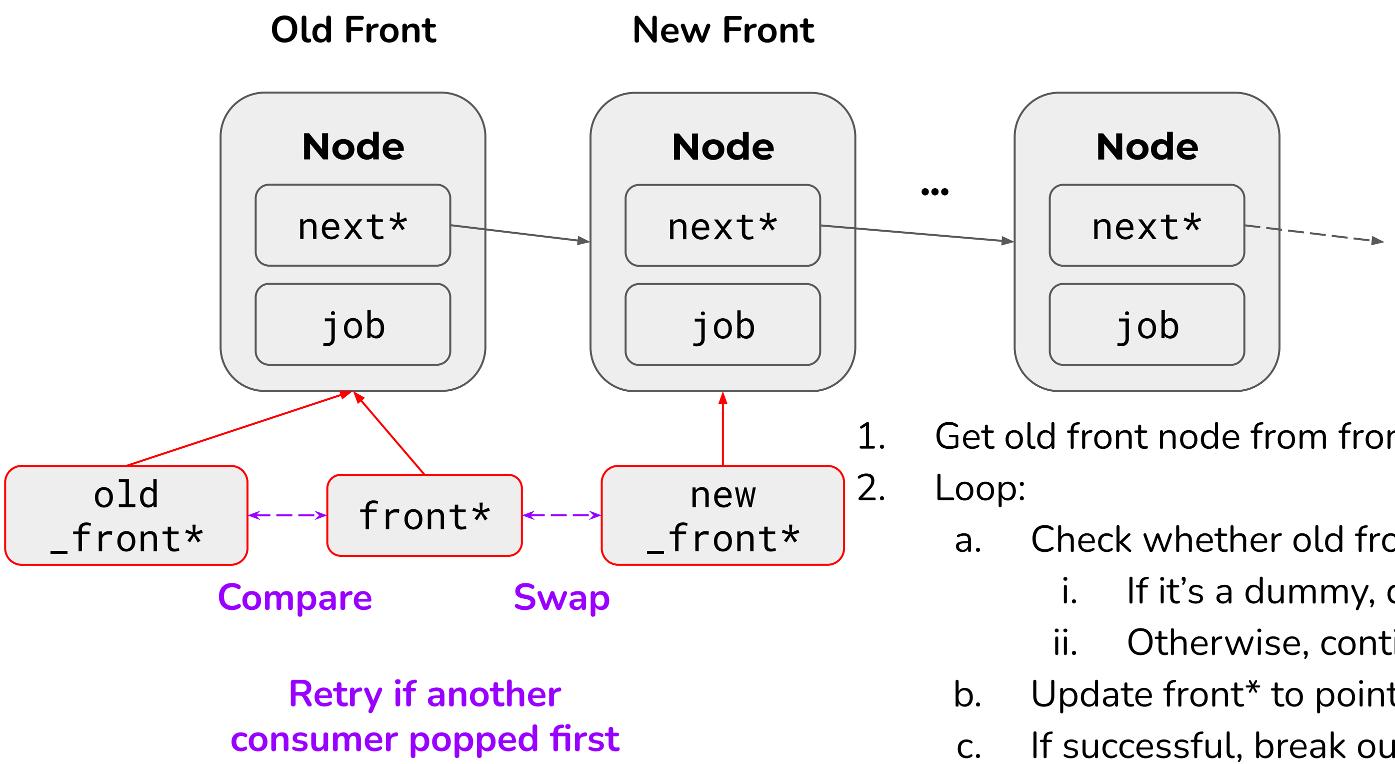
Consumers try_pop()



Read next* into new_front* and check if nullptr

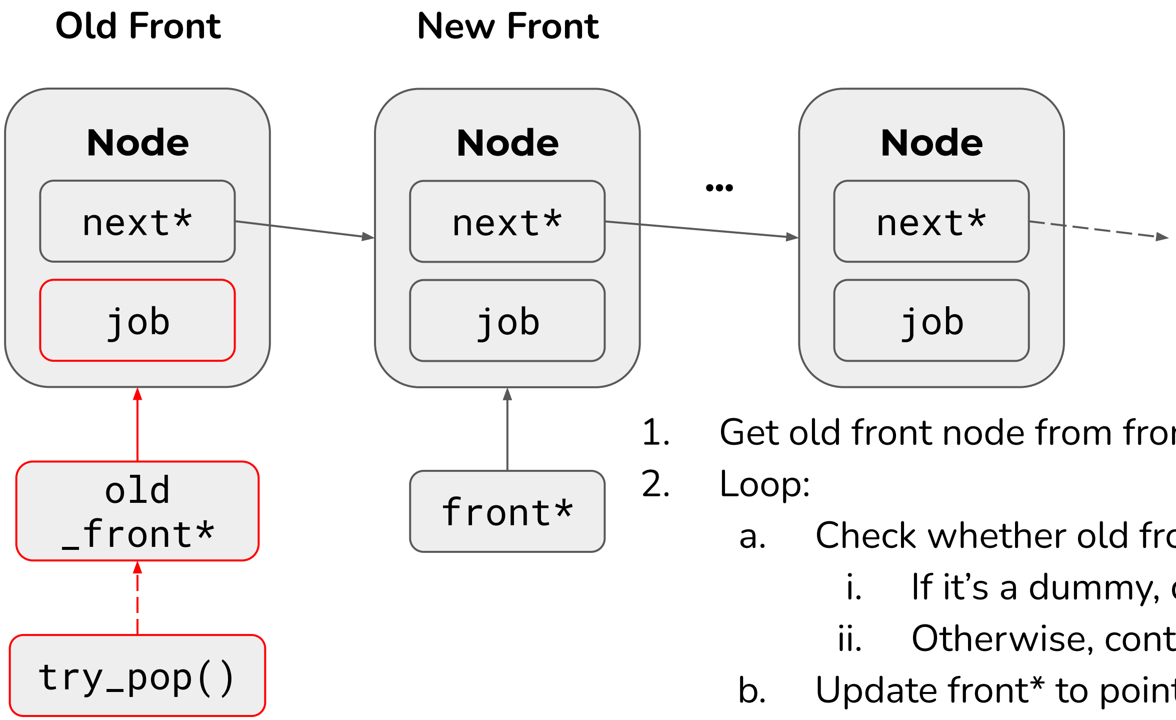
1. Get old front node from front*
2. Loop:
   a. Check whether old front node is a dummy
      i. If it's a dummy, queue is empty, return empty
      ii. Otherwise, continue

Consumers try_pop()



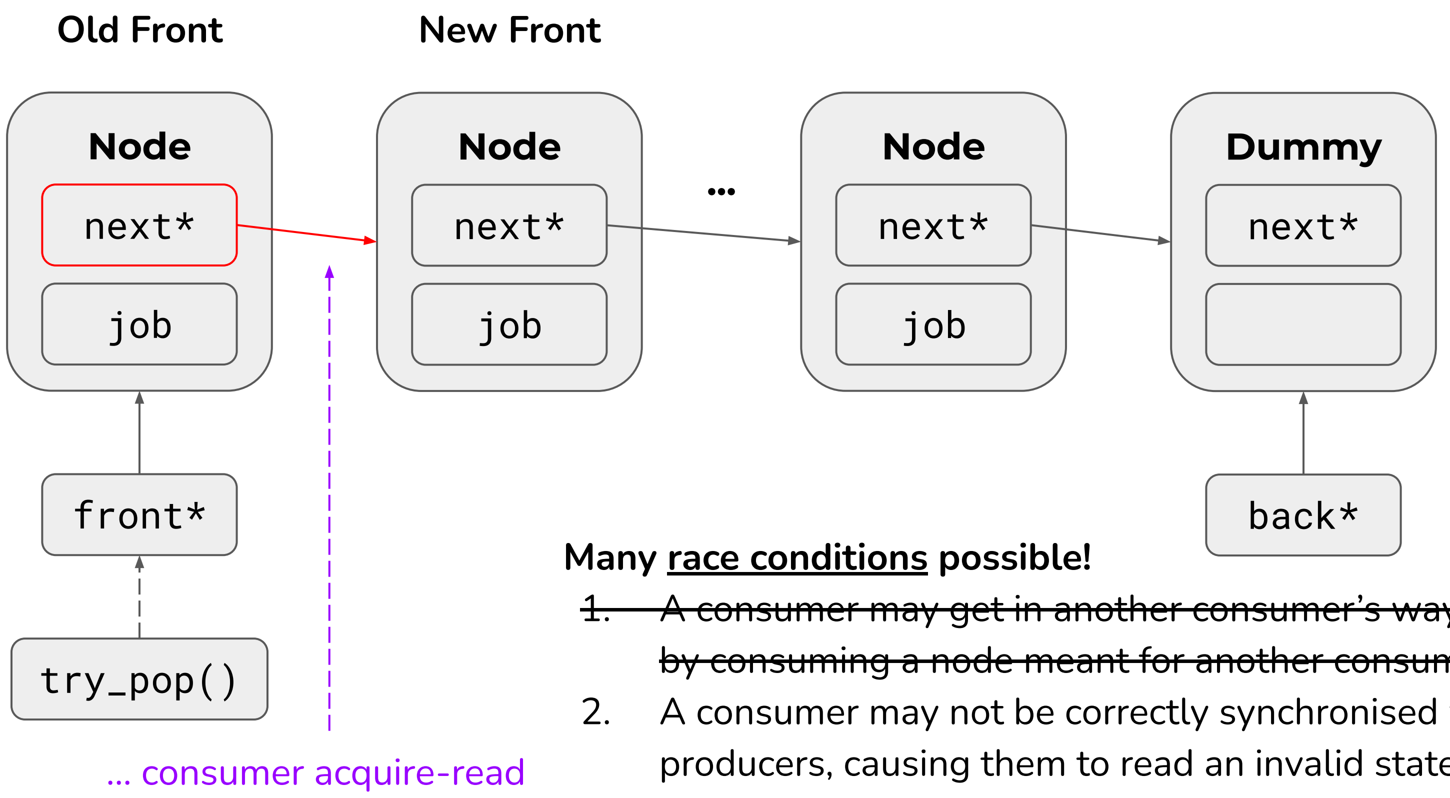
1. Get old front node from front*
2. Loop:
   a. Check whether old front node is a dummy
      i. If it's a dummy, queue is empty, return empty
      ii. Otherwise, continue
   b. Update front* to point at the next node with CAS
   c. If successful, break out of the loop

Consumers try_pop()



1. Get old front node from front*
2. Loop:
   a. Check whether old front node is a dummy
      i. If it's a dummy, queue is empty, return empty
      ii. Otherwise, continue
   b. Update front* to point at the next node with CAS
   c. If successful, break out of the loop
3. Return the job in old front node

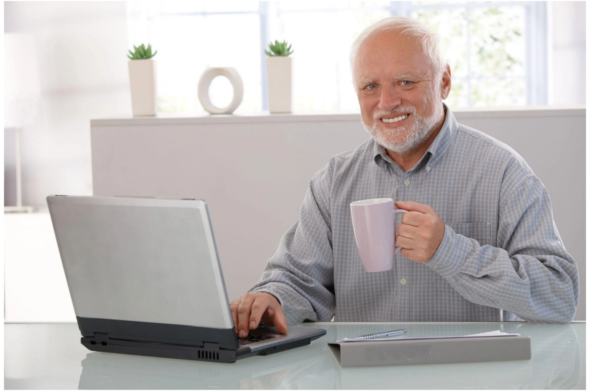
Consumers try_pop()



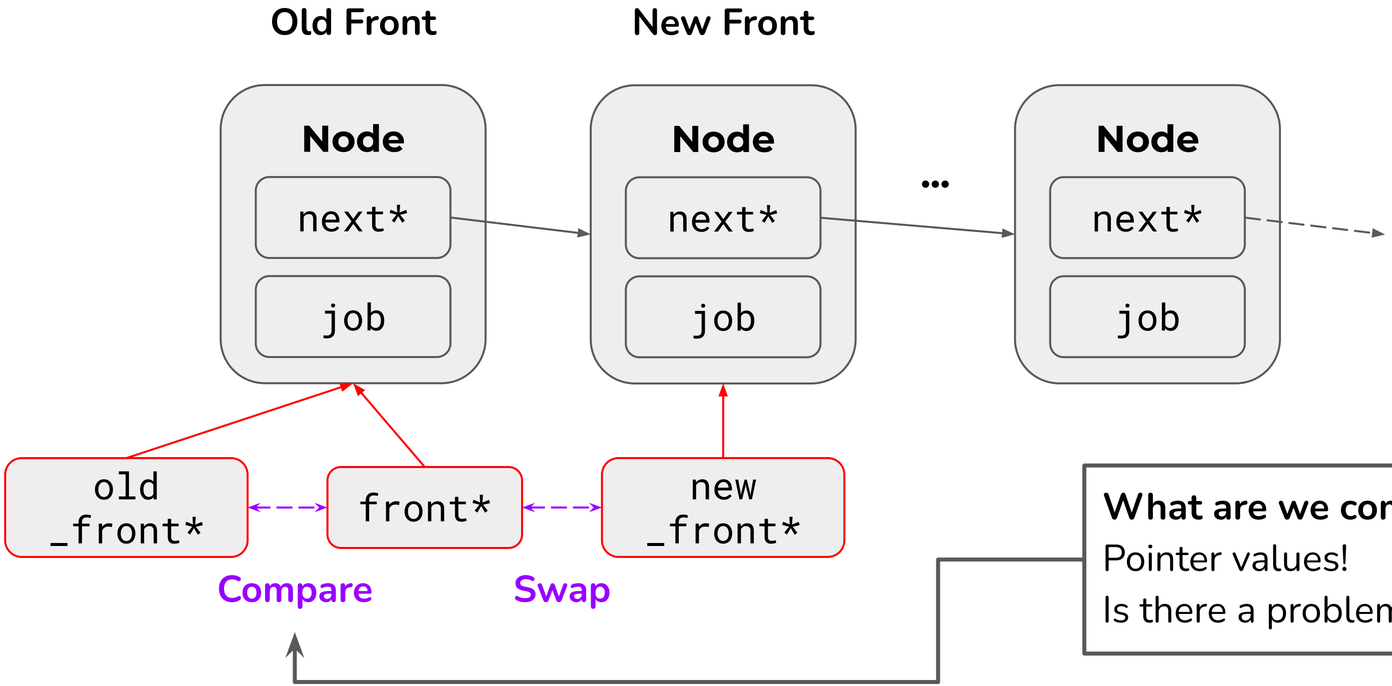
Many race conditions possible!

1. ~~A consumer may get in another consumer's way, eg. by consuming a node meant for another consumer~~
2. A consumer may not be correctly synchronised with producers, causing them to read an invalid state.

Problem #1: The ABA problem



Problem #1: the ABA problem

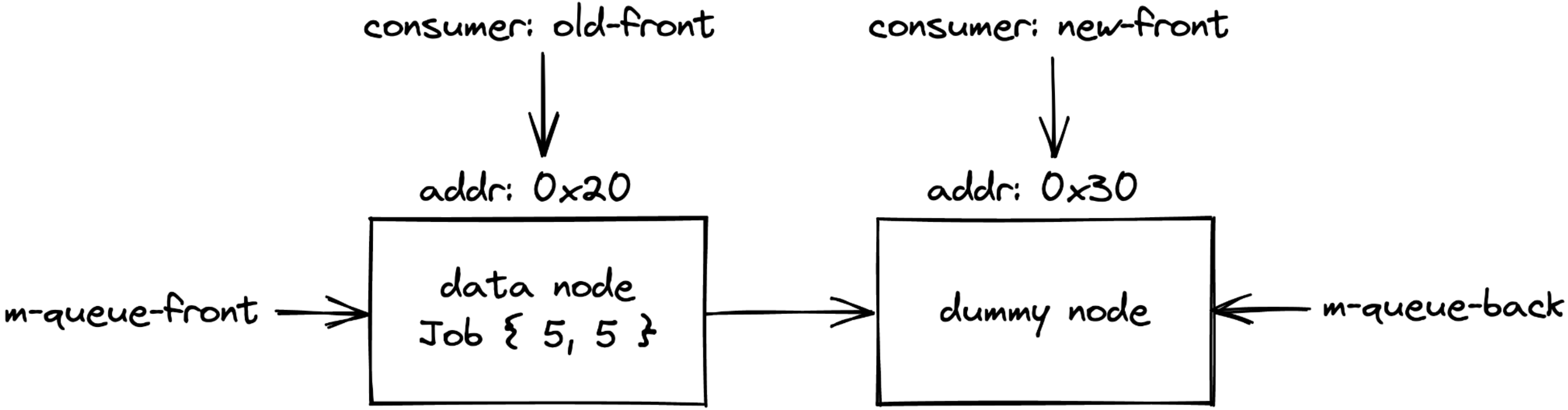


What are we comparing?

Pointer values!

Is there a problem?

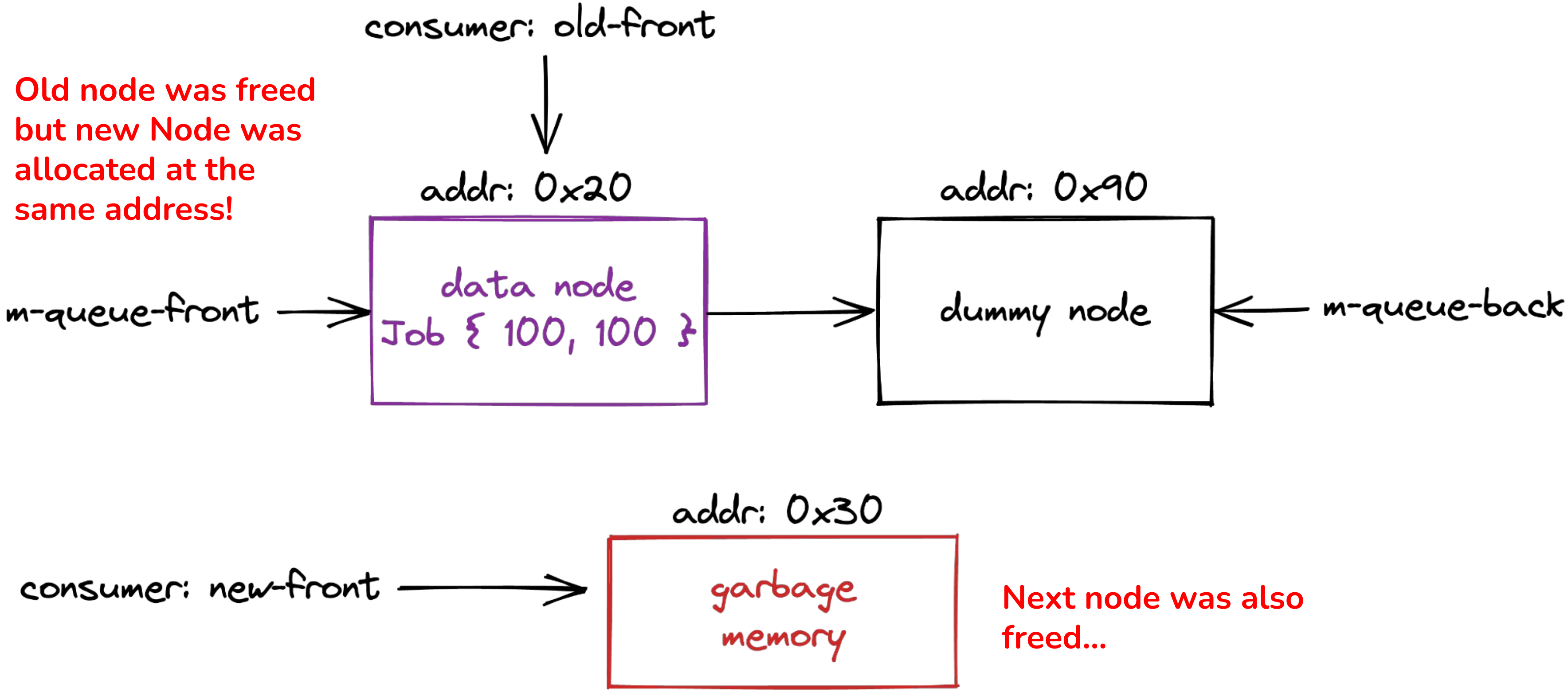
Problem #1: the ABA problem




A FEW
MOMENTS LATER

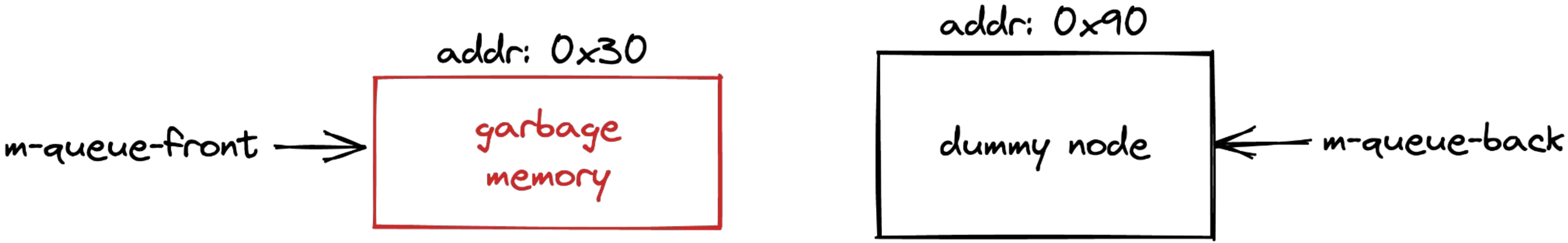
Problem #1: the ABA problem

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



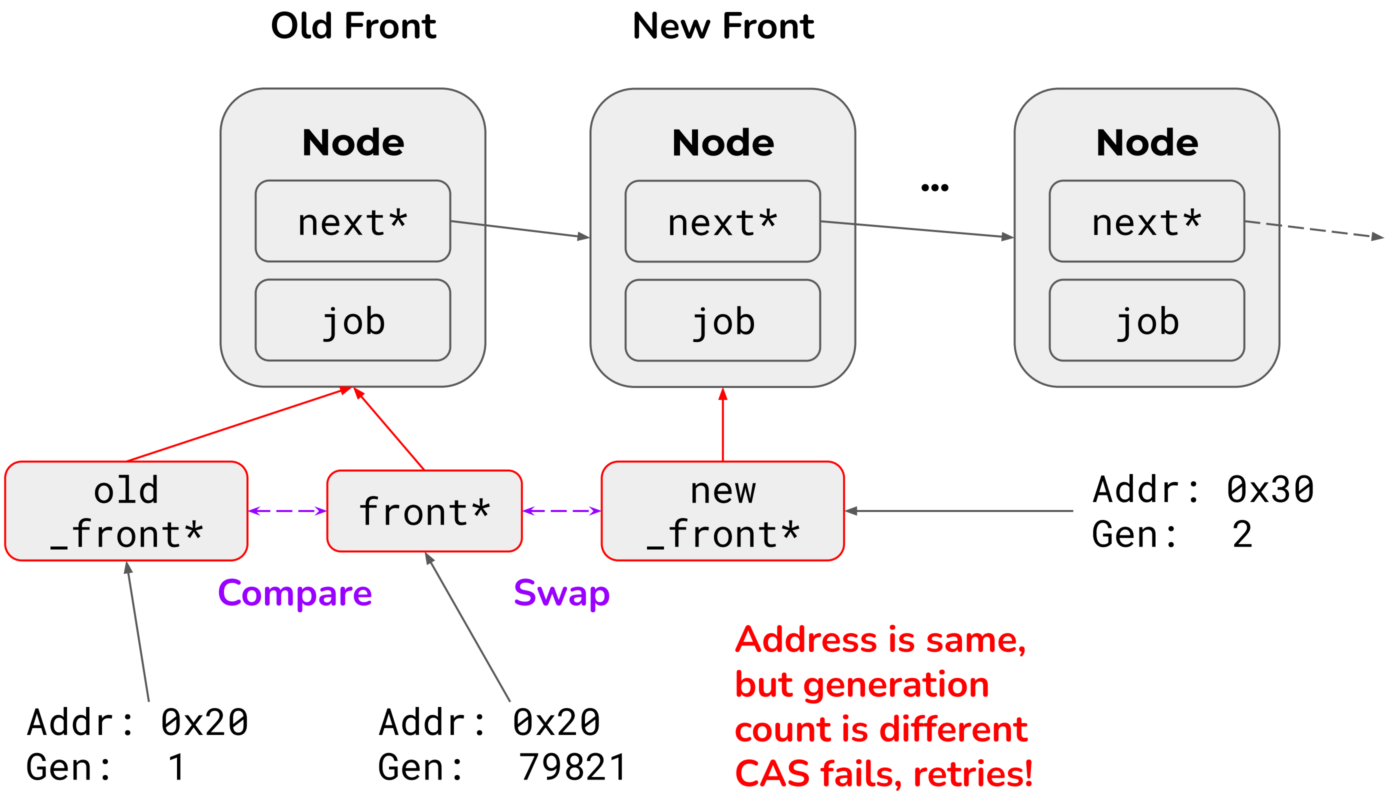
Next node was also freed...

Problem #1: the ABA problem

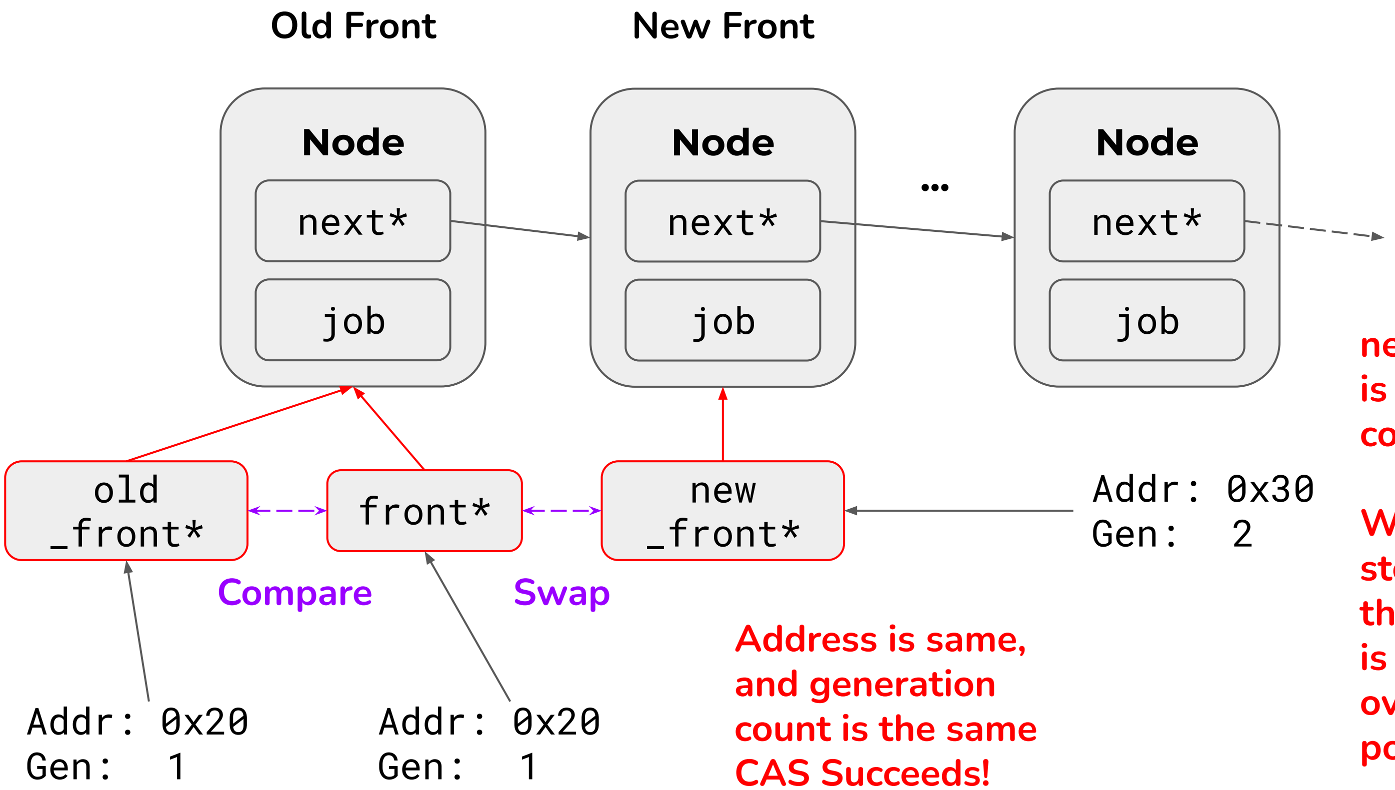


CAS Succeeds!
Everything has gone wrong

Problem #1: the ABA problem Solution



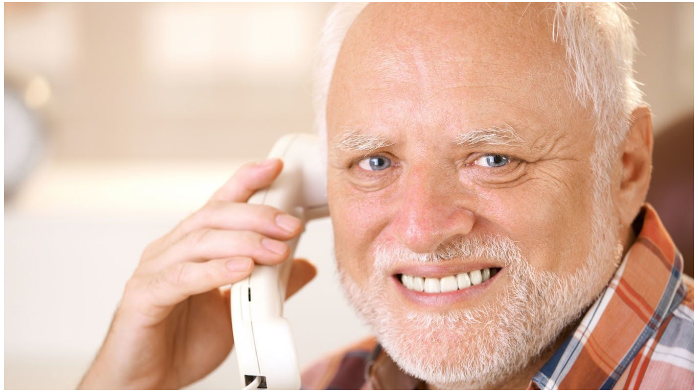
Problem #1: the ABA problem Solution



Address is same, and generation count is the same CAS Succeeds!

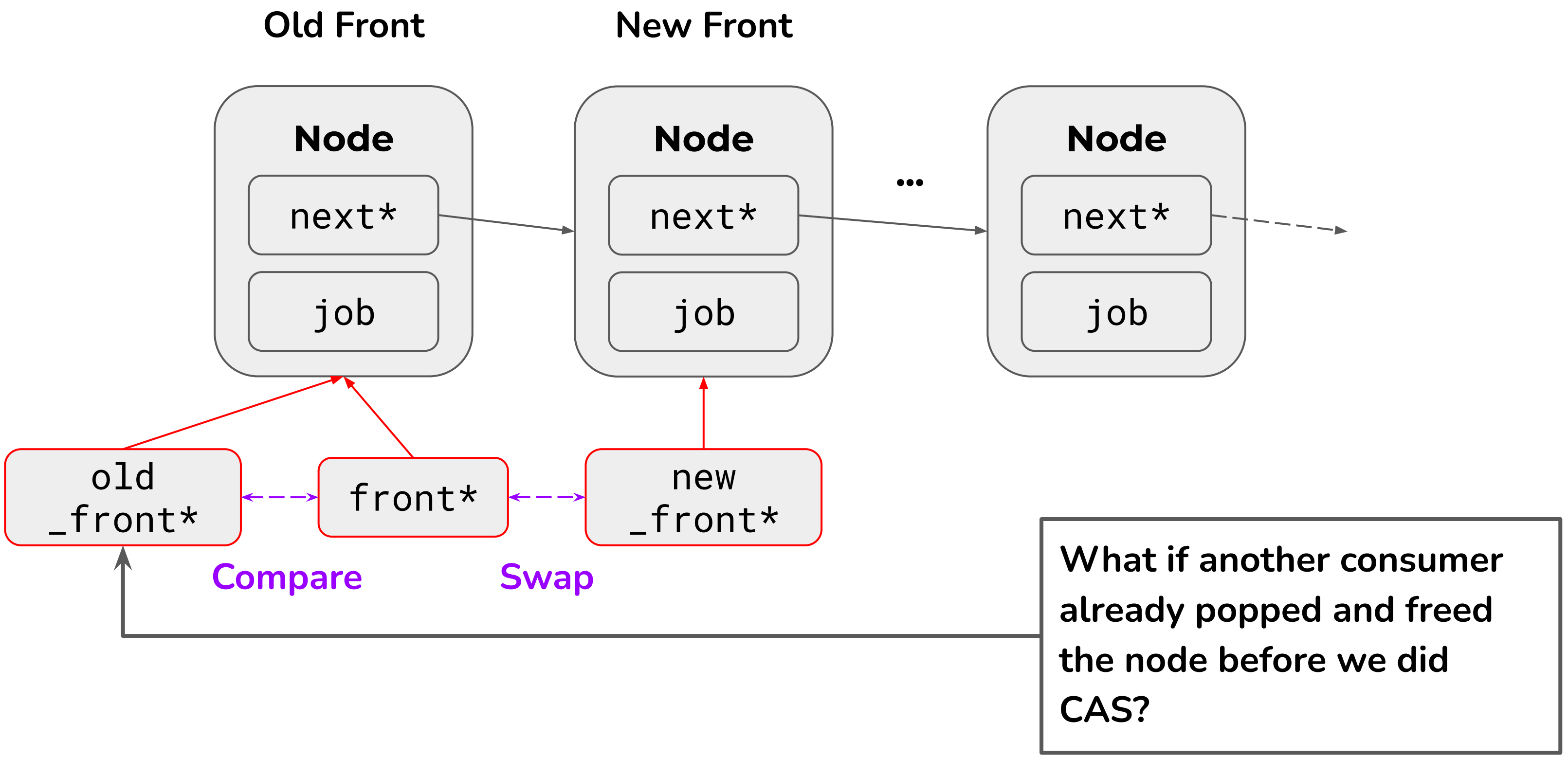
new_front generation count is old_front generation count + 1

We use 64 bit integer to store generation count so the likelihood that address is the same AND count overflows (2^64 nodes popped) is low!



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

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



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

Solutions?

1. Never free anything (ie. just leak all the memory).

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

Solutions?

1. Never free anything (ie. just leak all the memory).
2. Mark nodes for deletion while there are still threads in try_pop, and when the last one leaves, we free all of them at once.

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

Solutions?

1. Never free anything (ie. just leak all the memory).
2. Mark nodes for deletion while there are still threads in try_pop, and when the last one leaves, we free all of them at once.
3. Use reference counting (ie. an atomic shared_ptr) to know when there are no more remaining references to a particular object.

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

Solutions?

1. Never free anything (ie. just leak all the memory).
2. Mark nodes for deletion while there are still threads in try_pop, and when the last one leaves, we free all of them at once.
3. Use reference counting (ie. an atomic shared_ptr) to know when there are no more remaining references to a particular object.
4. Use **hazard pointers** to track which threads have references to which objects.

For those who are curious what hazard pointers are: https://melodiessim.netlify.app/intro-hazard-ptrs/

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

Solutions? (We will discuss no 2)

1. ~~Never free anything (ie. just leak all the memory).~~
2. Mark nodes for deletion while there are still threads in try_pop, and when the last one leaves, we free all of them at once.
3. ~~Use reference counting (ie. an atomic shared_ptr) to know when there are no more remaining references to a particular object.~~
4. ~~Use **hazard pointers** to track which threads have references to which objects.~~

   ~~For those who are curious what hazard pointers are: https://melodiessim.netlify.app/intro-hazard-ptrs/~~

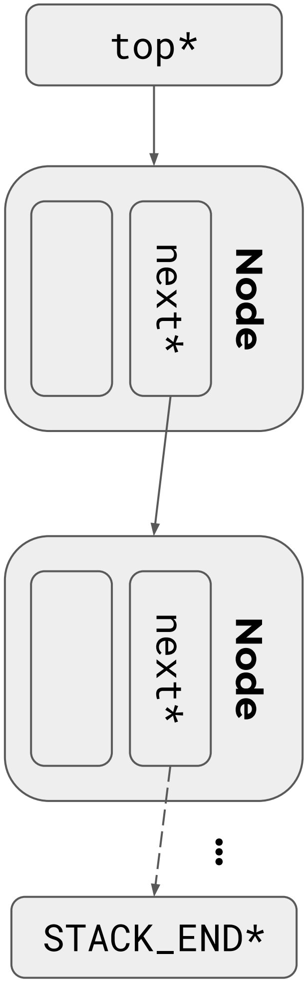
Problem #2: use-after-free (UAF)

Recycle nodes to save memory!

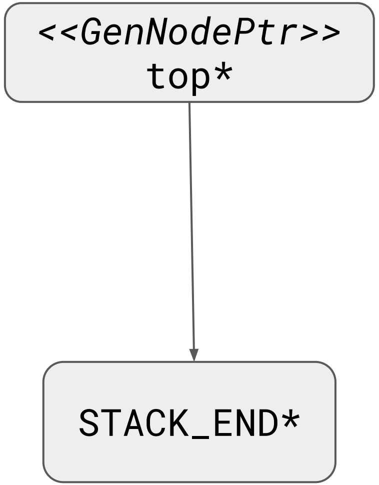
We'll use a concurrent stack to hold our "deleted" nodes!

...

It's actually just the queue but sideways and with only one end :)



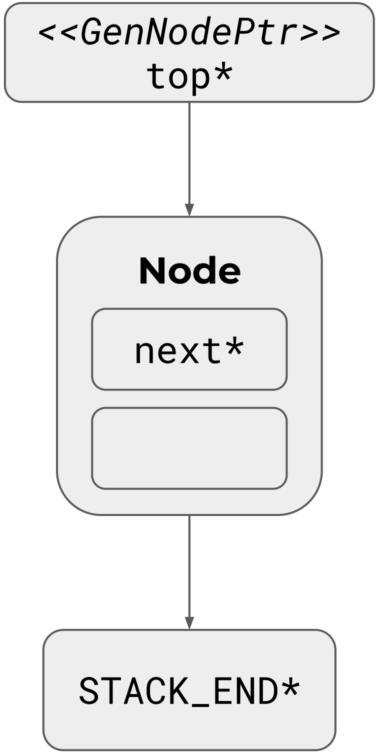
Problem #2: Node Allocation (Consuming from Stack)



Case 1:

If the stack is empty (i.e. `top == STACK_END`) then return a new Node

Problem #2: Node Allocation



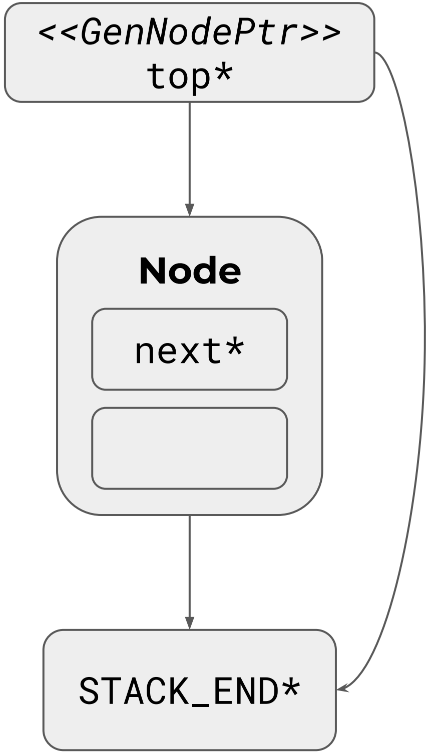
Case 1:

If the stack is empty (i.e. `top == STACK_END`) then return a new Node

Case 2:

If the stack is non-empty, similar to `try_pop` , then we try use CAS to pop the

Problem #2: Node Allocation



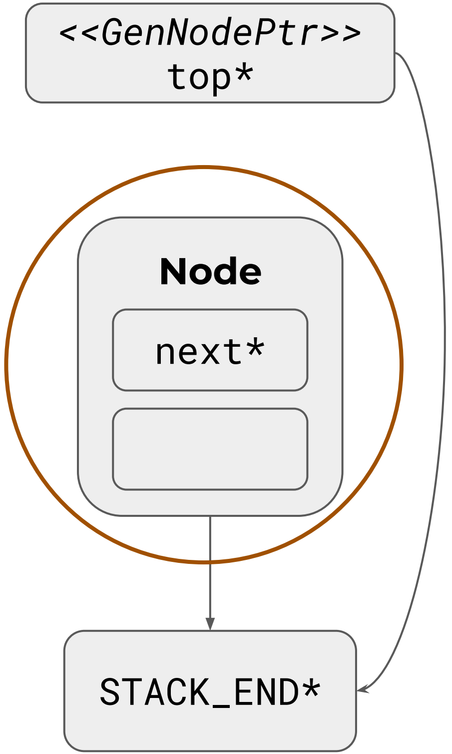
Case 1:

If the stack is empty (i.e. `top == STACK_END`) then return a new Node

Case 2:

If the stack is non-empty, similar to `try_pop` , then we try use CAS to pop the

Problem #2: Node Allocation



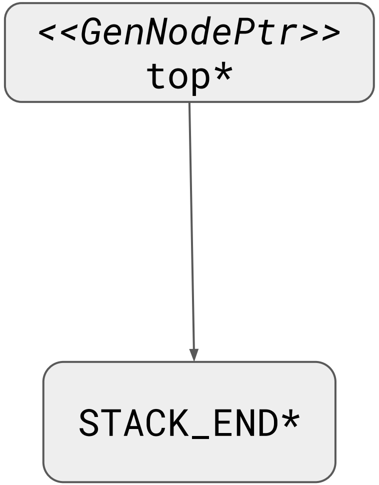
Case 1:

If the stack is empty (i.e. `top == STACK_END`) then return a new Node

Case 2:

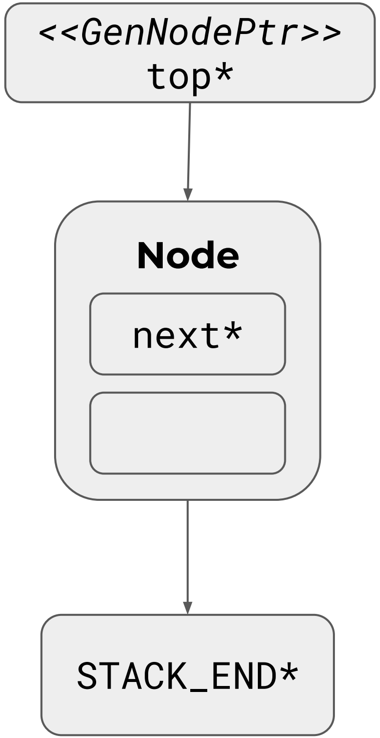
If the stack is non-empty, similar to `try_pop` , then we try use CAS to pop the

Problem #2: Node Deletion (Producing to Stack)



1. We cannot immediately use exchange like one in queue because the top is both modified by push and pop to the stack.
2. Therefore, we need to use CAS pattern utilizing `GenNodePtr`

Problem #2: Node Deletion (Producing to Stack)



1. We cannot immediately use exchange like one in queue because the top needs to be checked in producing to the stack
2. Therefore, we need to use CAS pattern utilizing `GenNodePtr`

Full code can be found in **t4.html**

Problem #3: Data race in recycling stack

Problem #3: Data race in recycling stack

```
WARNING: ThreadSanitizer: data race (pid=3978284)
  Write of size 8 at 0x7b0400000008 by thread T1:
    #0 JobQueue11B::push(Job) demo3.cpp:134 (demo3.tsan+0xe163d)
    <...>

  Previous read of size 8 at 0x7b0400000008 by thread T3:
    #0 JobQueue11B::try_pop() demo31.cpp:171 (demo3.tsan+0xe0ecd)
    <...>
```

Problem #3: Data race due to recycling stack

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 **A**

↓ 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( … )` **B**

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 **C**
`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

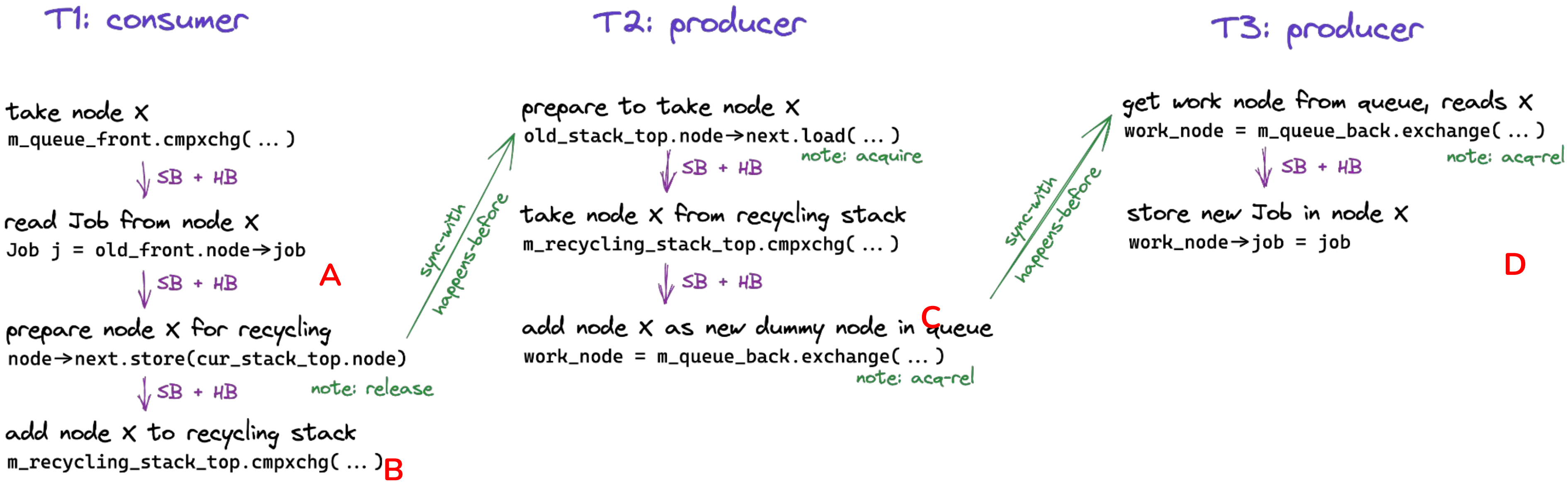
↓ SB + HB

store new Job in node X
`work_node→job = job` note: the race **D**

A. Node X is read by consumer T1
B. Node X is put into the recycling stack by consumer T1
C. Node X is taken out of the recycling stack by producer T2 and used as new dummy node
D. Node X is wrote to by producer T3

But no happens-before relationship between A (L154) and D (L134)! Data race possible! => Initially both Line A and D have **stdmo::relaxed**

Problem #3: Data race due to recycling stack



A. Node X is read by consumer T1
B. Node X is put into the recycling stack by consumer T1
C. Node X is taken out of the recycling stack by producer T2 and used as new dummy node
D. Node X is wrote to by producer T3

Happens-before relationship between A and D! No data race!

Problem #4: Internal data race

Problem #4: Internal data race

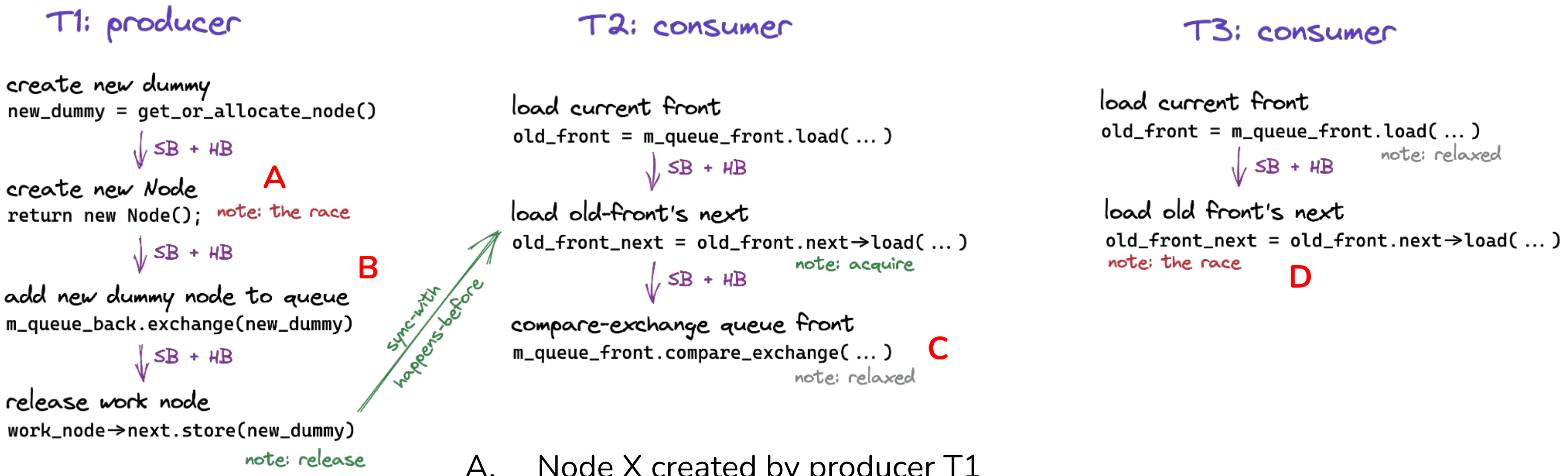
Another data race... This time between:

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

and

```
Node* old_front_next = old_front.node->next.load(stdmo::acquire);
```

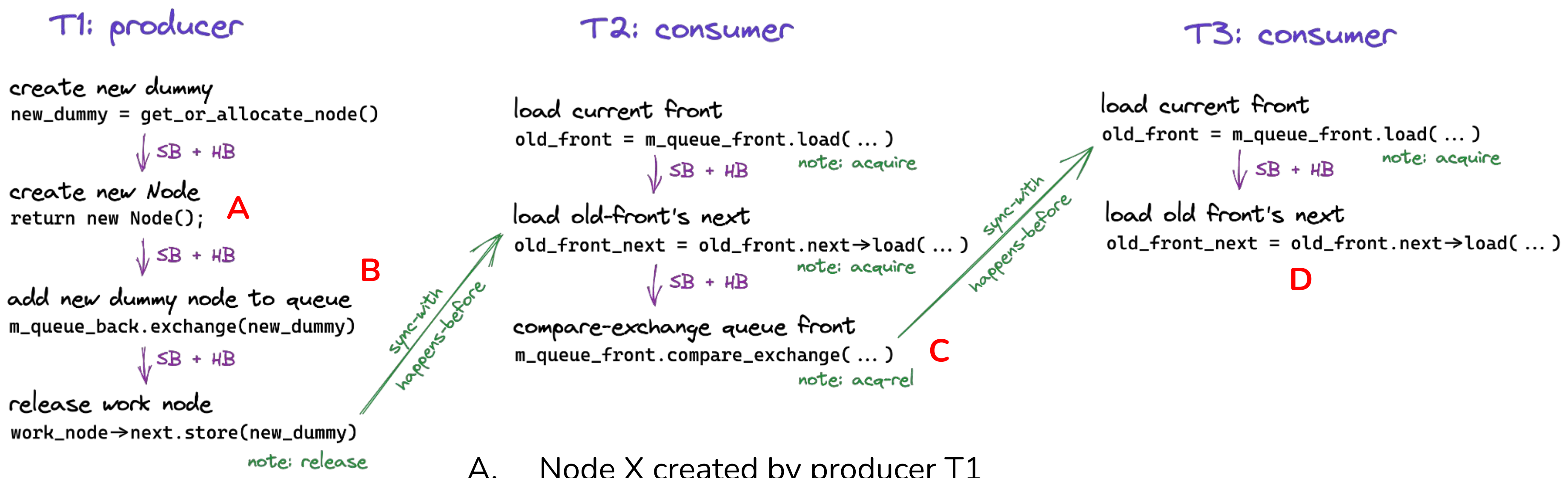
Problem #4: Internal data race



A. Node X created by producer T1
B. Node X is pushed to queue by producer T1 as dummy node
C. Node X is made front by consumer T2
D. Node X is read from by consumer T3

But no happens-before relationship between A and D! Data race possible!
Even if it seems time travel would need to happen…

Problem #4: Internal data race



A. Node X created by producer T1
B. Node X is pushed to queue by producer T1 as dummy node
C. Node X is made front by consumer T2 (L163)
D. Node X is read from by consumer T3 (L154)

Happens-before relationship between A and D! No data race!

Problem #5: Lack of Linearizability

What’s wrong with this?

```
int main() {
    JobQueue5 queue;
    auto t1 = std::jthread{[&queue]{
        queue.push(Job{1, 1});
    }};
    auto t2 = std::jthread{[&queue]{
        queue.push(Job{2, 2});
        if (!queue.try_pop()) {
            printf("saw empty queue\n");
        }
    }};
}
```

There are 3 important operations here:

1. push(Job{1,1})
2. push(Job{2,2})
3. try_pop()

What’s wrong with this?

There are 3 important operations here:

1. push(Job{1,1})
2. push(Job{2,2})
3. try_pop()

If they are linearizable, the possible orderings are (respecting happens before)

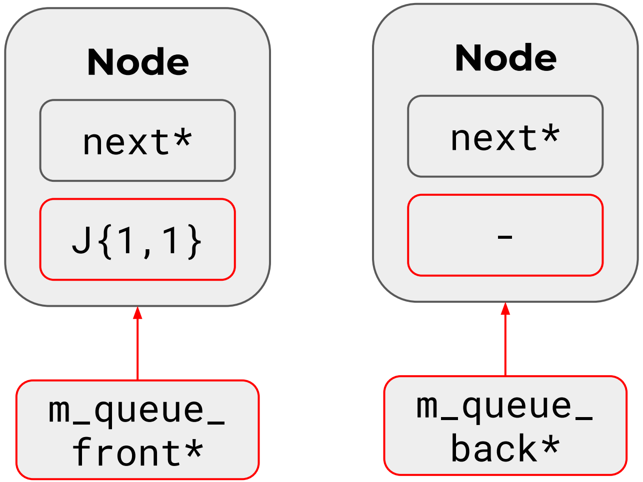
1. push(Job{1,1}) -> push(Job{2,2}) -> try_pop()
2. push(Job{2,2}) -> push(Job{1,1}) -> try_pop()
3. push(Job{2,2}) -> try_pop() -> push(Job{1,1})

All in all, try_pop() cannot be empty in all cases

What's wrong with this?

But consider cases where:

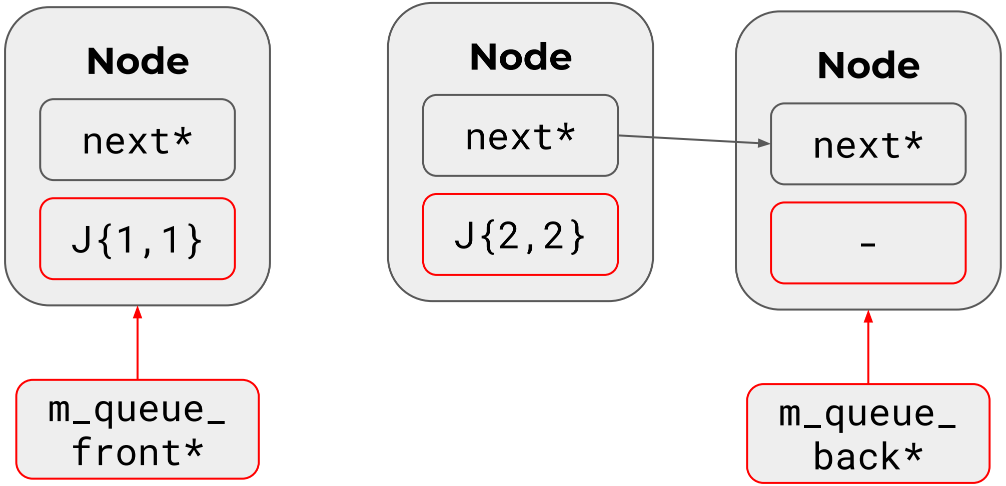
1. T1 => start to push Job{1,1} => Allocate `new_dummy` => exchange `m_queue_back` with `new_dummy` and store `Job{1,1}` into `work_node` => **SUSPENDED (Haven't Linked yet)**



What's wrong with this?

But consider cases where:

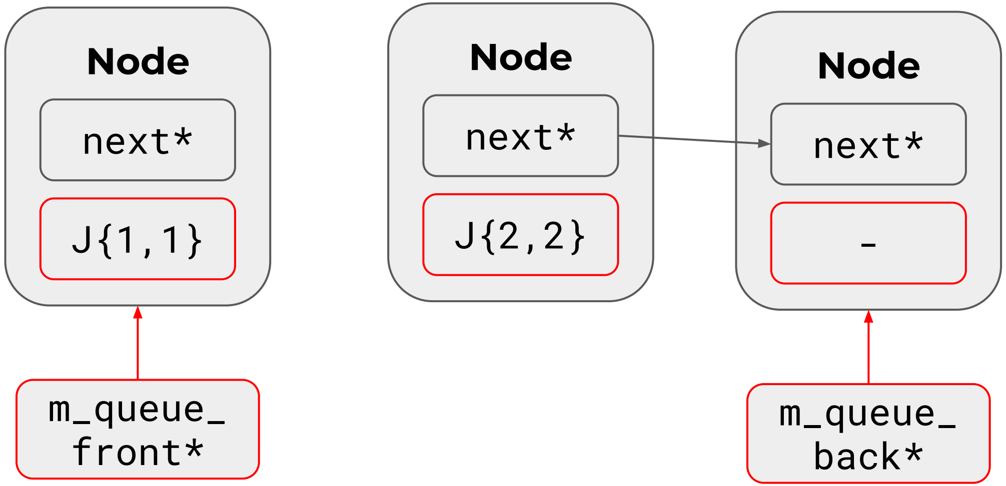
1. T2 => Push Job{2,2} (into the m_queue_back) => finish pushing



What’s wrong with this?

But consider cases where:

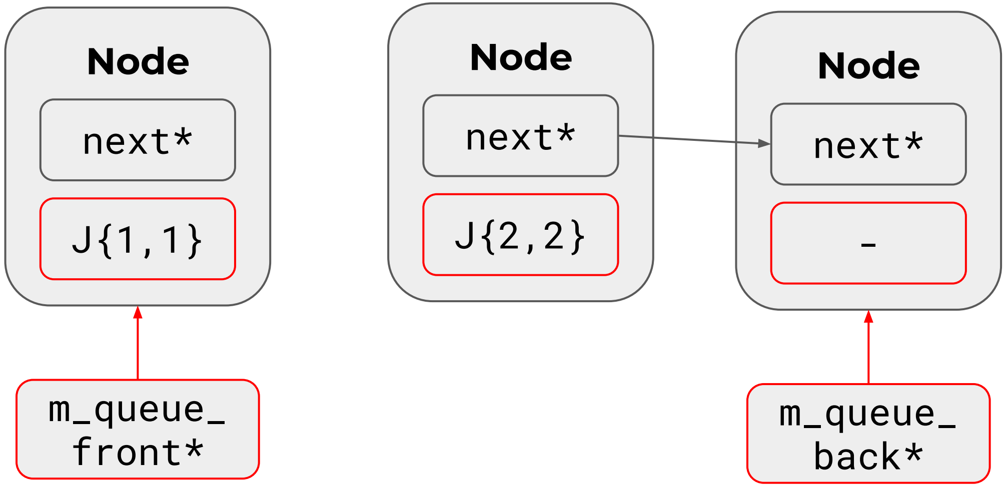
1. T2 => Push Job{2,2} (into the `m_queue_back`) => finish pushing => Try popping
2. But the next* of m_queue_front is `nullptr`, **so it must be dummy node, hence it saw empty queue => impossible for a linearizable queue**



What's wrong with this?

Solution: Michael-Scott Queue → maintaining Linked List as a priority, so such case is impossible (https://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf)

If you are interested, you can read how it's being implemented and summarize it for me afterward



Takeaways

- Lock free programming is hard… Mad respect to C++ ladz
- Compare-And-Swap Pattern
- ABA Problem and Generation Count
- Recycling Data Structure Memory
- Intro to “Linearizability” of the Queue
- Data Races are hard to solve… but drawing diagrams helps!