CS3211 Tutorial 4

(Failing to do) Lock-Free Programming in C++

(AY 25/26 Semester 2)

March 5, 2026

(Compiled by Benson, thanks to all past and current TAs!)
(Also thanks to this extremely good talk on Michael-Scott Queue)

Join our telegram group!
https://t.me/+MqCIT5ObVgxiMzRl

Contents

Contents

Disadvantages of Blocking

- **Deadlocks**
- **Priority Inversion**
  - A low priority process holds a lock.
  - High priority process cannot make progress.
- **(Lack of) Reliability**
  - A process that holds a lock dies.
- **(Suboptimal) Performance**
  - A process that holds a lock is kicked out by the scheduler.
  - No parallelism inside critical section.

Non-blocking Guarantees

Wait-Free	**Per-thread progress** is guaranteed i.e. Every operation completes in a finite number of steps
Lock-Free	**Overall progress** is guaranteed i.e. At least one thread complete in a finite number of steps
Obstruction-Free	**Overall progress** is guaranteed if threads **don't interfere** with each other

Compare-and-swap (CAS) Pattern

```
1  int old = x.fetch_add(1);
```

```
1  int old = x.load();
2  int new_val = old + 1;
3  x.store(new_val);
```

Is this correct?

Compare-and-swap (CAS)

```
int val = 0;
if (x == val) {
  x = 5;
} else {
  val = x;
}
```

```
int val = 0;
x.compare_and_exchange_weak(val, 5);
// or
x.compare_and_exchange_strong(val, 5);
```

The `_weak` version can *spuriously fail*, but the `_strong` version cannot.

Compare-and-swap (CAS) Pattern

```
1  int old = x.fetch_add(1);
```

```
1  int v = x.load();
2  while (true) {
3    if (x.compare_exchange_weak(v, v + 1)) {
4      return v;
5    }
6  }
```

This solution guarantees...

A. Wait-Free

B. Lock-Free

C. Obstruction-Free

Lock-free: When many threads are running this code snippet, at least one thread will be able to increment x.

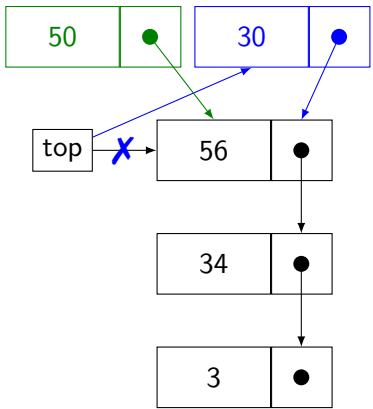
Not wait-free: Suppose other threads keep coming in and increment x successfully, an individual thread may starve indefinitely.

Lock-Free Stack

```
1  void push(Job job) {
2    Node* newTop = new Node{job, nullptr};
3    Node* oldTop = top.load();
4    newTop->next.store(oldTop);
5    top.store(newTop);
6  }
```

⇓

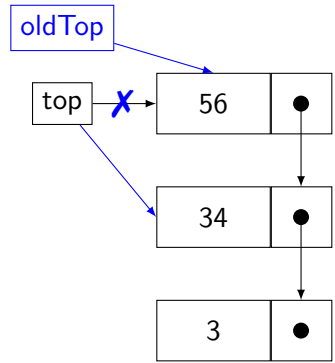
```
1  void push(Job job) {
2    Node* newTop = new Node{job, nullptr};
3    Node* oldTop = top.load();
4    while (true) {
5      newTop->next.store(oldTop);
6      if (top.compare_exchange_weak(oldTop, newTop)) {
7        break;
8      }
9    }
10 }
```



Checkpoint: Lock-Free Stack

Exercise (5 mins): Implement the `try_pop` method of the lock-free stack. Modify from this version.

```
1  std::optional<Job> try_pop() {
2    Node* oldTop = top.load();
3    if (oldTop == nullptr) return std::nullopt;
4    Node* newTop = oldTop->next.load();
5    top.store(newTop);
6    Job job = oldTop->job;
7    delete oldTop;
8    return oldTopValue;
9  }
```



Checkpoint: Lock-Free Stack

Solution. You should have gotten this:

```
std::optional<Job> try_pop() {
  Node* oldTop = top.load();
  while (true) {
    // Caution: We might get nullptr later on!
    if (oldTop == nullptr) return std::nullopt;
    Node* newTop = oldTop->next.load();
    if (top.compare_exchange_weak(oldTop, newTop)) {
      Job job = oldTop->job;
      delete oldTop;
      return oldTopValue;
    }
  }
}
```

Contents

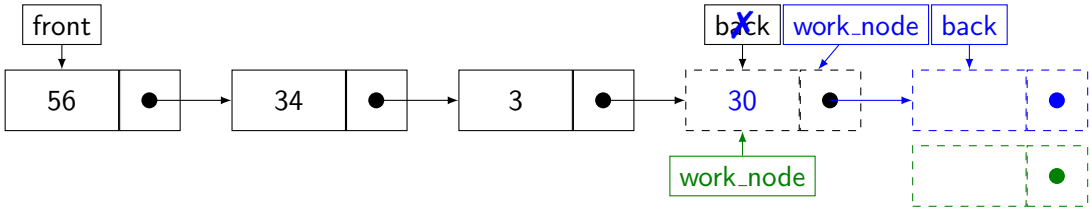
How hard is it to implement a lock-free queue?

Fast forward to 1+ hour later...

> Sadly, despite our heroic efforts, the queue is *still* incorrect.
> ...
> ... other changes have to be made to account for this fact that we don't have time to cover in this tutorial.

So why are we here?

Initial Attempt: Enqueue



Enqueue Tasks:

[A] Create a new dummy node `new_dummy`.

[B+E] Get current `work_node` (= `back`), and update `back` to `new_dummy`.

[C] Set the item in `work_node`.

[D] Set the next pointer of `work_node` to point to `new_dummy`.

Initial Attempt: Enqueue

```
1  void push(Job job) {
2    Node* new_dummy = new Node();                                    // A
3    Node* work_node = m_queue_back.exchange(new_dummy, stdmo::relaxed); // B+E
4    work_node->job = job;                                            // C
5    work_node->next.store(new_dummy, stdmo::relaxed);                // D
6  }
```

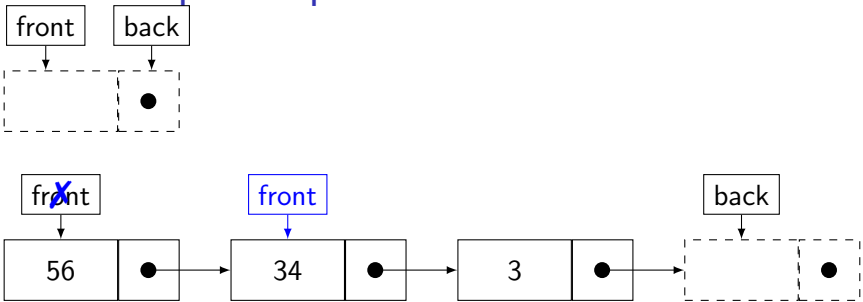
[A] Create a new dummy node `new_dummy`.

[B+E] Get current `work_node` (= `back`), and update `back` to `new_dummy`.

[C] Set the item in `work_node`.

[D] Set the next pointer of `work_node` to point to `new_dummy`.

Initial Attempt: Dequeue



Dequeue Tasks:

[A] Read `front`.

[B] If the next pointer of `front` is null, then the queue is empty.

[C] Update `front` to its next node.

[D] Return the old value of `front`.

} Done in a CAS loop!

Initial Attempt: Dequeue

```
1  std::optional<Job> try_pop() {
2    Node* old_front = m_queue_front.load(stdmo::relaxed);              // A
3    while (true) {
4      Node* new_front = old_front->next.load(stdmo::relaxed);
5      if (new_front == QUEUE_END) {                                    // B
6        return std::nullopt;
7      }
8
9      if (m_queue_front.compare_exchange_weak(old_front, new_front,
10                                             stdmo::relaxed)) {       // C+D
11       break;
12     }
13   }
14   Job job = old_front->job;
15   delete old_front;
16   return job;
17 }
```

- [A] Read `front`.
- [B] If the next pointer of `front` is null, then the queue is empty.
- [C+D] Update `front` to its next node, and return the old value of `front`.

Problem #1: Lots of Data Races

How can a data race occur?

Producer

- Create new node

```
new_dummy = new Node()
```

- Add new dummy node to queue

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

- Store job in old dummy node

```
work_node->job = job
```

- Link work node to new dummy

```
work_node->next.store(new_dummy)
```

Consumer

- Load current front

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

- Load old front's next

```
new_front = old_front->next.load()
```

- Compare-exchange queue front

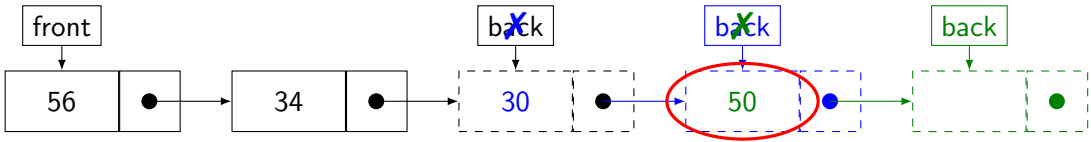
```
m_queue_front.CAS(...)
```

- Read job from node

```
job = old_front->job
```

Problem #1: Lots of Data Races

Producer-Producer Race



T1: Producer

- Create new node

```
new_dummy = new Node()
```

- Add new dummy node to queue

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

- Store job in old dummy node

```
work_node->job = job
```

- Link work node to new dummy

```
work_node->next.store(new_dummy)
```

acq_rel SW

T2: Producer

- Create new node

```
new_dummy = new Node()
```

- Add new dummy node to queue acq_rel

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

- Store job in old dummy node

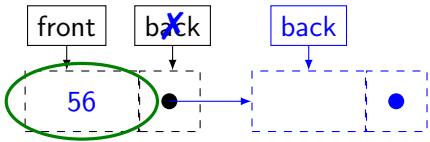
```
work_node->job = job
```

- Link work node to new dummy

```
work_node->next.store(new_dummy)
```

Problem #1: Lots of Data Races

Producer-Consumer Race



T1: Producer

- Create new node

```
new_dummy = new Node()
```

- Add new dummy node to queue

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

- Store job in old dummy node

```
work_node->job = job
```

- Link work node to new dummy

```
work_node->next.store(new_dummy)
```

release

SW

T2: Consumer

- Load current front

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

- Load old front's next (NOT NULL) acquire

```
new_front = old_front->next.load()
```

- Compare-exchange queue front

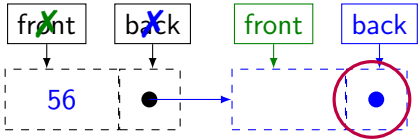
```
m_queue_front.CAS(...)
```

- Read job from node

```
job = old_front->job
```

Problem #1: Lots of Data Races

Producer-(Consumer after Consumer) Race



T1: Producer

- Create new node

```
new_dummy = new Node()
```

- Add new dummy node to queue

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

- Store job in old dummy node

```
work_node->job = job
```

- Link work node to new dummy

```
work_node->next.store(new_dummy)
```

T2: First Consumer

- Load current front

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

- Load old front's next

```
new_front = old_front->next.load()
```

- Compare-exchange queue front

```
m_queue_front.CAS(...)
```

- Read job from node

```
job = old_front->job
```

SW

T3: Second Consumer

- Load current front

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

acquire

- Load old front's next

```
new_front = old_front->next.load()
```

- Compare-exchange queue front

```
m_queue_front.CAS(...)
```

acquire

(loads front as well!)

- Read job from node

```
job = old_front->job
```

SW

release

Problem #1: Lots of Data Races

Producer

- Create new node

```
new_dummy = new Node()
```

- Add new dummy node to queue (acq_rel)

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

- Store job in old dummy node

```
work_node->job = job
```

- Link work node to new dummy (release)

```
work_node->next.store(new_dummy)
```

Consumer

- Load current front (acquire)

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

- Load old front's next (acquire)

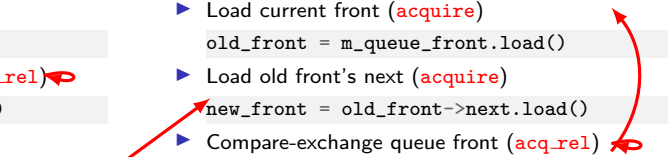
```
new_front = old_front->next.load()
```

- Compare-exchange queue front (acq_rel)

```
m_queue_front.CAS(...)
```

- Read job from node

```
job = old_front->job
```



Checkpoint

Exercise + Break (5 mins): Refer to your lock-free stack implementation.

- Illustrate a data race if we use `std::memory_order_relaxed`.
  Then, suggest what we should do to fix it (should be very simple!).

Checkpoint

Solution.

```
1  void push(Job job) {
2    Node* newTop =
3        new Node{job, nullptr};
4    // std::memory_order_relaxed
5    Node* oldTop = top.load();
6    while (true) {
7      // std::memory_order_relaxed
8      newTop->next.store(oldTop);
9      // std::memory_order_relaxed
10     if (top.CAS(oldTop, newTop)) {
11       break;
12     }
13   }
14 }
```

```
1  std::optional<Job> try_pop() {
2    // std::memory_order_relaxed
3    Node* oldTop = top.load();
4    while (true) {
5      if (oldTop == nullptr) {
6        return std::nullopt;
7      }
8      // std::memory_order_relaxed
9      Node* newTop = oldTop->next.load();
10     // std::memory_order_relaxed
11     if (top.CAS(oldTop, newTop)) {
12       Job job = oldTop->job;
13       delete oldTop;
14       return oldTopValue;
15     }
16   }
17 }
```

Producer-consumer race: Line 3 of `push` races with Line 9 & 12 of `try_pop`.

Checkpoint

Solution. Solution to data race:

```
void push(Job job) {
  Node* newTop =
      new Node{job, nullptr};
  // std::memory_order_relaxed
  Node* oldTop = top.load();
  while (true) {
    // std::memory_order_relaxed
    newTop->next.store(oldTop);
    // std::memory_order_release
    if (top.CAS(oldTop, newTop)) {
      break;
    }
  }
}
```

```
std::optional<Job> try_pop() {
  // std::memory_order_acquire
  Node* oldTop = top.load();
  while (true) {
    if (oldTop == nullptr) {
      return std::nullopt;
    }
    // std::memory_order_relaxed
    Node* newTop = oldTop->next.load();
    // std::memory_order_acquire
    if (top.CAS(oldTop, newTop)) {
      Job job = oldTop->job;
      delete oldTop;
      return oldTopValue;
    }
  }
}
```

Checkpoint

Solution. Is this correct?

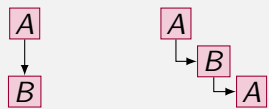
```
1  void push(Job job) {
2    Node* newTop =
3        new Node{job, nullptr};
4    // std::memory_order_relaxed
5    Node* oldTop = top.load();
6    while (true) {
7      // std::memory_order_release
8      newTop->next.store(oldTop);
9      // std::memory_order_relaxed
10     if (top.CAS(oldTop, newTop)) {
11       break;
12     }
13   }
14 }
```

```
1  std::optional<Job> try_pop() {
2    // std::memory_order_relaxed
3    Node* oldTop = top.load();
4    while (true) {
5      if (oldTop == nullptr) {
6        return std::nullopt;
7      }
8      // std::memory_order_acquire
9      Node* newTop = oldTop->next.load();
10     // std::memory_order_relaxed
11     if (top.CAS(oldTop, newTop)) {
12       Job job = oldTop->job;
13       delete oldTop;
14       return oldTopValue;
15     }
16   }
17 }
```

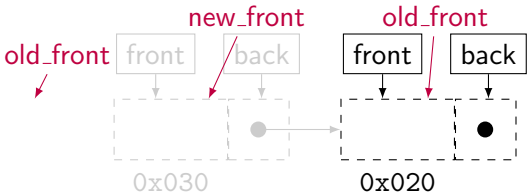
Problem #2: ABA Problem

Recall our compare-and-swap (CAS) loop:

```
1  while (true) {
2    old_value = value.load();
3    // do something with old_value
4    if (value.compare_exchange_weak(old_value, new_value)) {
5      break;
6    }
7  }
```



Problem #2: ABA Problem



Consumer

- Keep Trying:
  - Load current front

    ```
    old_front = m_queue_front.load()
    ```
  - Load old front's next

    ```
    new_front = old_front->next.load()
    ```
  - Compare-exchange queue front

    ```
    m_queue_front.CAS(...)
    ```
- Read job from node

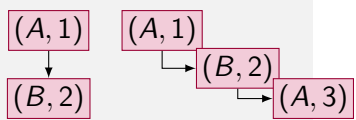
  ```
  job = old_front->job
  ```

Problem #2: ABA Problem

Solution: Generation Counters

Intuition. Ban duplicate values completely.

```
1  while (true) {
2    old_value = value.load();
3    // do something with old_value
4    if (value.compare_exchange_weak(old_value, new_value)) {
5      break;
6    }
7  }
```



Checkpoint

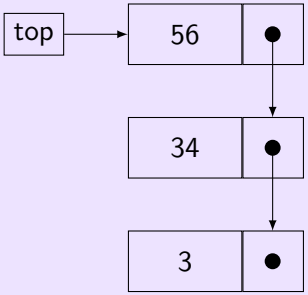
Exercise (5 mins): Refer to your lock-free stack implementation.

- Does it suffer from the ABA problem?
  If yes, draw a diagram to illustrate how the ABA problem could break its state and suggest a solution.
  If no, you should reconsider your choice.

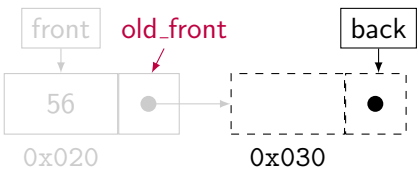
Checkpoint

Solution. Yes, the stack suffers from ABA Problem.

```
1  std::optional<Job> try_pop() {
2    Node* oldTop = top.load();
3    while (true) {
4      if (oldTop == nullptr) {
5        return std::nullopt;
6      }
7      Node* newTop = oldTop->next.load();
8      if (top.CAS(oldTop, newTop)) {
9        Job job = oldTop->job;
10       delete oldTop;
11       return oldTopValue;
12     }
13   }
14 }
```



Problem #3: Use-after-free (UAF) Problem



Consumer

- Keep Trying:
  - Load current front

    ```
    old_front = m_queue_front.load()
    ```
  - Load old front's next

    ```
    new_front = old_front->next.load()
    ```
  - Compare-exchange queue front

    ```
    m_queue_front.CAS(...)
    ```
- Read job from node

  ```
  job = old_front->job
  ```

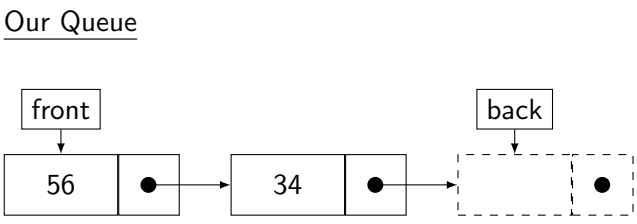
Problem #3: Use-after-free (UAF) Problem

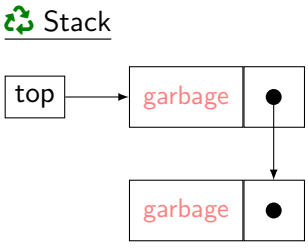
Possible Solutions:

- Never free anything (i.e. just leak all the memory).
- 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.
- Use reference counting (i.e. an atomic `shared_ptr`) to know when there are no more remaining references to a particular object.
- Use hazard pointers to track which threads have references to which objects.

Problem #3: Use-after-free (UAF) Problem

Solution: Recycling Centre





Problem #3: Use-after-free (UAF) Problem

Solution: Recycling Centre

Producer

- Try to get a recycled node:
  - Load current top
    ```
    old_top = m_stack_top.load()
    ```
  - Load current top's next
    ```
    new_top = old_top->next.load()
    ```
  - Compare-exchange stack top
    ```
    m_stack_top.CAS(...)
    ```
- Add new dummy node to queue
  ```
  m_queue_back.exchange(new_dummy)
  ```
- Store job in old dummy node
  ```
  work_node->job = job
  ```
- Link work node to new dummy
  ```
  work_node->next.store(new_dummy)
  ```

Consumer

- Load current front
  ```
  old_front = m_queue_front.load()
  ```
- Load old front's next
  ```
  new_front = old_front->next.load()
  ```
- Compare-exchange queue front
  ```
  m_queue_front.CAS(...)
  ```
- Read job from node
  ```
  job = old_front->job
  ```
- Recycle the node:
  - Load current top
    ```
    old_top = m_stack_top.load()
    ```
  - Link recycled node to old top
    ```
    new_top->next.store(old_top)
    ```
  - Insert recycled node onto stack
    ```
    m_stack_top.CAS(...)
    ```

Checkpoint

Select the correct statement(s) about the recycling centre!

A. The recycling centre suffers from the ABA problem.

B. We should add generation counters for the recycling centre.

C. The recycling centre suffers from the use-after-free (UAF) problem.

D. We should build a recycling centre for the recycling centre.

A and B only. Our stack suffers from the ABA problem, so does the recycling centre. Nodes are never freed from the stack (they get added back to the queue), so there is no use-after-free.

Problem #4: Data Race in Recycling Centre

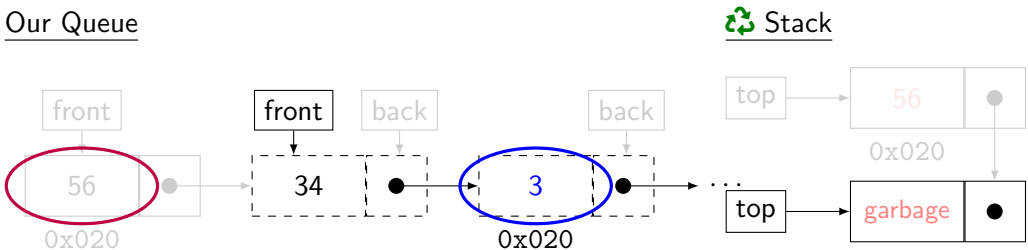
Producer

- Try to get a recycled node:
  - Load current top
    ```
    old_top = m_stack_top.load()
    ```
  - Load current top's next
    ```
    new_top = old_top->next.load()
    ```
  - Compare-exchange stack top
    ```
    m_stack_top.CAS(...)
    ```
- Add new dummy node to queue
  ```
  m_queue_back.exchange(new_dummy)
  ```
- Store job in old dummy node
  ```
  work_node->job = job
  ```
- Link work node to new dummy
  ```
  work_node->next.store(new_dummy)
  ```

Consumer

- Load current front
  ```
  old_front = m_queue_front.load()
  ```
- Load old front's next
  ```
  new_front = old_front->next.load()
  ```
- Compare-exchange queue front
  ```
  m_queue_front.CAS(...)
  ```
- Read job from node
  ```
  job = old_front->job
  ```
- Recycle the node:
  - Load current top
    ```
    old_top = m_stack_top.load()
    ```
  - Link recycled node to old top
    ```
    new_top->next.store(old_top)
    ```
  - Insert recycled node onto stack
    ```
    m_stack_top.CAS(...)
    ```

Problem #4: Data Race in Recycling Centre



Problem #4: Data Race in Recycling Centre

T1: Consumer

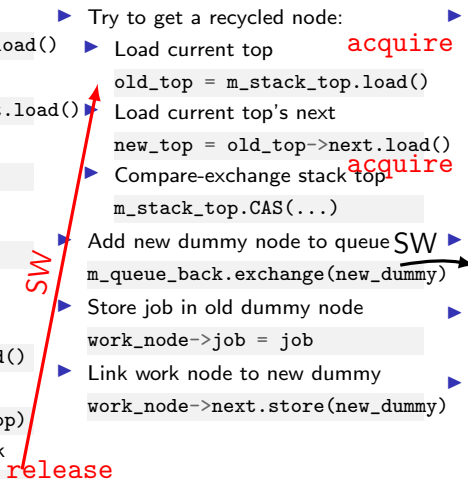
- Load current front
  `old_front = m_queue_front.load()`
- Load old front's next
  `new_front = old_front->next.load()`
- Compare-exchange queue front
  `m_queue_front.CAS(...)`
- Read job from node
  `job = old_front->job`
- Recycle the node:
  - Load current top
    `old_top = m_stack_top.load()`
  - Link recycled node to old top
    `new_top->next.store(old_top)`
  - Insert recycled node onto stack
    `m_stack_top.CAS(...)`

T2: Producer

- Try to get a recycled node:
  - Load current top
    `old_top = m_stack_top.load()`
  - Load current top's next
    `new_top = old_top->next.load()`
  - Compare-exchange stack top
    `m_stack_top.CAS(...)`
- Add new dummy node to queue
  `m_queue_back.exchange(new_dummy)`
- Store job in old dummy node
  `work_node->job = job`
- Link work node to new dummy
  `work_node->next.store(new_dummy)`

T3: Producer

- Try to get a recycled node:
  - Load current top
    `old_top = m_stack_top.load()`
  - Load current top's next
    `new_top = old_top->next.load()`
  - Compare-exchange stack top
    `m_stack_top.CAS(...)`
- Add new dummy node to queue
  `m_queue_back.exchange(new_dummy)`
- Store job in old dummy node
  `work_node->job = job`
- Link work node to new dummy
  `work_node->next.store(new_dummy)`



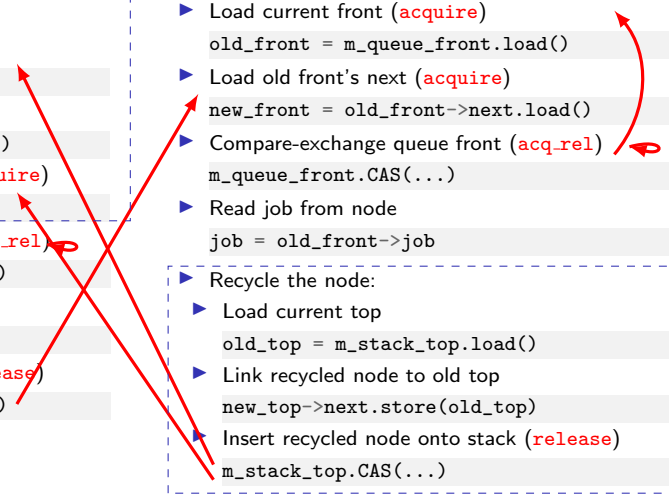
Our Final Implementation

Producer

- Try to get a recycled node:
  - Load current top (acquire)
    ```
    old_top = m_stack_top.load()
    ```
  - Load current top's next
    ```
    new_top = old_top->next.load()
    ```
  - Compare-exchange stack top (acquire)
    ```
    m_stack_top.CAS(...)
    ```
- Add new dummy node to queue (acq_rel)
  ```
  m_queue_back.exchange(new_dummy)
  ```
- Store job in old dummy node
  ```
  work_node->job = job
  ```
- Link work node to new dummy (release)
  ```
  work_node->next.store(new_dummy)
  ```

Consumer

- Load current front (acquire)
  ```
  old_front = m_queue_front.load()
  ```
- Load old front's next (acquire)
  ```
  new_front = old_front->next.load()
  ```
- Compare-exchange queue front (acq_rel)
  ```
  m_queue_front.CAS(...)
  ```
- Read job from node
  ```
  job = old_front->job
  ```
- Recycle the node:
  - Load current top
    ```
    old_top = m_stack_top.load()
    ```
  - Link recycled node to old top
    ```
    new_top->next.store(old_top)
    ```
  - Insert recycled node onto stack (release)
    ```
    m_stack_top.CAS(...)
    ```

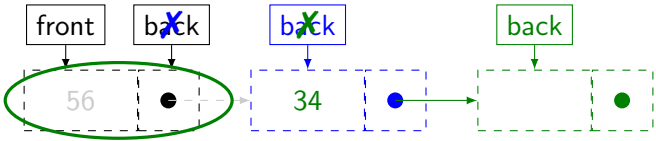


Our Final Implementation

ThreadSanitizer feels happy now.

Are we done?

Problem #5: Lack of Serializability



T1
`enqueue(56)`
(suspended)

T2
`enqueue(34)`
`try_pop()`

Producer

- Create new node

```
new_dummy = new Node()
```

- Add new dummy node to queue

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

- Store job in old dummy node

```
work_node->job = job
```

- Link work node to new dummy

```
work_node->next.store(new_dummy)
```

Consumer

- Load current front

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

- Load old front's next

```
new_front = old_front->next.load()
```

- Compare-exchange queue front

```
m_queue_front.CAS(...)
```

- Read job from node

```
job = old_front->job
```

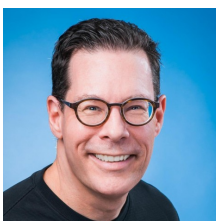
Problem #5: Lack of Serializability

Serializability *(intuitively)*: The results must correspond to an interleaved execution (which respects process order).

For this case, since `try_pop()` returns `std::nullopt`, our queue is **not serializable**.

Summary

- Compare-and-swap (CAS) Pattern.
- Common problems when implementing lock-free data structures:
  - ABA Problem $\Rightarrow$ Avoided by generation counters.
  - Use-after-free (UAF) $\Rightarrow$ Avoided by recycling memory.
- Lock-free programming is far from trivial.



"If you think using atomics is easy, you are either jaded, or not being careful enough." — Herb Sutter

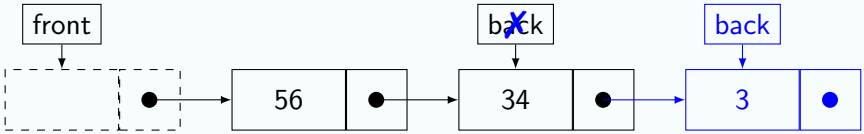
Attendance

Session ID: 1939882
CS3211 Tutorial 4
5 March 2026 14:00-16:00
BIZ2-02-02 - SEMINAR ROOM 2-2

Contents

Michael-Scott Queue: Intuition

Extra Slide

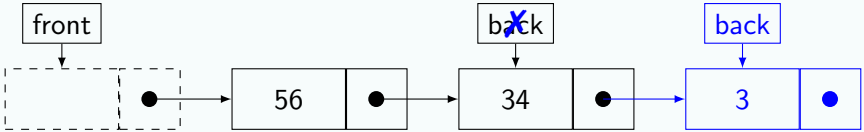



```
1  void enqueue(Job job) {
2    Node* newBack = new Node{job, nullptr};
3    bool success = false;
4    while (!success) {
5      Node* oldBack = m_queue_back.load();
6      success = oldBack->next.compare_exchange_weak(nullptr, newBack);
7      m_queue_back.compare_exchange_weak(oldBack, oldBack->next.load());
8    }
9  }
```

Michael-Scott Queue: Intuition

Extra Slide

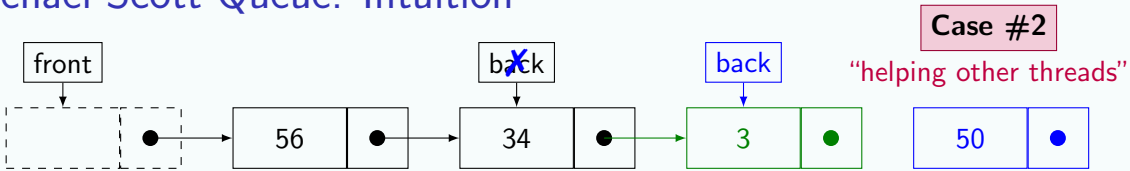
Case #1




```
1  void enqueue(Job job) {
2    Node* newBack = new Node{job, nullptr};
3    bool success = false;
4    while (!success) {
5      Node* oldBack = m_queue_back.load();
6      success = oldBack->next.compare_exchange_weak(nullptr, newBack);    // true
7      m_queue_back.compare_exchange_weak(oldBack, oldBack->next.load()); // true
8    }
9  }
```

Michael-Scott Queue: Intuition

Extra Slide



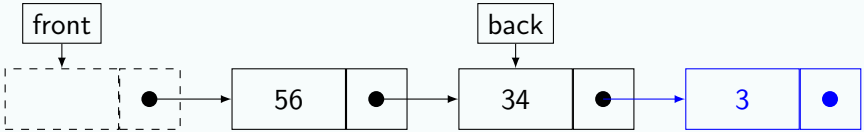

```
1  void enqueue(Job job) {
2    Node* newBack = new Node{job, nullptr};
3    bool success = false;
4    while (!success) {
5      Node* oldBack = m_queue_back.load();
6      success = oldBack->next.compare_exchange_weak(nullptr, newBack);    // false
7      m_queue_back.compare_exchange_weak(oldBack, oldBack->next.load()); // true
8    }
9  }
```

Michael-Scott Queue: Intuition

Extra Slide

Case #3

"need help"



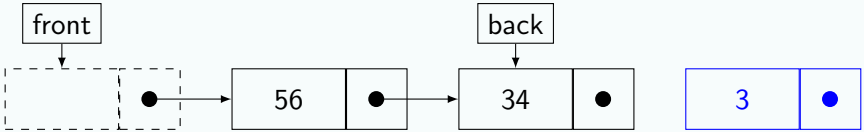

```
1  void enqueue(Job job) {
2    Node* newBack = new Node{job, nullptr};
3    bool success = false;
4    while (!success) {
5      Node* oldBack = m_queue_back.load();
6      success = oldBack->next.compare_exchange_weak(nullptr, newBack);    // true
7      m_queue_back.compare_exchange_weak(oldBack, oldBack->next.load()); // false
8    }
9  }
```

Michael-Scott Queue: Intuition

Extra Slide

Case #4

"nothing happened"




```
1  void enqueue(Job job) {
2    Node* newBack = new Node{job, nullptr};
3    bool success = false;
4    while (!success) {
5      Node* oldBack = m_queue_back.load();
6      success = oldBack->next.compare_exchange_weak(nullptr, newBack);   // false
7      m_queue_back.compare_exchange_weak(oldBack, oldBack->next.load()); // false
8    }
9  }
```

Michael-Scott Queue: Implementation

Extra Slide

```
1  void enqueue(Job job) {
2    Node* node, back, next;
3    node = new Node{job, nullptr};
4    while (true) {
5      back = m_queue_back.load();
6      next = back->next.load();
7      if (back == m_queue_back.load()) {
8        if (next != nullptr) {
9          m_queue_back.compare_exchange_weak(nullptr, next);
10       } else {
11         if (back->next.compare_exchange_weak(next, node)) {
12           break;
13         }
14       }
15     }
16   }
17   m_queue_back.compare_exchange_weak(back, node);
18 }
```

Michael-Scott Queue: Implementation

Extra Slide

```
std::optional<Job> dequeue() {
  while (true) {
    Node* front = m_queue_front.load();
    Node* back = m_queue_back.load();
    Node* next = front->next.load();
    if (front == m_queue_front.load()) {
      if (front == back) {
        if (next == nullptr) return std::nullopt;
        m_queue_back.compare_exchange_weak(back, next);
      } else {
        Job job = front->job;
        if (m_queue_front.compare_exchange_weak(front, next)) {
          return job;
        }
      }
    }
  }
}
```

Michael-Scott Queue: Notes

Extra Slide

- Loosened invariant for `m_queue_back`.
  - Required to be reachable from `m_queue_front` via the linked list.
  - No longer required to point to the last element of the queue.
- Introduced in the paper Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms (1996).
- It was only much later that this property is formally verified, using *trace reduction*, in the paper Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis (2019).

That's it!

Anonymous Feedback (throughout the semester):

https://forms.gle/6a9T4t88wNwYxG4G8

The link to the tutorial slides will be posted in telegram.