KAIST CS431: Concurrent Programming

Instructor: Jeehoon Kang (https://cp.kaist.ac.kr)

Logistics

- Homepage: https://github.com/kaist-cp/cs431
  - Read README.md carefully!
  - Announcement and question in the issue tracker (please watch the repo)
  - Office hours: Friday 9:15-10:15am

- Homework and attendance: https://gg.kaist.ac.kr/16

- Honor code: sign the KAIST CS honor code.

- Grading
  - Homework & project: 60%
  - Midterm & final exams: 40%
  - Attendance: ?

Large Language Model Policy

- Large language models (LLMs): [ChatGPT](), …
  - We assume all of us can use ChatGPT 3.5.

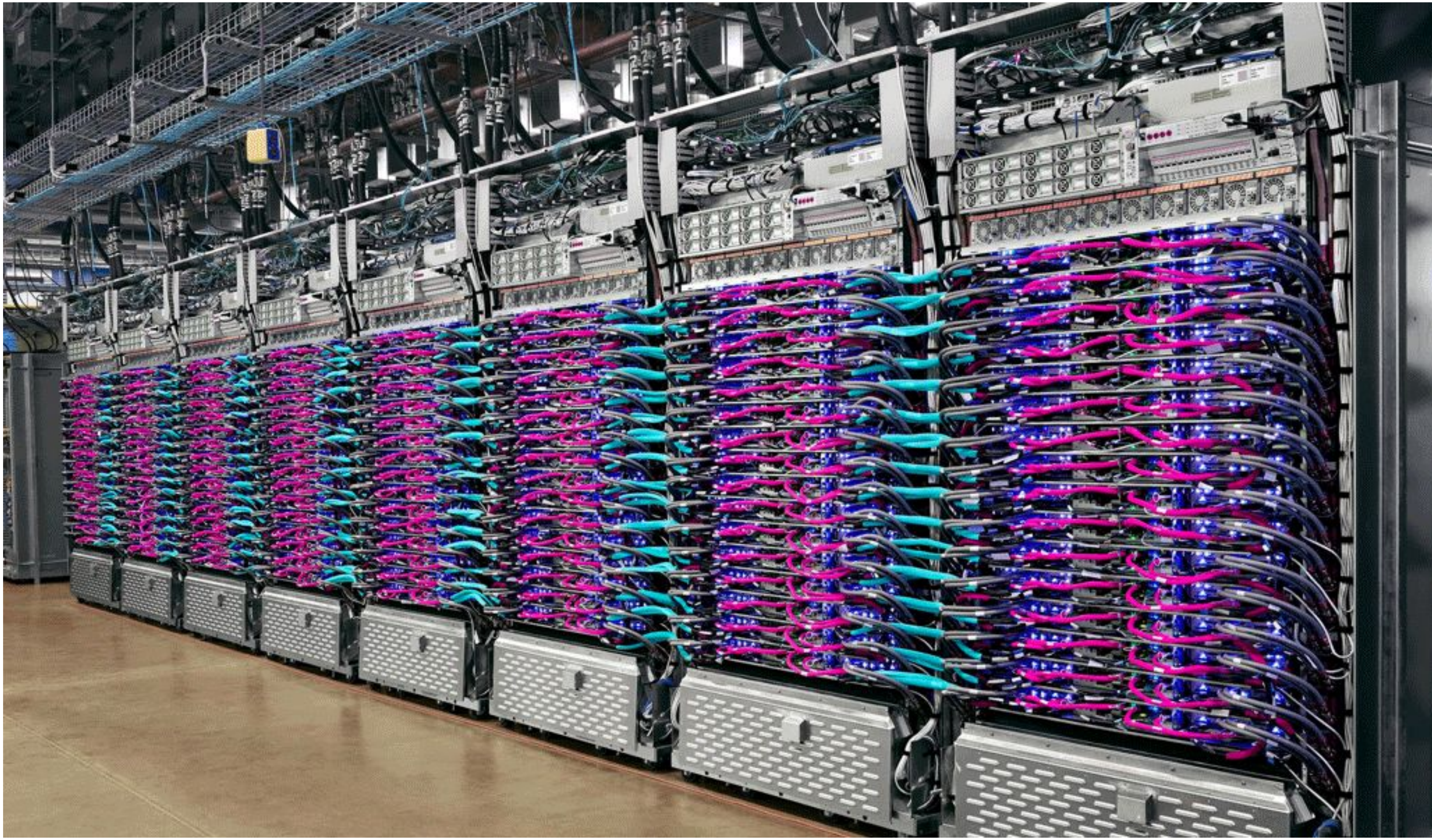
- You can use LLMs for homework, study, …
  - ChatGPT 4.0 doesn't help much for homework.
  - ChatGPT 3.5 helps much for studying CS431 materials.

- You cannot use LLMs for exams.

- We'll survey on your experience of LLMs for this course.

Introduction


Google DeepMind
Challenge Match
8 - 15 March 2016
AlphaGo
ALPHAGO
02:00:00
LEE SEDOL
01:59:42
AlphaGo
Lee Sedol



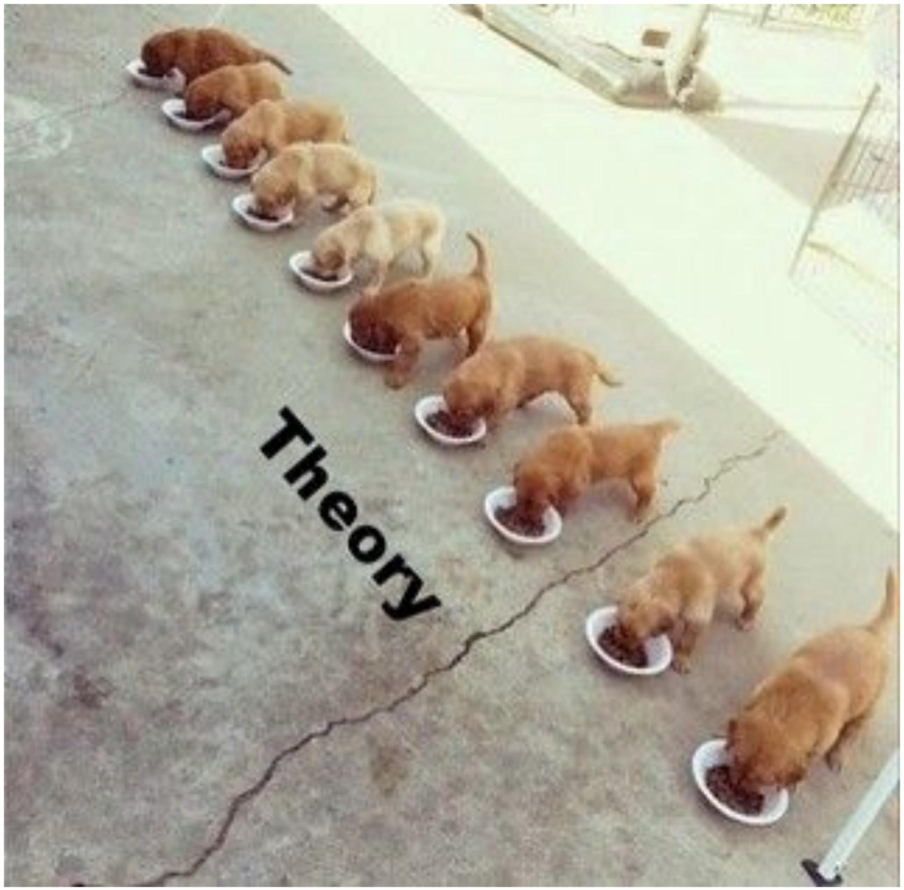
The era of parallelism

- **Context: we need to do more and more computations**
  - Especially so in the era of AI/IoT
- **Trend: computers become more and more parallel**
  - Parallel resources: CPUs, memory, I/O, …

- **Reason 1 (2005): the end of Dennard scaling**
  - Cannot increase the frequency of the circuits
  - Breakthrough: multi-core systems
- **Reason 2 (2018): the end of Moore's law**
  - Cannot increase # of transistors for fixed area
  - Breakthrough: accelerators (specialized, extremely parallel H/W)

- **How to coordinate parallel resources to achieve higher performance?**

Parallel computing, theory and practice



Concurrency: synchronizing parallel resources

- **Parallelism:** multiple resources

- **Concurrency: shared mutable resources (states)**
  - E.g. CPU, GPU, memory, server, database, datacenter, ...
  - Parallelism : concurrency = 찐빵 : 팥소
    - Shared immutable resources: constant
    - Exclusive mutable resources: sequential

- **Example 1: lock-protected inode**
  - Inode: file system metadata
  - Serializes file accesses of multiple threads

- **Example 2: lock-free hash table**
  - Ensures correctness of simultaneous reads/writes of multiple threads

Challenge in concurrency: nondeterminism

- **Challenge: combinatorially explosive nondeterminism**
- **Source 1: interleaving**
  - E.g., “X=1 || X=2”: the end memory depends on the order of execution
- **Source 2: optimization by hardware/compiler**
  - E.g., a=b=0 is possible in modern architecture by reordering:

    X=1 || Y=1

    a=Y || b=X
- **We need to tame the nondeterminism**

Approaches to taming nondeterminism

- **Enclosing nondeterminism within safe API**
  - Hiding too low-level nondeterminism (e.g., interleaving of instructions)
  - While exposing high-level one (e.g., interleaving of queue operations)
  - Most people need to understand API, not implementation
  - E.g. safe API of locks, conditional variables, concurrent data structures

- **Reasoning with synchronization patterns in implementation**
  - Someone needs to implement lock, condvars, data structures, …
  - Use only well-studied synchronization patterns

- **This course: learning API and implementation of concurrency libraries**

Two modes of concurrency: “easy” and “difficult”

- **“Easy” lock-based concurrency**
  - Locks, conditional variables, …
  - Pros & cons: simplicity & low scalability (losing parallelism opportunities)
  - Applicability: covering the majority🤔 of use cases (in terms of # of lines)

- **“Difficult” lock-free concurrency**
  - **Theory:** semantics and reasoning principles
    (characterizing the nondeterminism)
  - **Tools:** synchronization patterns
    (building block for lock-free concurrency)
  - **Practice:** API and implementation of lock-free data structures
    (e.g., stack, queue, list, hash table, radix tree, balanced tree)

General advices for concurrent programming

- **“Easy” concurrency is the first to study**
  - Go for “difficult” concurrency only when it’s a bottleneck
  - “Premature optimization is the root of all evil”

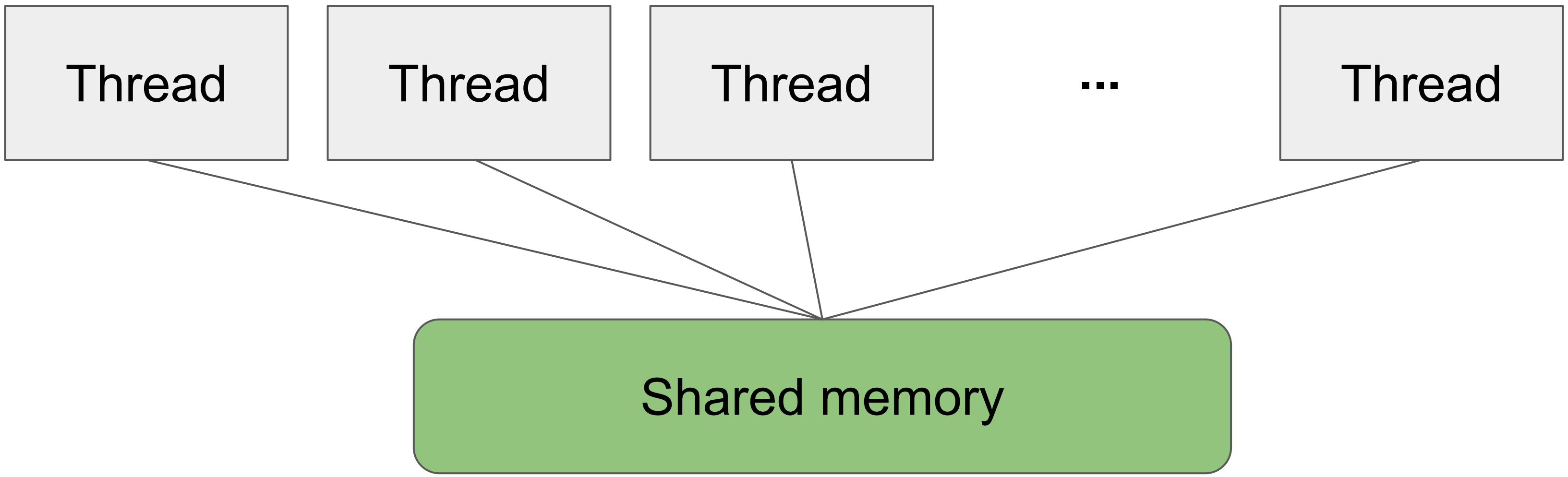
- **“Difficult” concurrency is not that difficult once you understand theory**
  - In other words, understanding theory may be difficult…
  - The key is taming the right amount of nondeterminism
    (Too much => scalability problem, too less => correctness problem)

- **Not so much to code, too much to debug**
  - Use sanitizer, stress testing, assert, line-by-line debugging, …
  - Mathematical and PL thinking helps a lot

Lock-based concurrency

Part 1: motivation for safe API

Context: shared-memory concurrency



- **Thread:** agent of execution reading/writing to shared memory
- **Shared memory:** shared storage of data

- **Other kinds of concurrency:** among CPU, memory, GPU/FPGA, persistent memory, distributed nodes, ...

Lock-based shared-memory concurrency

- **Definition:** at any moment, a location is accessed by a single agent
- **Pros & cons:** simple & possibly inefficient
- **Mechanism:** locks (at any moment, only one thread holds the lock)

- **Examples**
  - r1=X || r2=X
    X=r1+1 || X=r2+1
    // Not always X=2: unfortunate interleaving produces X=1

  - **L.acquire()** || **L.acquire()**
    r1=X || r2=X
    X=r1+1 || X=r2+1
    L.release() || **L.release()**
    // Now always X=2: lock prevents unfortunate interleaving

Why are locks “easy”?

- Recall: concurrency’s challenge is nondeterminism
  - Thread interleaving
  - Instruction reordering

- Lock constrains thread interleaving to acquire/release points
  - No interleaving between acquire and release

- Lock removes instruction reordering
  - Thread A’s release strictly happens before Thread B’s acquire
  - Threads are executed **AS IF** they are the same thread.

- **Lock reduces nondeterminism to the minimum.**

Lock-based concurrency’s low-level API

- **Lock.acquire():** Blocks until acquiring the lock.
- **Lock.try_acquire():** Returns whether a lock is acquired. Doesn’t block.
- **Lock.release():** Releases the acquired lock.

- **Challenge:** the API is extremely error-prone.
  - **Relating lock and resource:** users should access X only when L is held.
  - **Matching acquire/release:** users should release only acquired locks.

- **Consequence:** the API incurs high cost.
  - **Attention:** programmers should always be concerned with the API.
  - **Potential bugs:** there are typically many bugs that remain.

Lock-based concurrency's high-level API

- **We want an easy-to-use, always-safe high-level API.**
  - Acquire/release are automatically matched.
  - Lock and resource are explicitly related.

- **Benefits of high-level API:** low cost with less attention and bugs
  - **Less attention:** Programmers don't need to worry about API misuses.
  - **Less bugs:** It's highly unlikely that a program using the API is buggy.

- **Design of high-level API (from C++/RAII)**
  - **LockGuard:** automatically releasing a lock using an RAII type
    - Lock.acquire() returns "lock guard"
    - When a lock guard is destructed, the corresponding lock is released.
  - **Lock<T> (= (Lock, T)):** relating a lock and a resource with a new type

Lock-based concurrency's safe(ish) API: lock guard

- **Lock guard: holding a lock**
  - https://en.cppreference.com/w/cpp/thread/lock_guard
  - ```
    #include <thread>
    #include <mutex>
    #include <iostream>

    int g_i = 0;
    std::mutex g_i_mutex;  // protects g_i

    void safe_increment()
    {
        const std::lock_guard<std::mutex> lock(g_i_mutex);
        ++g_i;

        std::cout << std::this_thread::get_id() << ": " << g_i << '\n';

        // g_i_mutex is automatically released when lock
        // goes out of scope
    }

    int main()
    {
        std::cout << "main: " << g_i << '\n';

        std::thread t1(safe_increment);
        std::thread t2(safe_increment);

        t1.join();
        t2.join();

        std::cout << "main: " << g_i << '\n';
    }
    ```
    ```

Lock-based concurrency’s safe(ish) API: locked data

- **Locked data: a pair of lock and data**
- **Benefit: API mandates the inner data is (quite) safely protected by a lock**

- **// type**
  template<typename T> class Lock<T> { RawLock lock; T data; }
- **// acquire and create lock guard**
  LockGuard<T> Lock<T>::lock(this) {
      this->lock.acquire(); LockGuard { this }
  }
- **// dereference data from lock guard**
  &T LockGuard<T>::operator->(this) { &this->0.data }
- **// release automatically when guard is dropped**
  LockGuard<T>::~LockGuard() { this->0.lock.release(); }

Lock-based concurrency's safe(ish) API: **not safe!**

- // data: Lock<int>
  auto data_guard = data.lock();
  auto data_ptr = (int *) &data_guard;
  …
  // data_guard is dropped, lock is released
  ***data_ptr = 666; // UNSAFE!**

- **Root cause:** data_ptr should not outlive data_guard
- **Error-prone:** happening in the production code, causing lots of troubles

- **Solution: Rust's type system** based on **ownership and lifetime**
  - https://github.com/kaist-cp/cs431/blob/main/src/lock/api.rs
  - Rust implementation of lock has proven-safe API
  - => Let's first study Rust and then resume studying concurrency

Lock-based concurrency

Part 2: foundation for safe API in Rust

https://docs.google.com/presentation/d/1LbiQ1Z3FTjp1144GRwEj3EPNj-RspAthlsq3a0PCQHw/edit#slide=id.p

Rust: **safe** systems programming language

- **Motivation:** achieving **safety & control** at the same time
  - Safety: compiled programs don't go wrong
  - Control: language supports low-level features
  - Prior art: C/C++ (unsafe)

- **Best fit for this course:** ownership and lifetime captures
  the essence of concurrency

- **Reading assignments:**
  - Read the book. Homework 1 is about the book's final project.
  - Read Rust by example.
- **Programming assignments will be in Rust**
  - Set up programing environment on the provided server.

Rust example 1: iteration invalidation (1/3)

```
fn main() {
    let v = vec![1, 2, 3];

    let p = &v[1];
    v.push(4);
    println!("v[1]: {}", *p);
}
```

- https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=08f5870c40f7afdfd7a2fab9d7815f9f

- This code would be compiled in C++, but it may fail at runtime
    - "v.push(4)" may relocate "v", Invalidating "p"

- This code is not compiled in Rust
    - "cannot borrow `v` as mutable because it is also borrowed as immutable"

Rust example 1: iteration invalidation (2/3)

```
fn main() {
    let mut v = vec![1, 2, 3];

    let p = &v[1];
    v.push(4);
    println!("v[1]: {}", *p);
}
```

- “v”: the owner of the vector
- “p”: immutably borrowing it
  from “let p = …” to “println!(...)”
- “v.push(4)”: mutably borrowing it for the line

- Compile fails because type checker detects shared mutable accesses (SMA) to the vector
  - “p” and “v.push”
- It’s precisely the reason it may go wrong

- **Q: how to detect SMA’s?**

Rust example 1: iteration invalidation (3/3)

```
fn main() {
    let v = vec![1, 2, 3];

    let p = &v[1];
    v.push(4);
    println!("v[1]: {}", *p);
}
```

- Calculate each owner/borrower's "**lifetime**"
  - "v": L1-L5
  - "p": L3-L5
  - "v.push": L4
- List up the pairs of **overlapping lifetimes**
  - "v" and "p" (L3-L5), "v" and "v.push" (L4)
  - "p" and "v.push" (L4)
- Remove pairs of borrowee & borrower
  - "p" and "v.push" (L4) remained
- Remove pairs of immutable borrows

- The remaining pairs are **regarded as SMA's**
- **Statically sound:** detecting all SMA's at compile time
- Incomplete: not all pairs are actually SMA's

Ownership for analyzing shared mutable accesses

- **“Ownership”: the ability (of an agent) to access and destroy a resource**
  - **Exclusive**: if I own a resource, no one else can own it
  - **Borrowable:** *mutably* borrowed by single agent or *immutably* by multiple
  - **Fit for concurrency:** each thread is an agent

- **Enforcing discipline:** no shared mutable accesses to a resource (by default)
  - **Static**: ownership discipline is enforced by types
  - **Easy to use:** compilers will report every violation of the discipline
  - **Correct:** if type-checked, a program doesn’t go wrong

- **Bending the discipline** with “*interior mutability*”
  - **Necessary:** shared mutable accesses are inevitable in concurrency
  - **Modular:** enveloping implementation within safe API
    (“as if” there are no shared mutable accesses)

Rust example 2: RefCell (1/4)

- **Context:** it is unrealistic to forbid SMA's altogether for, e.g., concurrency

- **Solution:** interior mutability
  - Enveloping SMA's in a safe API **as if** there are no SMAs

- **Example:** RefCell<T>
  - Checking ownership at runtime (not compile time)
  - RefCell<T>::try_borrow(), RefCell<T>::try_borrow_mut():
    trying to borrow the inner value, immutably or mutably (resp.)
  - https://doc.rust-lang.org/stable/std/cell/struct.RefCell.html
    https://doc.rust-lang.org/book/ch15-05-interior-mutability.html

Rust example 2: RefCell (2/4)

```
fn f1() -> bool { true }
fn f2() -> bool { !f1() }

fn main() {
        let mut v1 = 42;
        let mut v2 = 666;

        let p1 = if f1() { &v1 } else { &v2 };

        if f2() {
                let p2 = &mut v1;
                *p2 = 37;
                println!("p2: {}", *p2);
        }

        println!("p1: {}", *p1);
}
```

- https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=c07efb0ed16980ef85d09568382114f9

- Suppose f1() and f2() are complex and yet exclusive conditions (not f1() && f2())
- Safe because p1 and p2 are not aliased

- Compile error because
  the type checker cannot deduce the safety due to the complexity of conditions
- “cannot borrow `v1` as mutable because it is also borrowed as immutable”

Rust example 2: RefCell (3/4)

```
use std::cell::RefCell;

fn f1() -> bool { true }
fn f2() -> bool { !f1() }

fn main() {
        let v1 = RefCell::new(42);
        let v2 = RefCell::new(666);

        let p1 = if f1() { &v1 } else { &v2 }
                .try_borrow().unwrap();

        if f2() {
                let mut p2 = v1
                        .try_borrow_mut().unwrap();
                *p2 = 37;
                println!("p2: {}", *p2);
        }

        println!("p1: {}", *p1);
}
```

- https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=00e64c84fc7b31b5080c4d386add8e9e

- Ownership is checked at runtime (try_borrow(), try_borrow_mut())
- Compiled and executed as expected "p1: 42"

- If f1() && f2(), try_borrow_mut() fails at runtime (not compile time)
- "thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value"

Rust example 2: RefCell (4/4)

- **Interior mutability:** encapsulating SMA's in a non-SMA type

- **Safe API:** virtually w/o SMA's
  - pub fn try_borrow_mut(**&self**) -> Result<RefMut<T>, BorrowMutError> (**immutably** borrowing self)

- **Potentially unsafe implementation:** w/ SMA's
  - ... **unsafe { &mut *self.value.get() }**, … (https://doc.rust-lang.org/1.63.0/src/core/cell.rs.html#1732)
  - **"Unsafe"**: bridge between API w/o SMA's and impl. w/ SMA's (needs manual inspection, should be explicitly annotated)

Rust example 3: Lock

- https://github.com/kaist-cp/cs431/blob/main/src/lock/api.rs#L105

```rust
impl<'s, L: RawLock, T> Deref for LockGuard<'s, L, T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        unsafe { &*self.lock.data.get() }
    }
}
```

- // data: Lock<int>
  let data_guard = data.lock();
  let data_ref = data_guard.deref();
  …
  drop(data_guard); // lock is released
  ***data_ref = 666; // NOT COMPILED: deref target shall not outlive guard**

Summary of Rust’s ownership type

- **Motivation:** achieving **safety & control** over shared mutable resources

- **Key ideas:**
  - **Discipline:** disallowing shared mutable accesses by default
  - **Interior mutability:** allowing them in a controlled way

- **Benefits:**
  - Statically analyzing the safety of shared mutable accesses (both for sequential and concurrent programs)
  - Explicitly marking those code that needs manual inspection

- **What we’ll do:** understanding lock-based concurrency with safe API

Lock-based concurrency

Part 3: safe API

Rust concurrency libraries (1/3)

- **Potentially-unsafe implementations are enveloped within safe API**
  - If libraries are correct, the users don't need to worry about safety at all
- **Rust std**
  - Thread: agent of execution
    - Safety: 'static closure (not function pointer), typed join handle
  - Scoped thread: restricting thread's lifetime within a scope
    - Motivation: safe sharing of non-'static data
    - Safety: thread should be joined before the scope ('s) ends
  - Arc: reference counter, immutably sharing data among multiple threads
    - Safety: Deref, not DerefMut
  - Send: transferable to other thread
    - **Implementers:** usize, &usize, Arc<T>, &Arc<T> (but not Rc<T>, &Rc<T>)
  - Sync: concurrently accessible from multiple threads
    - **Implementers:** usize, Arc<T> (but not Rc<T>)
    - **Property:** `T: Sync` if and only if &T: Send

Rust concurrency libraries (2/3)

- CS431 Lock API: a safe API for lock (see below for implementations)

- **Lock<L: RawLock, T>:** owns T that is protected by an L lock
  - Guarantee: the T object is not concurrently accessed (not code region)
  - Examples: Lock<SpinLock, Vec<usize>>, Lock<ClhLock, &'t TLS>
  - Property: Send + Sync if T is Send (i.e., meaningful only if T is Send)

- **LockGuard<'s, L: RawLock, T>:** proves the lock is acquired
  - Guarantee: the lock is held, T is accessible w/ Deref/DerefMut
  - RAII: it releases the lock when dropped
  - Property: Send if T is Send, Sync if T is Sync (i.e., transparent accessor)

- The API's guarantees/safety are proven w.r.t. Rust's ownership type system (as opposed to C/C++)

Rust concurrency libraries (3/3)

- **More std**
  - Mutex: mutual exclusion w/ various strategies
  - Condvar: conditional variable, waiting for an event (condition)
    - Safety: Condvar::wait() gets `&mut MutexGuard`, forbidding reuse of protected data
  - RwLock: reader-writer lock, allowing multiple readers **OR** one writer

- **crossbeam**
  - Channel: sending/receiving values among threads
  - CachePadded: align with 128 bytes
    - Motivation: to defeat "false sharing"

- **rayon**
  - into_par_iter: executing a function for each element in parallel
    - Motivation: parallelism made easy

Lock-based concurrency

Part 4: implementation

Several lock implementations

- **Implementations:** https://github.com/kaist-cp/cs431/tree/main/src/lock
  - Spinlock, ticket lock, CLH lock, MCS lock, MCS parking lock

- **RawLock trait:** defines the API of the "raw" lock
  - **APIs:** lock(), unlock(), Token, Default, Send, Sync
  - **Guarantee:** only one agent acquires the lock at a time
  - **Implementers:** spinlock, ticket lock, CLH lock, ...

- **Tradeoffs among locks**
  - Simple, fast (when uncontended), compact, scalable, fair, energy-efficient, …

Spinlock implementation

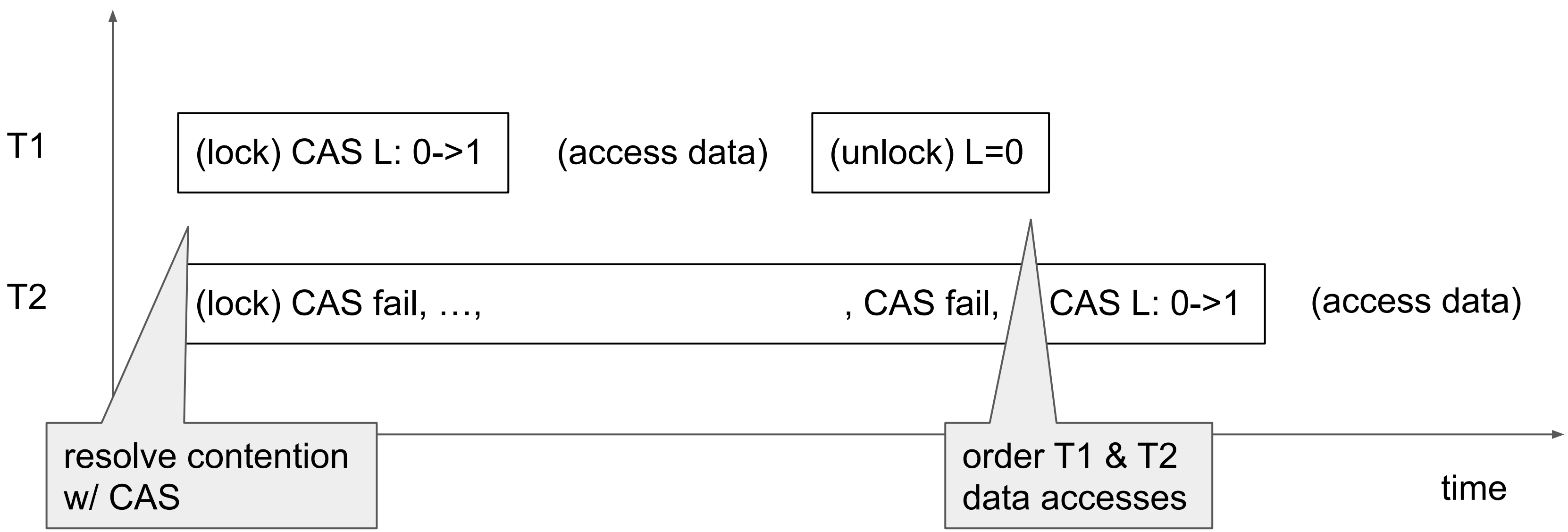
- https://github.com/kaist-cp/cs431/blob/main/src/lock/spinlock.rs
  (for now, ignore memory orderings: acquire, release, …)

- ```
  pub struct RawSpinLock {
      inner: AtomicBool, // true means locked, false means unlocked
  }
  pub fn lock(&self) {
      while self.inner.compare_and_swap(false, true).is_err() {} // rmw
  }
  pub fn unlock(&self) {
      self.inner.store(false); // not rmw, thanks to exclusiveness of lock
  }
  ```
  ```

Spinlock correctness

- pub fn lock(&self)   { while self.inner.cas(false, true, acquire).is_err() {} }
  pub fn unlock(&self) { self.inner.store(false, release); }

- If a lock has already been acquired, lock() will spin.
- Only one thread can hold a lock at a time (see below).



The key ideas of the other locks

- https://github.com/kaist-cp/cs431/tree/main/src/lock

- **Guaranteeing mutual exclusion w/ CAS**
  - Ordering from the end of a critical section to the beginning of another
  - **`curr`** in ticket lock, **a new location** for each waiter in CLH/MCS lock

- **Guaranteeing fairness by ordering & waiting w/ different locations**
  - Ordering w/ fair instructions (e.g. swap, fetch-and-add)
  - Ticket lock: ordering w/ **`next`** and waiting w/ **`curr`**
  - CLH/MCS lock: ordering w/ **`tail`** and waiting w/ **a new location**

- **Homework: reasoning the correctness of locks**

Lock tradeoffs

- **Ticket lock:** guaranteeing fairness by ticket queueing
  - Lock order is decided by a fair instruction (fetch-add or swap) beforehand
  - Cons: a slightly complicated API (returning ticket)
- **CLH lock:** improving scalability by using per-critical section spinning location
  - Queue of spinning locations
  - Cons: O(n) space overhead, where n is the number of critical sections
- **MCS lock:** awaring NUMA by spinning on self-allocated location
  - Cons: possibly an additional compare-and-swap in unlock()
- **MCS parking lock:** reducing energy consumption by thread parking
  - Thread parking (intentionally blocking) instead of spinning
  - Cons: sacrificing performance in modest contention cases
- A paper on performance evaluation

Lock questions

- **In ticket lock, why issuing a new ticket can be relaxed?**
  - In CLH lock, why swapping the tail can be relaxed?
  - In MCS (parking) lock, why swapping the tail can be relaxed?

- **In MCS parking lock, after unparked, why do we check if it's unlocked?**
- **In MCS parking lock, why is `thread` cloned?**

- **What else is in the literature?**
  - Reentrant lock: calling lock()s in a nested fashion
  - Reader-writer lock: allowing multiple readers or a single writer
  - Hierarchical lock: combining requests in a single NUMA node
  - Backoff strategy: spinning? yielding? parking? exponential backoff?
  - Conditional variable: guaranteeing order in addition to mutual exclusion
  - https://docs.rs/parking_lot/0.11.0/parking_lot/ (Rust impl of locks)
  - ...

Lock-based concurrency

Part 5: fine-grained locking

Motivation: coarse-grained locking is unscalable

- **Coarse-grained locking:** protecting a large object w/ a single lock
  - Easy: the straightforward path towards concurrent programming
  - Unscalable: all accesses to the object are serialized w/o parallelism.

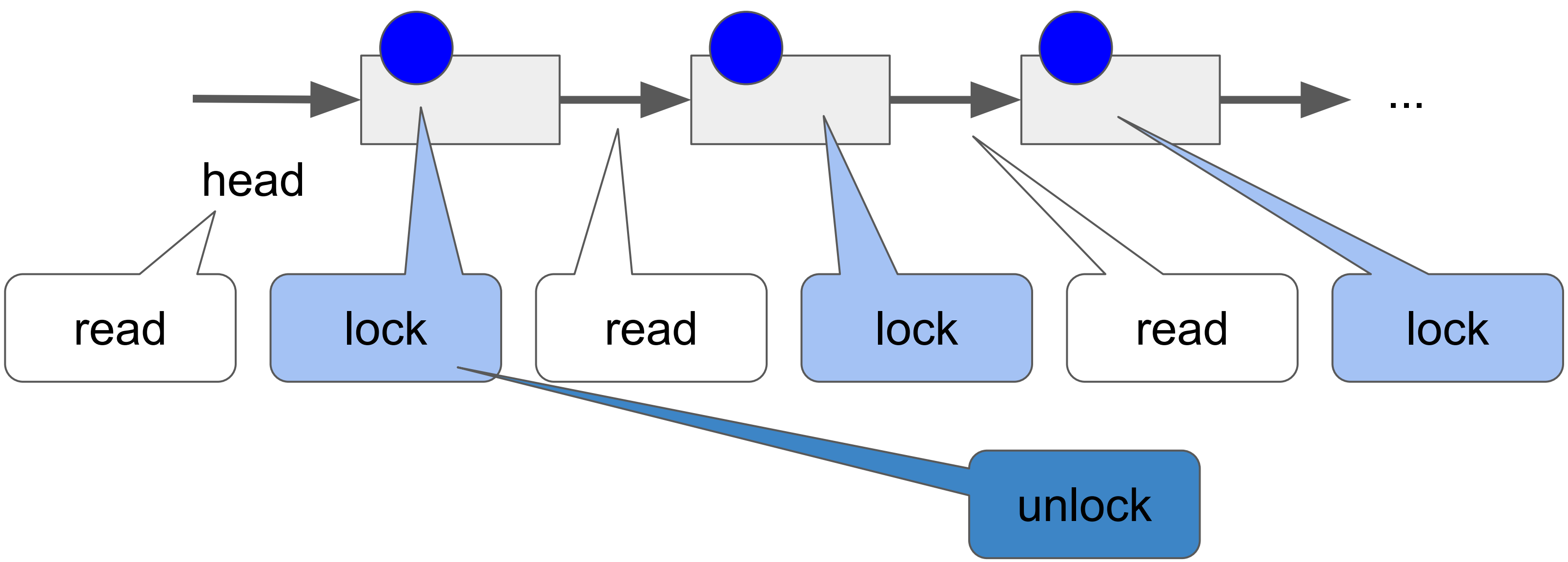
- **Fine-grained locking:** protecting many small objects w/ separate locks
  - More scalable: accesses are “distributed” to multiple locks
    (ideally, minimal synchronization overhead)

- **Drawbacks** of fine-grained locking (compared to coarse-grained one)
  - Single-thread overhead (ideally, only modest)
  - Complexity

- **Key design questions**
  - What should be done in sequence? -> lock required
  - What can be done in parallel? -> lock not required

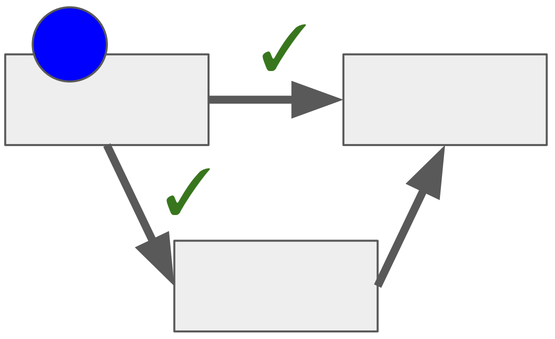
Example: lock-coupled linked list (structure)

- Each node has a lock
- When accessing the “next” pointer, lock should be held
  - A node’s lock protects the node’s next pointer
- Acquire the next node’s lock before releasing the current’s (“hand-over-hand”)
  - Make sure the current node is not detached

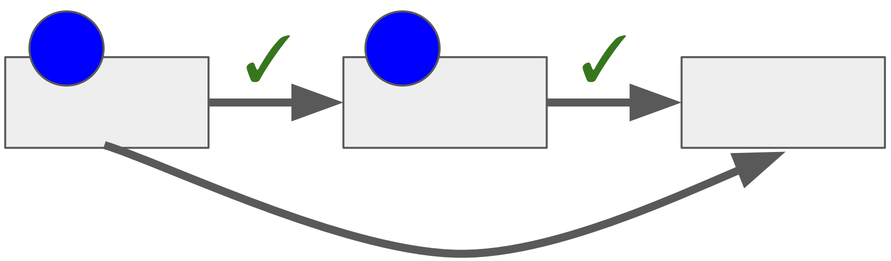


Example: lock-coupled linked list (operations)

- **Insert B between A and C**
    - Acquire A’s lock
    - Read A.next (=C)
    - Allocate B with B.next=C
    - Write A.next = B



- **Remove B between A and C**
    - Acquire A’s lock
    - Read A.next (=B)
    - Acquire B’s lock
        - To ensure B.next is in the list
    - Read B.next (=C)
    - Write A.next=C
    - (Release B’s lock and) free B



Lock-based concurrency

Part 5: managing multiple locks w/ BoC

Deadlock and livelock bugs

- **Deadlock:** each thread is blocked by another
  - e.g., A holding L1, acquiring L2
    B holding L2, acquiring L1
  - Countermeasure: detecting deadlock and killing an operation(transaction)

- **Livelock:** operations keeps getting killed without meaningful progress
  - A: L1 -> L2 -> L3
    B: L2 -> L3 -> L1
    C: L3 -> L1 -> L2
  - B holding L2, C holding L3/L1 -> C killed ->
    A holding L1, B holding L2/L3 -> B killed ->
    C holding L3, A holding L1/L2 -> A killed -> ...

How to fight with them?

Deadlock and livelock avoidance

- Order locks: L1, L2, L3
  - e.g., according to their pointer address

- Acquire locks in the order
  - e.g., A holding L1, acquiring L2
    B holding L2**, acquiring L1 => NO!!!**

  - A: L1 -> L2 -> L3
    B: L2 -> L3 **-> L1 => NO!!!**
    C: **L3 -> L1** -> L2 **=> NO!!!**

- Theorem: no deadlock/livelock happens
  - Proof sketch: the thread holding the biggest lock will go unblocked

BoC: Behavior-oriented Concurrency (Basics)

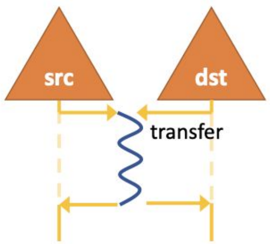
- https://dl.acm.org/doi/10.1145/3622852

- Motivations
  - Ordering locks automatically
  - Coupling lock-based synchronization and thread-based parallelization
    (cf: coupling synchronization and permission in high-level lock API)
    => Natural representation of task dependencies

- Example

List. 5. Spawn a behaviour that requires both accounts

```
transfer(src: cown[Account], dst: cown[Account], amount: U64) {
  when (src, dst) { // withdraw and deposit
    if (src.balance >= amount && !src.frozen && !dst.frozen) {
      src.balance -= amount;
      dst.balance += amount;
} } }
```

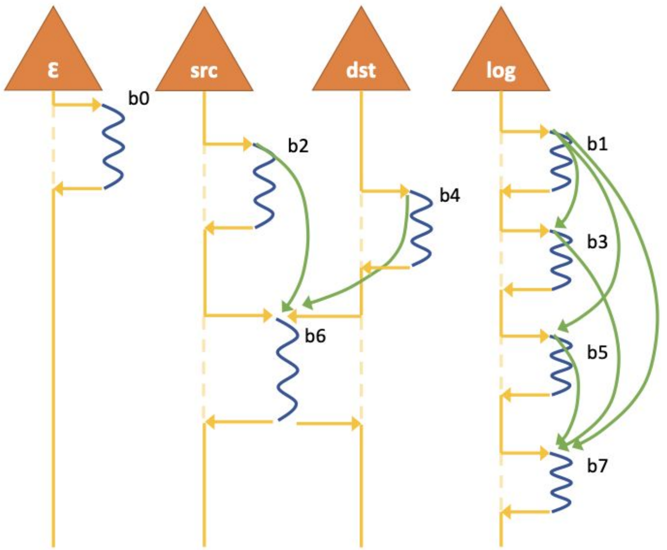


BoC: Behavior-oriented Concurrency (Semantics)

- A behavior is **queued** for the (multiple) Cowns it uses
- Queueing guarantees inter-behavior dependencies and mutual exclusion

List. 8. Creating an accurate log

```
main(src: cown[Account], dst: cown[Account],
    log: cown[OutStream]) { /* b0 */
 when(log) { /* b1 */ log.log("begin") }
 when(src) { /* b2 */ ...
   when(log) { /* b3 */ log.log("deposit") }
 }
 when(dst) { /* b4 */ ...
   when(log) { /* b5 */ log.log("freeze") }
 }
 when(src, dst) { /* b6 */ ...
   when(log) { /* b7 */ log.log("transfer") }
} }
```



BoC: Behavior-oriented Concurrency (Impl)

- Mutual exclusion for each Cown (similar to MCS lock's queueing)
- Each behavior requests Cowns in a fixed order (e.g., increasing order of addr)
- Each behavior acquires multiple cowns **atomically** by "2-phase enqueuing"
    - **Start phase** for all Cowns => **finish phase** for all Cowns
    - A behavior exclusively accesses a cown from start to finish phases
- Without 2-phase enqueuing, there can be a deadlock…
    - b1, b2 queued for c1, c2
    - c1 queue: b1, b2; c2 queue: b2, b1
    - **=> DEADLOCK!**

But what if we want even better performance?

- **Performance drawbacks of fine-grained locking**
  - Write operations have heavy synchronization costs due to lock ops
  - Read operations also writes to lock leading to cache invalidation

- **Alternative 1: advanced locks for read-mostly workloads**
  - reader-writer lock (“Rust concurrency libraries”),
    optimistic locking (not in semester), … (still unscalable for writers)

- **Alternative 2: “lock-free” concurrent data structures**
  - Pros: even better performance
    - Write operations have lightweight synchronization costs
    - Read operations perform just reads w/o cache invalidation
  - Cons: *much* (×100) more difficult than lock-based concurrency…

MIDTERM EXAM

Everything you’ve learned so far

Lock-free concurrency

Part 1: definition

Lock freedom is not about the existence of locks

- **What the hell?**
  - It was, but it isn't: the word's meaning has evolved over time.

- **Lock's limitations:** no guarantee of progress
  - Q: What if a thread holds a lock and sleeps forever?
    A: All threads waiting for the same lock stops making progress.

- **Lock freedom:** a progress guarantee
  - Definition: one of the ongoing operation is eventually completed
  - Intuition: the whole system's progress is guaranteed
  - Examples: Treiber stack, Michael-Scott queue, circular buffer,
    work-stealing deque, …

- Enemies: thread stall due to I/O, crash, non-preemptive scheduling;
  locks (deadlock and livelock)

Other progress guarantees

- **Obstruction freedom:** weaker than lock freedom
  - Definition: if only a single operation is run, it is eventually completed
  - Intuition: we can always “recover” an object to a stable state

- **Wait freedom:** stronger than lock freedom
  - Definition: every ongoing operation is eventually completed
  - Intuition: every thread’s progress is guaranteed
  - Examples: Yang-Mellor-Crummey queue, ...

- Wait-free ⊆ lock-free ⊆ obstruction-free ⊆ “nonblocking”
- Reference: Herlihy and Shavit. On the Nature of Progress. OPODIS'11

Key idea for lock freedom: single-instruction commit

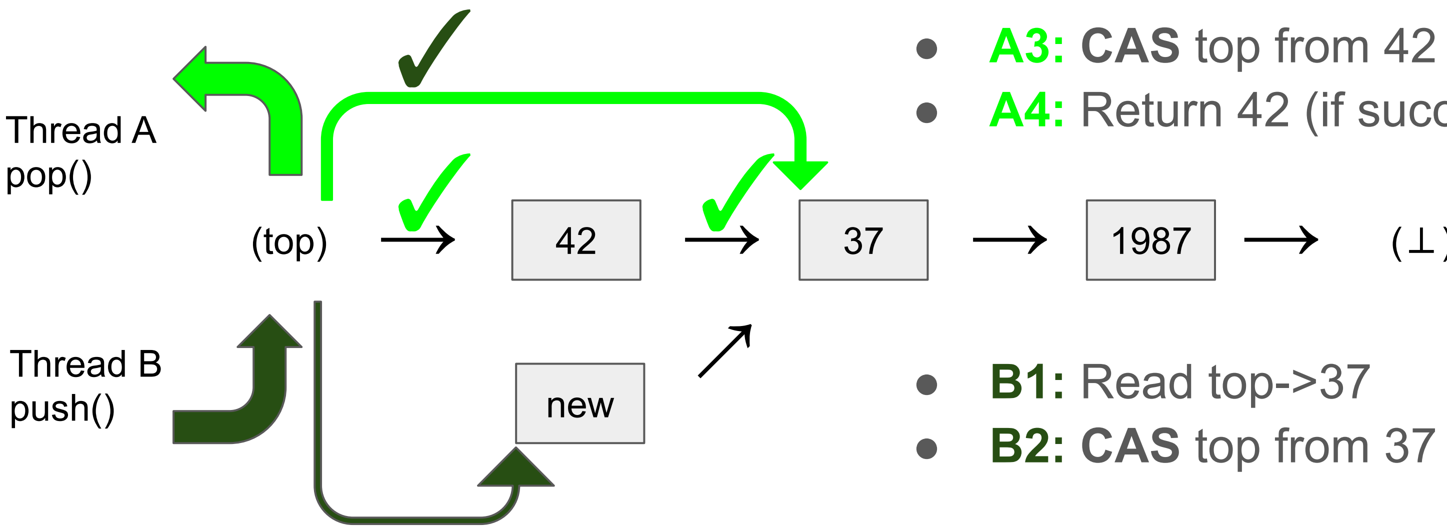
- **Leveraging architectural guarantee of lock freedom**
  - At any moment, at least one CPU core executes its instruction.
- **Designating an RMW instruction as an operation's commit point**
  - Definition: atomically reading from & writing to a location (e.g., CAS)
  - Property: capable of expressing synchronization protocols (e.g., spinlock)
  - History: single-instruction commit is RMW's motivation!
- **Defeating the "enemies" of lock freedom**
  - Thread pause/stall doesn't bother with a single RMW instruction.
  - Deadlock/livelock doesn't happen due to the absence of locks.
- **Side benefit: scalability** (reducing contention to a single instruction)

Lock-free concurrency

Part 2: data structures

Example: Treiber's stack

- **Singly linked list** w/ list head = stack top
- https://github.com/kaist-cp/cs431/blob/main/src/lockfree/stack.rs



- **A1:** Read top->42
- **A2:** Read 42->37
- **A3: CAS** top from 42 to 37
- **A4:** Return 42 (if successful)

- **B1:** Read top->37
- **B2: CAS** top from 37 to new

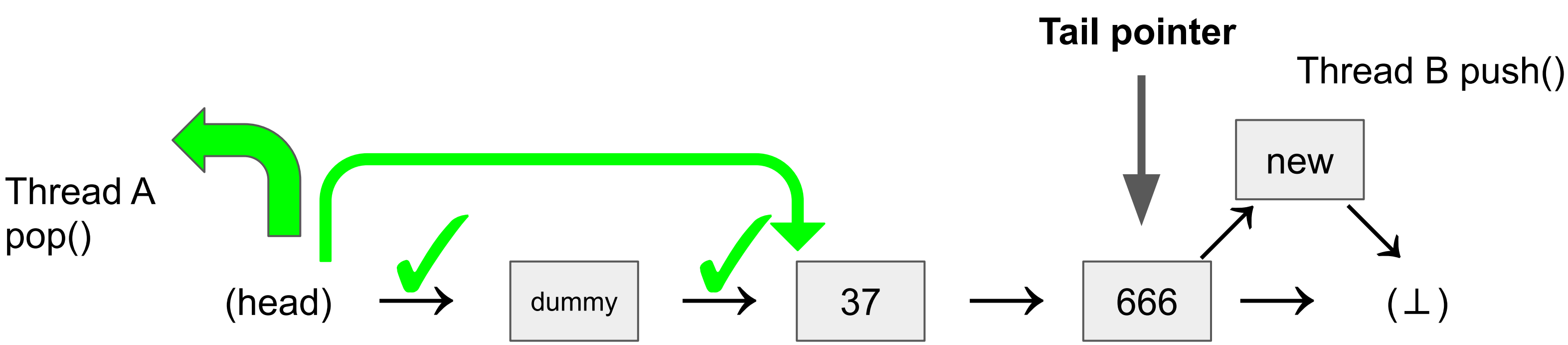
Push/pop synchronizes with CAS on the top location

Questions for Treiber’s stack

- Why data: ManuallyDrop<T> in Node<T>?
  - Because data’s ownership is returned by pop(),
    while Node<T> will be deallocated later
  - Common in lock-free data structures

Example: Michael-Scott's queue

- **Singly linked list** w/ list head/tail = queue pop/push ends (1 dummy at head)
- **Tail pointer:** push() doesn't need to traverse from the beginning
- https://github.com/kaist-cp/cs431/blob/main/src/lockfree/queue.rs
- **Pop(): CAS** on head (the same with Treiber's stack pop())
- **Push(): CAS** on tail from null to new



- **Challenge:** tail can be stale (e.g. right after pushing)
- **Solution:** relaxed invariant for tail (tail is reachable from head)

Algorithm of Michael-Scott’s queue

- **Push 1:** find the actual tail
  - Update the “tail” pointer if necessary
- **Push 2:** try to append a new node
  - If it fails, retry from the beginning
- **Push 3:** update the “tail” pointer
  - Perform a CAS from the old to the new tail
  - It’s okay to fail the CAS (we just want tail pointer is sufficiently recent)

- **Pop 1: update the “tail” pointer if it points to the dummy node**
  - Ensure the head does not catch up to the tail
  - It’s okay to fail the CAS (we just want tail pointer is sufficiently recent)
- **Pop 2: update the “head” pointer**
  - Perform CAS from the old to the new head
  - If it fails, retry from the beginning

Questions for Michael-Scott’s queue

- Why can a tail pointer be stale?
  - Right after a new node is inserted, the tail pointer becomes stale.
- Why have a dummy node at the head?
  - To make sure that the tail points to some node even for empty queues.
- In pop, why update the tail if it points to the dummy?
  - To make sure head doesn’t “overtake” tail (why? see the GC part).
- Why are head & tail: CachePadded<Node<T>>?
  - To make sure that head and tail are not falsely shared.

Example: sorted singly linked lists

- Linked list whose nodes (consisting of key, value, next) are **sorted by keys**
- https://github.com/kaist-cp/cs431/blob/main/src/lockfree/list.rs

- **Finding a node:** iterating list and finding the matching node
  - Used in lookup, insert, delete

- **Deleting a node:**
  - Step 1: tagging its next pointer by 0x1 (logical deletion)
  - Step 2: detaching it in another iteration (physical deletion)

- **Iteration strategies:** how to deal with logically deleted nodes?
  - Harris's: detaching consecutive logically deleted nodes at once
  - Harris-Michael's: detaching logically deleted nodes individually
  - Harris-Herlihy-Shavit's (wait-free): ignore logically deleted nodes
    - Can be used only for lookup (not insert/delete)

Questions for sorted singly linked lists

- Why do logical deletion and then physical deletion?
  - To synchronize with insert right after the deleted node.
- What's the motivation of Harris-Michael's list?
  - To support hazard pointers; more on it later.
- What's the motivation of Harris-Herlihy-Shavit's list?
  - For wait-free lookup.
- Why can't HHS be used for insert/delete?
  - Cursor's "prev" may have already been deleted and cannot be updated.

More examples

- **Circular buffer**
  - https://people.mpi-sws.org/~dreyer/papers/gps/paper.pdf

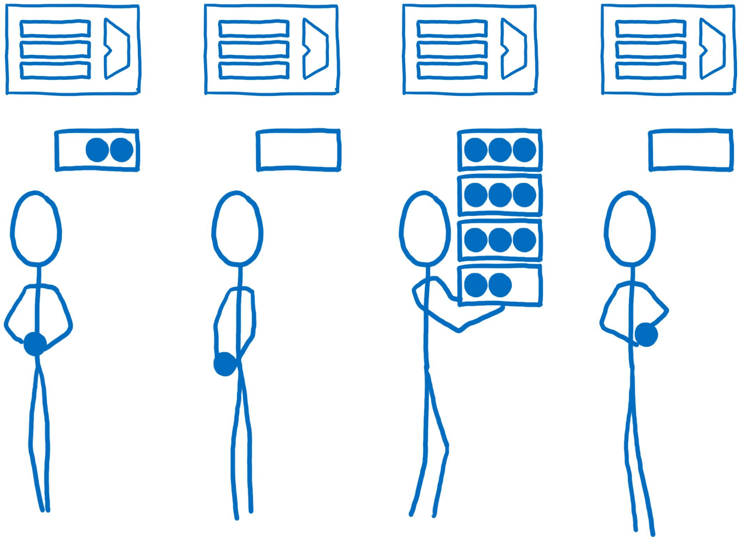
- **Hash table, binary trees (AVL/red-black), radix trees, …**

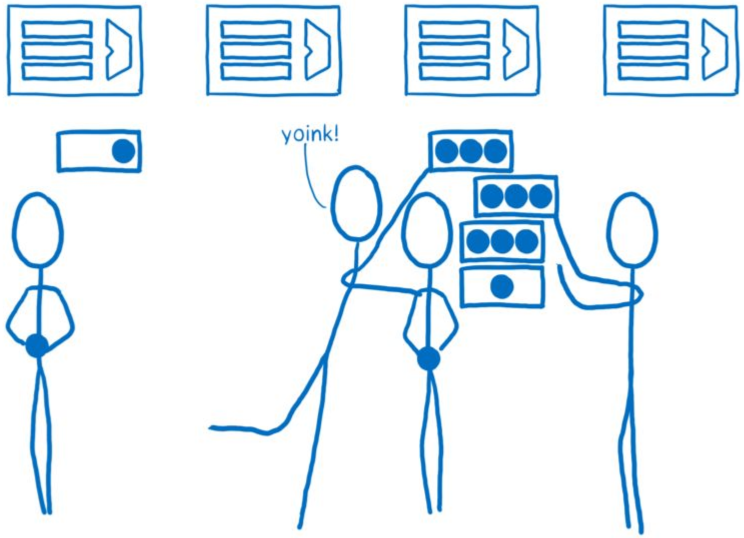
- **Chase-Lev work-stealing deque**
  - https://www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-dequeue.pdf
  - https://fzn.fr/readings/ppopp13.pdf
  - https://github.com/crossbeam-rs/crossbeam/tree/master/crossbeam-deque
  - https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md

More about work stealing

- Originated from Cilk: http://supertech.csail.mit.edu/cilk/
- Dynamic balancing of workloads across parallel resources
  - By “stealing” work from the other threads

- (figures from [Lin Clark's blog post]())





Lock-based concurrency

Part 5: optimistic locking

Optimistic concurrency control (OCC)

- Observation: frequently, multiple operations are not conflicting with each other
- Idea: optimistically assumes the success of operation, posthumously recovers

- Primitive: sequence lock
  - https://github.com/kaist-cp/cs431/blob/main/src/lock/seqlock.rs
  - Observation: all read operations are not conflicting with each other
  - Idea: optimistically reads, posthumously validates with sequence number

Sequence lock

- Optimistic reader-writer lock

- Writer: almost the same with spinlock
  - Managing sequence number (usize) instead of lock flag (boolean)
  - Even: consistent state between c.s., odd: inconsistent state inside c.s.
  - E.g. W(0): acquires 0->1, releases 2,
    W(2): acquires 2->3, releases 4, ...

- Reader: trying to read a consistent state (e.g. 2)
  - **Reading** sequence number @ beginning & end (should be even & same)
  - **Req 1:** W(0)'s end happens before R(2)'s beginning
    Req 2: R(2) doesn't see W(2)'s writes as far as R(2) is validated

Example: coarse-grained optimistic list

- algorithm (그림): global read lock -> find -> finish/upgrade
- memory reclamation problem
  - 위 그림 또는 GC 슬라이드 이용
  - RCU API만
  - old slide https://docs.google.com/presentation/d/1NMg08N1LUNDPuMxNZ-UMbdH13p8LXgMM3esbWRMowhU/edit#slide=id.ga54eefc4bc_1_651
- https://git.fearless.systems/kaist-cp-class/cs431-private/-/blob/d92dac01a8036c7abdb6ccf13a90fcc338332c47/src/list_set/optimistic.rs (TODO: publish)
  - crossbeam_epoch API
    - using Atomic<T> to explicitly allow race
    - reclamation

Lock-free concurrency

Part 3: garbage collection

In a separate slide

Lock-free concurrency

Part 4: linearizability

Key objectives of concurrent data structures (CDS)

- **Progress:** guaranteeing the completion (or progress) of operations
  - Lock freedom: progress of at least one
  - Wait freedom: progress of everyone

- **Scalability:** showing better performance as the number of cores grow
  - Ideal: linear scaling
  - Reality: e.g. sublinear scaling after 16 threads
  - Key idea: reducing critical sections (fine-grained locks and CAS) & writes

- **Correctness:** working “as expected”
  - Safety: doesn’t go wrong (i.e., no segmentation fault)
  - Sequential specification: e.g. works like a queue
  - Synchronization: e.g. matched push and pop are synchronizing

Scalability of CDS

- **Key idea 1: reduce contention by shrink lock protection scopes**
  - E.g. hand-over-hand locking, lock coupling, read-write locking

- **Key idea 2: reduce cache invalidations by avoiding writes**
  - E.g. “optimistic concurrency control”
  - E.g. avoiding writes in readers (especially for read-mostly workloads)
  - E.g. lightweight custom synchronization protocols

- **Case study: optimistic lock coupling**
  - Lock coupling + optimistic concurrency control
  - ~~Homework 2: Optimistic lock coupling for binary search tree~~

Safety of CDS

- **Key idea: protect CDS w/ locks or more primitive synchronization**
  - Protecting a sequential DS with a global lock
  - Protecting a DS with more fine-grained locks
  - Protecting a DS with custom synchronization protocols

- **Specification: “linearizability”** (https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md)

Linearizability: the “right” correctness specification

- **Key idea:** the only complication of CDS over sequential DS is
  the order of operations (not the order of instructions)

- **Contextual refinement:** CDS works “as if” an abstract DS
  - Abstract DS: DS operations are single instructions

- **Linearizability:** a key lemma for contextual refinement
  - In an execution, there exists a total order R among CDS operations s.t.:
  - (VIEW) If o1 happens before o2, then o1 R o2.
  - (SEQ) The results of operations are as if they are executed to sequential DS in the order R.
  - (SYN) E.g. a push operation happens before its matching pop operation.

  - More detail in [this document]

How to prove linearizability?

- **Key question: how to find linearization order?**
  - Linearization point: a point in an operation that determines lin. order
  - Then what's the linearization point of each operation?

- **Key idea 1:** Write operation's commit point is its linearization point.
  - Recall: many lock-free DSs have single-instruction commit point

- **Key idea 2:** Read operation's *critical read* is its linearization point.
  - Example: [read of head = null] for a pop from the empty Treiber's stack

- **Challenges & research questions**
  - Some data structures don't have commit point (e.g. Chase-Lev deque). How to find the linearization order for such data structures?
  - How to *prove* linearization?

Linearization examples

- **Treiber's stack**
  - Key idea 1: DS operation order = the order of successful CAS
  - Key idea 2: empty pop's order = between the surrounding pop and push

- **Michael-Scott's queue**
  - Key idea 1: DS operation order = the order of successful CAS on the head & node's next pointers (but not on tail)

- **Harris-*'s lists**
  - Key idea 1: DS write operation order = the order of successful CAS/FAO On the head & node's next pointers
    - Except for detaching logically deleted nodes (it's not commit point)
  - Key idea 2: DS read operation order = active research area
    - Cf. paper1 paper2

Towards formal verification of linearizability

- Video lecture by Dr. Derek Dreyer (MPI-SWS)
- RustBelt: Logical Foundations for the Future of Safe Systems Programming
- At Jane Street (2019)

Relaxed-memory behaviors

Part 1: semantics

Motivation: lock-freedom reveals gory details

- Recall: concurrency is challenging due to nondeterminism arising from…
  - Interleaving: thread A and B’s executions are interleaved; and
  - Reordering: thread A’s instructions may be reordered.

- Reordering example
  DATA=42 || if FLAG==1
  FLAG=1 || assert(DATA==42) // may fail due to reordering!

- Lock would hide such reordering: X and Y are NOT concurrently accessed!
  - Reordering doesn’t happen across lock/unlock.
  - Each variable (X or Y) is accessed only with holding a lock.
  - => all accesses to each variable is completely ordered

- Remaining questions
  - How to forbid reordering across lock/unlock?
  - How to forbid undesirable reordering in lock-free programming?

Nondeterminism: the challenge of shared memory

- Memory: location -> byte w/ concurrent load/store instructions
- The single-most widely-used SMS
- **Challenge: highly nondeterministic**

- **Source 1 of nondeterminism: thread interleaving**
  - Load/store instructions of multiple threads are interleaved
  - Resulting in combinatorially explosive # of behaviors

- **Source 2 of nondeterminism: reordering**
  - Load/store instructions inside a single thread may be reordered
  - Resulting in unintuitive behaviors

- **Strategy: taming nondeterminism by forbidding unintended behaviors**

Nondeterminism due to reordering

- **Motivation:** performing optimizations as if it's a sequential program
- **E.g.** message passing (FLAG is a shared location, e.g. AtomicUsize)

```
DATA = 42;       ||   if FLAG.load() {
FLAG.store(1);   ||       assert(DATA == 42);
                 ||   }
```

- **Problem:** unintended behaviors due to reordering by compiler/hardware
  - Assertion failure due to reordering in the left thread or in the right thread
  - **Relaxed behaviors:** observable behaviors
    not captured in the "interleaving semantics"
- **Solution:** forbidding such a reordering with **ordering primitives**
  - **Fence:** fence(SC) between stores and between loads, or
  - **Access ordering:** FLAG.store(1, release) and FLAG.load(acquire)

Enforcing Order in Spinlock (1/2)

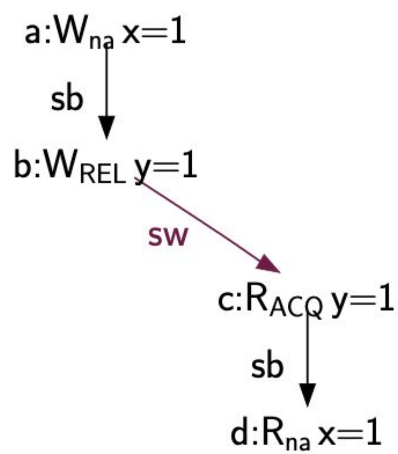
- https://github.com/kaist-cp/cs431/blob/main/src/lock/spinlock.rs
-
```rust
unsafe impl RawLock for SpinLock {
    type Token = ();

    fn lock(&self) {
        let backoff = Backoff::new();

        while self
            .inner
            .compare_exchange(false, true, Acquire, Relaxed)
            .is_err()
        {
            backoff.snooze();
        }
    }

    unsafe fn unlock(&self, _token: ()) {
        self.inner.store(false, Release);
    }
}
```
- “Ordering::Acquire” and “Ordering::Release” enforces the order.

Enforcing Order in Spinlock (1/2)

- // Thread 1
  DATA = 42
  LOCK = false (L.unlock()) // **RELEASE** to prevent reordering w/ above

- // Thread 2
  if (LOCK.cas(false, true)) // **ACQUIRE** to prevent reordering w/ below
  assert(DATA == 42)

- Release/acquire synchronization
  - If a release write is read by an acquire read,
    Then the write is **strictly happening before** the read.



More relaxed behaviors due to reordering

- **Reordering:** unless accessing the same location,
  any two load/store/rmw instructions can be reordered.
  - E.g. X=1; Y=1 -> Y=1; X=1 (store-store reordering)

- **E.g. load hoisting** [r1=r2=0 allowed by store-load reordering]
  - X=1 || Y=1
    r1=Y || r2=X
- **E.g. store hoisting** [r1=r2=1 allowed by load-store reordering]
  - r1=X || r2=Y
    Y=1 || X=1
- **E.g. Java causality test cases** (#16, #19, #20 are wrong)
  http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html

- **Homework:** (informally) making sense of the above examples

Forbidding reordering w/ ordering primitives

- **E.g. message passing w/ release/acquire synchronization**
  - DATA = 42; || if FLAG.load(**acquire**) {
    FLAG.store(1, **release**); || assert(DATA == 42);
    || }
  - **Release store:** forbidding reordering itself w/ earlier instructions
  - **Acquire load:** forbidding reordering itself w/ later instructions

- **E.g. message passing w/ sequentially consistent (SC) synchronization**
  - DATA = 42; || if FLAG.load(**relaxed**) {
    fence(SC); || **fence(SC);**
    FLAG.store(1, **relaxed**); || assert(DATA == 42); }
  - **SC fence:** forbidding any reordering across itself
  - **Relaxed:** imposing no orderings

- **Q: What is the precise meaning of relaxed behaviors and orderings?**

Challenge: modeling relaxed behaviors & orderings

- **Approach 1: avoiding concurrent accesses altogether with locks**
  - DRF (data-race freedom): no concurrent accesses, no relaxed behaviors.
  - Problem: concurrent accesses are essential

- **Approach 2: constraining executions by "axioms"**
  - Representing an execution as a value-flow graph, validated w/ axioms
  - Problem 1: not operational semantics (less intuitive)
  - Problem 2: allowing the bad "out-of-thin-air" behaviors

- **Approach 3: modeling reorderings w/ operational semantics**
  - "Promising semantics": https://sf.snu.ac.kr/promise-concurrency/
  - Operational semantics

Promising semantics

- Kang et al. A promising semantics for relaxed-memory concurrency. POPL 2017
- Interleaving operational semantics modeling relaxed behaviors and orderings

- **Key idea 1:** modeling load hoisting w/ **multi-valued memory**
  - Allowing a thread to read an old value from a location

- **Key idea 2:** modeling read-modify-write w/ **message adjacency**
  - Forbidding multiple read-modify-writes of a single value

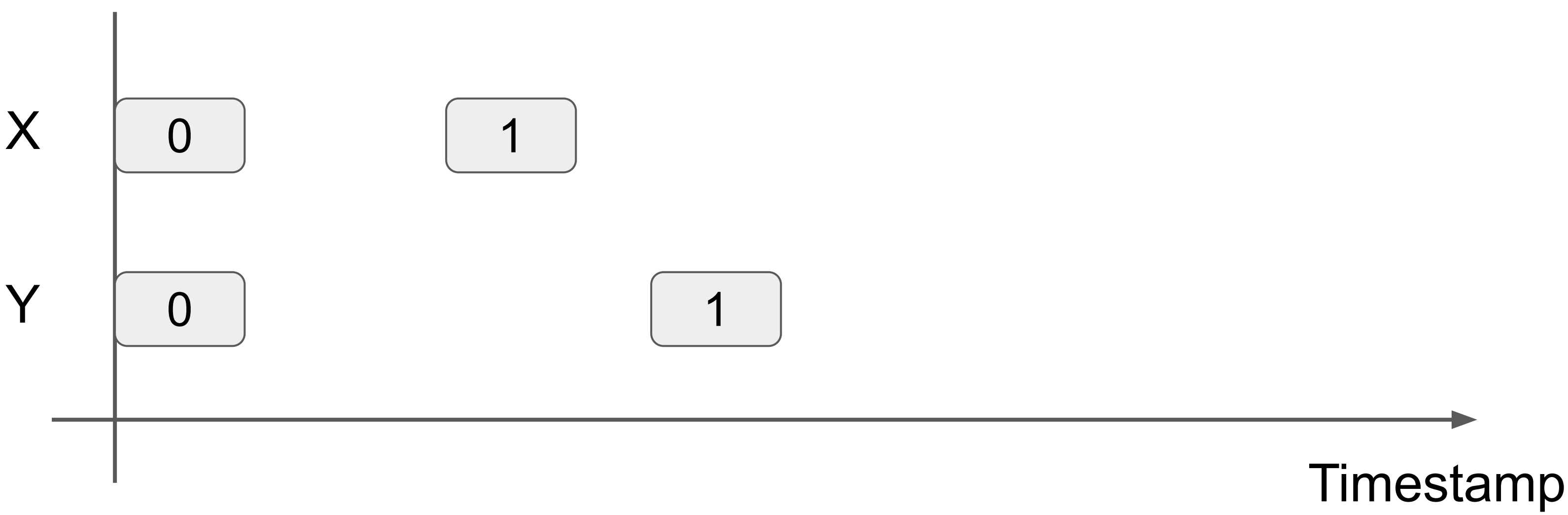
- **Key idea 3:** modeling coherence & ordering w/ **views**
  - Constraining a thread's behavior

- **Key idea 4:** modeling store hoisting w/ **promises**
  - Allowing a thread to speculatively write a value

Promising semantics 1: multi-valued memory

- **Memory:** location → list message, **message:** value & timestamp
- Threads may read an old value from a location (effectively hoisting loads)

- E.g. load hoisting [r1=r2=0 allowed by reading old values from X and Y]
  - X=1 || Y=1
    r1=Y 0 || r2=X 0

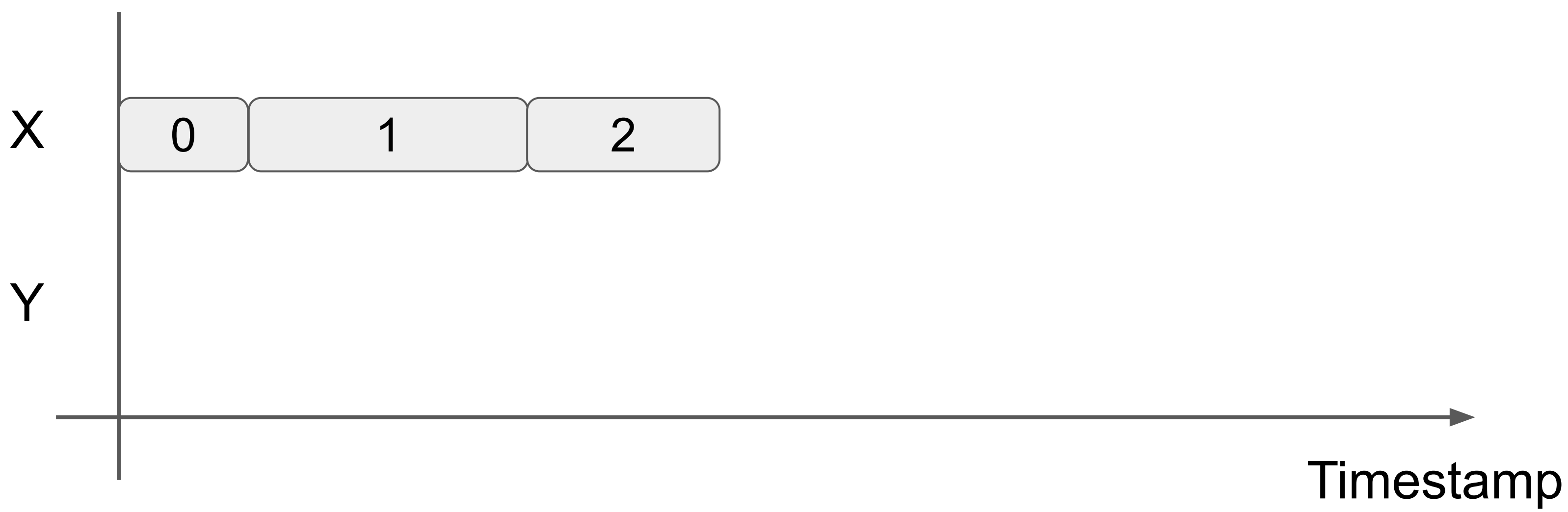


Promising semantics 2: message adjacency

- **Message:** value & timestamp range
- Read-modify-writes should put the new msg. **adjacent** to the old one (no gap)
- A message occupies a range of timestamps (e.g. (10, 20])

- E.g. counter [r1=r2=0 forbidden]
  - ● r1=X.fetch_add(1) 0 || ● r2=X.fetch_add(1) 1

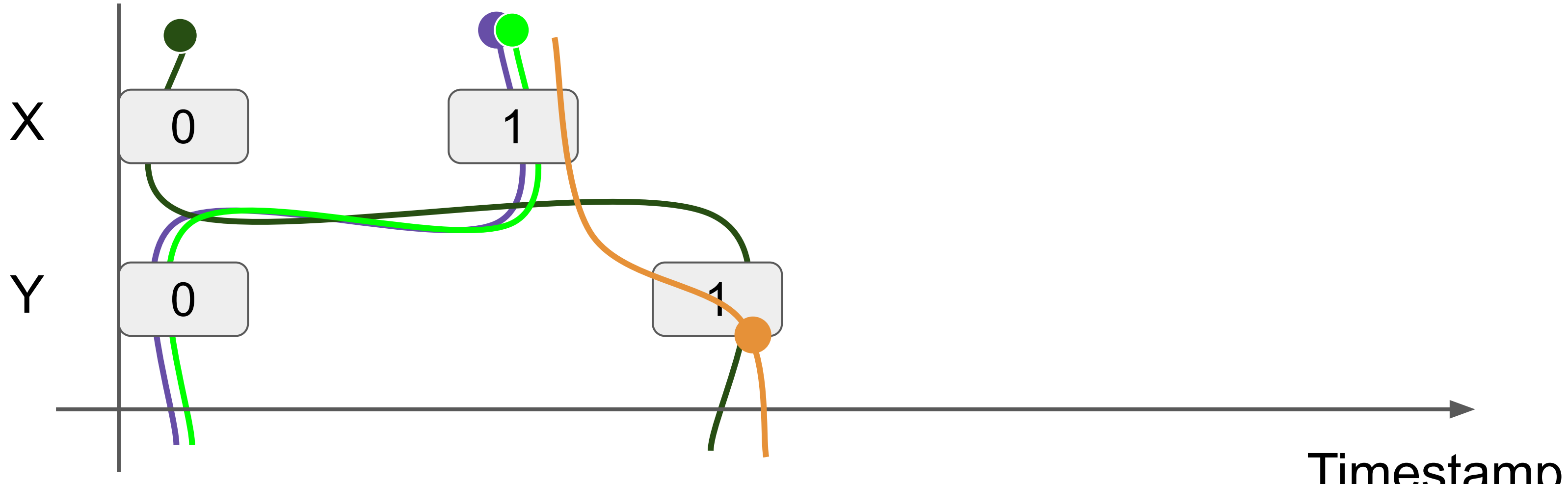
    ● ●



Promising semantics 3: views (1/4)

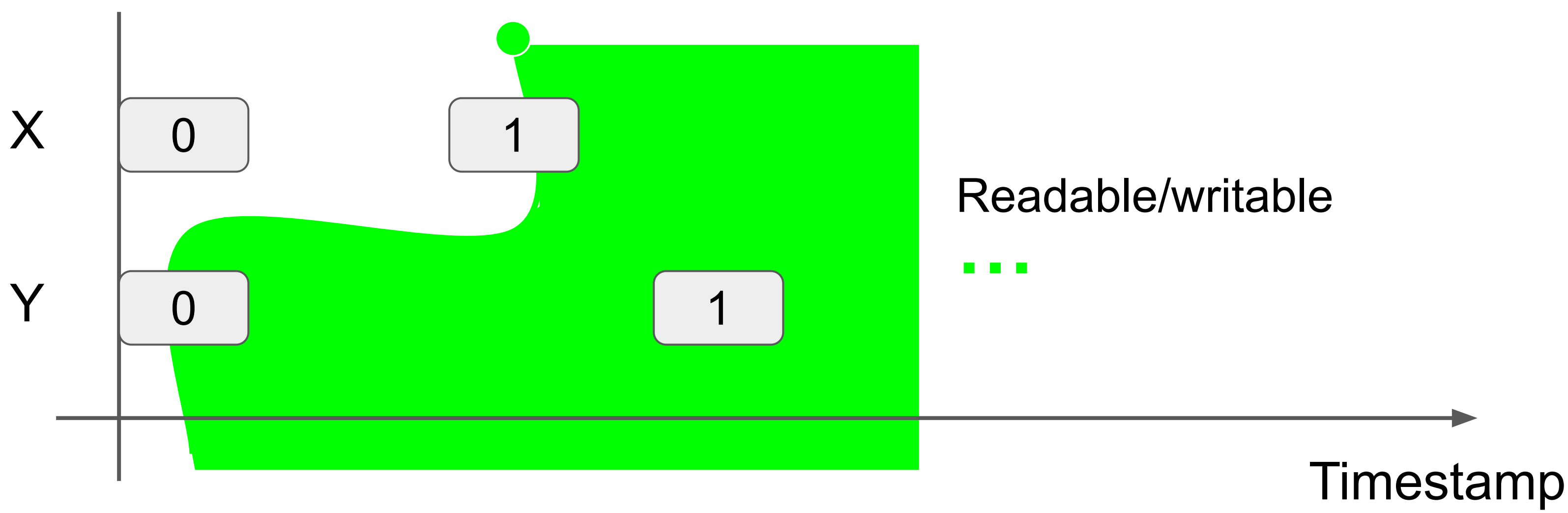
- Multi-valued memory allows too many, unintended behaviors
- Needs to constrain the behaviors w.r.t. coherence and synchronization

- **View:** location -> timestamp (acknowledging messages for each location) representing acknowledgement of messages
  - **Per-thread view** for coherence
  - **Per-message view** for release/acquire synchronization
  - **A global view** for SC synchronization



Promising semantics 3: views (2/4)

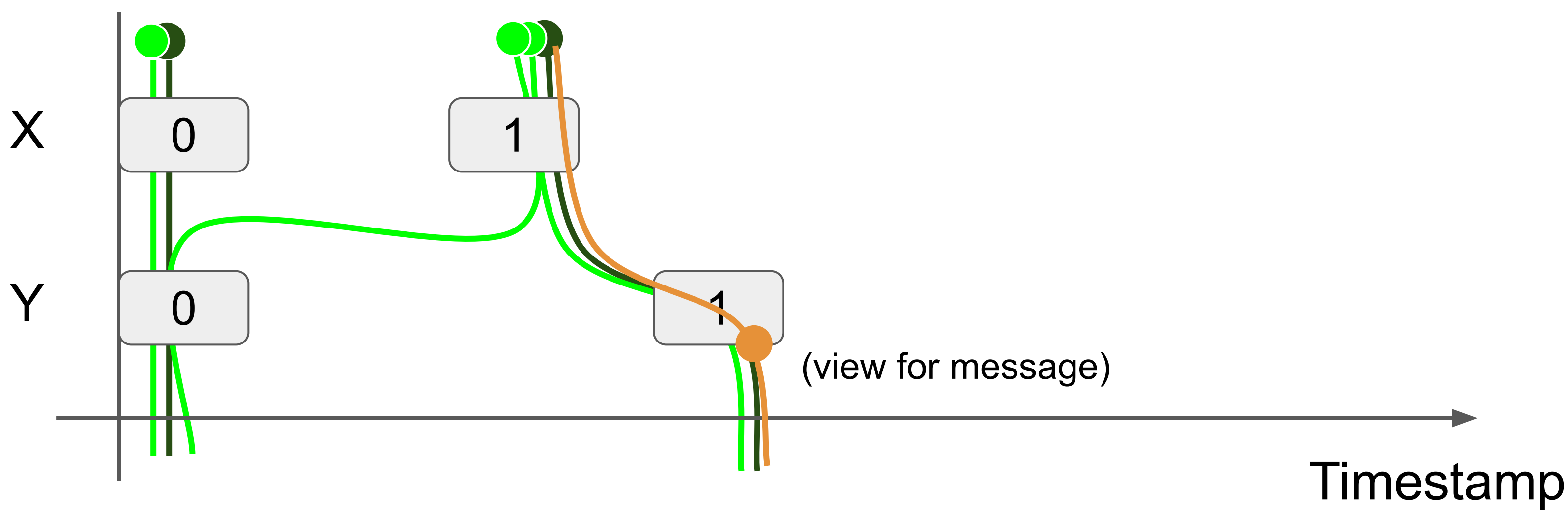
- **Per-thread view** representing a thread's acknowledgement of messages
- For modeling per-location coherence
  - RR coherence: X=1 || r1=X; r2=X [r1=1,r2=0 impossible]
  - RW coherence: r=X; X=1 [r=0]
  - WR coherence: X=1; r=X [r=1]
  - WW coherence: X=1; X=2 [X=2 at the end]
- Reading/writing happens after the current thread's view
- Reading/writing changes the current thread's view



Promising semantics 3: views (3/4)

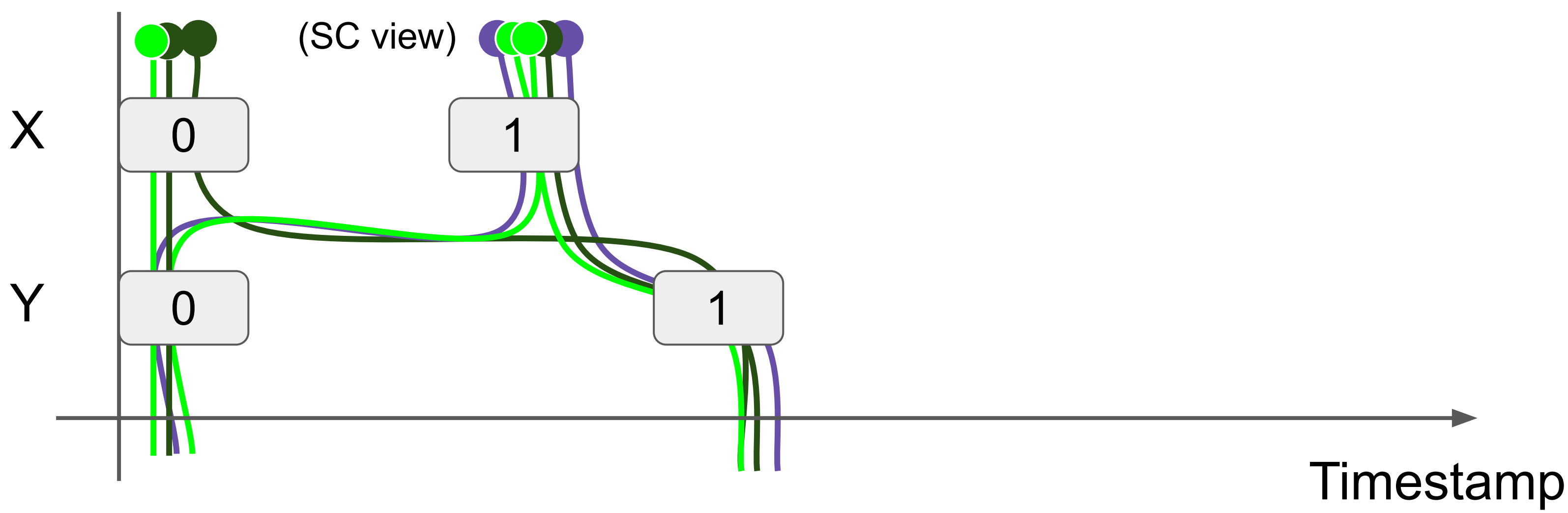
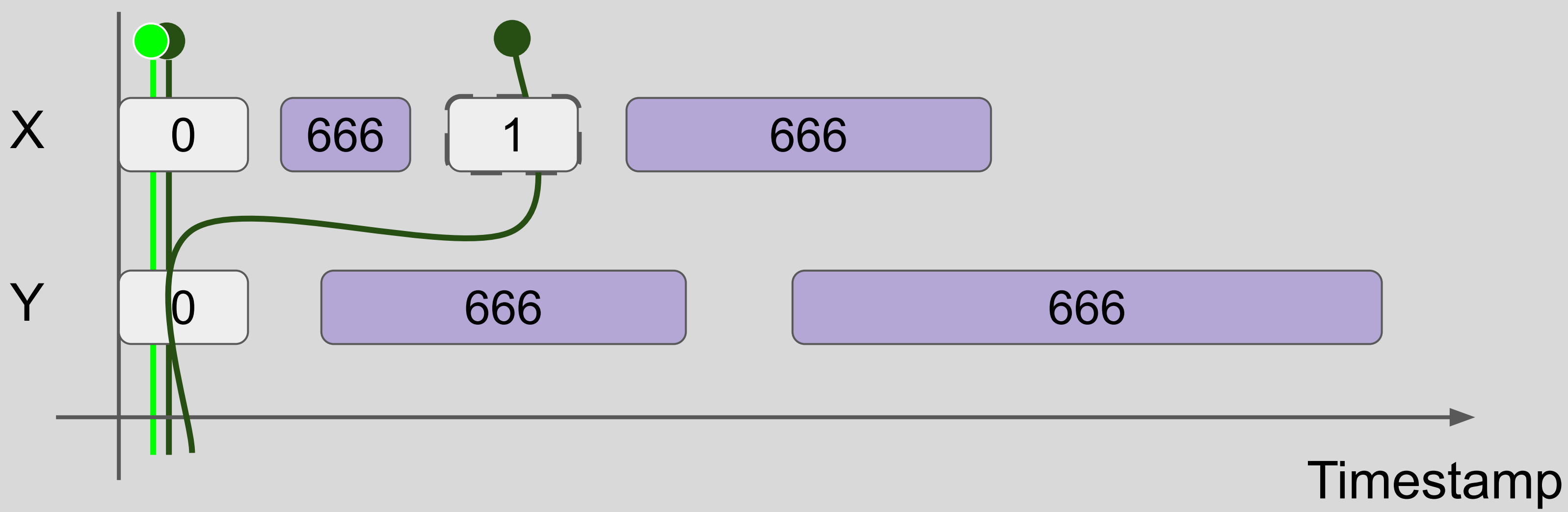
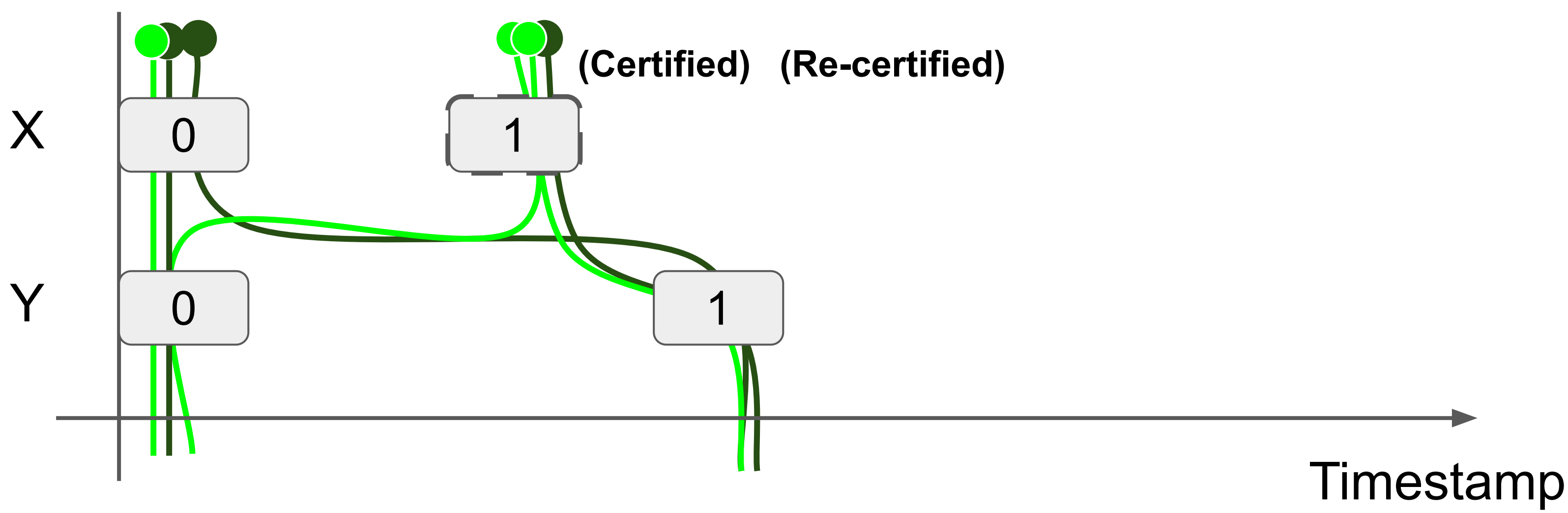
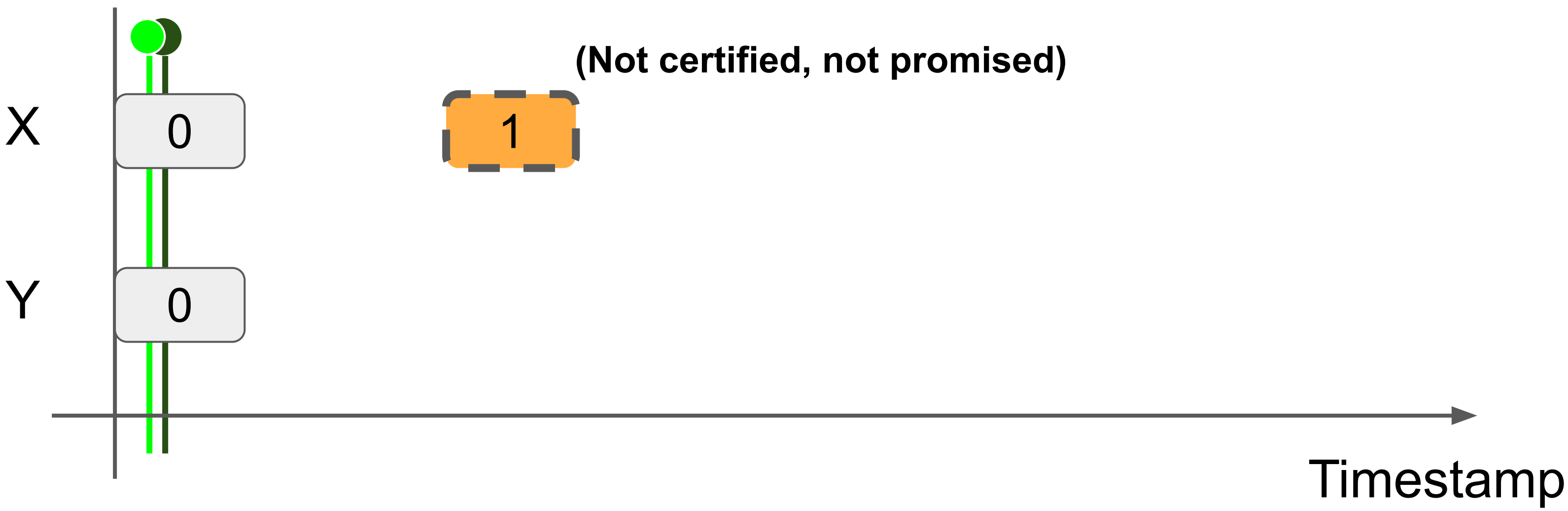
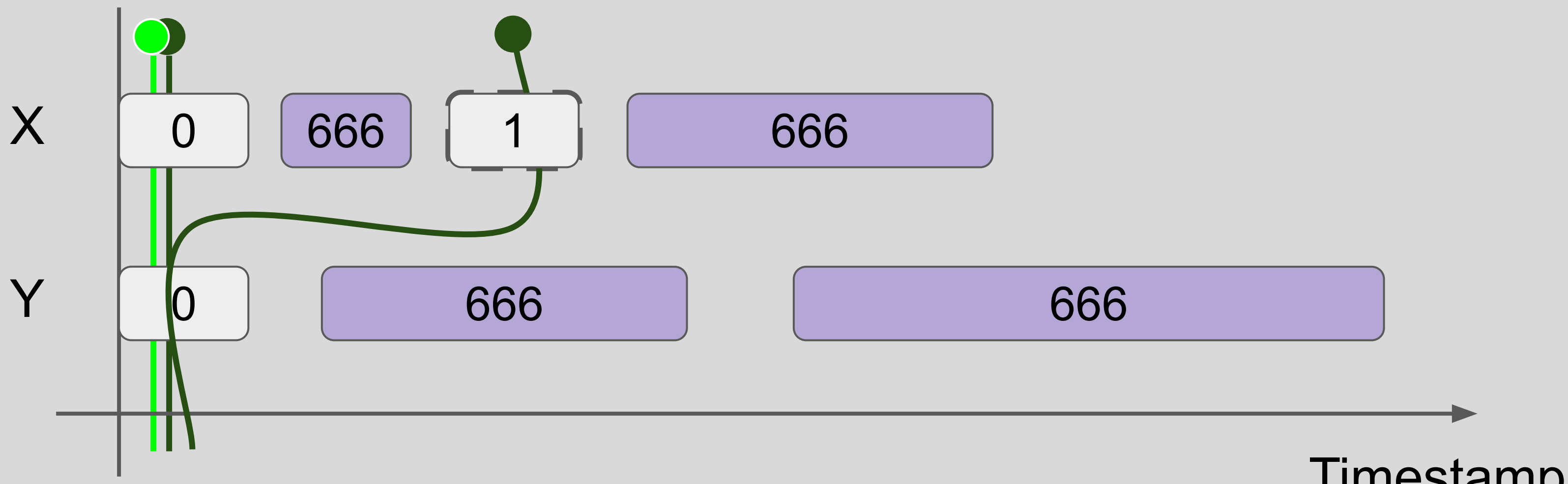
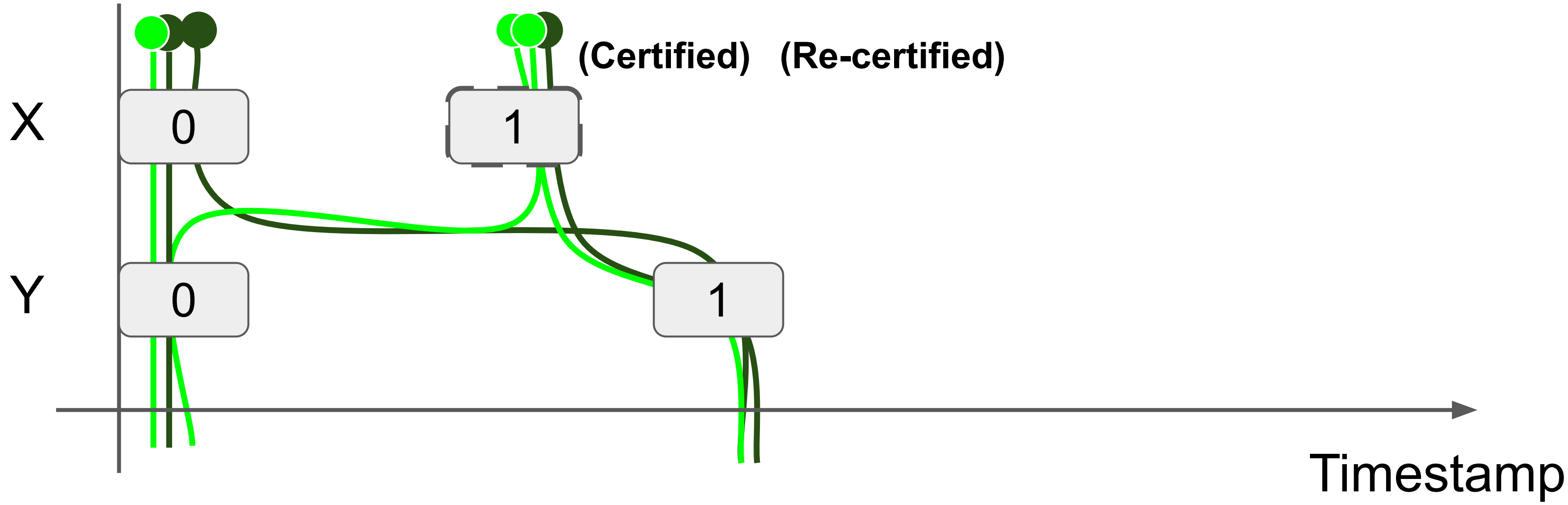
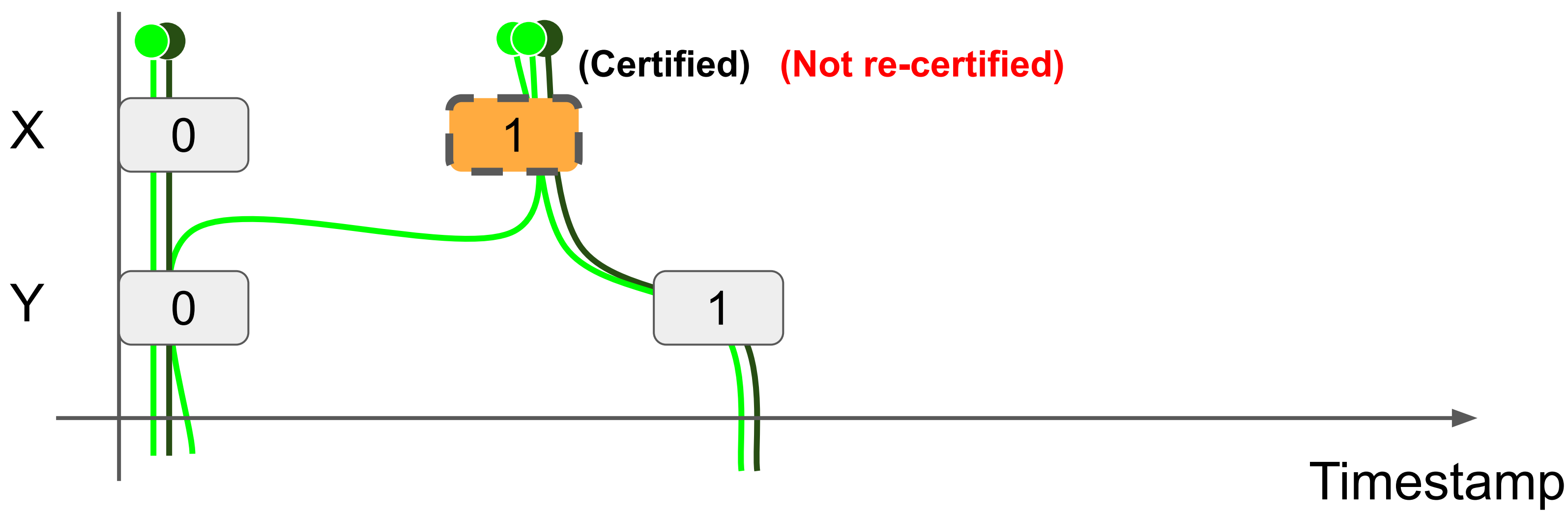
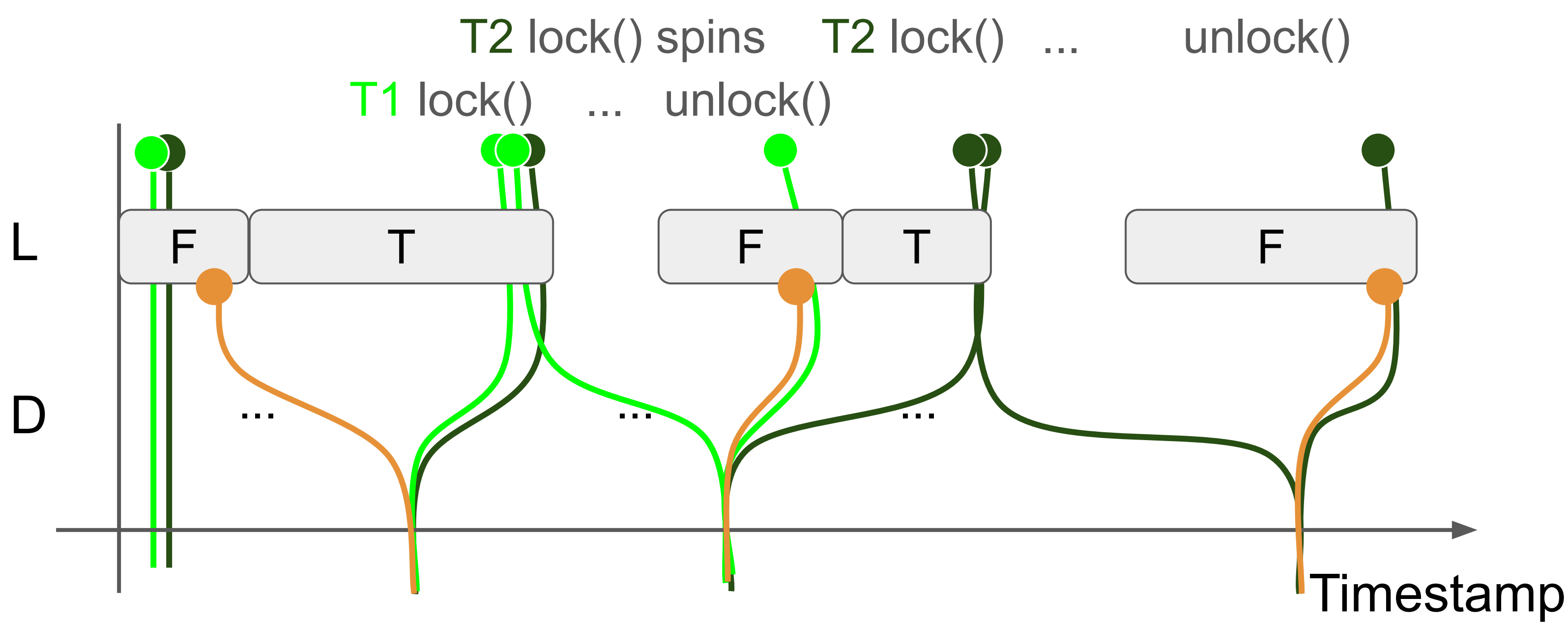
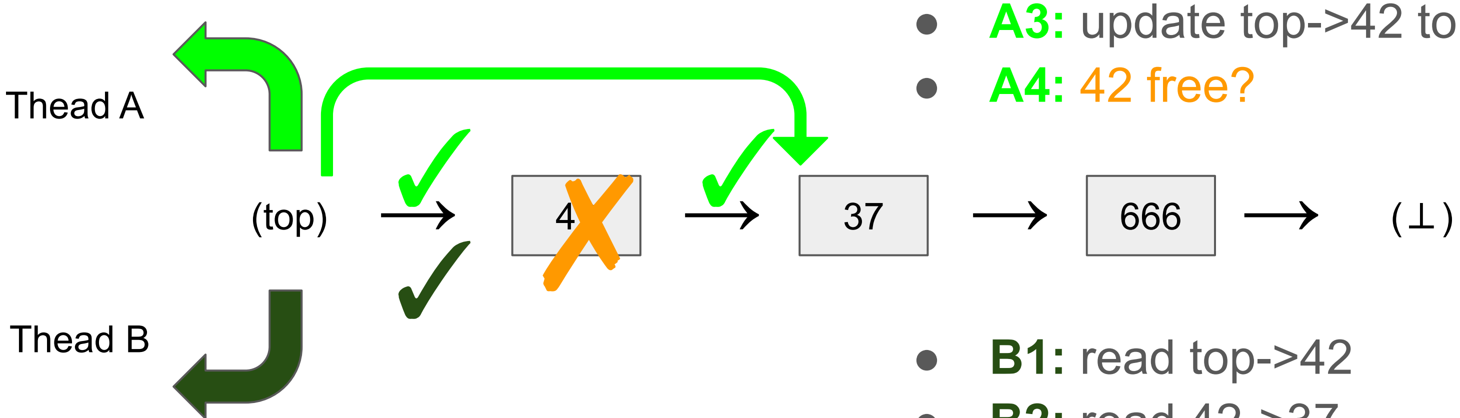
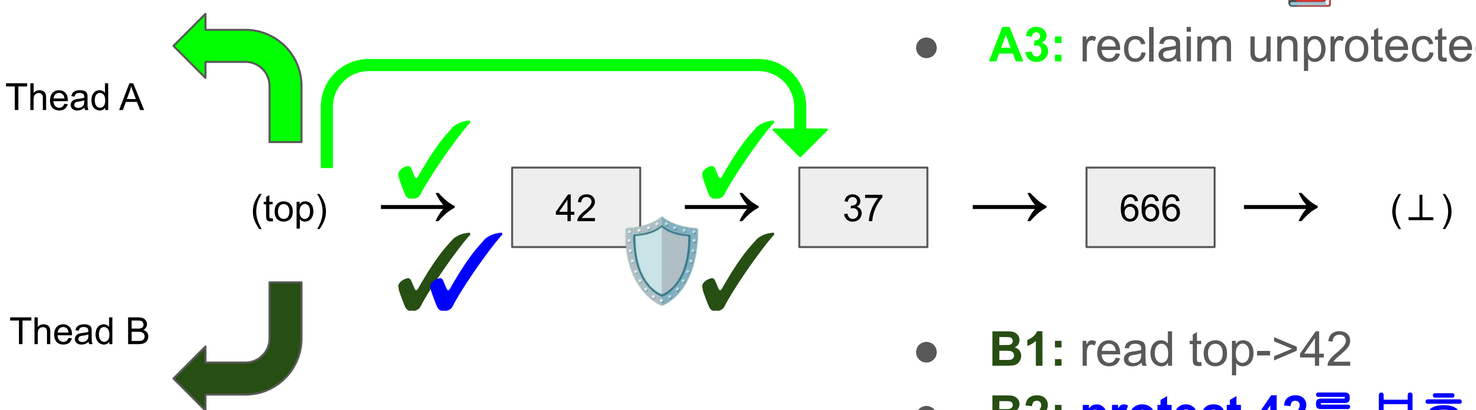
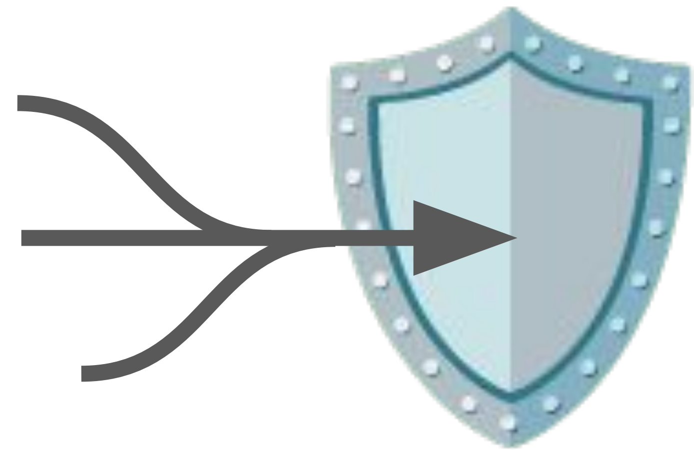
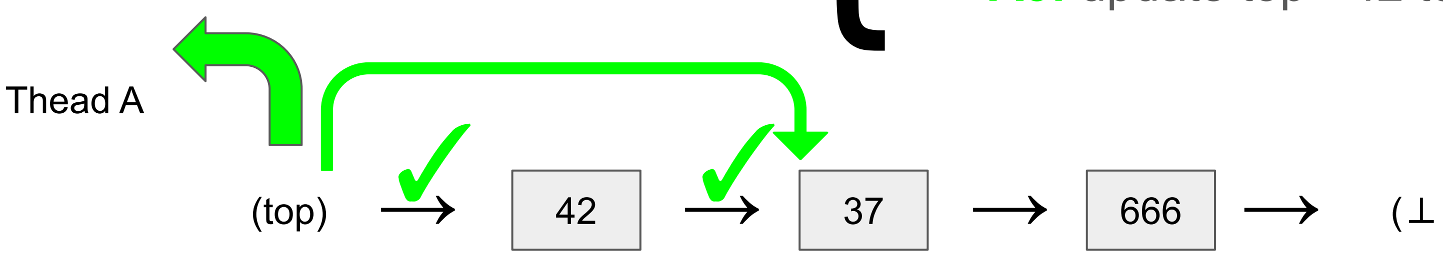
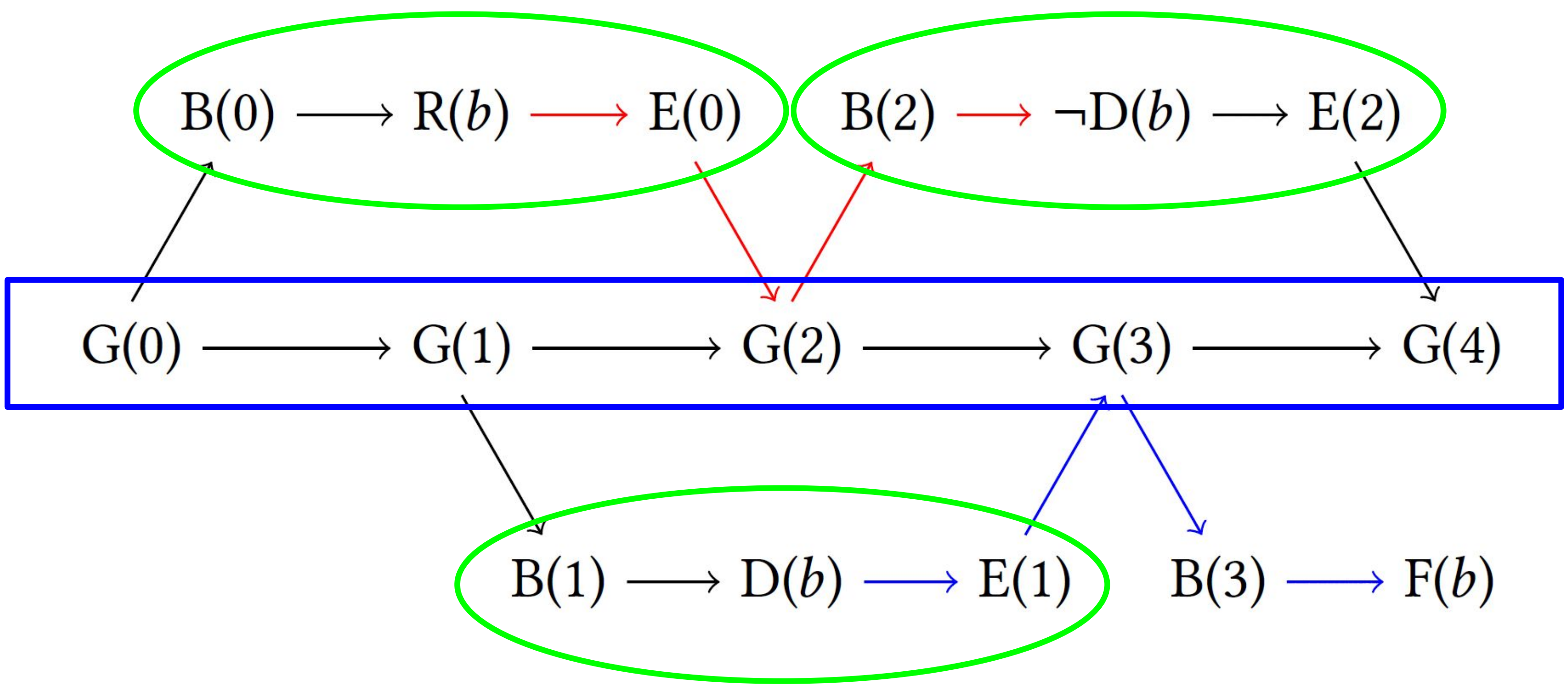
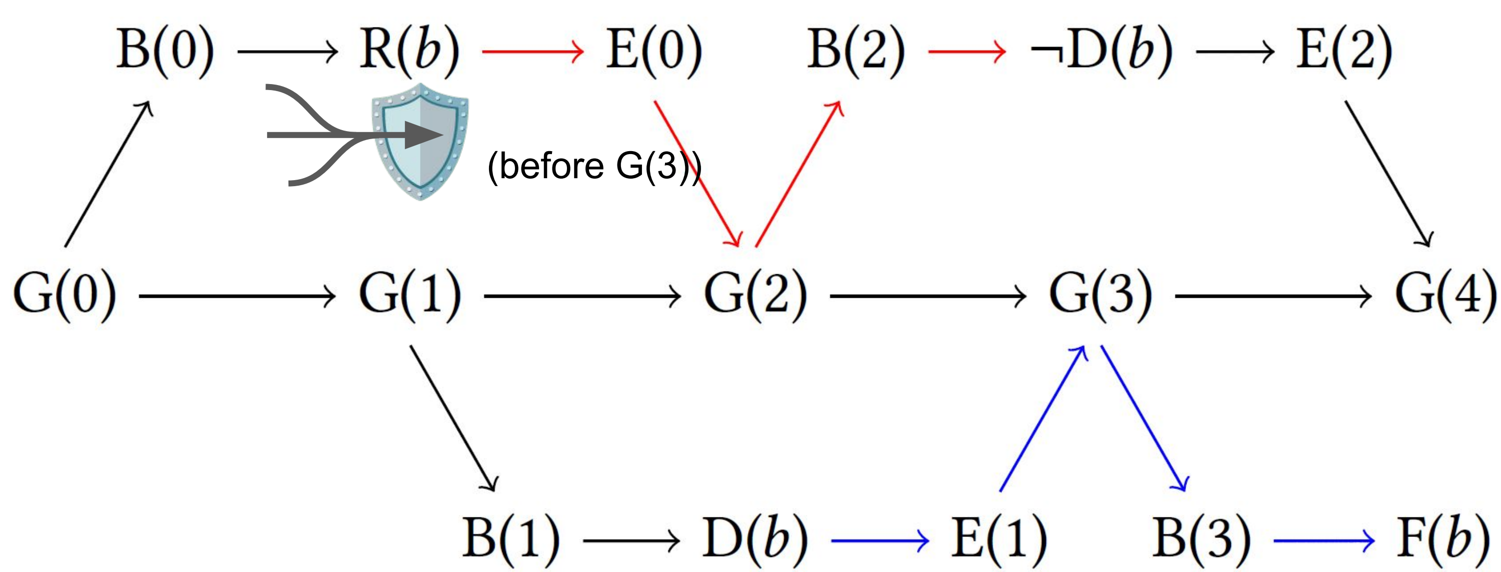
- **Per-message view** representing the released view of the corresponding store
- For modeling release/acquire synchronization

- E.g. message passing (X=1 should be acknowledged after reading Y=1)
  -
    ```
    X = 1;                  || if Y.load(acquire) {   1
    Y.store(1, release);    ||     assert(X == 1);    1
                            || }
    ```

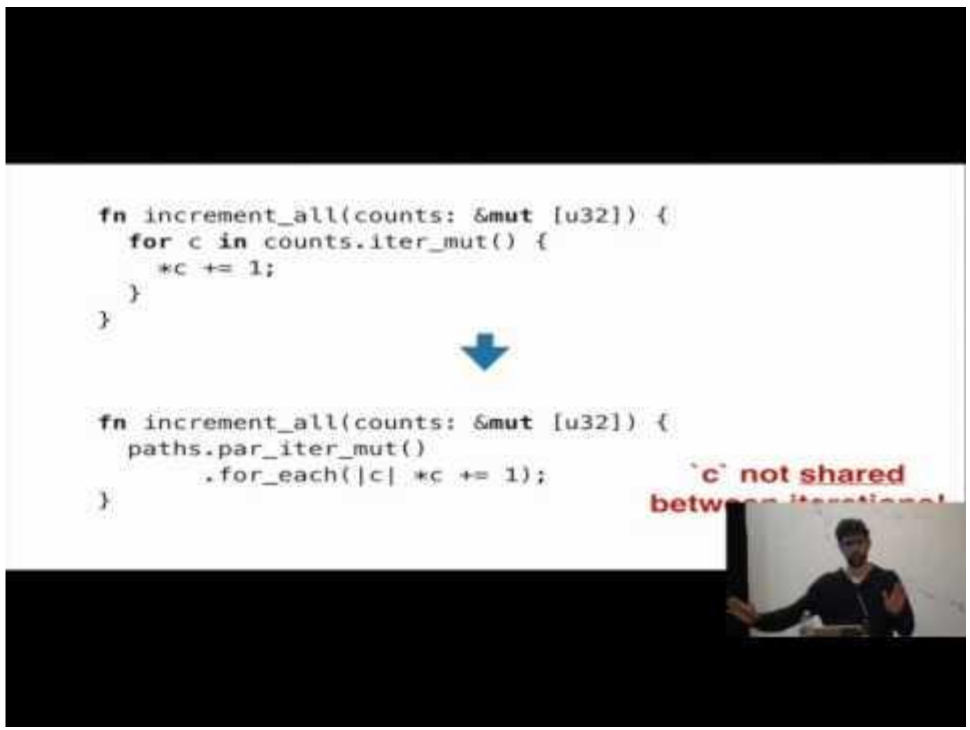
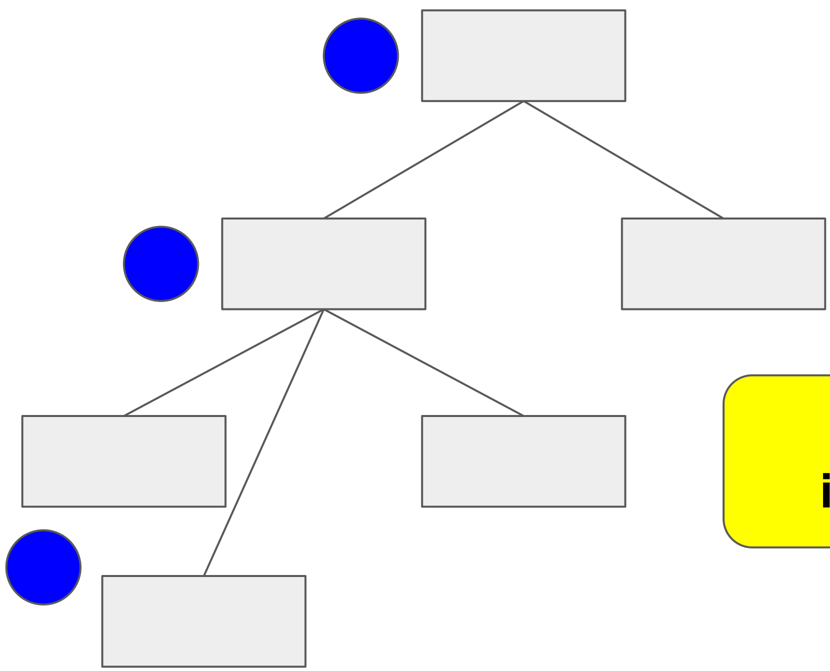
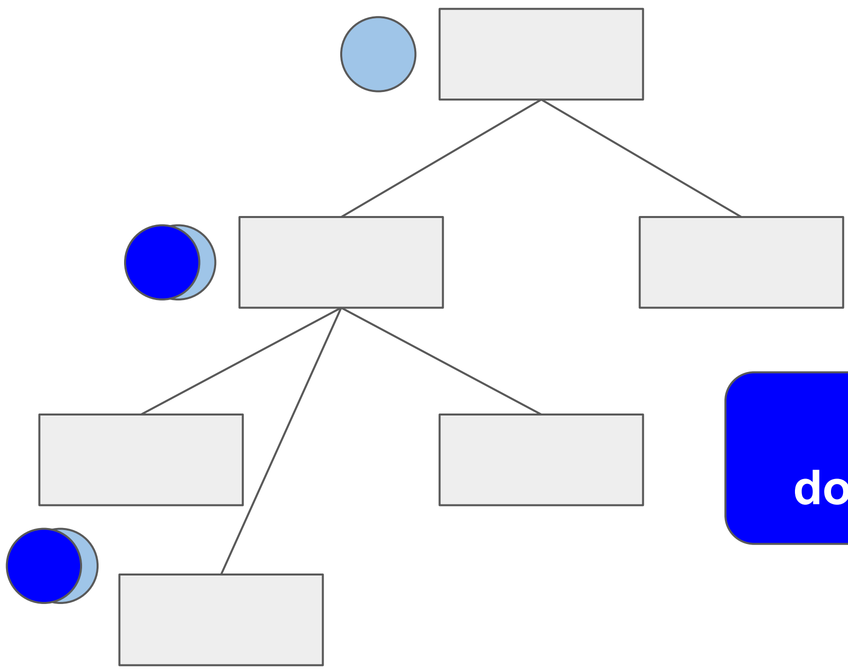
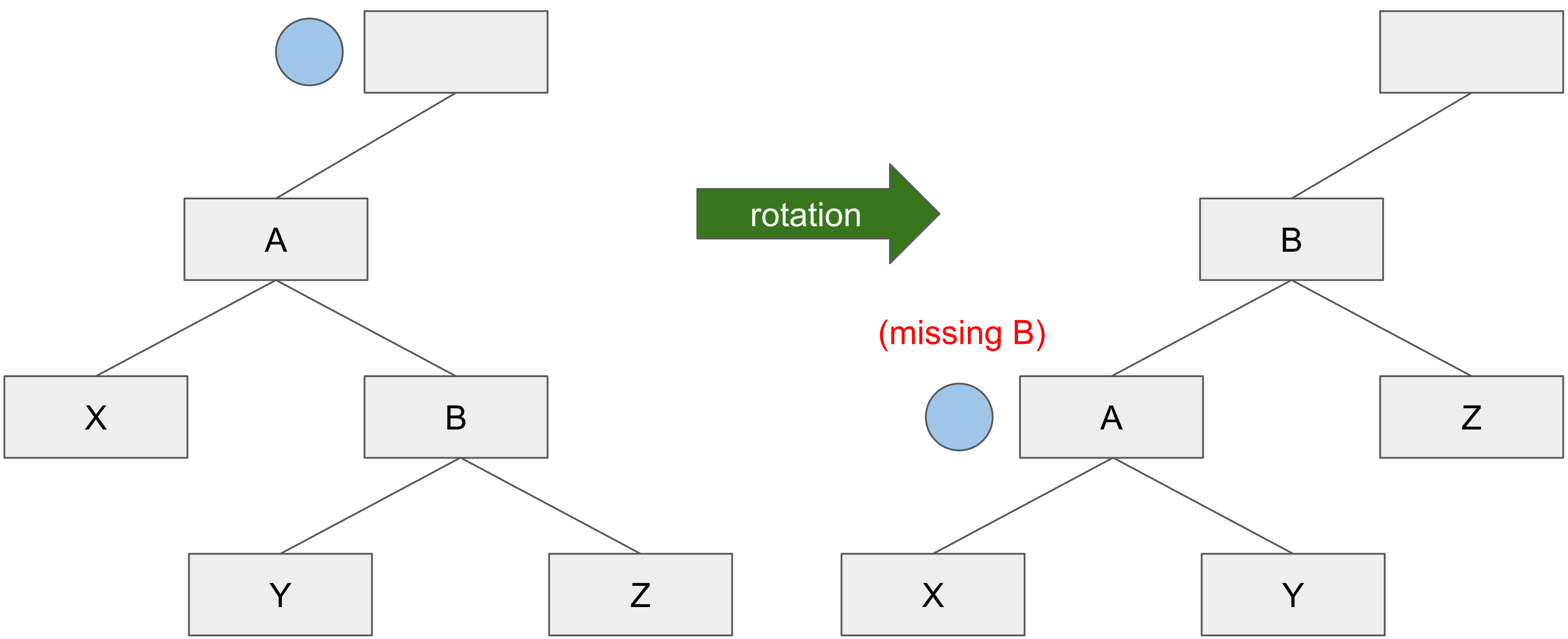


Promising semantics 3: views (4/4)

- **A global view** representing the currently accumulated view of SC fences
- For modeling SC-fence synchronization (strict order among SC fences)
- After an SC fence, SC view and thread's view become the maximum of them
- E.g. message passing (X=1 should be acknowledged after reading Y=1)
  - ```
    X = 1;                  || if Y.load(relaxed) {   1
    fence(SC);              ||     fence(SC);
    Y.store(1, relaxed);    ||     assert(X == 1);    1
    ```


    ```

Promising semantics 4: promises (1/5)

- **Challenge: modeling store hoisting**
  ("A major open problem for PL semantics", Batty et al., ESOP 2015)

- **Store hoisting w/o dependency** [r1=r2=1 allowed by reordering in the right]

```
r1=X   ||  r2=Y
Y=r1   ||  X=1
```

- **Store hoisting w/ dependency** [r1=r2=1 disallowed, "out of thin air" (OOTA)]

```
r1=X   ||  r2=Y
Y=r1   ||  X=r2
```

- **Store hoisting w/ syntactic dependency** [r1=r2=1 allowed by compiler opt.]

```
r1=X   ||  r2=Y
Y=r1   ||  if r2==1  { X=r2 }   // "if" should be taken for the behavior,
            else      { X=1  }   // but looks like OOTA
```

  - compiler may (1) forward r2=1 in the then branch and (2) hoist X=1 out

Promising semantics 4: promises (2/5)

- **Goal:** allowing the hoisting of **semantically independent writes** only
- We **will** cover promises in the final exam.
- **Idea:** “semantically independent writes” are **always writable** in the future
- **Mechanism**
    - A thread may speculatively write a value (“promise to write”)
    - A thread should **always be able to write its promises** in the future
- **Examples:** store hoistings w/ & w/o (syntactic) dependency

Promising semantics 4: promises (3/5)

- **Store hoisting w/o dependency** [r1=r2=1 allowed by reordering in the right]
  - ● r1=X || ● r2=Y

    Y=r1 || X=1

Promising semantics 4: promises (3/5), certification

- **Store hoisting w/o dependency** [r1=r2=1 allowed by reordering in the right]
  - (alone) r2=Y; X=1



Promising semantics 4: promises (3/5)

- **Store hoisting w/o dependency** [r1=r2=1 allowed by reordering in the right]
  - r1=X || r2=Y
    Y=r1 || X=1



Promising semantics 4: promises (4/5)

- **Store hoisting w/ dependency** [r1=r2=1 disallowed, “out of thin air” (OOTA)]

  r1=X || r2=Y

  Y=r1 || X=r2



Promising semantics 4: promises (5/5)

- **Store hoisting w/ syntactic dependency** [r1=r2=1 allowed]
  - r1=X || r2=Y
    Y=r1 || if r2==1 { X=r2 }
    else { X=1 }

Promising semantics 4: promises (5/5), certification

- **Store hoisting w/ syntactic dependency** [r1=r2=1 allowed]
  - (alone)

```
r2=Y
if r2==1  { X=r2 }
else      { X=1  }
```



Promising semantics 4: promises (5/5)

- **Store hoisting w/ syntactic dependency** [r1=r2=1 allowed]
  - ● r1=X || ● r2=Y
    ● Y=r1 || ● if r2==1 { X=r2 }
    ● else { X=1 }
    ●

Certification and real execution may diverge to capture semantic dependency only



Bonus: behaviors constrained by re-certification

- **Store hoisting w/ syntactic dependency** [r1=r2=r3=1 disallowed]
  - ● r1=X || ● r2=Y; r3=X // due to RW coherence
    ● Y=r1 || ● if r2==1 { X=r2 }
    ● else { X=1 }



Summary of promising semantics

- An operational semantics modeling relaxed behaviors and orderings

- **Key ideas**
  - **Multi-valued memory:** modeling load hoisting
  - **Message adjacency:** modeling read-modify-write
  - **Views:** modeling coherence and synchronization
  - **Promises:** modeling store hoisting (**covered** in the final exam)

- **Homework:** making sense of the examples in the promising semantics
  - Message passing
  - Load hoisting
  - Store hoisting w/ & w/o (syntactic) dependency
  - Java causality test cases (#16, #19, #20 are wrong) — already done!

Bonus: more on promising semantics

- https://sf.snu.ac.kr/promise-concurrency/ for more details
- Designated courses for details on lower-level (e.g. (micro-)architecture)

- Related work
  - View-based semantics for ARM/RISC-V architectures
    https://sf.snu.ac.kr/promising-arm-riscv/
  - View-based semantics for persistent memory
    https://cp.kaist.ac.kr/pmem
  - A program logic (formal reasoning principle) for promising semantics
    https://people.mpi-sws.org/~viktor/papers/esop2018-slr.pdf

- Research ideas
  - View-based semantics for interrupts, non-cacheables, I/O, PCIe, CXL, ...
  - Verified compilation for promising semantics
  - ...

Relaxed-memory behaviors

Part 2: revisiting locks and objects

Spinlock Correctness, Revisited

- pub fn lock(&self) { while self.inner.cas(false, true, **acquire**).is_err() {} }
  pub fn unlock(&self) { self.inner.store(false, **release**); }

- If a lock has already been acquired, lock() will spin
- Events between lock & unlock are transferred via release/acquire synch.
- When holding the lock, you'll access the latest value of D (no shared access)



Treiber's Stack Correctness, Revisited

- **Invariant:** head's pointer value has release view $\sqsupseteq$ (>= for each loc.)
  the messages of all node's value & next pointer

- **Release:** for maintaining the invariant (L54)
- **Acquire:** for exploiting the invariant (L68; reads at L70,84 are safe)

- **L74 can be relaxed:** the messages of later node's value & next pointer
  are already released
  - **"Release sequence":** a msg's relview is transferred to the adjacent msg
  - E.g. B's next pointer and data is released at Head
    - [Head -> B -> ⊥] // ∵ release at L50
    - => [Head -> A -> B -> ⊥] // ∵ head is CASed
    - => [Head -> B -> ⊥] // ∵ head is CASed
    - => [Head -> ⊥] // ∴ safe to read B's value and next pointer

MS Queue Correctness, Revisited

- **Invariant:** a pointer value has release view $\sqsupseteq$
  the messages of the pointed node's value & next pointer
  - A pointer value can be head, tail, or a node's next pointer
  - E.g. ptr's message has release view $\sqsupseteq$ A's value and & pointer (to B)

- **Release:** for maintaining the invariant
  - MS queue: L88, 101, 111, 140, 148

- **Acquire:** for exploiting the invariant
  - MS queue: L77, 81, 125, 128)

Sequence lock's synchronization

- W(0) happens before R(2): release/acquire synchronization
- R(2) doesn't see W(2)'s writes: release/acquire **fence** synchronization

```
// R(2)'s end            ||    // W(2)'s beginning
… // reading value       ||    update seq 2->3
fence(acquire)           ||    fence(release)
read seq (== 2?)         ||    … // writing value
```

(**Red**) if R(2) reads from any of W(2)'s writes,
(**Blue**) R(2)'s acquire fence happens after W(2)'s release fence,
(**Green**) invalidating R(2).

- Reads of R(2) and writes of W(2) should be "[atomic]" (no undefined behavior)
- R(2) may observe inconsistent state partially modified by W(2)
  (R(2) needs to be **resilient to such inconsistency**)

Closing remarks

What did you learn?

- **Concurrency:** about shared mutable resources
  - Difficult due to interleaving’s nondeterminism
- **Shared memory:** the most widely-used shared mutable resources
  - More difficult due to reordering’s nondeterminism

- **Design patterns**, both low-level (e.g. rel/acq) and high-level (e.g. helping)
- **Implementation** of locks, concurrent data structures, garbage collectors
  - Motivation, use cases, applications, tradeoffs, pitfalls
- **Promising semantics** for shared-memory concurrency

Expected outcome achieved?

- **Don't be scared of concurrent programming**
  - In theory, it's just dealing with shared mutable states
- **Don't be scared of systems programming**
  - Systems programming: about low-level resources
  - Especially for parallel systems
- **Don't be scared of programming**
  - You'll learn systematic approach to concurrent programming
  - Which is also beneficial for programming in general

What can I learn more about concurrency?

- **Hardware concurrency**
  - Interrupt, I/O, system calls, persistent memory, ...
  - X86, ARM, RISC-V concurrency
  - Hardware description w/ Verilog, VHDL, ShakeFlow
  - Compiler correctness

- **Concurrent data structures**
  - … (cannot summarize)
  - OS kernel, DBMS, …

Visit https://cp.kaist.ac.kr
Contact jeehoon.kang@kaist.ac.kr

- **Concurrent separation logic for verification**
  - "Separation": dealing with exclusive ownership of a part of resources
  - The most widely-used reasoning principle for concurrency
  - Iris: the most advanced CSL (https://iris-project.org/)

Thank you

FINAL EXAM

Everything you've learned after midterm

Backup slides

Example: memory reclamation in Treiber's stack

- Treiber's stack: singly linked list w/ head = stack top



- **A1:** read top->42
- **A2:** read 42->37
- **A3:** update top->42 to top->37
- **A4:** 42 free?

- **B1:** read top->42
- **B2:** read 42->37

Use-after-free error!

Question: how to ensure 42 is freed after finished being accessed?

Answer: by synchronization among threads

- **Memory reclamator:** library/runtime dedicated for such synchronization
  - Motivation: too costly to solve the problem for each data structure

- **Many reclamation schemes**
  - Pointer-based reclamation (PBR)
  - Epoch-based reclamation (EBR)
  - Hybrids: QSBR, Snowflake, QSense, IBR, Hazard Era, **PEBR**, ...

- **Key ideas**
  - **Protect** blocks being accessed (by each thread)
  - **Retire** unlinked blocks (instead of immediately reclaiming them)
  - **Reclaim** retired blocks that are not protected by any threads

Hazard pointer (HP)

- **Key idea 1: protect** blocks before accessing them

- **Key idea 2:** retire blocks Instead of immediately reclaiming them
- Reclaim those retired blocks that are not protected by any threads

HP's example: Treiber's stack



- **A1:** … (detaching 42)
- **A2: retire 42** 📕
- **A3:** reclaim unprotected blocks

- **B1:** read top->42
- **B2: protect 42를 보호** 🛡️
- **B3: validate top->42**
- **B4:** read 42->37

- **Case 1:** B2🛡️ => A2📕
  - B2🛡️ => A2📕 => A3
  - A3 doesn't reclaim 42
- **Case 2:** A2📕 => B2🛡️
  - A1 => A2📕 => B2🛡️ => B3
  - B3 cannot read top->42
  - B4 doesn't read 42

HP’s API

- **Data**
  - PL: Per-thread protected block (hazard pointer) list
  - RL: Per-thread retired block list
- **Protect(block)**
  - P1: add block to PL
  - P2: SC fence
  - P3: validate if the block is still pointed to by the pointer; otherwise, retry
- **Retire(block)**
  - R1: remove all references of the block from the shared memory
  - R2: SC fence
  - R3: add block to RL
- **Collect()**
  - C1: read every thread’s PL
  - C2: reclaim those blocks in RL that are not in every thread’s PL

HP’s problem: expensive synchronization

- **Synchronization points:** Inserting to RL 📕, Inserting to PL 🛡️
  - To order 📕 and 🛡️
  - Reordering may happen without synchronization

- **Synchronization cost:** fence in both A and B
  - 100+ cycles (e.g.: x86 mfence, ARMv8 dmb sy, POWER sync)
  - **Critical:** each traversal executes a fence in 🛡️ (⅓ throughput)

Epoch-based reclamation: reducing synchronization cost

Epoch-based reclamation (EBR)

{



- **Key idea 1: critical region** of multiple accesses
- **Key idea 2: epoch consensus** (concurrent critical regions have "similar" epochs)
- **Protect epochs**, not pointers (amortizing synch. cost)

EBR’s key idea 1: critical region

- **Critical region (c.r.):** memory-accessing period inside a thread
- **User-defined:** may differ among data structures and applications
  - E.g. during "stack.pop()", during processing a DB query
- **Memory traversal within a critical region**
  - Shared memory access should be inside a c.r.
  - A pointer read inside a c.r. may be dereference inside the same c.r.
  - E.g. stack.pop() should be inside a c.r.

- **A1:** read top->42
- **A2:** read 42->37
- **A3:** update top->42 to top->37



EBR's key idea 2: epoch (1/2)

- **Epoch:** assigned to each c.r. (0, 1, 2, 3, ...), →: happens-before
- **B(i), E(i):** the beginning/end of c.r. with epoch i, **G(i):** the beginning of epoch i
- **Epoch consensus:** G(i) → B(i), E(i) → G(i+2)
  - Corollary: epochs of concurrent critical regions may differ only by 1
- **SC fence:** only at the beginning of critical regions



EBR’s key idea 2: epoch (2/2)

- **Protect:** blocks retired 📕 at i are protected until G(i+3)
- **Safety:** (1) inaccessible in c.r. @ i+2, (2) live in c.r. @ i+1

- **R(b):** retirement of b after detaching it from memory
  D(b): dereference of b
  F(b): free(reclamation) of b



EBR’s problem: not robust (memory leak)

- **When:** long critical region hinders epoch advancement and reclamation
  - Because E(i) → G(i+2)

- **Case 1:** user-defined long c.r. (e.g. OLAP, object cache, I/O)
- **Case 2:** unscheduled threads (e.g. oversubscription)

  Case 3: stalled threads (e.g. bugs, crash-only distributed systems)

Question: fast & robust memory reclamation?

PEBR: marriage of HP and EBR

- **Goal:** fast like EBR and robust like HP
- **Key idea: hybrid of HP and EBR**
  - EBR (fast) at first, ejected to PBR (robust) when blocks are not reclaimed

- (2019) Jeehoon Kang and Jaehwang Jung.
  A marriage of pointer- and epoch-based reclamation. Submitted.
  https://cp.kaist.ac.kr/gc/

Shared memory low-level synchronization pattern

Shared memory low-level synchronization patterns

- https://jeehoonkang.github.io/2017/08/23/synchronization-patterns.html

- **Pattern 1:** "positive" release/acquire synchronization
  - Positive: if A, then B
  - If a release store is read by an acquire load, view is transfered

- **Pattern 2:** "contrapositive" release/acquire synchronization
  - Contrapositive: if not B, then not A
  - If view is not transferred, a release store is not read by an acquire load

- **Pattern 3:** "negative" SC synchronization
  - Negative: either A or B
  - Either F1 happens before F2 or F2 happens before F1

Covering "most" low-level synchronizations

Positive release/acquire synchronization

- If a release store is read by an acquire load,
  Then the view "released" at the store is "acquired" at the load

- E.g. message passing (X=1 should be acknowledged after reading Y=1)
  - ```
    X = 1;                ||  if Y.load(acquire) {
    Y.store(1, release);  ||      assert(X == 1); // should not fail
                          ||  }
    ```
  - Transferring data (X) w/ release/acquire synchronization of flag (Y)

- **The most widely-used low-level synchronization pattern**
  - Used in spinlock, channel, ...
    ```

Negative SC synchronization

- All SC fences are strictly ordered w.r.t. the per-thread views
- E.g. message passing (X=1 should be acknowledged after reading Y=1)
  - ```
    X = 1;                  ||   if Y.load(relaxed) {
    fence(SC);              ||       fence(SC);
    Y.store(1, relaxed);    ||       assert(X == 1);  }
    ```
  - Transferring data (X) or ignoring flag (Y) w/ SC synchronization
- **An advanced low-level synchronization pattern**
  - Used in Peterson's mutex, memory reclamator, work-stealing deque, ...
    ```

Example: Peterson's mutex (implementation)

- let flag: [AtomicBool; 2];   // whether a thread wants to begin a critical section
  let turn: AtomicUsize = 0; // who has the precedence?

- fn begin(id: Usize) { // thread id: 0 or 1 **(two threads, T0 and T1)**
  ```
      flag[id].store(true);
      fence(SeqCst);              // A
      turn.store(1 - id);
      fence(SeqCst);              // B
      while (flag[1 - id].load(acquire) && turn.load() == 1 - id) {}
  }
  ```

- fn end(id: Usize) {
  ```
      flag[id].store(false, release);
  }
  ```

Example: Peterson’s mutex (correctness)

- **Case analysis for the order of fences**
- **Case 1: A0 -> B0 -> A1 -> B1**
  - flag[0] = true and turn = 1 should be ack. at A1
  - turn = 1 before turn = 0 w.r.t. Coherence
- **Case 2: A0 -> A1 -> B0 -> B1 or A0 -> A1 -> B1 -> B0**
  - Both flag[0] = true and flag[1] = true should be ack. at B0 and B1
  - W/o loss of generality, assume turn = 1 before turn = 0 w.r.t coherence
- … (similar cases)

- flag[0] = true is ack. at B1 && turn = 1 before turn = 0 w.r.t coherence
- T1 should spin until T0 calls end() writing flag[0] = false
- Mutex thanks to release/acquire synch. from T0 to T1 via flag[0]

- **Homework: Is each of the fences (A and B) necessary?**

Example: Peterson’s mutex (discussion)

- **Not practical due to drawbacks**
  - Unnecessarily complicated
  - Not reusable: cannot begin() after end() (hence not “lock”)

- **Theorem:** reusable mutex requires atomic rmw instructions
  (e.g. swap, compare-and-swap, or other rmw instructions)
  - Peterson’s mutex is an artifact before rmw instructions

- **Next section: more lock implementations w/ tradeoffs**
  - Spinlock’s advantages: simple, fast (when uncontended), compact
  - Spinlock’s drawbacks: unfair, inscalable, energy-inefficient

Videos on data parallelism and async I/O

Rayon: Data Parallelism for Fun and Profit

- Nicholas Matsakis @ Rust Belt Rust 2016
- (Turn on English caption)



Zero-Cost Async IO

- Boats @ Rust LATAM 2019
- (Turn on English caption)

Optimistic lock coupling

Lock coupling (“hand-over-hand locking”)

- **Holding at most 2 locks at a time during traversal**
  - Each node has its own lock
  - During traversal, hold locks for the “current” node and its parent
  - Holding the lock for the parent to protect structural changes



Read operations invalidates cache!

Optimistic lock coupling (OLC)

- **Holding at most 2 sequence locks at a time during traversal**
  - Upgrading to write lock to protect structural changes



Read operations
do not invalidate cache!

Why acquiring locks of two nodes simultaneously?

- Otherwise, we may miss values (e.g. B below) during traversal.



When is it safe to deallocate a node?

- **Question:** Read doesn't leave any footprint on memory. How to make sure concurrent threads don't deallocate the accessed node?

- **Approach:** separate synchronization mechanism for safe deallocation
  - "Safe memory reclamation scheme" or just garbage collection
  - E.g. epoch-based reclamation, hazard pointers, …
  - Will be discussed later in this course

- **Let's assume epoch-based reclamation (EBR)**
  - Operations are delimited by "critical sections"
  - Memory blocks are accessed and *retired* only inside critical sections
  - Retired blocks are automatically deallocated **later when safe**
  - Implementation: Crossbeam (concurrency library in Rust)

Old Annoucements

Homework 1 (due: Nov. 4, 20% of the credits)

- **Implement (sequential) adaptive radix tree in Rust**
  - https://github.com/kaist-cp/cs431/issues/6
  - Read paper: https://db.in.tum.de/~leis/papers/ART.pdf
  - Read skeleton: [link]

- **Read skeleton's README.md for specifications and recommendations**

Homework 2 (due: Nov. 29, 20% of the credits)

- **Synchronize binary search tree with OLC**
    - **IMPORTANT: not AVL, but plain binary search tree**
    - Read paper: https://stanford-ppl.github.io/website/papers/ppopp207-bronson.pdf
    - Read skeleton: [link]
    - Grading: TBA

- **Tip: read paper early!**

Midterm exam (Oct. 21, 20% of the credits)

- **Date & time:** 9:00am-11:00am, October 21th, 2019
- **Place:** Rm. 2111, Bldg. E3-1
- **Coverage:** everything you've learned in this course
- **Bring a black/blue pen and your student ID**

- **Study guide**
  - You will be asked to "explain" the reasons of a phenomena. Your explanation should be an informal proof, but with a few gaps. Your informal proof should be clear and precise.
  - You will be asked about the implementations details of the code presented in class (e.g. locks and ART). You may be asked to hand-write an implementation.
  - You will be asked about the video lecture.
- **Questions or comments?**

Final exam (Dec. 16, 20% of the credits)

- **Date & time:** 9:00am-11:00am, December 16th, 2019
- **Place:** Rm. 2111, Bldg. E3-1
- **Coverage:** everything you've learned after the midterm exam
- **Bring a black/blue pen and your student ID**

- **Study guide**
  - You will be asked to "explain" the reasons of a phenomena. Your explanation should be an informal proof, but with a few gaps. Your informal proof should be clear and precise.
  - You will be asked about the implementations details of the code presented in class. You may be asked to hand-write an implementation.
  - You will be asked about the video lecture.

- **Questions or comments?**

Garbage slides

Nondeterminism due to thread interleaving (TODO)

- **E.g.** concurrent counter: multiple threads incrementing a shared location:

```
static COUNTER = AtomicUsize::new(0);
// thread A & B
let c = COUNTER.load();
COUNTER.store(c + 1);
```

- **Problem:** unintended behaviors due to unfortunate scheduling
  - [COUNTER=0] A load, B load, A store, B store [COUNTER=1]

- **Solution:** forbidding such a scheduling by **atomically reading & writing**
  - **"Read-modify-write"**, e.g. swap, compare-and-swap, fetch-and-add
  - ```
    // thread A & B
    let c = COUNTER.fetch_and_add(1);
    ```
  - [COUNTER=0] A fetch_and_add, B fetch_and_add [COUNTER=2]
    ```

Optimizations introducing? relaxed behaviors

- **Reordering:** unless accessing the same location,
  any two load/store/rmw instructions can be reordered.
  - E.g. X=1; r=Y -> r=Y; X=1 (load-store reordering)
  - E.g. X=1; Y=1 -> Y=1; X=1 (store-store reordering)

- **Merging:** if accessing the same location, TODO: not introducing??
  two load/store/rmw instructions **may** be merged.
  (only when it makes sense for sequential program)
  - E.g. X=1; X=2; r=X -> X=2; r=X (store-store, store-load merging)
  - E.g. X=1; X.fetch_and_add(1) -> X=2 (store-rmw merging)

- **Eliminating:** if the result of a load is not used, the load may be eliminated.
  - E.g. r=X; -> nop (load elimination) TODO: not introducing??

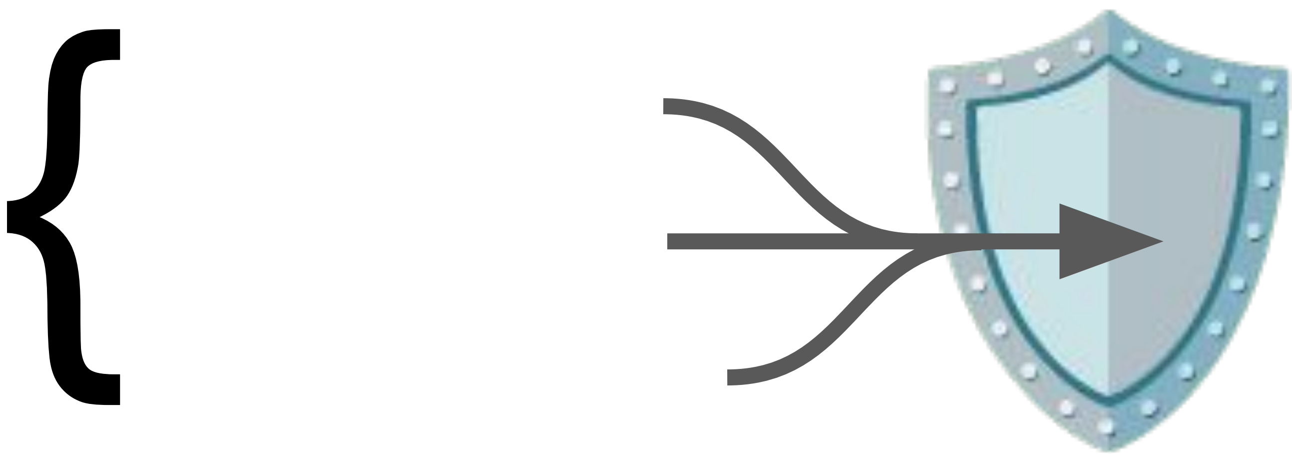
Wait-free data structures

- Key idea: each thread publishes its ongoing operation,
  the “winning” thread helps the others
  by finishing the published operations by the others

- Helping: https://dl.acm.org/citation.cfm?id=102808
- Flat combining: https://www.cs.bgu.ac.il/~hendlerd/papers/flat-combining.pdf
- Example: A Wait-free Queue as Fast as Fetch-and-Add

포인터/시대 기반 수집기법 (PEBR)

- **아이디어 :** PBR과 EBR의 혼종



- 처음에는 EBR (빠름), 메모리 수집이 잘 안되면 PBR로 **강퇴** (수집 보장)

Intermission: Crossbeam (TODO)

- Crossbeam: most widely used concurrency library for Rust

- **Utilities** (e.g. CachePadded)
- **Pointer value manipulation APIs**
  - Owned: TODO
  - Atomic: TODO
  - Shared: TODO

- **Garbage collection APIs** (epoch-based reclamation)
  - Guard: TODO
  - defer_destroy(): TODO
- **lock-free data structures**
  - Stack, queue, work-stealing deque, channel, …

Pointer/epoch-based reclamation (PEBR)

- https://cp.kaist.ac.kr/gc/

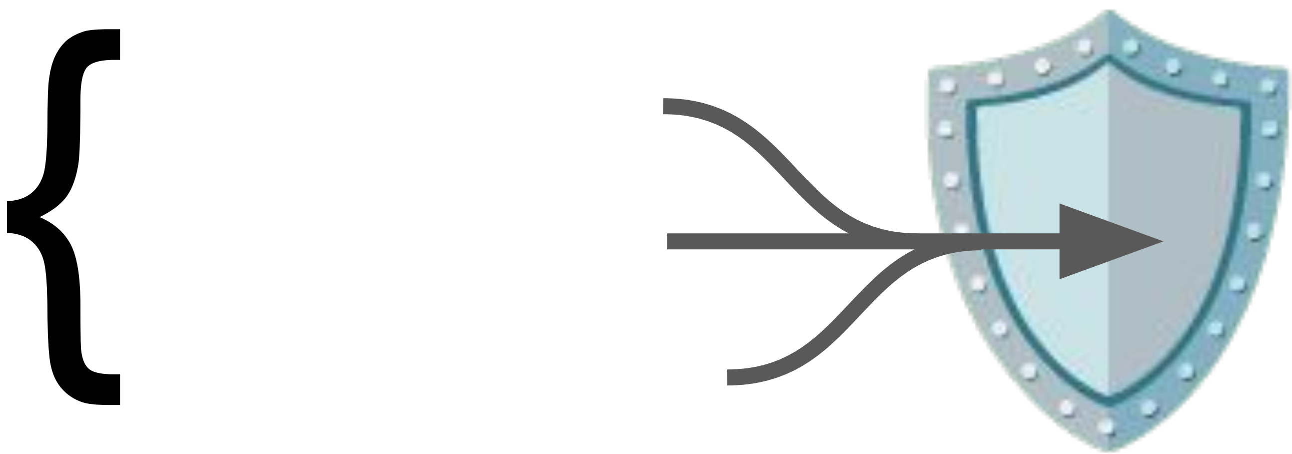
- **Idea: hybrid of PBR and EBR**
  - EBR (fast) at first, ejected to PBR (robust) when blocks are not reclaimed

- **Results**
  - As fast as EBR (85%-90% throughput)
  - Robust as PBR (guaranteeing reclamation of blocks)
  - Portable, generally applicable to many data structures, compact
  - **The first scheme that satisfies all the criteria above (i.e. superior)**

- We will learn PBR and EBR (as background), and PEBR

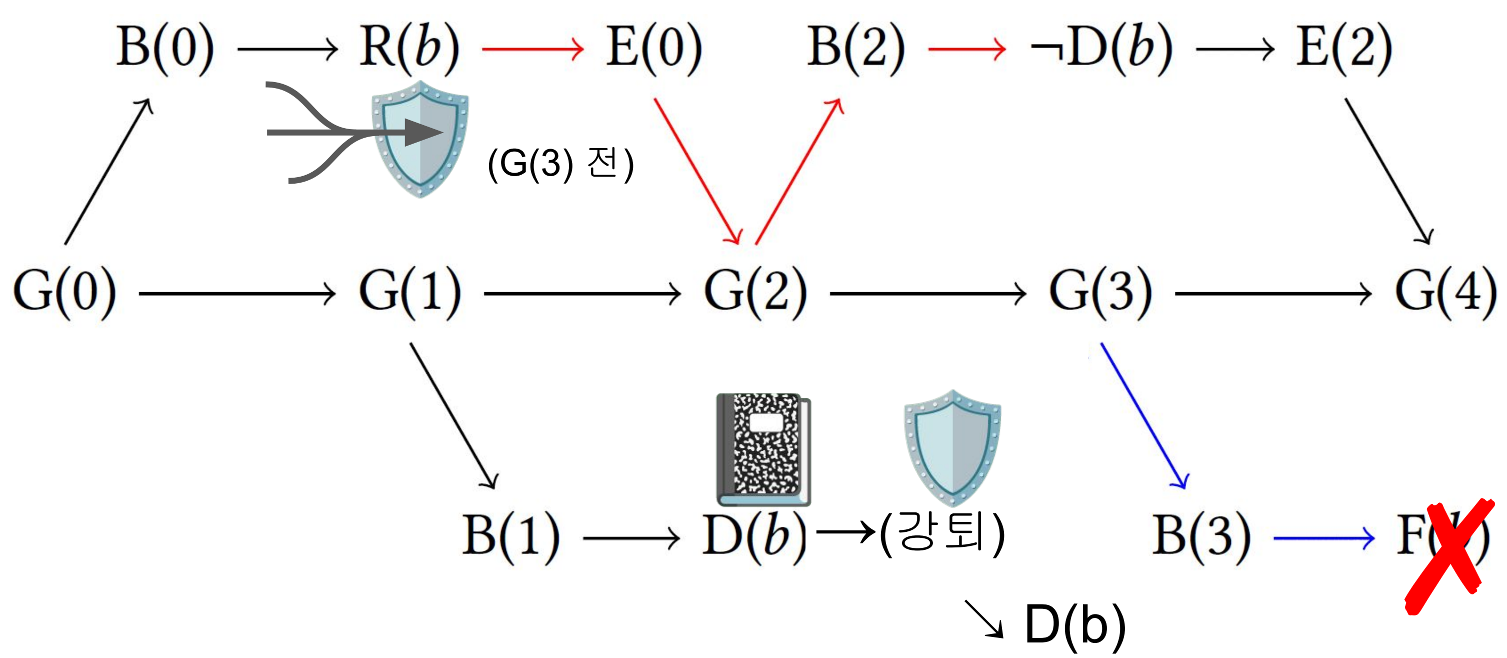
포인터/시대 기반 수집기법 (PEBR)

- **아이디어 :** PBR과 EBR의 혼종



- 처음에는 EBR (빠름), 메모리 수집이 잘 안되면 PBR로 **강퇴** (수집 보장)

포인터/시대 기반 수집기법 아이디어: 강퇴



- EBR에 기반, 묶음이 너무 길어지면 강퇴하고 새 시대 시작
- 강퇴시 묶음에서 접근한 포인터 다시 접근 가능, G(3) 이후에도 해제하면 안됨
- 접근한 포인터를 (쓰레드 내에) 기록해뒀다가📓 강퇴시 보호🛡️

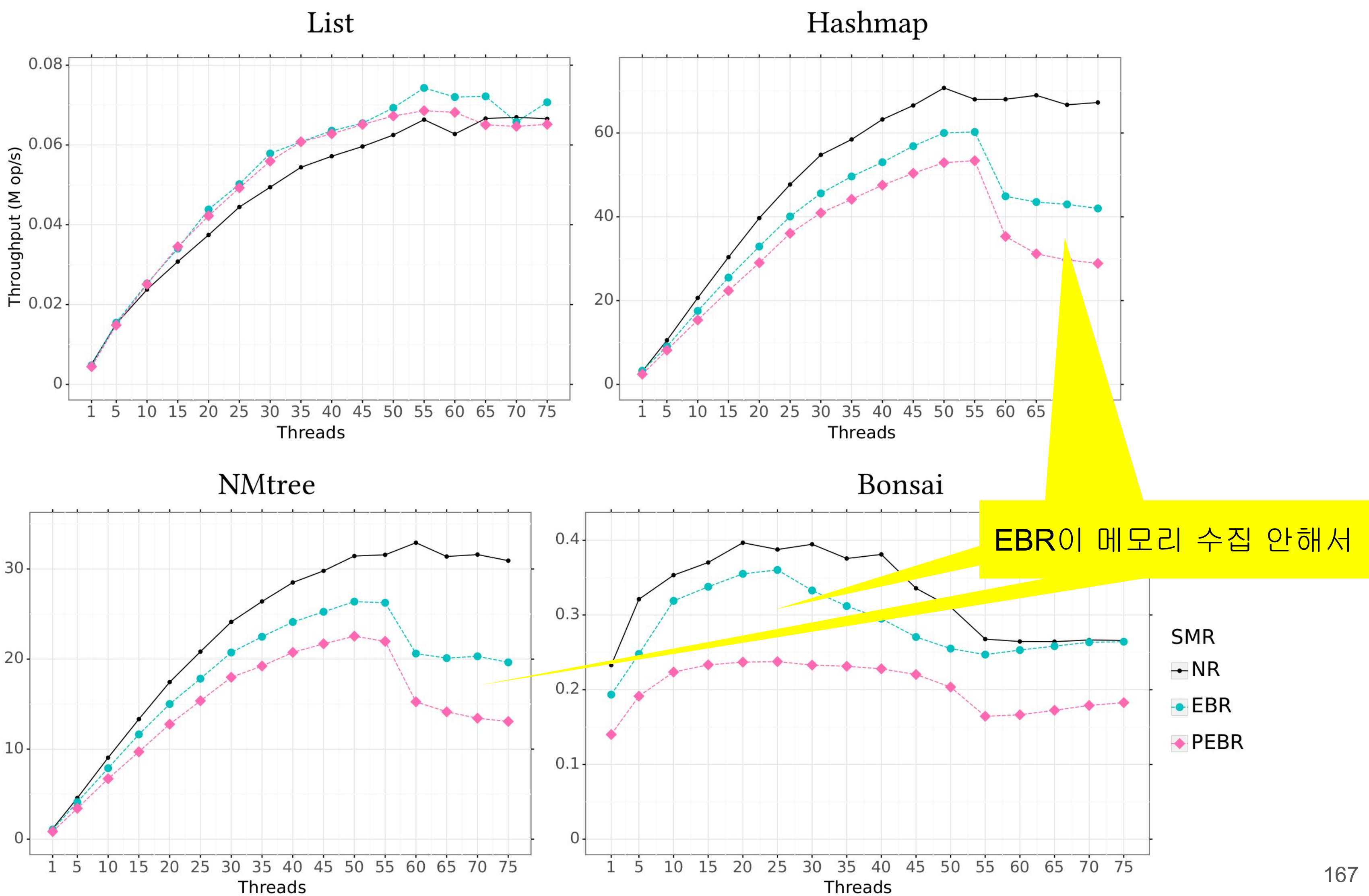
포인터/시대 기반 수집기법 장단점

- **장점 1: 빠르고 메모리 수집 보장 (강퇴를 통해)**
- **장점 2:** 강퇴 과정이 락없고 이식성 높음
- **장점 3:** 포인터를 오래 들고있어야 하는 경우 고의강퇴 가능

- **단점 1:** 접근한 포인터 기록📓하는 비용 (10-15%)
- **단점 2:** 프로그래밍하기 PBR/EBR보다 약간 복잡함

장점 1이 모든 단점을 커버하고도 남음

포인터/시대 기반 수집기법 결과 1: Throughput

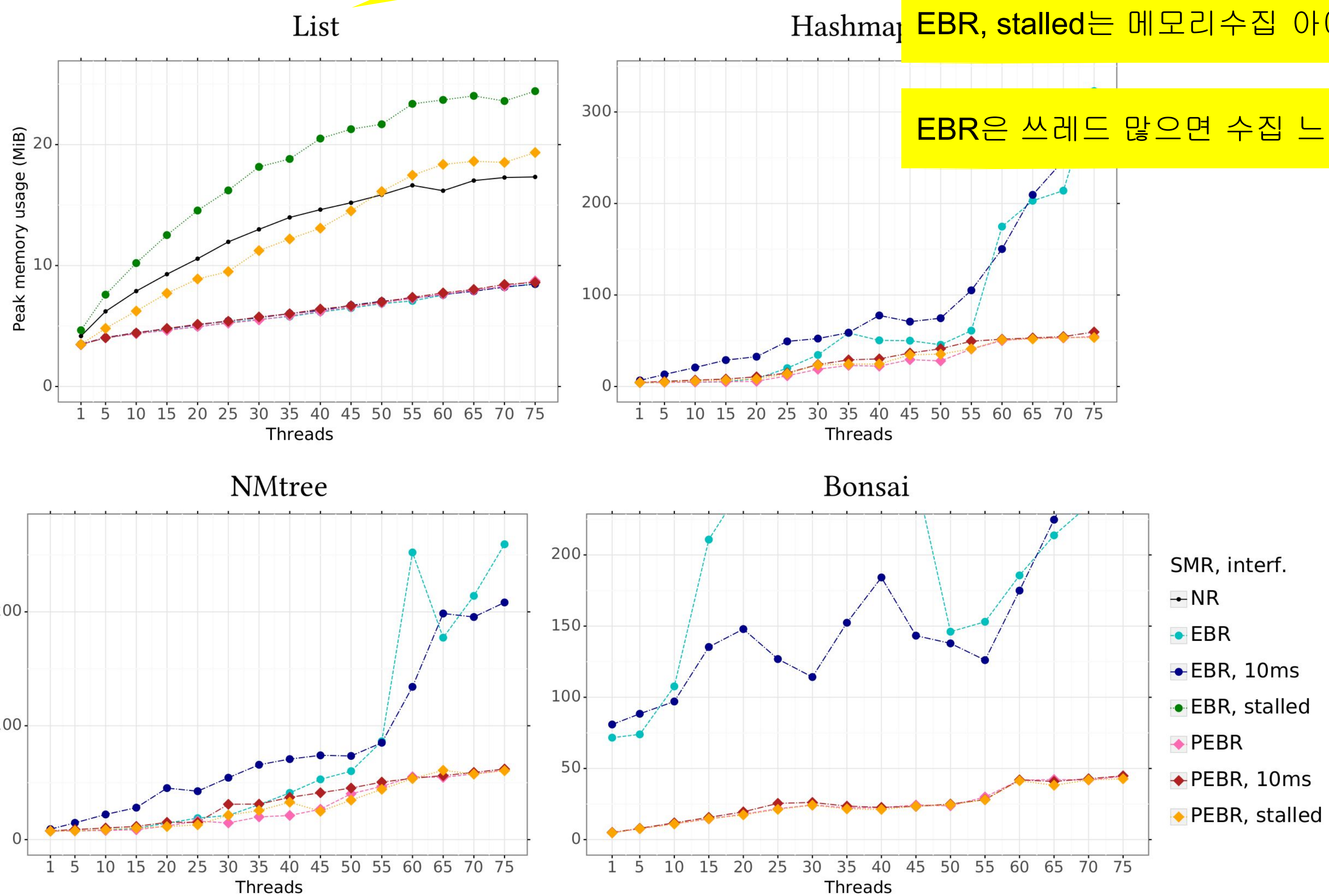


포인터/시대 기 메모리 사용량

Throughput이 적어 경향성 파악 실패

EBR, stalled는 메모리수집 아예 안함

EBR은 쓰레드 많으면 수집 느려짐



Synchronization in HP: use-after-free never happens

- Suppose T1 retires and reclaims a block B, and T2 protects and uses B.

- **Case 1: T2's P2 happens before T1's R2**
  - P1's write to PL is visible to C2
  - To reclaim B, T1's C2 should acknowledge the removal of B from PL (T2 finishes accessing B and then remove B from PL)
  - T2's access to B happens before T1's reclamation of B

- **Case 2: T1's R2 happens before T2's P2**
  - P3's validation fails because it cannot read B from the memory
  - T2 cannot access B

Analyzing shared mutable states with ownership

- **Application to concurrency:** fit for analyzing synchronization among threads
  - E.g., if thread A owns a buffer, thread B cannot access it

- **Its ownership type is an extremely useful abstraction for SMS**
  - **Statically proving the safety** of accesses to SMS
  - Not only for sequential but also for concurrent programs

- **Key idea 1: disallowing shared mutable operations (SMOs) by default**
  - A resource is either exclusive or immutable (but not both)
  - **Exclusive:** resource is read/written by its designated **owner** or its exclusive **borrower**; or
  - **Immutable:** resource is read by its shared borrowers

- **Key idea 2: allowing SMOs in a controlled way**
  - Motivation: SMOs are essential and unavoidable (e.g. in concurrency)
  - Mechanism: **interior mutability** via **unsafe** blocks

Rust and concurrent programming

- Implementation w/ SMOs
  - Essential in concurrent programming

- Interface virtually w/o SMOs
  - For the ease of concurrent programming

- E.g. concurrent stack w/ non-SMO interface
  - fn Stack::push(**&self**, value: T);
  - fn Stack::pop(**&self**) -> option<T>;

- Using compile-time lifetime as a static verification of that of the critical sections (more later)

Bonus: foundation of Rust's ownership type

- **Layer 3: ownership type**
  - Computational checker / mathematical "library" for proving the safety of accesses to SMS
- **Layer 2: lifetime logic**
  - Mathematical library for proving the safety specialized for lifetimes and borrows
- **Layer 1: concurrent separation logic (CSL)**
  - Mathematical proof system for proving the safety
  - Ownership type checker as a "lemma" for the safety proof in CSL
  - We'll see more later...

Q: why are you so narcissistic…?

- This course deals with work by myself only.

- **Reason 1: it's the material I can teach at my best**
  - I know the in and out of the work, including pitfalls and key points

- **Reason 2: I want to deliver a coherent narrative about concurrency**
  - No existing such a coherent narrative
  - I needed to build up my own self (semantics, design patterns, library, …)
  - This course: a first attempt