

KAIST CS431: 并发编程

讲师：姜济雄 (<https://cp.kaist.ac.kr>)

物流

- 主页: <https://github.com/kaist-cp/cs431>
 - 仔细阅读 README.md!
 - 在问题追踪器中发布公告和提问 (请关注仓库)
 - 办公时间: 周五 9:15-10:15am
- 作业和出勤: <https://gg.kaist.ac.kr/16>
- 荣誉准则: 签署KAIST CS荣誉准则。
- Grading
 - 作业 & 项目: 60%
 - 期中 & 期末考试: 40%
 - 出勤: ?

大型语言模型政策

- 大型语言模型 (LLMs): [ChatGPT, ...](#)
 - 我们假设我们都可以使用 ChatGPT 3.5。
- 你可以使用 LLMs 做作业、学习、 ...
 - ChatGPT 4.0 对做作业帮助不大。
 - ChatGPT 3.5 对学习 CS431 材料帮助很大。
- 你不能使用 LLMs 参加考试。
- 我们将调查您在本课程中关于大型语言模型 (LLMs) 的体验。

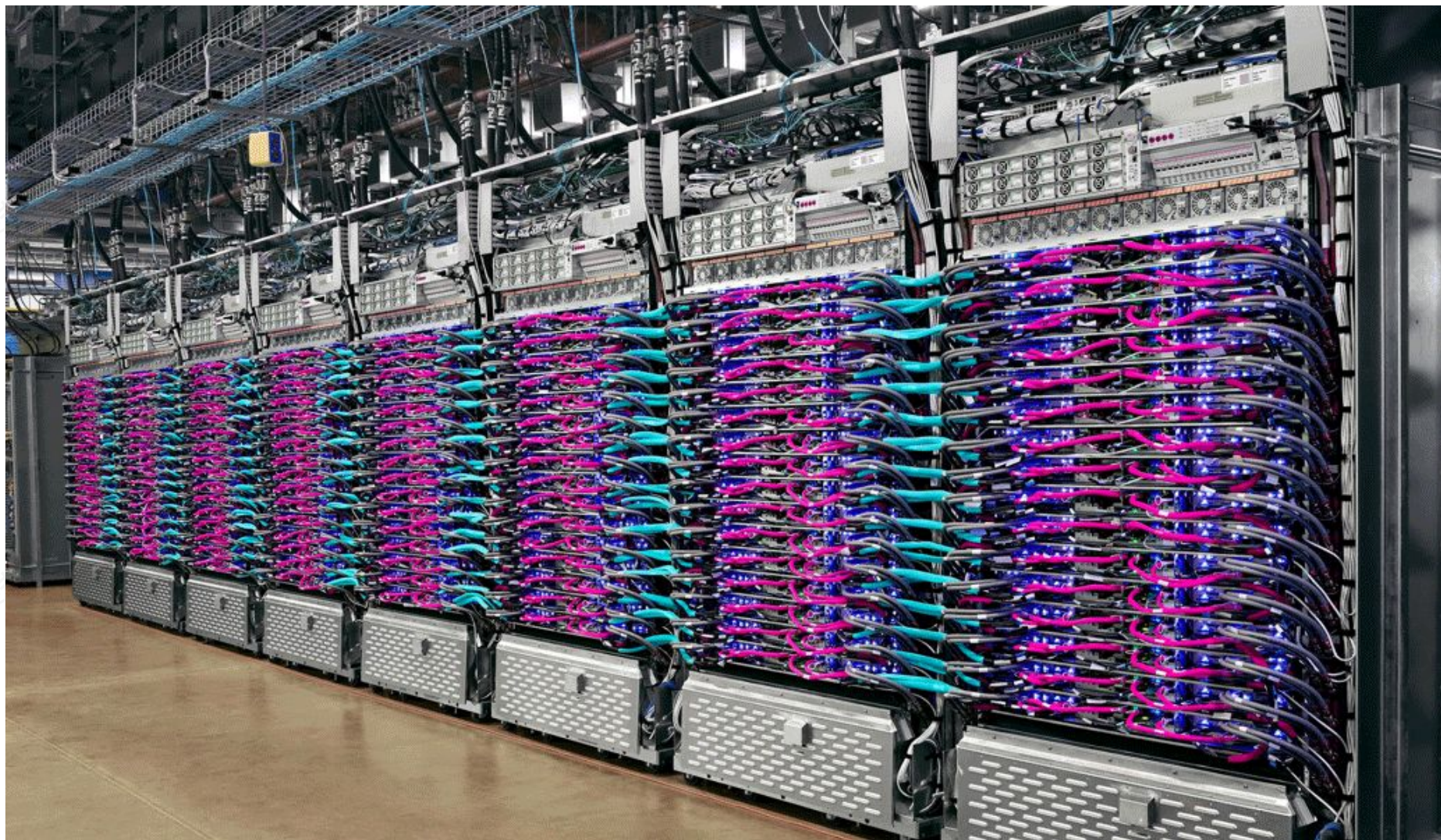
引言



Google DeepMind Challenge Match

8 - 15 March 2016





并行性时代

- **背景：我们需要进行越来越多的计算**
 - 尤其是在人工智能/物联网时代
- **趋势：计算机变得越来越并行**
 - 并行资源：CPU、内存、I/O、...
- **原因1（2005年）：德恩纳德缩放结束**
 - 不能提高电路的频率
 - 突破：多核系统
- **原因 2 (2018)：摩尔定律的终结**
 - 不能在固定面积下增加晶体管的数量
 - 突破：加速器（专用、高度并行硬件）
- **如何协调并行资源以实现更高性能？**

并行计算，理论与实践



并发：同步并行资源

- 并行性：多个资源

- 并发：共享可变资源（状态）

- 例如 CPU、GPU、内存、服务器、数据库、数据中心、 ...
- 并行性：并发 = 썬뻡：팔소

- 共享不可变资源：常量

- 独占可变资源：顺序

|

- 示例1：锁保护的inode

- inode：文件系统元数据
 - 序列化多个线程的文件访问

- 示例2：无锁哈希表

- 确保多个线程并发读写时的正确性

并发中的挑战：非确定性

- **Ch挑战：组合爆炸性非确定性**

- **源 1：交错**

- 例如，“ $X=1 \parallel X=2$ ”：最终内存取决于执行顺序

- **源 2：硬件/编译器优化**

- 例如，在现代架构中通过重排序可能实现 $a=b=0$ ： $X=1 \parallel Y=1$
 $a=Y \parallel b=X$

- **我们需要驯服非确定性**

控制非确定性方法

● 在安全的API中封装非确定性

- 隐藏过于低级的非确定性（例如，指令交错）
- 同时暴露高级的非确定性（例如，队列操作的交错）
- 大多数人需要理解API，而不是实现
- 例如，锁、条件变量、并发数据结构的安全API

● 在实现中运用同步模式进行推理

- 有人需要实现锁、条件变量、数据结构、...
- 仅使用经过充分研究的同步模式

● 本课程：学习并发库的API和实现

两种并发模式：“简单”和“困难”

● “简单”基于锁的并发

- 锁、条件变量、...
- 优缺点：简单性与低可扩展性（丢失并行性机会）
- 适用性：涵盖大多数 用例（按代码行数计算）

● “困难”无锁并发

- 理论：语义学和推理原则（表征非确定性）
- 工具：同步模式（无锁并发的构建块）
- 实践：无锁数据结构的API和实现（例如，栈、队列、列表、哈希表、基数树、平衡树）

并发编程的一般建议

- “简单”并发是首先要学习的

- 只有在它是瓶颈时才去追求“困难”并发
- “过早优化是万恶之源”

- “困难”并发一旦理解了理论就不那么困难了

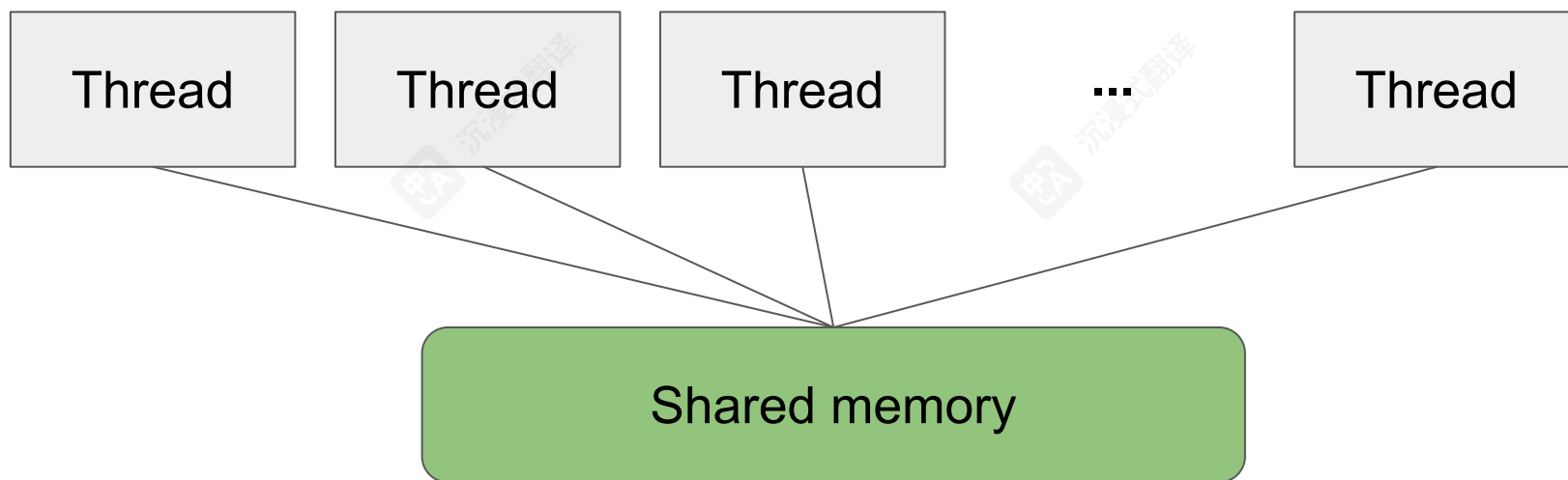
- 换句话说，理解理论可能很困难...
- 关键在于控制适量的非确定性（太多 => 可扩展性问题，太少 => 正确性问题）

- 编码不多，调试太多

- 使用净化器、压力测试、断言、逐行调试，...
- 数学思维和编程语言思维非常有帮助

基于锁的并发第一部分：安全 API的动机

Context: 共享内存并发



- **线程:** 执行代理读取/写入共享内存
- **共享内存:** 数据共享存储
- **其他并发类型:** CPU、内存、GPU/FPGA、持久内存、分布式节点、...

基于锁的共享内存并发

- **定义：**在任何时刻，一个位置被单个代理访问
- **优缺点：**简单且可能低效
- **机制** **m：**锁（在任何时刻，只有一个线程持有loc k)

- **Examples**

○ $r1=X \parallel r2=X \quad X=r1+1 \quad \parallel \quad X=r2+1$ // 不总是 $X=2$ ：不幸的交错产生 $X=1$

○ **L.acquire()** **||** **L.acquire()**
 $r1=X$ **||** $r2=X$
 $X=r1+1$ **||** $X=r2+1$
L.release() **||** **L.release()**
// 现在 $X=2$: 锁防止不幸的交错

ng

为什么锁“简单”？

- 回忆：并发的挑战是非确定性
 - 线程交错
 - 指令重排
- 锁约束线程交错为获取/释放点
 - 获取和释放之间没有交错
- 锁移除指令重排
 - 线程A的释放严格发生在线程B的获取之前
 - 线程被**AS IF**执行为同一个线程。
- 锁将非确定性降低到最小。

基于锁的并发低级API

- **Lock.acquire():** 阻塞直到获取锁。
- **Lock.try_acquire():** 返回是否获取了锁。不阻塞。
- **Lock.release():** 释放已获取的锁。
- **挑战:** 该API极易出错。
 - **关联锁和资源:** 用户应 **Id access X only when L is held.**
 - **Matching acquire/release:** users should release only acquired locks.
- **后果:** 该 API 产生高成本。
 - **注意:** 程序员应该始终关注该 API。
 - **潜在错误:** 通常存在许多未解决的错误。

基于锁的并发的高级API

- 我们想要一个简单易用、始终安全的高级API。
 - 获取/释放自动匹配。
 - 锁和资源明确相关。
- 高级API的优点：低成本，注意力少且错误少
 - 注意力少：程序员无需担心API 误用。
 - 减少错误：使用该API的程序出现错误的可能性极低。
- 高级API的设计（源自C++/RAII）
 - LockGuard：使用RAII类型自动释放锁
 - Lock.acquire() 返回“锁保护”
 - 当锁保护被销毁时，相应的锁会被释放。
 - 锁<T> (= (锁, T))：将一个锁和一个资源与一个新的类型关联起来

基于锁的并发安全（-ish） API：锁保护

● 锁保护：持有锁

- https://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';
}
```


基于锁的并发安全 (-ish) API: 锁定数据

- **Locked data: a pair of lock and data**
- **Benefit: API manda测试内部 DATA 是否由一个 loc (相当) 安全地保护** k
- `// 类型template<typename T> class 锁<T> { RawLock 锁; T DATA; }`
- `// 获取并创建锁保护LockGuard<T> 锁
<T>::lock(this) { this->锁.acquire();
锁保护 { this } }`
- `// 从锁保护中解引用数据&T LockGuard<T>::
operator->(this) { &this->0.DATA }`
- `// 守卫丢弃时自动释放LockGuard<T>::~~LockGuard() {
this->0.lock.release(); }`

基于锁的并发安全 (-ish) API: 不安全!

● `// data: 锁<int>auto data_guard = lock(); auto data_ptr = (int *) &data_guard; ... // data_guard被丢弃, 锁被释放*data_ptr = 666; // 不安全!`

● 根本原因: `data_ptr` 不应比 `data_guard` 存活时间更长

● 易出错的: 在生产代码中发生, 导致很多麻烦

● 解决方案: 基于 Rust 的类型系统的所有权和生命周期

○ <https://github.com/kaist-cp/cs431/blob/main/src/lock/api.rs>

○ Rust的锁实现有 被证明是安全的 API

○ => Let's first study Rust and then resume studying concurrency cy

基于锁的并发Part 2: foundation for safe API in Rust

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

Rust: 安全系统编程语言

- **Motivation: achieving safety & control at the same time**
 - 安全性：编译的程序不会出错
 - 控制：语言支持低级特性
 - 现有技术：C/C++ (不安全)
- **最适合本课程的：**所有权和生命周期抓住了并发性的本质
- **阅读作业：**
 - 读取 [书籍](#)。作业1是关于 [书籍的最终项目](#)。
 - 读取 [Rust by example](#)。
- **编程作业将使用Rust**
 - 在 [提供的服务器](#)上设置编程环境。

Rust 示例1: 迭代失效 (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&版本=2018&代码片段=08f5870c40f7afdfd7a2fab9d7815f9f>

- 此代码将在 C++ 中编译，但在运行时可能会失败

- "v.push(4)" 可能会重新定位 "v", 使 "p" 失效

- 此代码不会在 Rust 中编译

- 不能借用 `v` 作为可变的因为它也被借用为不可变"

Rust 示例1: 迭代失效 (2/3)

```
fn main() {  
    let mut v = vec![1, 2, 3];  
  
    let p = &v[1];  
    v.push(4); println!("{}",  
    v[1]: {}", *p); }
```

- “v” : 向量的所有者
- “p” : 从 “let p = ...” 到 “println!(...)” 进行不可变借用
- “v.push(4)” : 为该行进行可变借用
- 编译失败, 因为类型检查器检测到对向量的共享可变访问 (SMA)
 - “p” 和 “v.push”
- 这正是它可能出错的原因
- Q: 如何检测SMA?

Rust 示例1: 迭代失效 (3/3)

```
fn main() { let v =  
vec![1, 2, 3]; let p = &  
v[1]; v.push(4);  
println!("v[1]: {}",  
*p); }
```

- 计算每个所有者/借用者的“生命周期”
 - “v” : L1-L5
 - “p” : L3-L5
 - “v.push” : L4
- 列出所有重叠的生命周期的配对
 - “v” 和 “p” (L3-L5), “v” 和 “v.push” (L4)
 - “p” 和 “v.push” (L4)
- 删除借用方 & 借用者的配对
 - “p” 和 “v.push” (L4) 仍然存在
- 删除不可变借用的配对
- 剩余的配对被 视为 SMA 的
- 静态安全: 在编译时检测所有SMA
- 不完整: 并非所有对都是SMA

所有权用于分析共享可变访问

- **“Ownership”: the a (代理的) 访问和销毁资源的能力**
 - **独占**: 如果我拥有一个资源, 其他人就不能拥有它
 - **可借用**: 可变地被单个代理借用或 不可变地被多个借用
 - **适合并发**: 每个线程都是一个代理
- **执行纪律**: 对资源没有共享可变访问 (默认情况下)
 - **静态**: 所有权纪律由类型强制执行
 - **易于使用**: 编译器将报告所有纪律违规行为
 - **正确**: 如果经过类型检查, 程序就不会出错
- **弯曲纪律 with “内部可变性”**
 - **必要**: 在并发中, 共享可变访问是不可避免的
 - **模块化**: 将实现封装在安全API中 (“好像没有共享可变访问”)

Rust示例2: RefCell (1/4)

- **Context:** it is unr对于并发等场景，完全禁止SMA的做法并不现实
- **解决方案:** 内部可变性
 - 将SMA封装在安全的API中 **仿佛** 没有SMA存在
- **示例:** RefCell<T>
 - 在运行时检查所有权（而非编译时）
 - RefCell<T>::try_borrow(), RefCell<T>::try_borrow_mut(): 尝试借用内部值，不可变或可变（分别）

KAIST CS431: 并发编程 Rust示例2: RefCell (1/4)

<https://doc.rust-lang.org/book/ch15-05-interior-mutability.htm> |

Rust 示例 2: RefCell (2/4)

函数 f1() 返回布尔值 > {true}

函数 f2() 返回布尔值 > {!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>

- 假设 f1() 和 f2() 是复杂且互斥的条件 (不是 f1() && f2())
- 安全的, 因为 p1 和 p2 没有别名
- 由于条件复杂, 类型检查器无法推断安全性, 导致编译错误
- “不能借用 `v1` 作为可变的, 因为它也被不可变地借用了”

Rust 示例 2: RefCell (3/4)

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

函数 f1() 返回布尔值 > {true}

函数 f2() 返回布尔值 > {!f1()}

```
fn main() { let v1 = RefCell::new(42);  
let v2 = RefCell::new(666); let p3 = if  
f1() { &v1 } else { &v2 }; .  
try_borrow().unwrap(); if f2() { let  
mut p4 = v1 .  
try_borrow_mut().unwrap(); *p5;  
println!("p2: {}", *p2); } println!("p1: {}",  
*p1); }
```

● <https://play.rust-lang.org/?version=stable&mode=debug&edition=gist>

- 所有权在运行时进行检查
(try_borrow(), try_borrow_mut())
- 编译并按预期执行 “p1: 42”
- 如果 f1() && f2(), try_borrow_mut()
在运行时失败 (不是编译时)
- “线程 'main' panic 在 '调用`
Result::unwrap()` 在 `Err` 值上”

Rust 示例 2: RefCell (4/4)

- **Inter借用可变性:** 将 SMA 封装在非 SMA 类型中
- **安全 API:** 几乎没有 SMA
 - `pub fn try_borrow_mut(&self) -> Result<RefMut<T>, BorrowMutError>`(不可变地借用 self)
- **潜在的不安全实现:** 使用 SMA
 - KAIST CS431: 并发编程 Rust示例2: RefCell (4/4) ○ ... 不安全 { &mut *self.value.get() }, ... (<https://doc.rust-lang.org/1.63.0/src/core/cell.rs.html#1732>)
 - “不安全”: 无SMA的API与有SMA的实现之间的桥梁
(需要手动检查, 应明确注释)

Rust 示例 3: 锁

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

```
102 impl<'s, L: RawLock, T> Deref for LockGuard<'s, L, T> {  
103     type Target = T;  
104  
105     fn deref(&self) -> &Self::Target {  
106         unsafe { &*self.lock.data.get() }  
107     }  
108 }
```

● // data: 锁<int>let data_guard = data.lock(); let data_ref =
data_guard解引用(); ... drop(data_guard); // 锁已释放*data_ref = 666; //
NOT COMPILED: 解引用目标不应比守卫存活更久

Rust 的所有权类型总结

- **Motivation:** achieving **safety & control** over shared mutable resources
- **关键思想:**
 - **纪律:** 默认禁止共享可变访问
 - **内部可变性:** 以受控方式允许
- **优点:**
 - 静态分析共享可变访问的安全性（适用于序列和并发程序）
 - 明确标记需要手动检查的代码
- **我们将做什么:** 使用安全API理解基于锁的并发

基于锁的并发Part 3: 安全 API

Rust 并发库 (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
 - 作用域线程: 限制线程生命周期在作用域内 e
 - Motivation: safe sharing of non-'static data
 - Safety: thread should be joined before the scope ('s) ends
 - Arc: 引用计数器, 不可变地共享数据于多个线程
 - 安全性: Deref, 而非 DerefMut
 - Send: 可传递到其他线程
 - 实现者: usize, &usize, Arc<T>, &Arc<T> (但非 Rc<T>, &Rc<T>)
 - 同步: 可从多个线程并发访问
 - 实现者: usize, Arc<T> (但不是 Rc<T>)
 - 属性: 'T: 同步' 当且仅当 &T: 发送

Rust 并发库 (2/3)

- **CS431 Lock API**: 一个安全的锁 API (具体实现见下文)
- **锁<L: RawLock, T>**: 拥有受 L 锁保护的 T
 - 保证: T 对象不会被并发访问 (非代码区域)
 - 示例: Lock<SpinLock, Vec<usize>>, Lock<ClhLock, &' t TLS>
 - 属性: 如果 T 是 Send, 则 Send + Sync (仅当 T 是 Send 时有意义)
- **Lock守卫<'s, L: RawLock, T>**: 证明锁已被获取
 - 保证: 锁被持有, T 可通过 Deref/DerefMut 访问
 - RAII: 丢弃时释放锁
 - 属性: 若 T 是 Send, 则为 Send; 若 T 是 Sync, 则为 Sync (即透明访问器)
- API 的保证/安全性是相对于 Rust 的所有权类型系统 (与 C/C++ 相反) 被证明的

Rust 并发库 (3/3)

- **More std**

- 互斥锁: 带有多重策略的互斥
- 条件变量: 条件变量, 等待一个事件 (条件)
 - 安全性: `Condvar::wait()` 获取 `&mut` 守卫, 禁止重用受保护的数据
- 读写锁: 读写 `Read-Writer` 锁, 允许多个读者 **或** 一个写者

ta

- **crossbeam**

- 通道: 在线程之间发送/接收值 缓存对齐: 与 128 字节对齐
- 节对齐
 - 动机: 为了击败 “false sharing”

- **rayon**

- into_par_iter: 并行地为每个元素执行一个函数
 - 动机: 并行性变得简单

基于锁的并发Part 4: 实现

几种锁实现

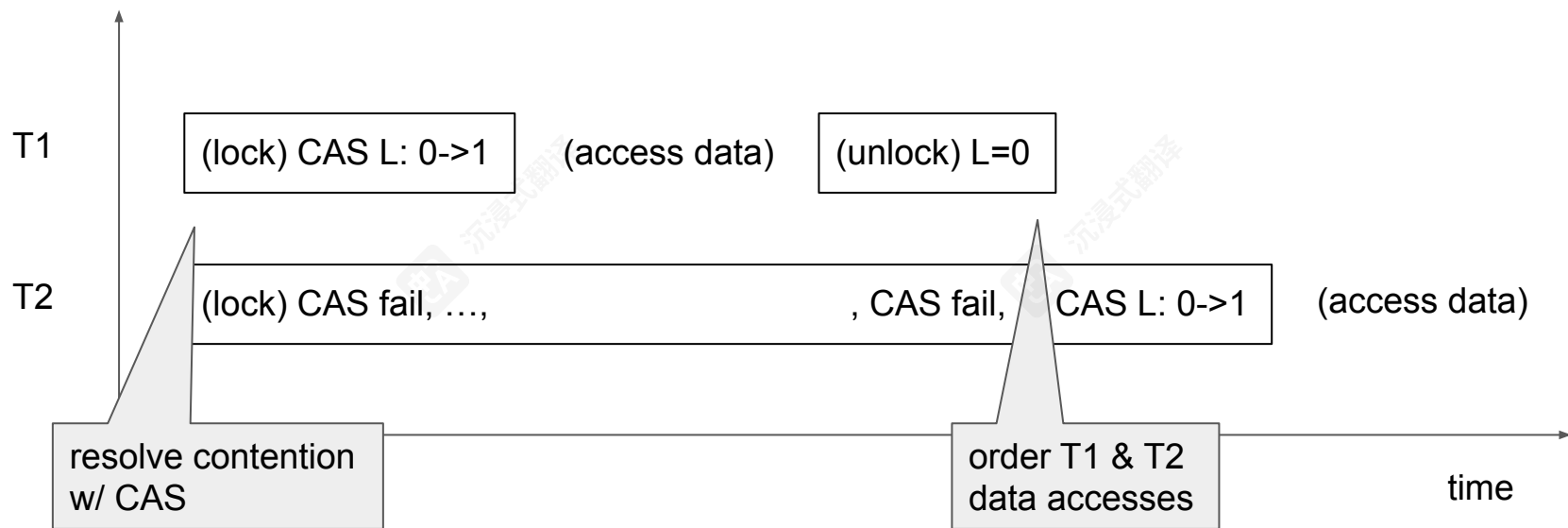
- **Implementations:** <https://github.com/kaist-cp/cs431/tree/main/src/loc>票 k
 - Spinlock, 锁, CLH锁, MCS锁, MCS停车锁
- **RawLock特征:** 定义了“原始”锁的API
 - **APIs:** lock(), unlock(), Token, 默认, 发送, 同步**保证:**
 - 一次只有一个代理获取**锁实现者:** 自旋锁, 票锁, CLH锁, ...
 -
- **锁之间的权衡**
 - 简单, 快速 (当无争用时), 紧凑, 可扩展, 公平, 节能, ...

自旋锁实现

- <https://github.com/kaist-cp/cs431/blob/main/src/lock/spinlock.rs>(目前, 忽略内存排序: acquire, release, ...)
- ```
pub struct RawSpinLock { inner: AtomicBool, // true 表示已锁定, false 表示未锁定 }
pub fn lock(&self) { while self.inner.compare_and_swap(false, true).is_err() {} // rmw }
pub fn unlock(&self) { self.inner.store(false); // 不是 rmw, 感谢锁的独占性 }
```

# 自旋锁正确性

- 公有的函数 `lock(&self) { while self.inner.cas(false, true, 获取).is_err() {} }` 公有的函数 `unlock(&self) { self.inner.store(false, 释放); }`
- 如果一个锁已经被获取, `lock()` 将自旋。
- 一次只有一个线程可以持有锁 (见下文) 。



# 其他锁的关键思想

- <https://github.com/kaist-cp/cs431/tree/main/src/锁>
- 使用CAS保证互斥
  - 从临界区的末尾到另一个临界区的开头进行排序
  - ``curr`` 在票锁中，每个等待者在CLH/MCS锁中的新位置
- 通过排序和不同位置等待来保证公平性
  - 使用公平指令（例如交换、获取并添加）的排序
  - 票锁：使用 ``next`` 进行排序，并使用 ``curr`` 进行等待
  - CLH/MCS锁：使用 ``tail`` 进行排序，并使用 **新位置** 进行等待
- 作业：推理锁的正确性

# 锁权衡

- **票锁**：通过票队列保证公平性
  - 锁顺序由公平指令（fetch-add 或 swap）事先决定
  - 缺点：一个稍微复杂的API（返回票）
- **CLH锁**：通过使用每个临界区的自旋位置提高可扩展性
  - 自旋位置队列
  - 缺点： $O(n)$ 空间开销，其中 $n$ 是临界区的数量
- **MCS锁**：通过在自分配位置自旋来感知NUMA
  - 缺点：unlock()中可能存在额外的比较交换
- **MCS停车锁**：通过线程停车来减少能耗
  - 线程停车（故意阻塞）而不是自旋
  - 缺点：在适度争用情况下牺牲性能
- [一篇关于性能评估的论文](#)

# 锁问题

- In 票锁，为什么发行新票可以宽松？
  - 在CLH锁中，为什么交换尾部可以宽松？
  - 在MCS（停车）锁中，为什么交换尾部可以宽松？
- 在MCS停车锁中，未停车后，为什么检查是否解锁？
- 在MCS停车锁中，为什么`线程`被克隆？
- 文献中还提到了什么？
  - 可重入锁：嵌套调用 lock() 读写锁：允许多个读者或单个写者 分层锁：
  - 在单个 NUMA 节点中组合请求 退避策略：旋转？让出？停车？指数退
  - 避？条件变量：除了互斥外还保证顺序
  - [https://docs.rs/parking\\_lot/0.11.0/parking\\_lot/](https://docs.rs/parking_lot/0.11.0/parking_lot/) (Rust锁实现) ...
  - 
  - 
  -

# 基于锁的并发Part 5: 细粒度锁

# 动机：粗粒度锁不可扩展

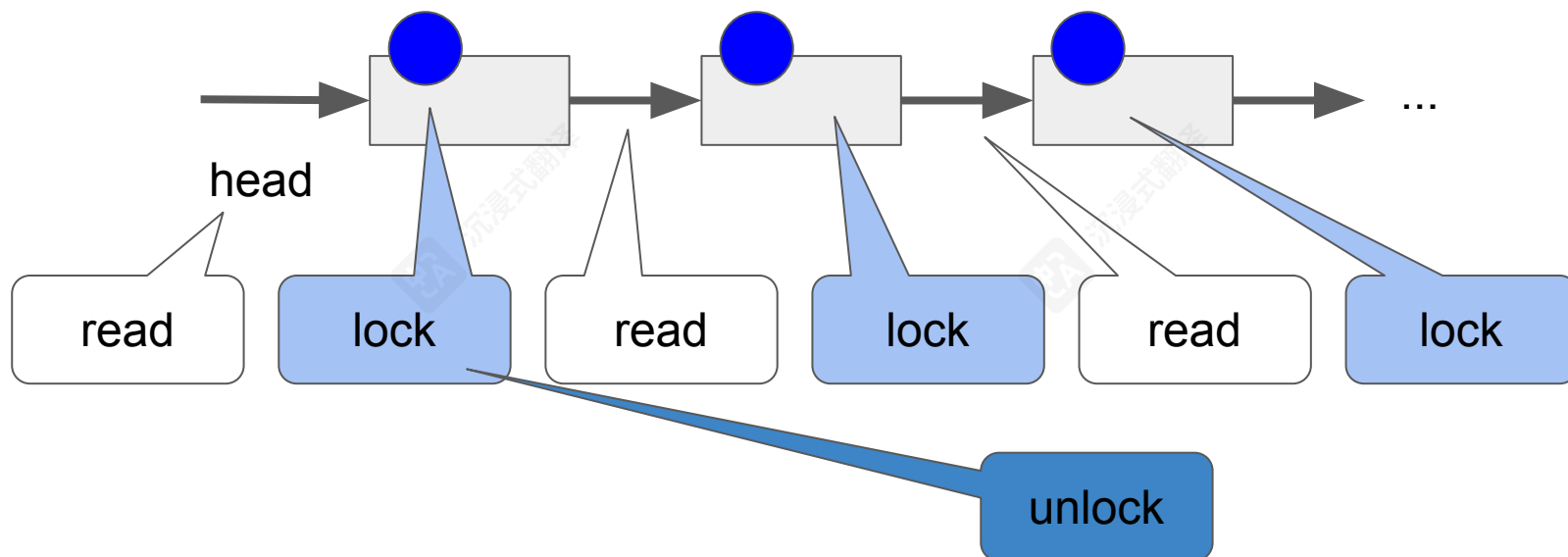
- **Coarse-grained locking:** protecting a large object w/ a single lock
  - 简单：通往并发编程的直路
  - 不可扩展的：对象的所有访问都序列化，无并行性。
- **细粒度锁：**用单独的锁保护许多小对象
  - 更可扩展的：访问被“分布”到多个锁（理想情况下，最小的同步开销）
- **细粒度锁的缺点（与粗粒度锁相比）**
  - 单线程开销（理想情况下，只有适度的）
  - 复杂性
- **关键设计问题**
  - 应该按顺序做什么？ -需要> 锁
  - 可以并行做什么？ -不需要> 锁

## 示例：锁耦合链表（结构）

- 每个节点都有一个锁
- 在访问“下一个”指针时，应持有锁
  - 节点的锁保护该节点的下一个指针

获取 next 节点      在释放当前（“手递手”）之前，先获取e的锁

- 确保当前节点不是分离的

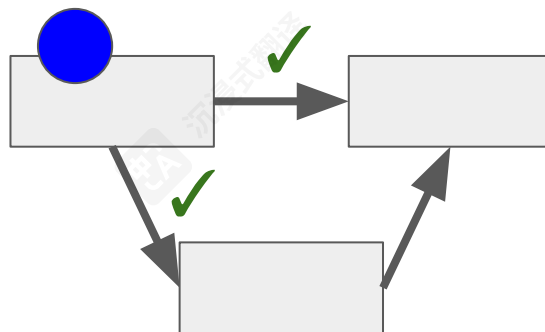




# 示例：锁耦合链表（操作）

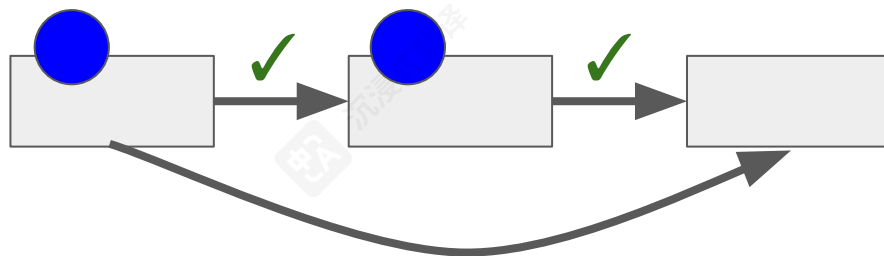
## ● 在 A 和 C 之间插入 B

- 获取 A 的锁
- 读取 A.next (=C)
- 分配 B with B.next=C
- 写入 A.next = B



## ● 删除 B 在 A 和 C 之间

- 获取 A 的锁
- 读取 A.next (=B)
- 获取 B 的锁
  - 为了确保 B.next 在列表中
- 读取 B.next (=C)
- 写入 A.next=C
- (释放 B 的锁和) 释放 B



# 基于锁的并发Part 5: managing multiple locks w/ BoC

# 死锁和活锁错误

## ● 死锁：每个线程被另一个线程阻塞

○ 例如，A 持有 L1，获取 L2 B 持有 L2，获取 L1

○ 对策

e: 检测死锁并终止一个操作（事务

n)

## ● 活锁：操作不断被终止而没有实质性进展

○ A: L1 -> L2 -> L3

B: L2 -> L3 -> L1

C: L3 -> L1 -> L2

○ B持有L2，C持有L3/L1 -> C被杀死 ->A持有L1，  
B持有L2/L3 -> B被杀死 ->C持有L3，A持有L1/L2 ->  
A被杀死 -> ...

如何与它们对抗？

# 死锁和活锁避免

- 顺序锁：L1、L2、L3

- 例如，根据它们的指针地址

- 按顺序获取锁

- 例如，A 持有 L1，获取 L2 B 持有 L2，获取

- L1 => NO!!!**

- A: L1 -> L2 -> L3 B: L2 ->  
L3 -> L -1 => NO!!! C: L3 ->  
L1 -> L2 => NO!!!

- 定理：无死锁/活锁发生

- 证明概要：持有最大锁的线程将不会被阻塞

# BoC: 面向行为并发 (基础)

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

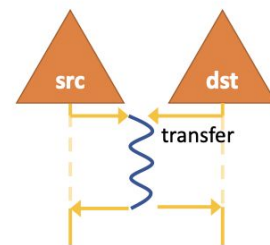
- 动机

- 自动排序锁
- 锁基础同步与线程基础并行化的耦合 (cf: 高级锁API中的同步与权限耦合)=> 任务依赖的自然表示

- 示例

List. 5. Spawn a behaviour that requires both accounts

```
1 transfer(src: cown[Account], dst: cown[Account], amount: U64) {
2 when (src, dst) { // withdraw and deposit
3 if (src.balance >= amount && !src.frozen && !dst.frozen) {
4 src.balance -= amount;
5 dst.balance += amount;
6 } } }
```

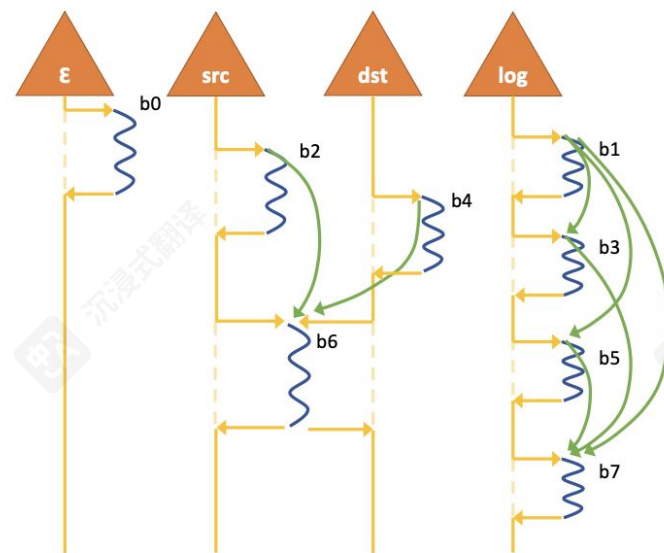


# BoC: 面向行为并发 (语义学)

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

List. 8. Creating an accurate log

```
1 main(src: cown[Account], dst: cown[Account],
2 log: cown[OutStream]) { /* b0 */
3 when(log) { /* b1 */ log.log("begin") }
4 when(src) { /* b2 */ ...
5 when(log) { /* b3 */ log.log("deposit") }
6 }
7 when(dst) { /* b4 */ ...
8 when(log) { /* b5 */ log.log("freeze") }
9 }
10 when(src, dst) { /* b6 */ ...
11 when(log) { /* b7 */ log.log("transfer") }
12 } }
```



# BoC: 面向行为并发 (Impl)

- Mutual ex每个Cown的结论（类似于MCS锁的队列）
- 每个行为以固定顺序请求Cown（例如，addr的升序）
- 每个行为通过“两阶段入队” **原子地**获取多个cowns
  - 所有Cown的**开始阶段** => **结束阶段**
  - 一个行为在开始到结束阶段独占访问一个cown
- 没有“两阶段入队”，可能会有死锁...
  - b1, b2 队列为 c1, c2
  - c1 队列: b1, b2; c2 队列: b2, b1
  - => **死锁!**

# 但如果我们想要更好的性能呢？

- **Performance drawbacks of fine-grained locking**

- 写入操作由于锁操作（lock ops）导致同步成本（synchronization costs）很高
- 读取操作也会写入锁（lock），导致缓存失效（cache invalidation）

- 替代方案1：为以读为主的工作负载（read-mostly workloads）使用高级锁

- 读写锁（reader-writer lock）（“Rust 并发库”），  
乐观锁（optimistic locking）（本学期不涉及），...（对于写者仍然不可扩展（unscalable））

- 替代方案2：“无锁”并发数据结构

- 优点：性能更好
  - 写入操作具有轻量级的同步成本
  - 读操作仅执行读取，无需缓存失效
- 缺点：要 ( $\times 100$ ) 比基于锁的并发困难得多

...



# 期中考试

到目前为止你学到的所有内容

# 无锁并发第一部分：定义

# 锁自由并非关于锁的存在

- **What the hell?**

- 它曾经是，但现在已经不是了：这个词的含义随着时间的推移而演变。

- **锁的限制：没有进度保证**

- Q：如果一个线程持有锁并永远睡眠怎么办？ A：所有等待相同锁的线程都会停止前进。

- **锁自由：进度保证**

- 定义：其中一个正在进行的操作最终完成

- 直觉：整个系统的进展是保证的

- 示例：特莱伯栈、迈克尔-斯科特队列、循环缓冲区、工作窃取双端队列、...

- **敌人：由于I/O导致的线程停滞、崩溃、非抢占式调度；锁（死锁和活锁）**

# 其他进展保证

- **阻塞自由：**弱于锁自由

- 定义：如果只运行单个操作，最终会完成
- 直觉：我们总能“恢复”一个对象到一个稳定状态

- **等待自由：**强于锁自由

- 定义：每个正在进行的操作最终会完成
- 直觉：每个线程的进展都会得到保证
- 示例：Yang-Mellor-Crummey 队列, ...

- 无等待  $\subseteq$  无锁  $\subseteq$  无阻塞  $\subseteq$  “非阻塞”

- 参考：Herlihy 和 Shavit. 关于进展的本质. OPODIS'11

---

# 锁自由的关键思想：单指令提交

- 利用锁自由架构保证

- 在一个时刻，至少有一个CPU核心执行它的指令

n.

- 将一个RMW指令指定为操作的提交点

- 定义：原子地从位置读取并写入（例如，CAS）
- 属性：能够表达同步协议（例如，自旋锁）
- 历史：单指令提交是RMW的动机！

- 克服锁自由的“敌人”

- 线程暂停/停滞不关心单个RMW指令。
- 死锁/活锁不是由于缺少锁而发生的。

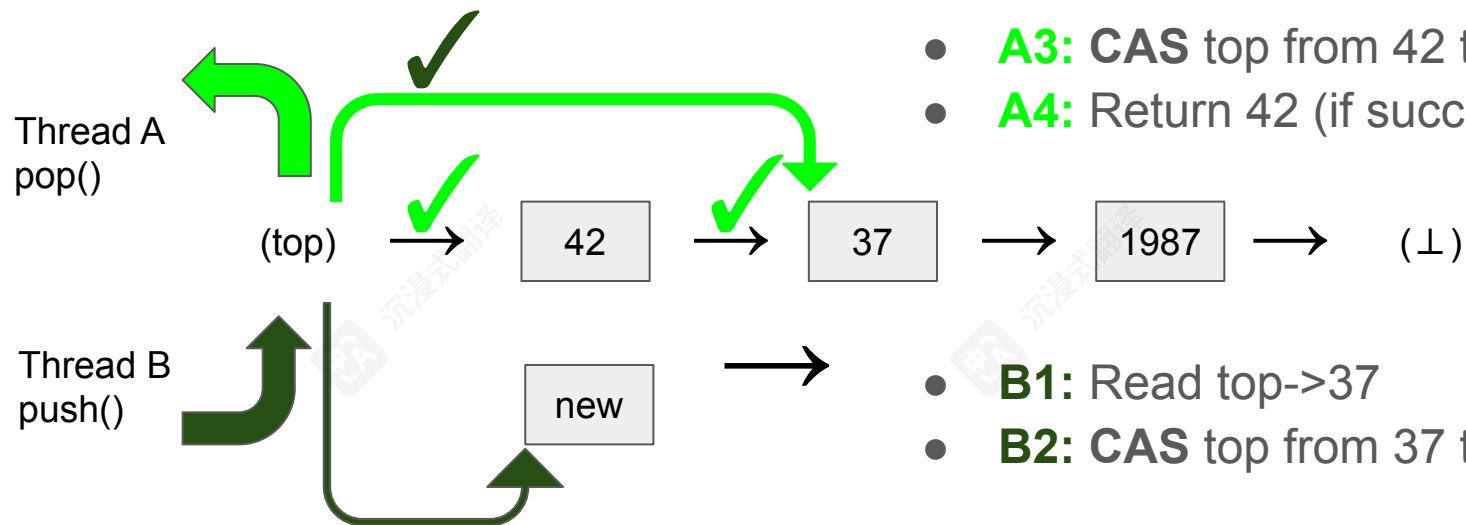
- 副作用：可扩展性（将争用减少为单个指令）

# 无锁并发Part 2: 数据结构

# 示例：特莱伯的栈

● 单链表 w/ 列表头 = 栈顶

● <https://github.com/kaist-cp/cs431/blob/main/src/无锁/栈.rs>



Push/pop 与栈顶位置的 CAS 同步

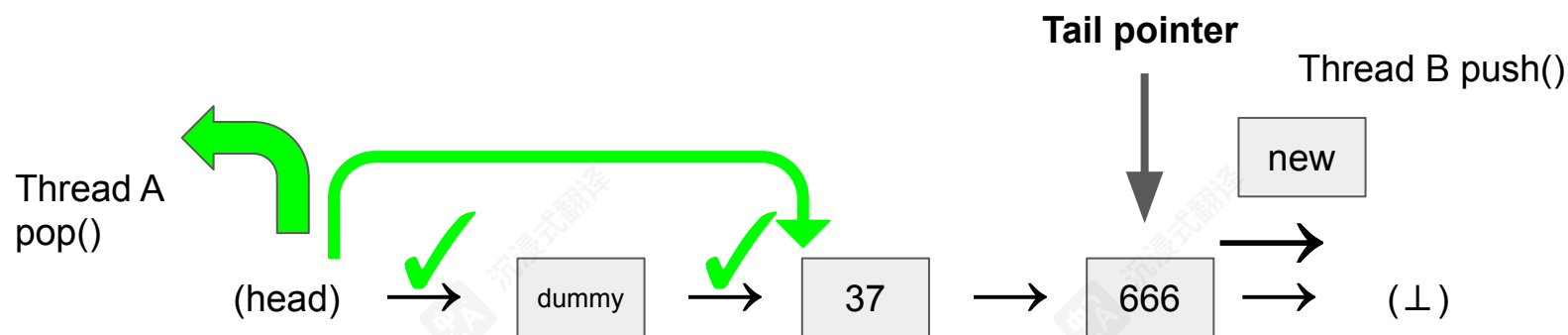
# 特莱伯的栈的问题

- 为什么数据：在Node<T>中手动Drop<T>？
  - 因为数据的所有权是通过pop()返回的，  
而Node<T> 稍后将被释放
  - 常见于无锁数据结构



# Michael-Scott队列

- **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>
- 弹出(): 比较并交换 头 (与 Treiber 栈弹出() 相同)      ))
- **Push(): CAS** on tail from null to new



- **挑战:** 尾部可能是过时的 (例如, 在推送之后)
- **解决方案:** 尾部宽松不变量 (尾部可以从头部访问)

# Michael-Scott队列的算法

- **Push 1:** 查找实际尾
  - 如果需要, 更新“尾”指针
- **Push 2:** 尝试追加一个新节点
  - 如果失败, 从头重试
- **Push 3:** 更新“尾”指针
  - 从旧尾到新尾执行CAS操作
  - CAS失败是允许的 (我们只需要尾指针足够新即可)
- **Pop 1:** 如果“尾”指针指向虚拟节点, 更新“尾”指针
  - 确保头不要追上尾
  - CAS失败是允许的 (我们只需要尾指针足够新即可)
- **Pop 2:** 更新“头”指针
  - 从旧头到新头执行CAS
  - 如果失败, 从头重试

# Michael-Scott队列的问题

- Why can a tail pointer be stale?
  - Right after a new node is inserted, the tail pointer becomes stale.
- 为什么头部有一个哑节点？
  - 为了确保即使对于空队列，尾部也指向某个节点。
- 在pop操作中，为什么如果尾部指向哑节点也要更新尾部？
  - 为了确保头部不会“超过”尾部（为什么？参见垃圾回收部分）。
- 为什么头和尾是：缓存对齐<节点<T>>？
  - 确保头和尾不会错误共享。

# 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>

## ● 查找节点：遍历列表并查找匹配节点

- 用于查找、插入、删除

## ● 删除节点：

- 第一步：通过0x1标记其下一个指针（逻辑删除）
- 第二步：在另一次遍历中将其分离（物理删除）

## ● 遍历策略：如何处理逻辑删除节点？

- 哈里斯的：一次性分离连续逻辑删除节点
- 哈里斯-迈克尔的：单独分离逻辑删除节点
- 哈里斯-赫尔利希-沙维特的（无等待）：忽略逻辑删除节点
  - 仅可用于查找（不可用于插入/删除）

# 排序单链表的问题

- 为什么先进行逻辑删除，然后进行物理删除？
  - 为了与被删除节点之后的插入操作同步。
- Harris-Michael列表的动机是什么？
  - 为了支持危险指针；稍后会有更多介绍。
- Harris-Herlihy-Shavit列表的动机是什么？
  - 用于无等待查找。
- 为什么HHS不能用于插入/删除？
  - 游标的“prev”可能已经被删除且无法更新。

# 更多示例

- 循环缓冲区

- <https://people.mpi-sws.org/dreyer/papers/gps/paper.pdf>

- 哈希表，二叉树（AVL/红黑树），基数树，...

- Chase-Lev 工作窃取双端队列

- <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-dequeue>
  - <https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md>

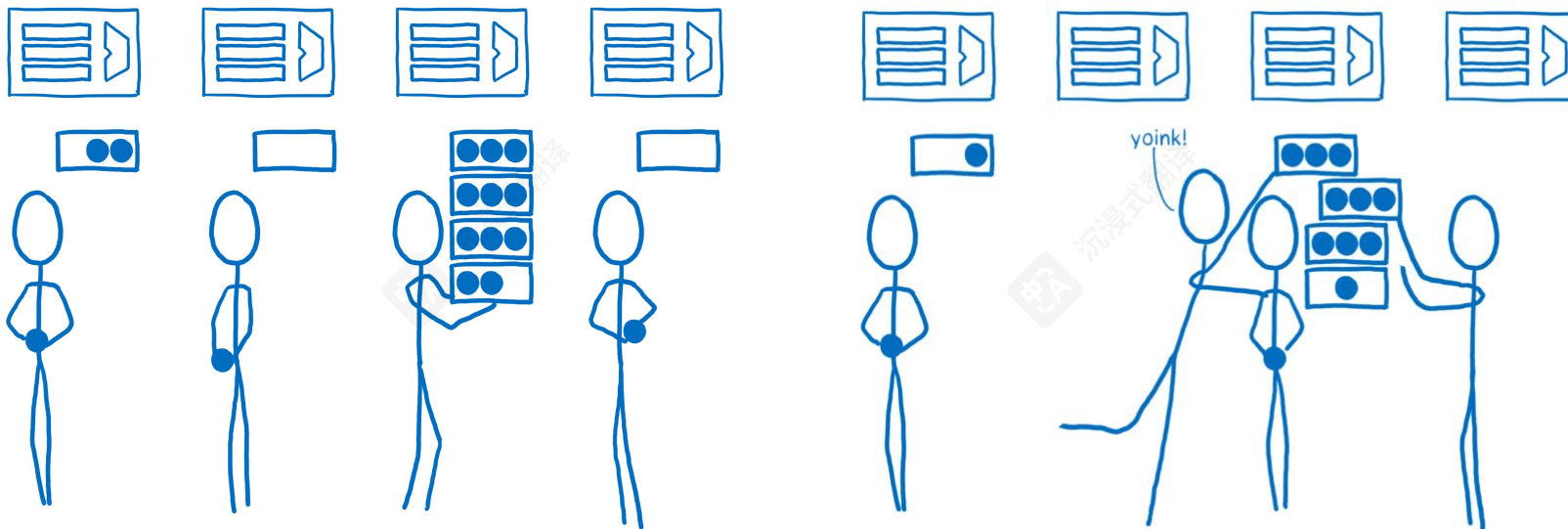
# 更多关于工作窃取的内容

- 源自 Cilk: <http://supertech.csail.mit.edu/cilk/>

- 在并行资源之间动态平衡工作负载

  - 通过“窃取”其他线程的工作

- (来自 [Lin Clark 的博客文章](#))



# 基于锁的并发Part 5: 乐观锁



# 乐观并发控制 (OCC)

- 观察：经常情况下，多个操作彼此不冲突
- 想法：乐观地假设操作成功，事后恢复
- 原语：序列锁
  - <https://github.com/kaist-cp/cs431/blob/main/src/lock/seqlock.rs>
  - 观察：所有读取操作彼此不冲突
  - 想法：乐观地读取，用序列号事后验证

# 序列锁

- 乐观读写锁

- 写者：几乎与自旋锁相同

- 管理序列号（无符号整数）而不是锁标志（布尔值） 偶数：c.s.之间的一致状态，奇数：c.s.内部的不一致状态。例如 W(0)：获取 0→1，
- 释放 2，W(2)：获取 2→3，释放 4，...

- 读者：尝试读取一致状态（例如 2）

- 读取 序列号 @ 开头和结尾（应该是偶数且相同）
- **Req 1:** W(0)'s end happens before R(2)'s beginning
- **Req 2:** R(2) doesn't see W(2)'s 写操作 as far as R(2) is 验证

# 示例：粗粒度乐观列表

- a算法 (图示)：全局读锁 -> 查找 -> 完成/升级

e

- memory reclamation problem

- 使用上方图示或GC幻灯片

- 仅使用RCU API

- 旧幻灯片

[https://docs.google.com/presentation/d/1NMg08N1LUNDPuMxNZ-UMbdH13p8LXgMM3esbWRMowhU/edit#slide=id.ga54eefc4bc\\_1\\_651](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](https://git.fearless.systems/kaist-cp-class/cs431-private/-/blob/d92dac01a8036c7abdb6ccf13a90fcc338332c47/src/list_set/optimistic.rs) (TODO: publish)

- crossbeam\_epoch API

- 使用原子<T> 显式允许竞争

- 回收

# 无锁并发Part 3: 垃圾回收

在 单独的幻灯片中

# 无锁并发Part 4: 线性化

# 并发数据结构（CDS）的关键目标

- **Progress:** 保证操作的完成（或进展）
  - 锁自由：至少有一个进展
  - 等待自由：所有人都有进展
- **可扩展性：**随着核心数量的增加，性能更好
  - 理想：线性扩展
  - 现实：例如，16个线程后的次线性扩展
  - 关键思想：减少临界区（细粒度锁和比较并交换）& 写操作
- **正确性：**按预期工作
  - 安全性：不会出错（即，没有段错误）
  - 顺序规范：例如，像队列一样工作
  - 同步：例如，匹配的 push 和 pop 是同步的

# CDS的可扩展性

- **关键思想 1：通过缩小锁保护范围来减少争用**

- 例如：手递手锁定、锁耦合、读写锁定

- **关键思想 2：通过避免写入来减少缓存失效**

- 例如：“乐观并发控制”
- 例如：读者中的避免写入（尤其是对于以读为主的工作负载）
- 例如。轻量级自定义同步协议

- **案例研究：乐观锁耦合**

- 锁耦合 + 乐观并发控制
  - 作业2：二叉搜索树的乐观锁耦合
-

# CDS的安全性

- **Key idea: prot带锁的CDS或更原始的同步**

- 用全局锁保护顺序数据结构
- 用更细粒度锁保护数据结构
- 用自定义同步协议保护数据结构

- 规范：“线性化”

(<https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md>)

---



# 线性化：‘正确’的正确性规范

- **关键思想：** CDS 相对于顺序数据结构的唯一复杂性在于操作的顺序（而不是指令的顺序）
- **上下文细化：** CDS “好像” 是在一个抽象数据结构上工作
  - 抽象数据结构：DS 操作是单条指令
- **线性化：** 上下文细化的一个关键引理
  - 在一个执行中，存在一个 CDS 操作之间的全序  $R$  使得：
  - (视图) 如果  $o1$  发生在  $o2$  之前，那么  $o1 R o2$ 。
  - (序列) 操作的结果就好像它们按  $R$  的顺序在顺序数据结构中执行一样。
  - (同步) 例如，一个 `push` 操作发生在它的匹配的 `pop` 操作之前。
  - 更多细节在 [本文件](#)

# 如何证明线性化？

- **关键问题：如何找到线性化顺序？**

- 线性化点：操作中的一个点，决定了线性化顺序
- 那么每个操作的线性化点是什么？

- **关键思想 1：写入操作的提交点是它的线性化点。**

- 回忆：许多无锁数据结构都有单指令提交点

- **关键思想 2：读操作的关键读是其线性化点。**

- 示例：[对头 = null] 的读取，从空 Treiber 栈中 pop

- **挑战与研究问题**

- 某些数据结构没有提交点（例如 Chase-Lev 双端队列）。如何为这类数据结构找到线性化顺序？
- 如何证明线性化？

# 线性化示例

- 特莱伯栈

- 关键思想 1: DS操作顺序 = 成功的CAS的顺序
- 关键思想 2: 空出栈的顺序 = 在周围的出栈和入栈之间

- 迈克尔-斯科特队列

- 关键思想 1: DS操作顺序 = 在头和节点的下一个指针上（但在尾部上不是）

- Harris-\*的列表

- 关键思想1: DS写入操作顺序 = 成功的CAS/FAO在头&节点的下一个指针的顺序
  - 除分离逻辑删除的节点外（它不是提交点）
- Key idea 2: DS读取操作顺序 = 活跃研究ar ea
  - Cf. [论文1](#) [论文2](#)

# Towards formal verification of linearizability

- 由德雷耶尔博士 (MPI-SWS) 主讲的视频讲座
- RustBelt: 面向未来安全系统编程的逻辑基础
- 在简街 (2019)



# 放松内存行为第一部分：语义学

# 动机：无锁自由揭示了血腥细节

- Recall: concurrency is challenging due to nondeterminism arising from...

- 交错：线程 A 和 B 的执行是交错的；和
- 重排序：线程 A 的指令可能被重排序。

● 重排序示例 `DATA=42 || if FLAG==1FLAG=1 ||`  
`assert(DATA==42) // 可能由于重排序而失败！`

- Lock would hide 这样的重排序：X 和 Y 不是并发访问的！
  - 重排序不会跨越锁/解锁。
  - 每个变量（X 或 Y）只有在持有锁的情况下才能被访问。
  - $\Rightarrow$  对每个变量的所有访问是完全有序的

## ● 剩余问题

- 如何禁止跨锁/解锁的重排序？
- 如何在无锁编程中禁止不良重排序？

# 非确定性：共享内存的挑战

- Me内存：位置  $\rightarrow$  字节 w/ 并发加载/存储指令
- 最广泛使用的单一SMS
- 挑战：高度非确定性

S

## ● 非确定性的来源1：线程交错

- 多个线程的加载/存储指令交错
- 导致组合爆炸性行为数量

## ● 非确定性的来源2：重排序

- 单个线程内的加载/存储指令可能被重排序
- 导致非直观行为

## ● 策略：通过禁止非预期行为来控制非确定性

# 由于重排序导致非确定性

- **动机：**像顺序程序一样执行优化

- **例如：**消息传递（FLAG是一个共享位置，例如 AtomicUsize）

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

- **问题：**由于编译器/硬件的重排序导致意外行为

- 由于重排序在左线程或右线程中导致的断言失败
- **宽松行为：**在“交错语义”中未捕获的可观察行为

- **解决方案：**禁止使用 排序原语

- **Fence:** fence(SC) between stores and between loads, or
- **Access ordering:** FLAG.store(1, release) and FLAG.load(acquire)



# 在自旋锁中强制执行顺序 (1/2)

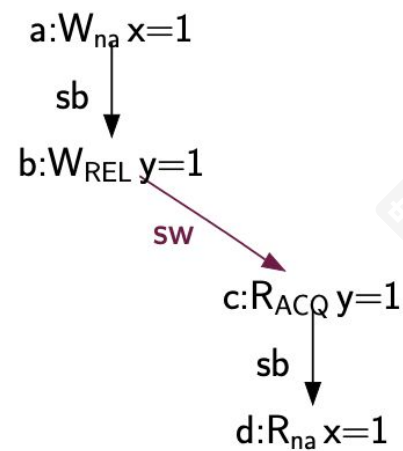
● <https://github.com/kaist-cp/cs431/blob/main/src/lock/spinlock.rs>

```
22 ✓ unsafe impl RawLock for SpinLock {
23 type Token = ();
24
25 ✓ fn lock(&self) {
26 let backoff = Backoff::new();
27
28 while self
29 .inner
30 .compare_exchange(false, true, Acquire, Relaxed)
31 .is_err()
32 {
33 backoff.snooze();
34 }
35 }
36
37 unsafe fn unlock(&self, _token: ()) {
38 self.inner.store(false, Release);
39 }
40 }
```

● “Ordering::获取” 和 “Ordering::释放” 强制执行顺序。

## 在自旋锁中强制执行顺序 (1/2)

- // 线程1 DATA = 42 LOCK = false (L.unlock()) // 释放 以防止与上述的重排序
- // 线程2 if (LOCK.cas(false, true)) // 获取 以防止与下文的断言 (DATA == 42)重排序
- 释放/获取同步
  - 如果一个释放写被获取读读取, 那么写操作 严格发生在读操作之前。



# 由于重排序导致的更宽松的行为

- **重排序**：除非访问相同位置，任何两个加载/存储/原子操作指令都可以重排序。

- 例如  $X=1; Y=1 \rightarrow Y=1; X=1$  (存储-存储重排序)

- **例如负载提升** [ $r1=r2=0$  允许通过存储-加载重排序]

- $X=1 \quad || \quad Y=1$

- $R1=Y \quad || \quad R2=X$

- **例如存储提升** [ $r1=r2=1$  允许通过加载-存储重排序]

- $r1=X \quad || \quad r2=Y$

- $Y=1 \quad || \quad X=1$

- **例如 Java 因果性测试用例** (#16, #19, #20 是错误的) <http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html>
- 

- **作业**：(非正式地) 理解上述示例

# 禁止使用排序原语进行重排序

- **E.g. 使用释放/获取同步的消息传递**

- DATA = 42;                      || if FLAG.load(获取)  
{FLAG.store(1, 释放);   ||     assert(DATA == 42); || }

- **释放存储**: 禁止使用排序原语与先前的指令进行重排序

- **获取加载**: 禁止使用排序原语与后继的指令进行重排序

- **E.g. message passing w/ sequentially consistent (SC) synchronization**

- DATA = 42;                      || if FLAG.load(宽松)  
{**fence(SC)**;                      ||     **fence(SC)**;FLAG.store(1, 宽松);  
||     assert(DATA == 42); }

- **SC栅栏**: 禁止在其自身上重排序

- **宽松**: 不强制排序

- **Q: What is the p宽松行为和排序的精确含义是什么?**

# 挑战：建模宽松行为 & 排序

- **Approach 1: 使用锁完全避免并发访问**

- DRF (数据竞争自由) 自由)：无并发访问，无宽松行为
- 问题：并发访问是必要的

S.

- **方法 2: 通过“公理”约束执行**

- 将执行表示为值流图，并使用公理进行验证
- 问题 1: 不是操作语义学（直观性较差）
- 问题 2: 允许“凭空出现”的坏行为

- **方法 3: 使用操作语义学对重排序进行建模**

- “Promise 语义学”：<https://sf.snu.ac.kr/promise-concurrency/>
- 操作语义学

# 承诺语义

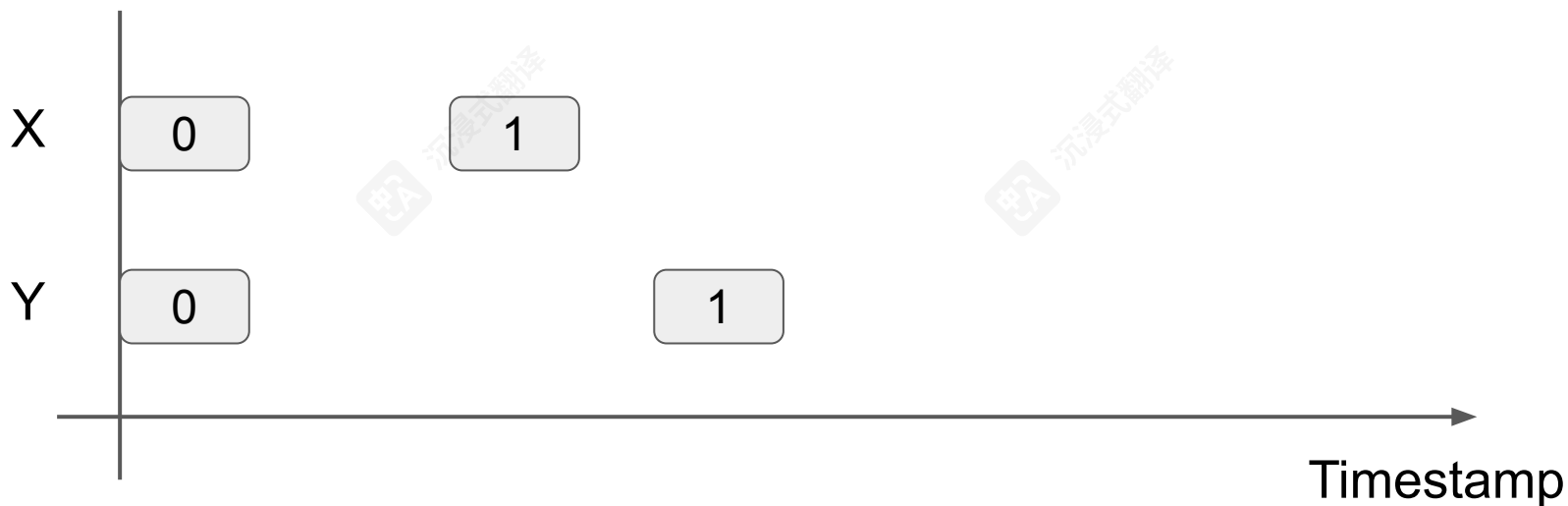
- Kang et al. A promising semantics for relaxed-memory concurrency. POPL 2017
- Interleaving operational semantics modeling relaxed behaviors and orderings
- **关键思想 1:**建模负载提升 w/ 多值内存
  - 允许一个线程从一个位置读取旧值
- **关键思想 2:**建模读改写 w/ 消息邻接
  - 禁止对单个值进行多次读改写
- **关键思想 3:**建模一致性 & 排序 w/ 视图
  - 限制线程的行为
- **关键思想 4:** 建模存储提升 w/ 承诺
  - 允许线程推测性写入一个值

# 承诺语义 1: 多值内存

- 内存: 位置  $\rightarrow$  列表消息, 消息: 值 & 时间戳
- 线程可能从一个位置读取旧值 (有效提升负载)

- 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]  
●                      ●

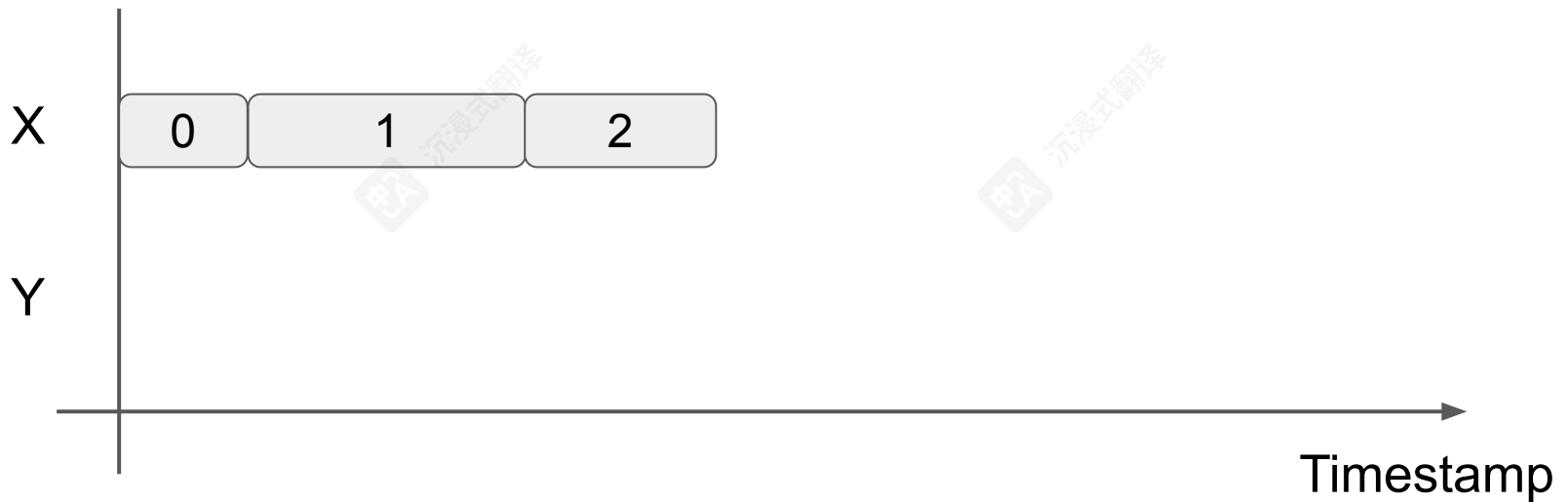


## 有前景的语义 2：消息邻接

- **消息：值 & 时间戳范围**
- **读-改-写应该将新的消息 邻接 到旧的（无间隙）**
- **消息占据一个时间戳范围（例如 (10, 20]**

- E.g. counter [r1=r2=0 **forbidden**]

- ● r1=X.fetch\_add(1) [0] || ● r2=X.fetch\_add(1) [1]  
● ●



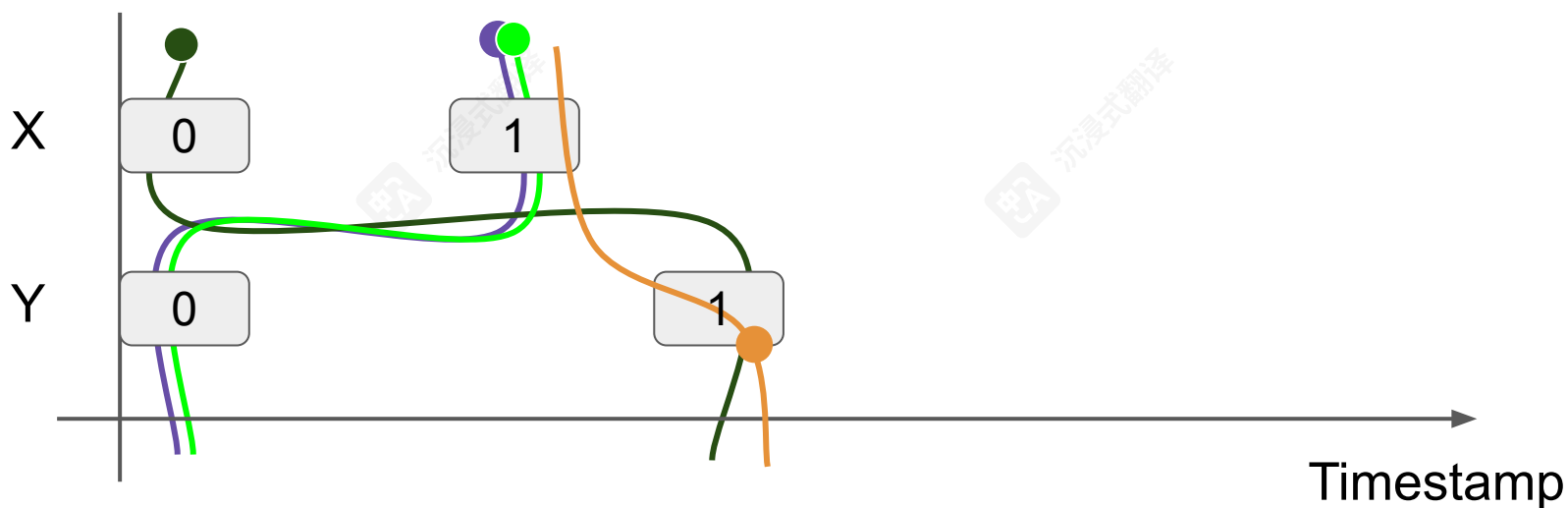


## 承诺语义 3：视图 (1/4)

- M多值内存允许太多、未预期的行为 rs
- Needs to con针对一致性和同步性的行为约束 n

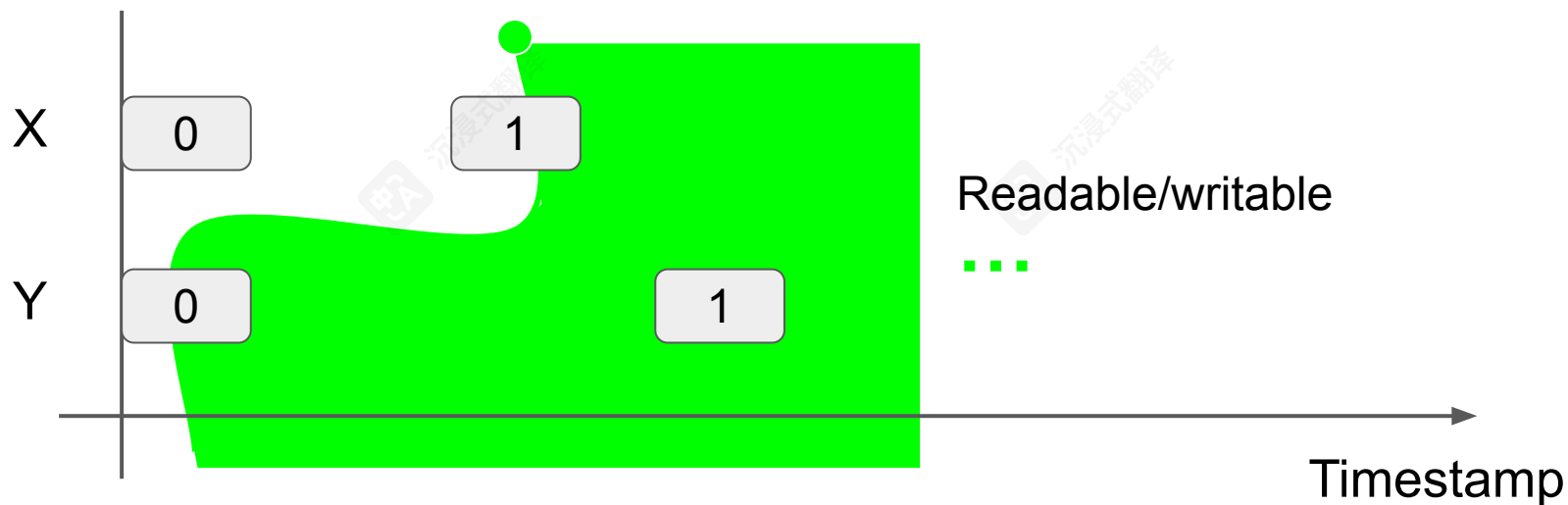
● 视图：位置  $\rightarrow$  时间戳（每个位置的确认消息）表示消息确认

- 按线程视图用于一致性
- 按消息视图用于释放/获取同步 n
- 全局视图 for SC同步



## 承诺语义 3：视图 (2/4)

- **Per-thread view** representing a thread's acknowledgement of messages
- 用于建模每个位置的连贯性
  - RR 一致性:  $X=1 \parallel r1=X; r2=X$  [ $r1=1, r2=0$  不可能]
  - 读写一致性:  $r=X; X=1$  [ $r=0$ ]
  - 读写一致性:  $X=1; r=X$  [ $r=1$ ]
  - WW 一致性:  $X=1; X=2$  [ $X=2$  在最后]
- 读取/写入发生在当前线程的视图之后
- 读取/写入改变了当前线程的视图



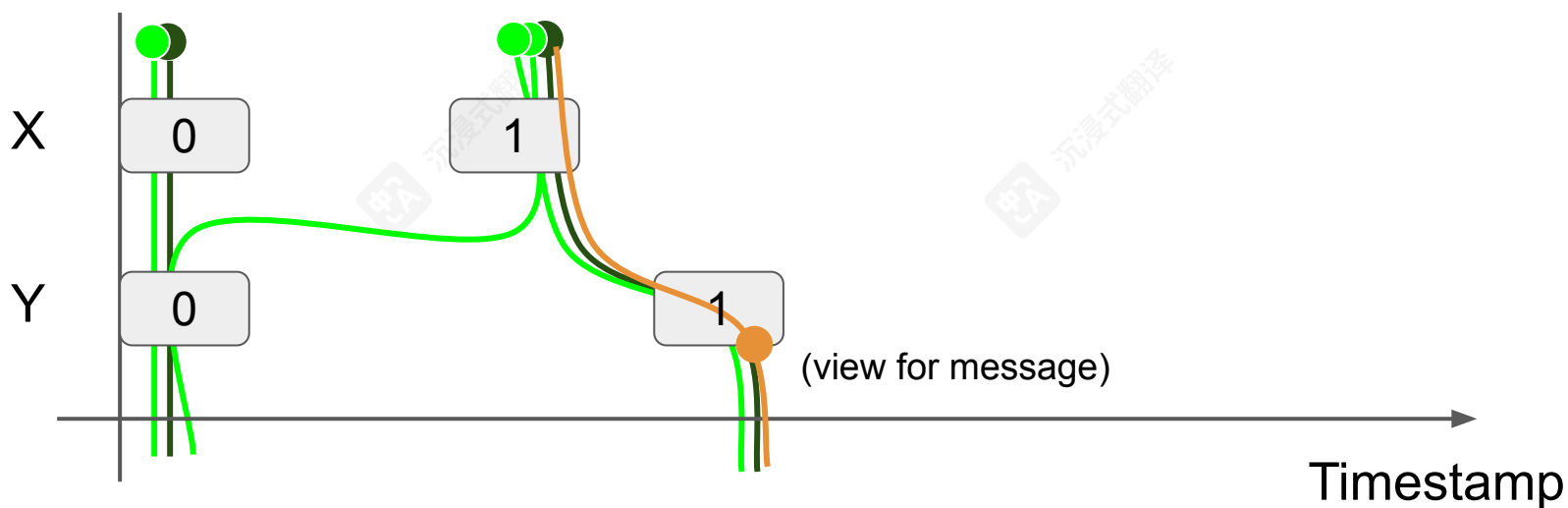
## 承诺语义 3：视图 (3/4)

- **按消息视图** 代表相应存储的已发布视图
- 用于建模释放/获取同步
- 例如：消息传递 (X=1 应在读取 Y=1后确认)

○ X = 1; || if Y.load(acquire){

● Y.store(1, release); || ● assert(X == 1);

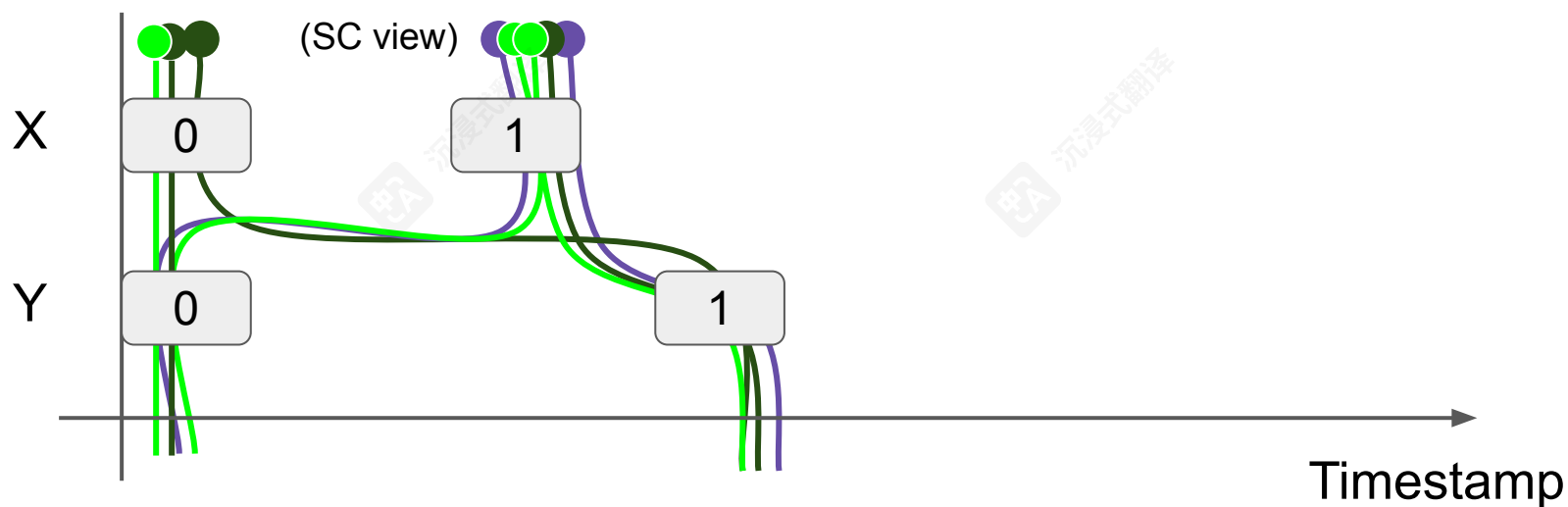
● || ● }



## 承诺语义 3：视图 (4/4)

- **全局视图** 表示当前累积的 SC 栅栏视图
- 用于建模 SC 栅栏同步 (SC 栅栏之间的严格顺序)
- 在 SC 栅栏之后, SC 视图和线程视图成为  $t$  的最大值
- 例如消息传递 ( $X=1$  应在读取  $Y=1$  后确认)

○ ●  $X = 1;$                       || ● if  $Y.\text{load}(\text{宽松}) \{$     1  
●  $\text{fence}(\text{SC});$                 || ●  $\text{fence}(\text{SC});$   
●  $Y.\text{store}(1, \text{relaxed});$  || ●  $\text{assert}(X == 1);$  } 1  
●



s

hem

## 承诺语义 4: 承诺 (1/5)

- **Challenge: modeling store hoisting**

(“A major 程序语言语义的开问题, Batty等人, ESOP 2015)

- 存储提升 w/o 依赖 [r1=r2=1 允许通过正确的重排序] r1=X || r2=YY=r

1 || X=1

- 存储提升 w/ 依赖 [r1=r2=1 不允许, 凭空出现 (OOTA)] r1=X || r2=YY=r

1 || X=r2

- 存储提升 w/ 语法依赖性 [r1=r2=1 允许通过编译器优化]

r1=X || r2=Y

Y=r1 || if r2==1 { X=r2 } // “if” 应该被视为行为,  
else { X=1 } // 但看起来像是凭空出现

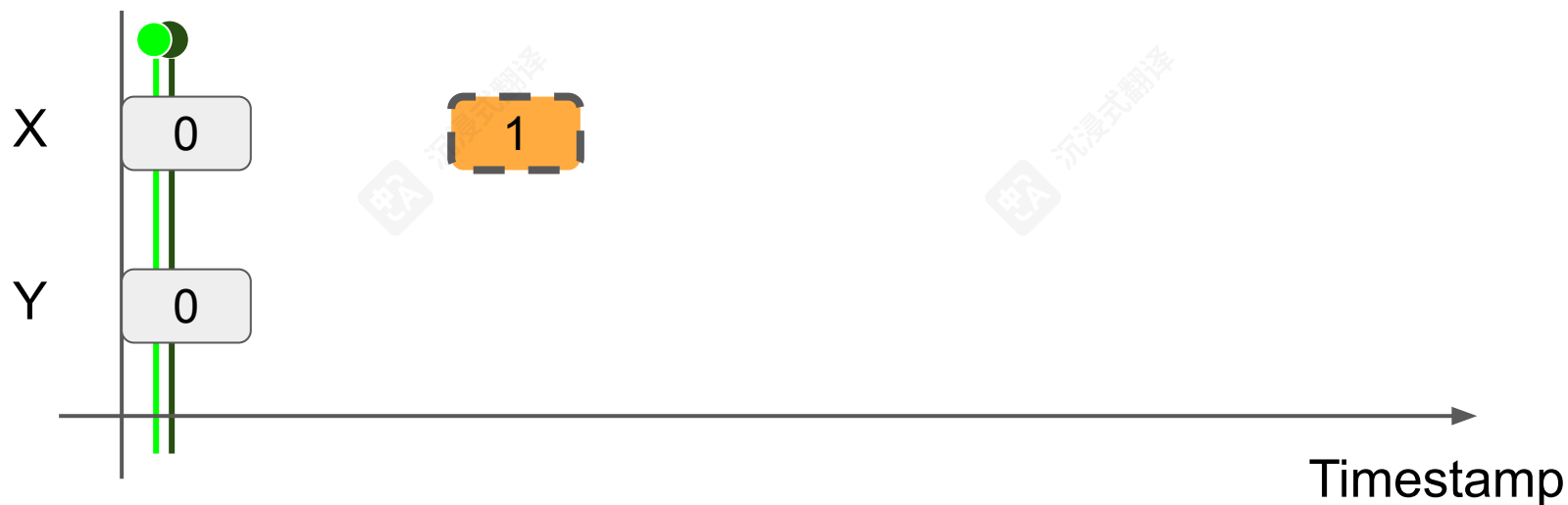
○ 编译器可能 (1) 在then分支中转发r2=1, 并且 (2) 提升X =1 out

## 承诺语义 4：承诺 (2/5)

- **Goal:** allowing the hoisting of **semantically independent writes** only
- 我们 **将** 在期末考试中涵盖承诺。
- **想法：** “语义独立写入” 在未来 **总是可写入**
- **Mechanism**
  - 一个线程可以推测性写入一个值（“写入承诺”）
  - 一个线程应该 **在未来总是能够写入它的承诺**
- **例子：** 存储提升 带 & 不带（句法）依赖

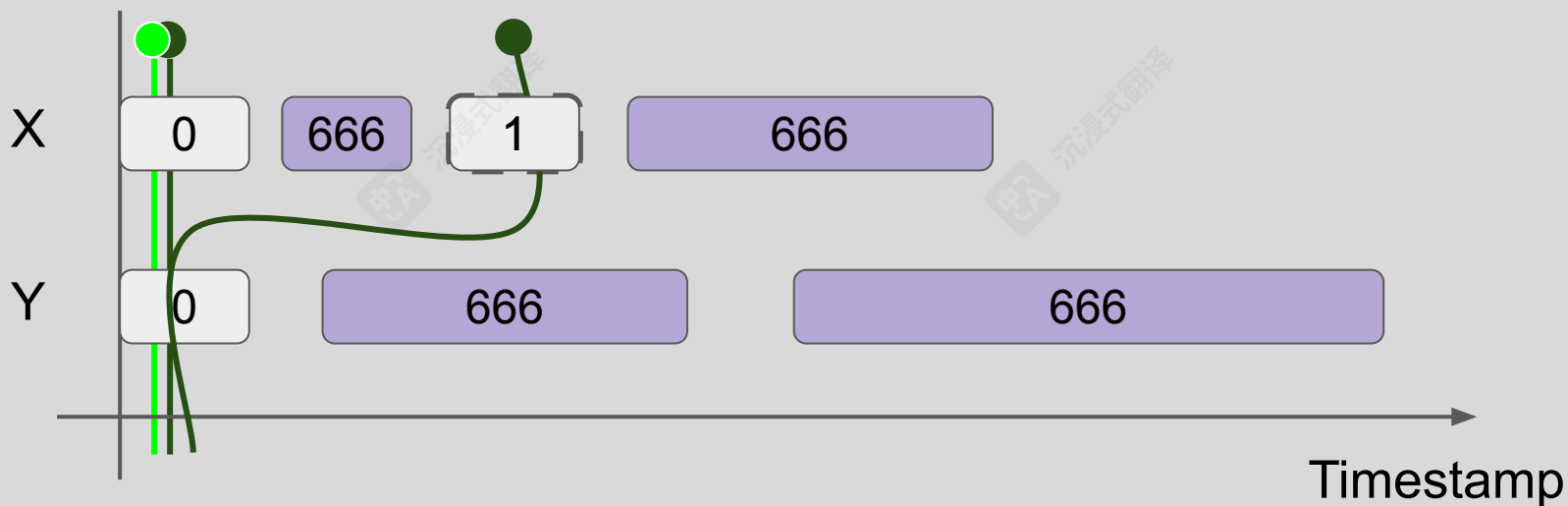
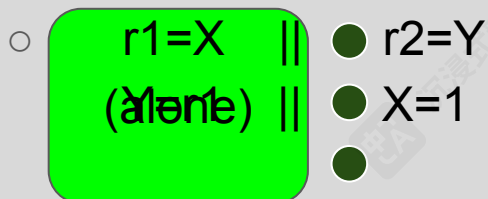
## 承诺语义 4: 承诺 (3/5)

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



## 承诺语义 4: 承诺 (3/5), 认证

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

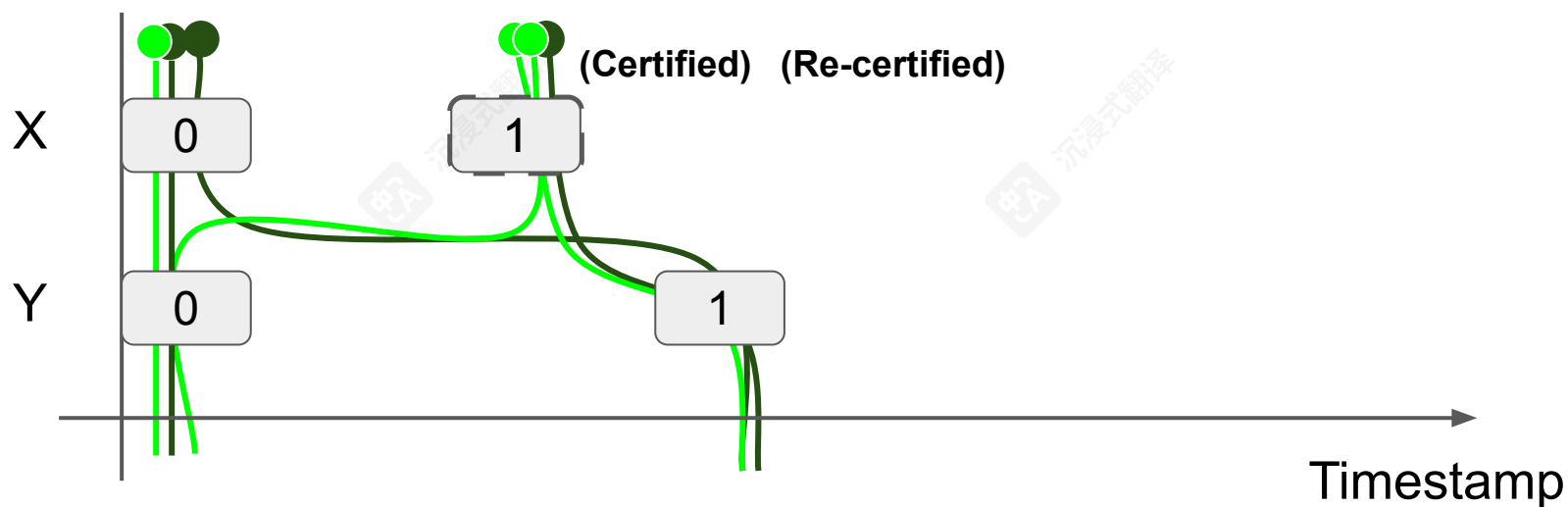




## 承诺语义 4: 承诺 (3/5)

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

○ ●  $r1=X$  || ●  $r2=Y$   
●  $Y=r1$  || ●  $X=1$   
● ●

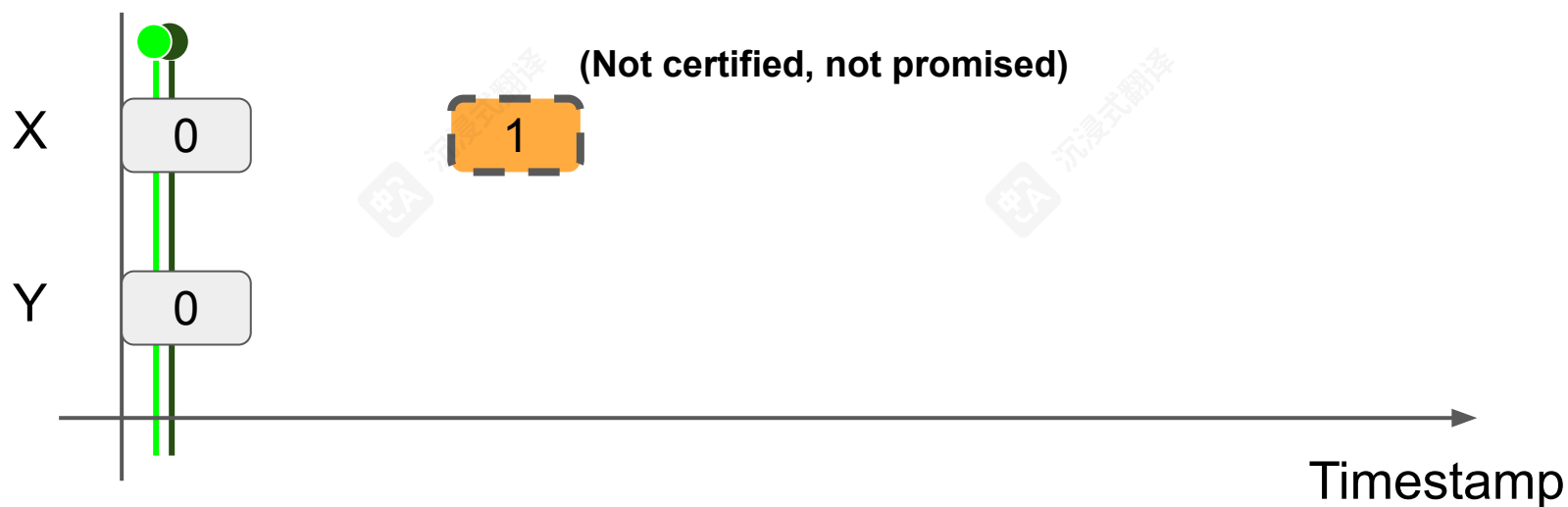


## 承诺语义 4: 承诺 (4/5)

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

$r1=X \parallel r2=Y \ Y=$

$r1 \parallel X=r2$

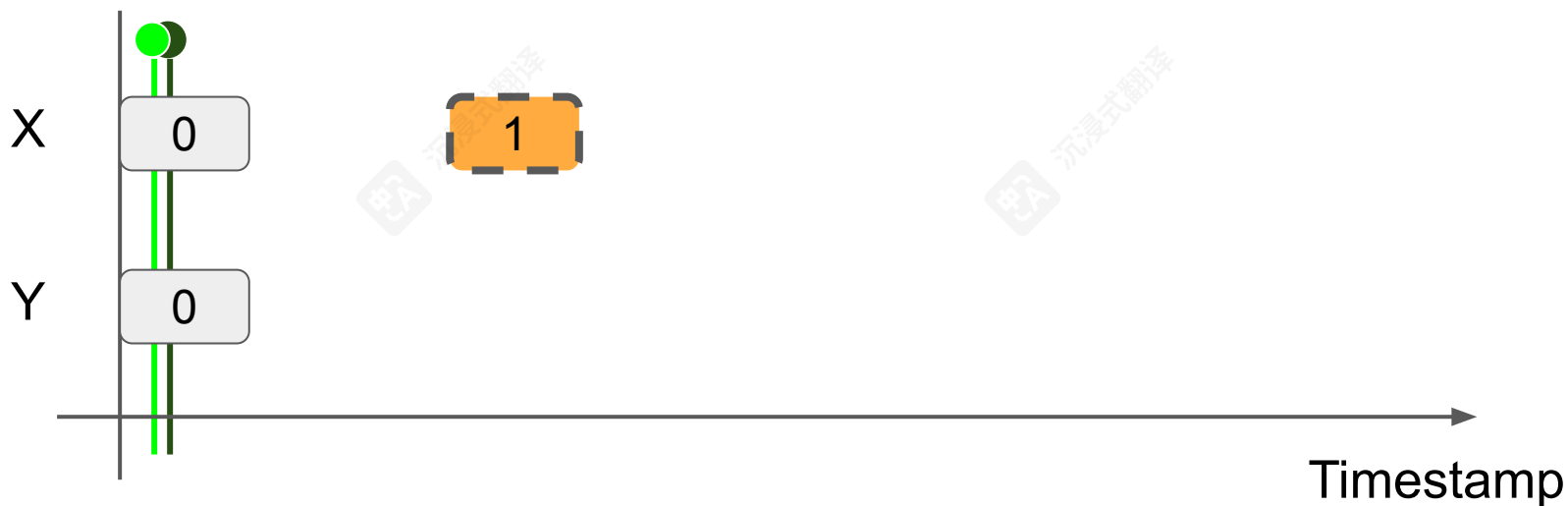


## 承诺语义 4: 承诺 (5/5)

- **Store 提升 w/ 语法依赖性**  $[r1=r2=1 \text{ 允许 } r1=X \parallel r2=Y]$

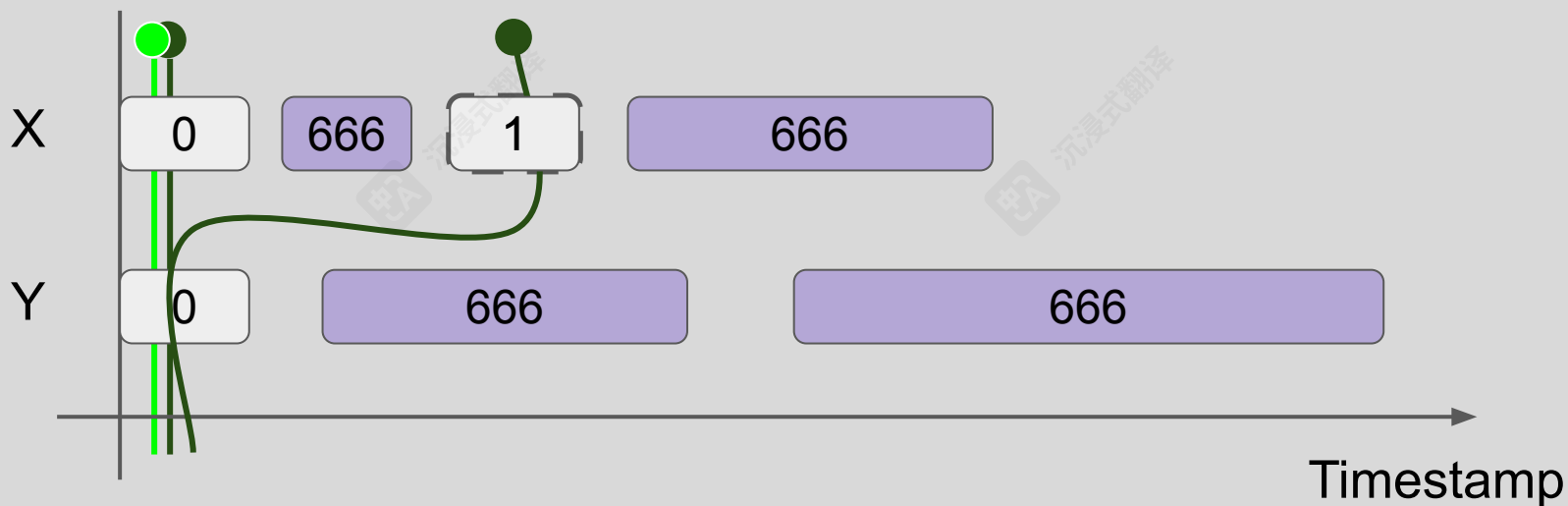
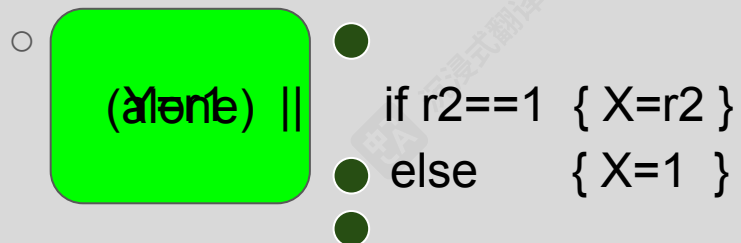
○ ●

$Y=r1 \parallel \begin{cases} \text{if } r2==1 \{ X=r2 \} \\ \text{else } \{ X=1 \} \end{cases}$



## 承诺语义 4: 承诺 (5/5), 认证

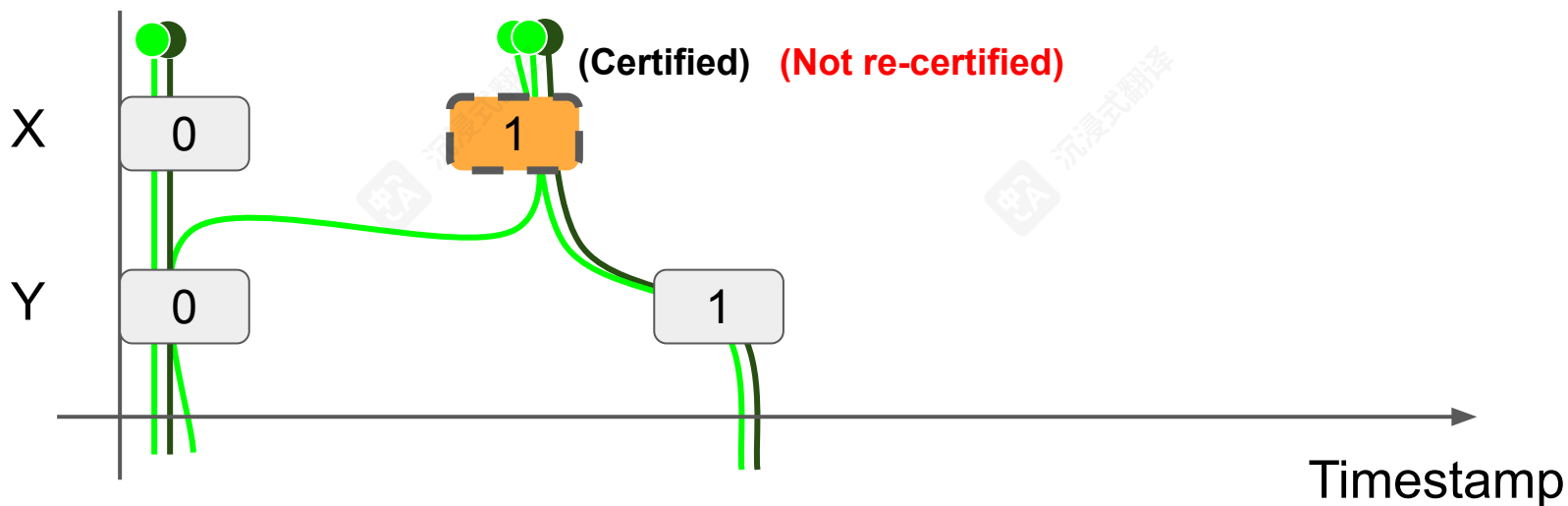
- Store 提升 w/ 语法依赖性  $[r1=r2=1 \text{ 允许 } r1=X \parallel r2=Y]$





## 加分项：受再认证约束的行为

- **Store hoisting w/ syntactic dependency** [ $r1=r2=r3=1$  **disallo** **d**]
  - ●  $r1=\text{weX}$  || ●  $r2=Y$ ;  $r3=X$  // 由于读写一致性
  - $Y=r1$  || ●  $\text{if } r2==1 \{ X=r2 \}$
  - $\text{else } \{ X=1 \}$



# Summary of promising semantics

- An operational semantics modeling relaxed behaviors and orderings

## ● 关键思想

- 多值内存：建模负载提升消息邻接：建模读改写视图
- 图：建模一致性和同步承诺
- 
- **es**：建模存储提升（已涵盖在期末考试）

## ● 作业：理解承诺语义中的示例

- 消息传递 负载提升 存储提升 带 & 不带（句法）依赖Java 因果性测试
  - 用例 (#16, #19, #20 是错误的) — 已完成!
  - 
  -
-

## 加分项：更多关于承诺语义的内容

- <https://sf.snu.ac.kr/promise-concurrency/> for more details
- Designated courses for details on lower-level (e.g. (micro-)architecture)
- Related work
  - 基于 ARM/RISC-V 架构的视图语义  
<https://sf.snu.ac.kr/promising-arm-riscv/>
  - 基于持久内存的视图语义  
<https://cp.kaist.ac.kr/pmem>
  - 承诺语义的程序逻辑（形式推理原则）  
<https://people.mpi-sws.org/~viktor/papers/esop2018-slr.pdf>
- 研究思路
  - 基于视图的语义（用于中断、非缓存、I/O、PCIe、CXL、...）
  - 验证编译对于承诺语义
  - ...

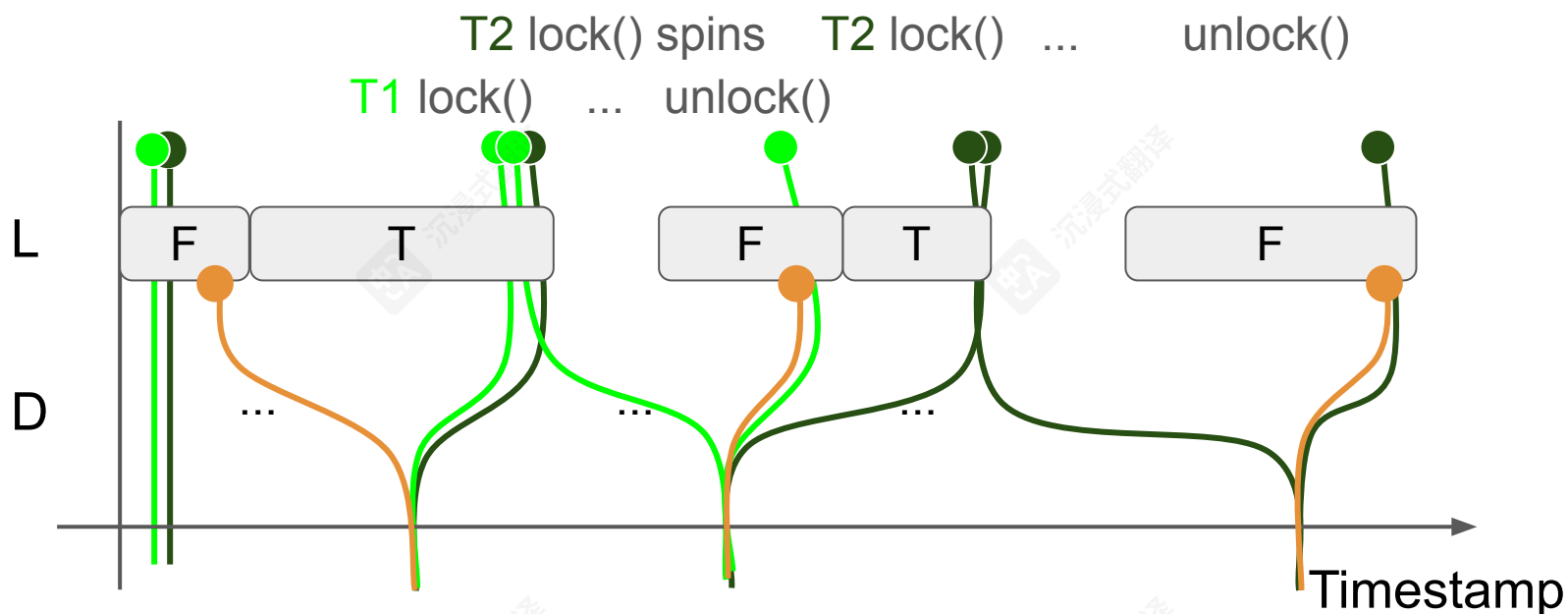


# 放松内存行为Part 2: revisiting locks and objects

# 自旋锁正确性，再探讨

● 公有的函数 `lock(&self) { while self.inner.cas(false, true, 获取).is_err() }`  
} } 公有的函数 `unlock(&self) { self.inner.store(false, 释放); }`

- 如果锁已经被获取，`lock()` 将自旋
- `lock` 与 `unlock` 之间的操作通过 `release/获取同步` 传递
- 持有锁时，你将访问 `D` 的最新值（无共享访问）



# Treiber堆栈正确性，再探讨

- **不变量：**头指针值具有释放视图  $\sqsupseteq$  ( $\geq$  每个位置loc.)所有节点值的消息 & 下一个指针
- **释放：**用于维护不变量 (L54)
- **获取：**用于利用不变量 (L68; L70,84处的读取是安全的)
- **L74可以宽松：**后续节点值的消息 & 下一个指针已经释放
  - **“Release sequence”：** a msg’s relview is transferred to the adjacent msg
  - E.g. B’s next pointer and data is released at Head
    - [头  $\rightarrow$  B  $\rightarrow \perp$ ] //  $\therefore$  L50处的释放
    - 头  $\rightarrow$  A  $\rightarrow$  B  $\rightarrow \perp$  //  $\therefore$  头是CAS操作
    - $\Rightarrow$  [头  $\rightarrow$  B  $\rightarrow \perp$ ] //  $\therefore$  头是CAS操作
    - $\Rightarrow$  [头  $\rightarrow \perp$ ] //  $\therefore$  读取B的值和下一个指针是安全的

# MS队列正确性，回顾

● **不变量：** 一个指针值具有释放视图  $\sqsupset$  指向的节点的值和下一个指针的消息

- 一个指针值可以是头、尾部或一个节点的下一个指针
- 例如。ptr 的消息具有释放视图  $\sqsupset$  A 的值和指针（指向 B）



● **释放：** 用于维护不变量

- MS队列：L88、101、111、140、148

● **获取：** 用于利用不变量

- MS队列：L77, 81, 125, 128)

# 序列锁的同步

- 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 |

(**红色**) 如果 R(2) 从 W(2) 的写操作中读取, (**蓝色**) R(2) 的获取栅栏发生在 W(2) 的释放栅栏之后, (**绿色**) 使 R(2) 无效。

- R(2) 的读取和 W(2) 的写操作应该是 “**原子**” (没有未定义行为)
- R(2) 可能观察到部分由 W(2) 修改的不一致状态 (R(2) 需要对该不一致具有 **弹性**)

# 结束语

# 你学到了什么？

- **并发：**关于共享可变资源
  - 由于交错非确定性而困难
- **共享内存：**最广泛使用的共享可变资源
  - 由于重排序非确定性而更困难
- **设计模式，**包括低级（例如 rel/acq）和高级（例如 helping）
- **实现 锁，**并发数据结构，垃圾收集器
  - 动机，用例，应用程序，权衡，陷阱
- **承诺语义** 对于共享内存并发

# 预期结果是否达成？

- 不要害怕并发编程

- 理论上，它只是处理共享可变状态

- 不要害怕系统编程

- 系统编程：关于低级资源
- 特别是对于并行系统

- 不要害怕编程

- 你将学习并发编程的系统化方法
- 这也有利于一般的编程



# 我还能学习更多关于并发的内容吗？

## ● 硬件并发

- 中断、I/O、系统调用、持久内存、...
- X86、ARM、RISC-V 并发
- 硬件描述 w/ Verilog、VHDL、[ShakeFlow](#)
- 编译器正确性

## ● 并发数据结构

- ... (cannot summarize)
- 操作系统内核，数据库管理系统，...

访问 <https://cp.kaist.ac.kr>  
联系 [jeehoon.kang@kaist.ac.kr](mailto:jeehoon.kang@kaist.ac.kr)

## ● 用于验证的并发分离逻辑

- “分离”：处理资源部分的所有权独占
- 并发最广泛使用的推理原则
- Iris: 最先进的 CSL (<https://iris-project.org/>)

谢谢

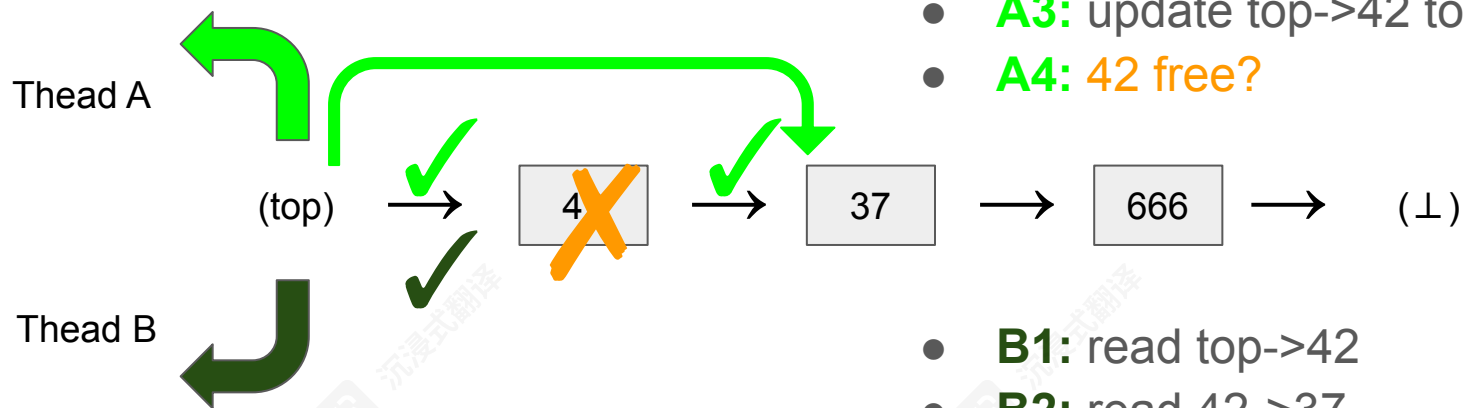
# 期末考试

期中考试之后你所学到的一切

# 备份幻灯片

# 示例：Treiber栈中的内存回收

- Treiber栈：带有头= 的单链表栈顶



- **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!**

问题：如何确保42在完成访问后释放？

# 答案：通过线程间的同步

- **内存回收器：** 专门用于此类同步的库/运行时

- 动机：为每种数据结构解决问题成本过高

- **许多回收方案**

- 基于指针的回收 (PBR)
- 基于时代的回收 (EBR)
- 混合：QSBR、雪花、QSense、IBR、危险时代、**PEBR**、...

- **关键思想**

- **保护** 被访问的块（由每个线程）
- **退役** 未链接的块（而不是立即回收它们）
- **回收** 未被任何线程保护的退役块

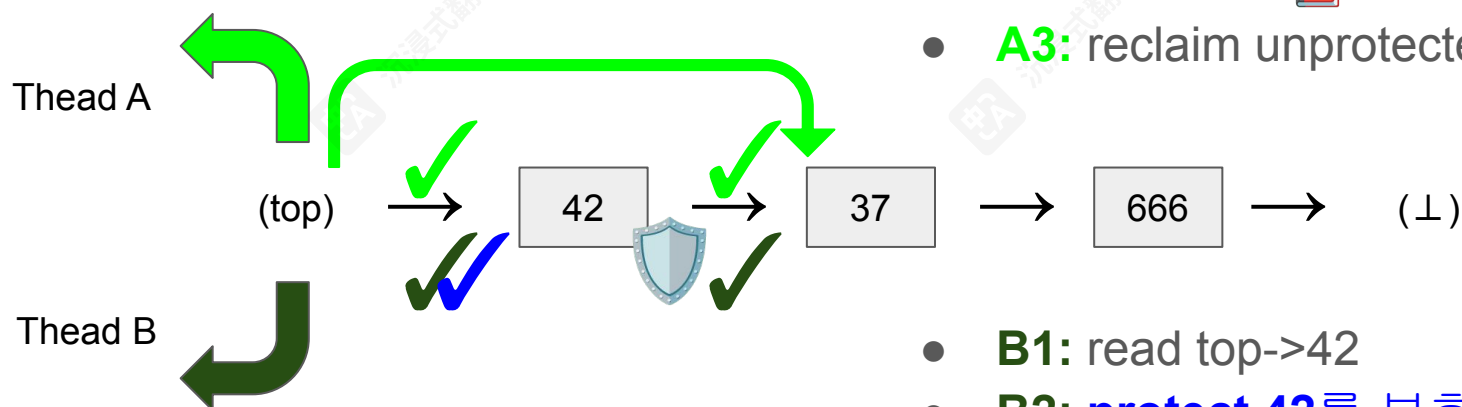
# 危险指针 (HP)

- **关键思想 1:** 保护 块, 然后再访问它们

- **关键思想 2:** 退役块, 而不是立即回收它们

- 回收那些没有被任何线程保护的退役块

# HP的示例：特莱伯的栈



- **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的API

## ● 数据

- PL: 线程级保护块（危险指针）列表
- RL: 线程级退役块列表

## ● 保护(block)

- P1: 将块添加到PL
- P2: SC栅栏
- P3: 验证块是否仍被指针指向；否则重试







## ● 退役(block)

- R1: 从共享内存中删除块的所有引用
- R2: SC栅栏
- R3: 将块添加到RL

## ● 收集()

- C1: 读取每个线程的 PL
- C2: 回收 RL 中不在每个线程的 PL 中的那些块

# HP的问题：昂贵的同步

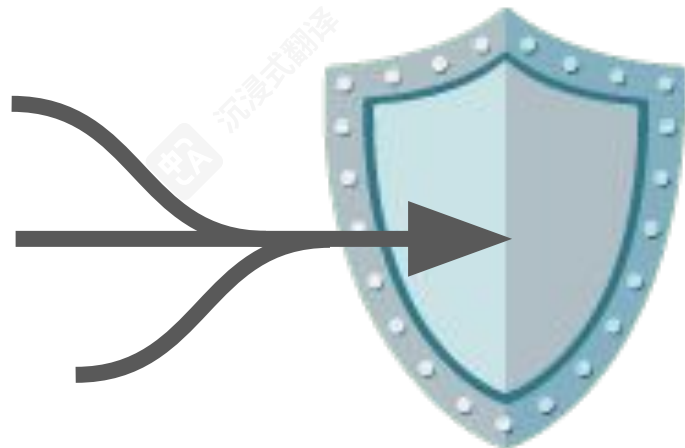
- **同步 同步点：**插入到RL ，插入到P  L 
  - 订购  和 
  - 重排序可能在没有同步的情况下发生
- **同步成本：**在A和B中都存在栅栏
  - 100+ 周期 (例如：x86 mfence, ARMv8 dmb sy, POWER syn c)
  - **关键：**每次遍历在 (1/3 吞吐量) 中执行一个  栅栏

**基于周期的回收：减少同步成本**

# 基于时代的回收 (EBR)



- 关键思想 1: 临界区域的多次访问



- 关键思想 2: 周期共识(并发临界区域具有“相似”的时代)
- 保护时代, 而非指针  
(摊销同步成本)

# EBR的关键思想1：临界区域

- **Critical 区域 (c.r.):** 线程内的内存访问周期
- **用户定义的:** 可能在不同的数据结构和应用程序中有所不同
  - 例如, 在 “stack.pop()” 期间, 在处理数据库查询期间

## ● 临界区域内的内存遍历

- 共享内存访问应在内
- A pointer r在c.r.内部读取时, 可能会在同一c.r.内解引用
- stack.pop() 应该在 c.r. 内部

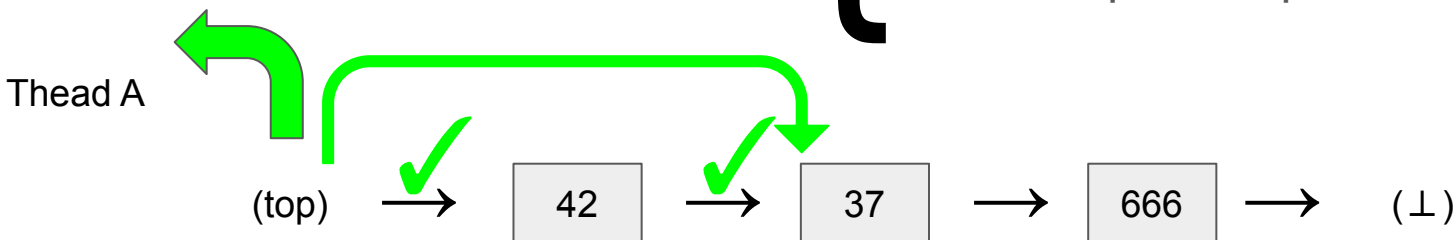
e a c.r.



KAIST CS431: 并发编程 ○ 共享内存访问  
应该在内部

● A2: 读取42-

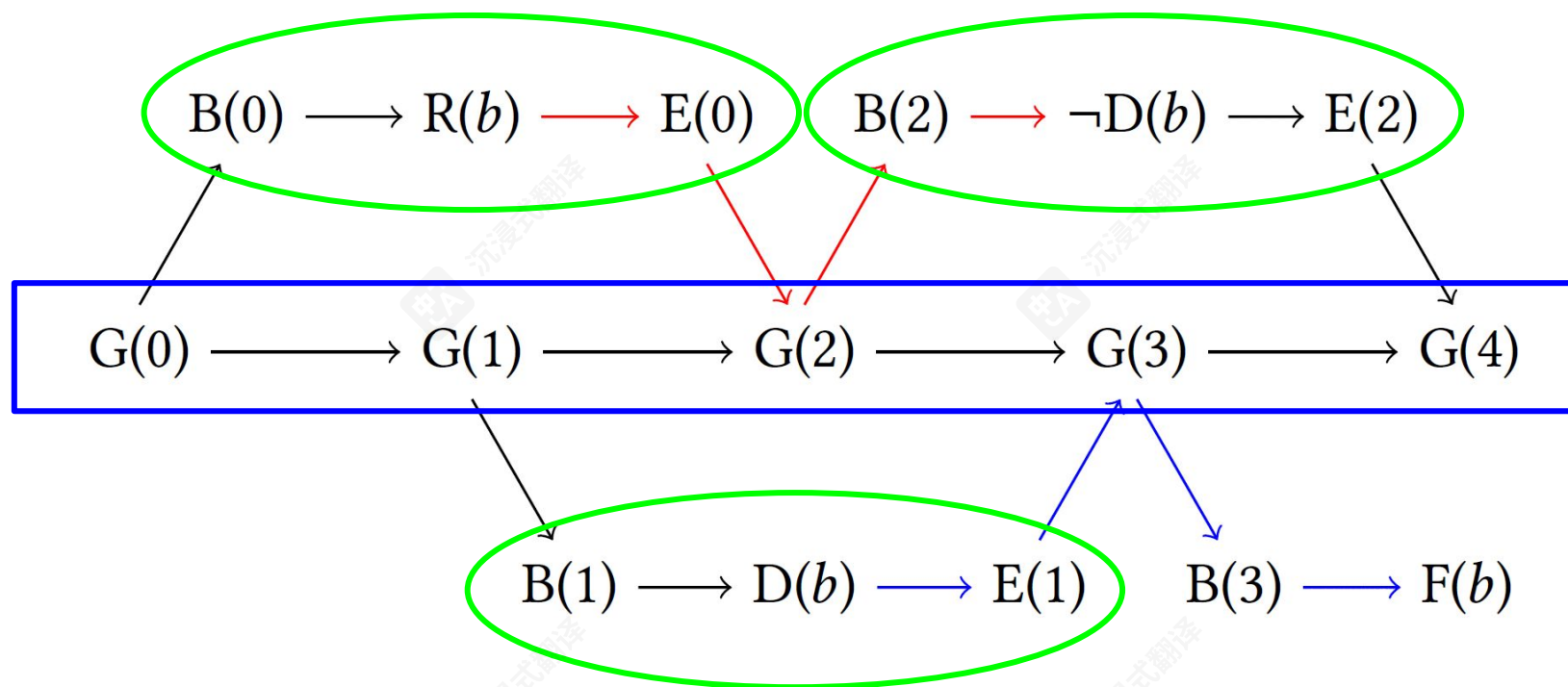
● **A3:** update top->42 to top->37



## EBR的关键思想2：时代（1/2）

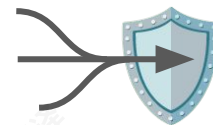
- **时代**：分配给每个临界区域（0, 1, 2, 3, ...）， $\rightarrow$ ：发生之前
- **$B(i)$ ,  $E(i)$** ：时代 $i$ 的临界区域的开始/结束， **$G(i)$** ：时代的开始
- **时代共识**： $G(i) \rightarrow B(i)$ ,  $E(i) \rightarrow G(i+2)$ 
  - 推论：并发临界区域的时代可能只相差1
- **SC栅栏**：仅在临界区域的开始处

och  $i$



## EBR的关键思想2：时代 (2/2)

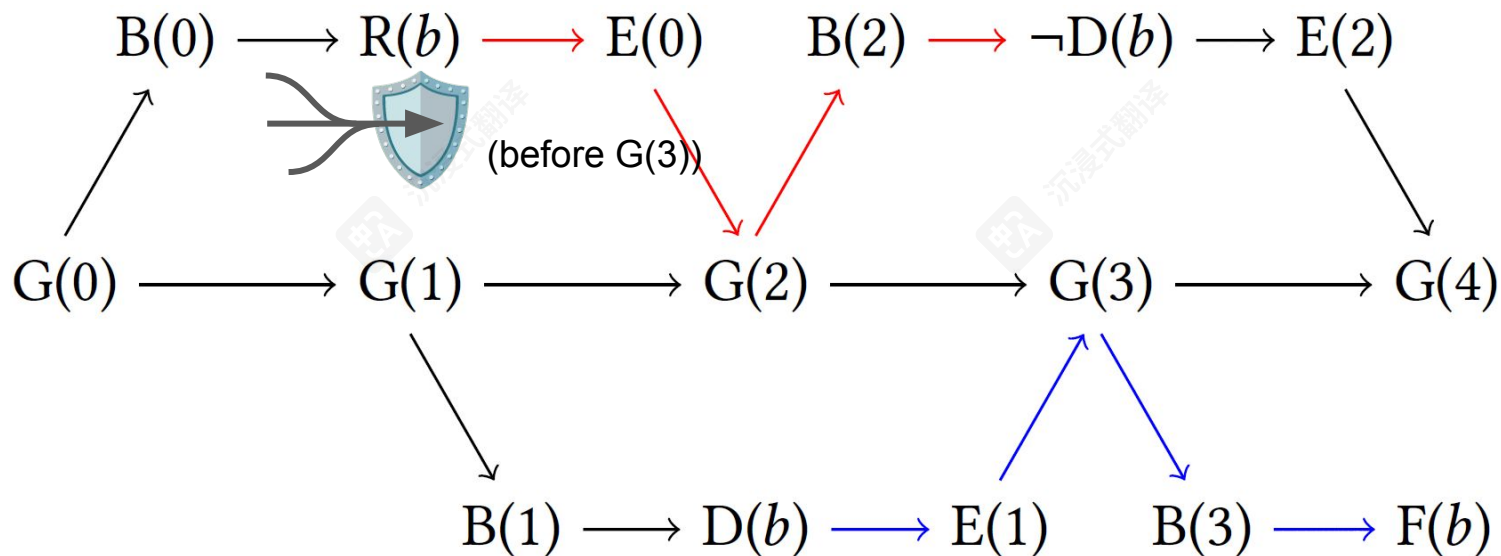
- 保护：在 $i$ 处退役的 块被保护，直到 $G(i+3)$
- 安全性： (1) 不可访问的 在c.r. @  $i+2$ , (2) 活跃的 在c.r. @  $i+1$



- $R(b)$ ：在内存中分离 $b$ 后的退役

$D(b)$ ：对 $b$ 的解引用

$F(b)$ ：对 $b$ 的释放（回收）



# EBR的问题：不鲁棒（内存泄漏）

- 当：长临界区阻碍纪元推进和回收
  - 因为  $E(i) \rightarrow G(i+2)$
- 情况1：用户定义的长临界区（例如OLAP、对象缓存、I/O）
- 情况2：非计划线程（例如过载订阅） 情况3：停滞线程（例如错误、仅崩溃分布式系统）

**问题：快速且鲁棒的内存回收？**

# PEBR: HP 和 EBR 的结合

- 目标: 像 EBR 一样快速, 像 HP 一样鲁棒
  - 关键思想: HP 和 EBR 的混合
    - 首先使用 EBR (快速), 当块未被回收时, 被弹出至 PBR (鲁棒)
  - (2019) 姜济雄和 Jaehwang Jung。指针和基于纪元的回收的结合。  
已提交。 <https://cp.kaist.ac.kr/gc/>
-



# Shared memory low-level synchronization pattern

# 共享内存低级同步模式

- <https://jeehoonkang.github.io/2017/08/23/同步模式.htm> |
- **模式 1: “正向” 释放/获取同步**
  - 正向: 如果 A, 则 B
  - 如果释放存储被获取加载读取, 视图被转移
- **Pattern 2: “contrapositive” release/acquire synchronization**
  - 逆否命题: 如果非 B, 则非 A
  - 如果视图没有传递, 获取加载不会读取释放存储
- **模式3: “负向” SC同步**
  - 负向: 要么A要么B
  - 要么F1发生在F2之前, 要么F2发生在F1之前

涵盖“大多数”低级同步

## 正向释放/获取同步

- 如果一个释放存储被一个获取加载读取，那么存储处的"已释放"视图在加载处被"已获取"

- 例如：消息传递 (X=1 在读取Y=1后应被确认)

```
○ X = 1; || if Y.load(acquire) {Y.store(1,
release); || assert(X == 1); // 应该不失败|| }
```

○ 传输 数据 (X) 使用 flag ( 的释放/获取同步

Y)

- ## ● 最广泛使用的低级同步模式

○ 用于自旋锁、通道、...

# 负向SC同步

- All SC栅栏相对于每个线程的视图是严格有序的

- 例如. 消息传递 (X=1 应在读取Y=1后确认)

- `X = 1;                      ||    if Y.load(宽松) {  
fence(SC); ||       fence(SC); Y.store(1, 宽松); ||  
assert(X == 1); }`

- Trans 传输数据 (X) 或忽略标志 (Y) w/ SC同步

on

- **An advanced low-level synchronization pattern**

- Used in Peterson's mutex, memory reclamator, work-stealing deque, ...

## 示例：Peterson互斥锁（实现）

● let flag: [原子布尔; 2]; // 是否有线程想要开始临界区 let turn: AtomicUsize = 0; // 谁具有优先权?

- 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) {}  
}

● 函数 end(id: 无符号整数) {  
    FLAG[id].存储(false, 释放); }

# 示例：Peterson互斥锁（正确性）

- 栅栏顺序的案例分析
- 案例1:  $A0 \rightarrow B0 \rightarrow A1 \rightarrow B1$ 
  - $\text{flag}[0] = \text{true}$  and  $\text{turn} = 1$  should be ack. at A1
    - $\text{turn} = 1$  before  $\text{turn} = 0$  w.r.t. 一致性
- 案例2:  $A0 \rightarrow A1 \rightarrow B0 \rightarrow B1$  or  $A0 \rightarrow A1 \rightarrow B1 \rightarrow B0$ 
  - 两者  $\text{flag}[0] = \text{true}$  和  $\text{flag}[1] = \text{true}$  都应在 B0 和 B1 处被确认
  - 不失一般性, 假设  $\text{turn} = 1$  在  $\text{turn} = 0$  关于一致性
- ... (类似情况)
- $\text{flag}[0] = \text{true}$  在 B1 处被确认 &&  $\text{turn} = 1$  在  $\text{turn} = 0$  关于一致性
- T1 应旋转直到 T0 调用 `end()` 写入  $\text{flag}[0] = \text{false}$
- Mutex 感谢 T0 到 T1 通过  $\text{flag}[0]$  的释放/获取同步
- 作业: 栅栏 (A 和 B) 中的每一个是否必要?

# 示例：Peterson互斥锁（讨论）

- 由于缺点，不实用

- 不必要的复杂
- 不可重用：在end()之后不能begin()（因此不是“锁”）

- **定理：**可重用互斥锁需要原子读-改-写指令（例如swap、比较并交换或其他读-改-写指令）

- Peterson互斥锁是读-改-写指令出现之前的产物

- 下一节：更多锁实现及权衡

- 自旋锁的优点：简单、快速（当无争用时）、紧凑
- 自旋锁的缺点：不公平、不可扩展、能效低下

# 数据并行和异步I/O的视频



# Rayon: 数据并行 乐趣与利润

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



The screenshot shows a presentation slide with two Rust code snippets. The top snippet is a sequential function that iterates over a mutable array and increments each element. A blue arrow points down to the bottom snippet, which uses Rayon's `par_iter_mut()` to parallelize the iteration. A red text overlay on the right side of the slide states: `'c' not shared between iterations!`. In the bottom right corner, there is a small video inset showing a man, presumably Nicholas Matsakis, gesturing while speaking.

```
fn increment_all(counts: &mut [u32]) {
 for c in counts.iter_mut() {
 *c += 1;
 }
}
```

↓

```
fn increment_all(counts: &mut [u32]) {
 counts.par_iter_mut()
 .for_each(|c| *c += 1);
}
```

`'c' not shared  
between iterations!`

# 零成本异步IO

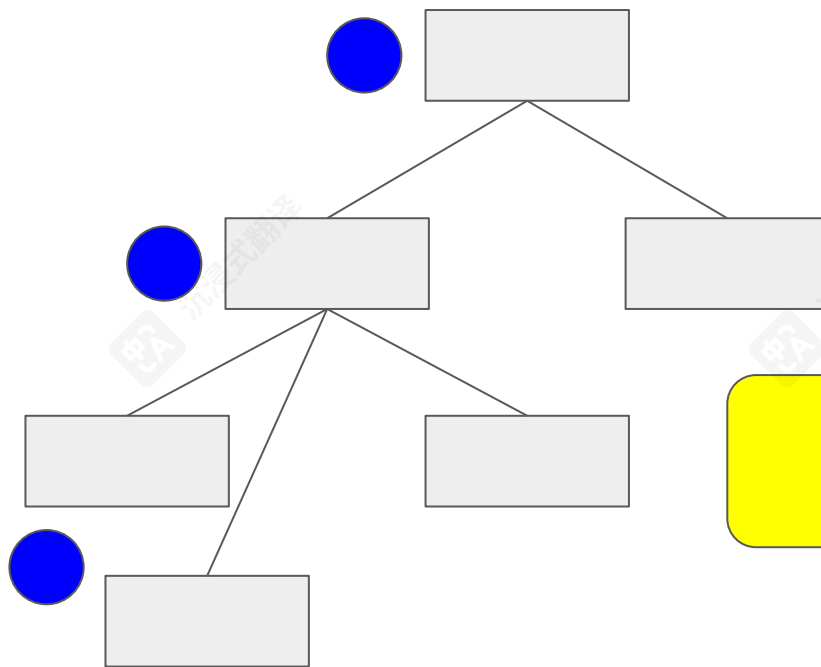
- Rust LATAM 2019
- (打开英文字幕)



# 乐观锁耦合

# 锁耦合（“手递手锁定”）

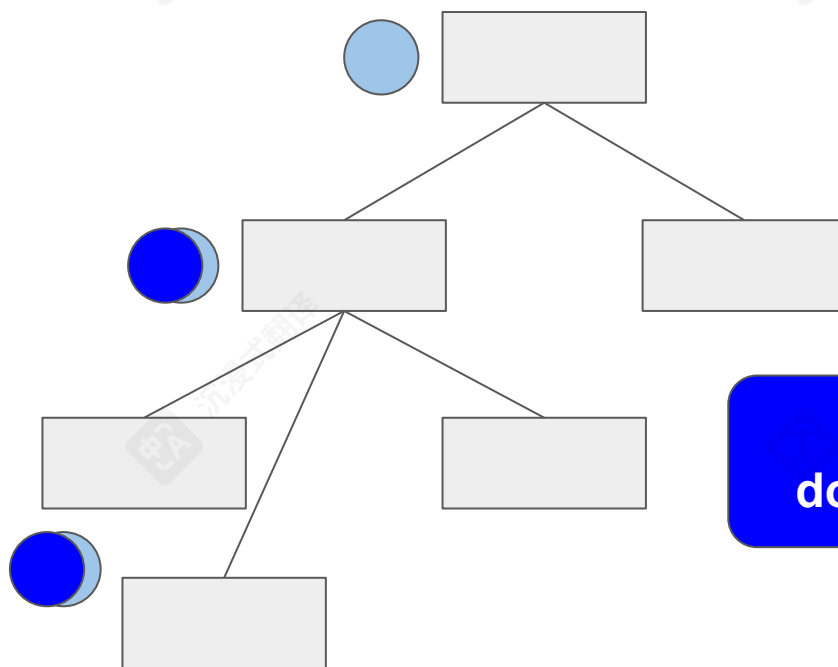
- 遍历时最多同时持有 2 把锁
  - 每个节点有自己的锁
  - 遍历时，持有“当前”节点及其父节点的锁
  - 持有父节点的锁以保护结构变更



**Read operations  
invalidates cache!**

# 乐观锁耦合 (OLC)

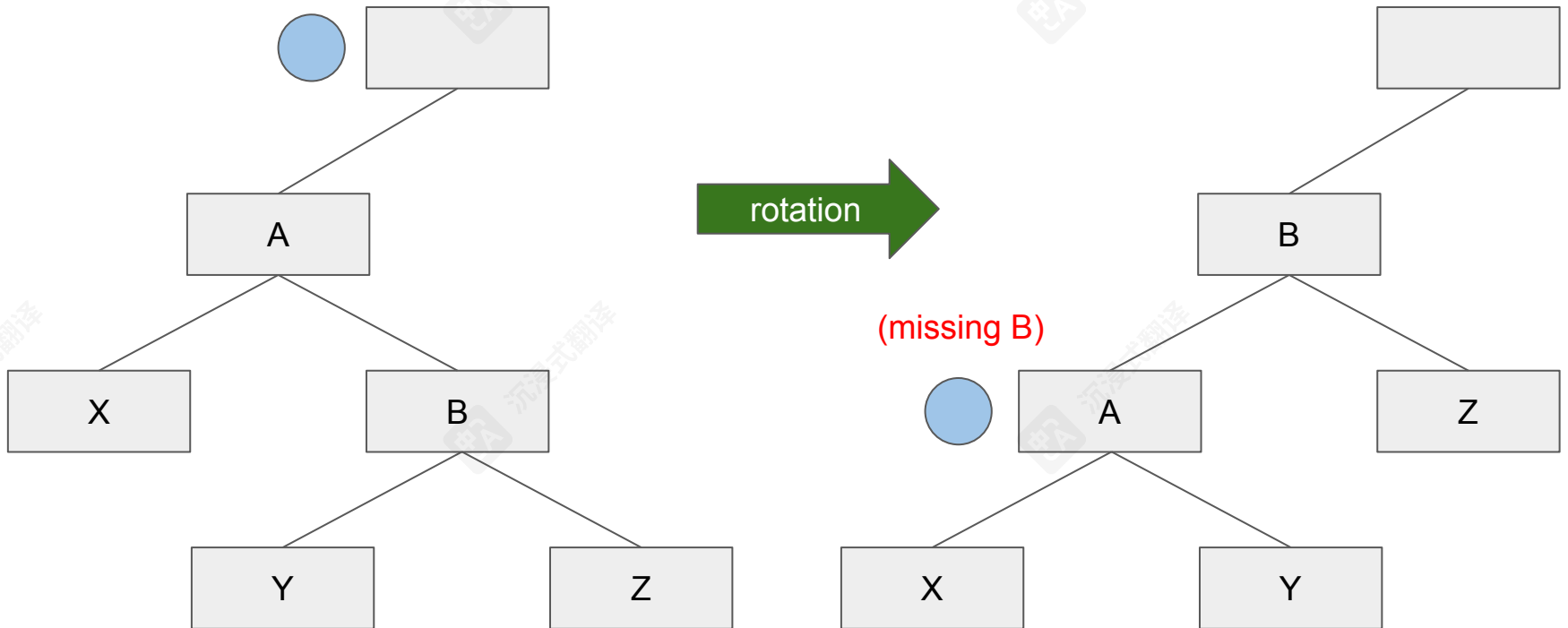
- 遍历时最多持有 2 个序列锁
  - 升级为写锁以保护结构变更



**Read operations  
do not invalidate cache!**

# Why acquiring locks of two nodes simultaneously?

- Otherwise, in traversal, we might miss some values (for example, the B below).



# 何时可以安全地释放节点？

- **问题：**读取不会在内存中留下任何痕迹。如何确保并发线程不会释放所访问的节点？
- **方法：**为安全释放设置独立的同步机制
  - “安全内存回收方案”或仅仅是垃圾回收
  - 例如：基于周期的回收、危险指针、...
  - 将在本课程后续部分讨论
- **假设基于时代的回收 (EBR)**
  - 操作由“临界区”分隔
  - 内存块仅在临界区内被访问和 退役
  - 退役块会在安全时自动 **稍后释放**
  - 实现： [Crossbeam](#) (Rust 中的并发库)

# Old Annoucements



# 作业1（截止日期：11月4日，占总成绩的20%）

- 在Rus中实现（序列）自适应基数树 t
  - <https://github.com/kaist-cp/cs431/issues/6>
  - 读取论文： <https://db.in.tum.de/~leis/papers/ART.pdf>读
  - 取骨架： [\[链接\]](#) \_\_\_\_\_
- 读取骨架的README.md以获取规范和建议

## 作业2（截止日期：11月29日，占总成绩的20%）

### ● 使用OLC同步二叉搜索树

- 注意：不是AVL树，而是普通二叉搜索树
- 阅读论文：<https://stanford-ppl.github.io/website/papers/ppopp207-bronson.pdf>
- 阅读骨架：[\[链接\]](#)
- 评分：待定

### ● 提示：尽早阅读论文！

# 期中考试（10月21日，占总成绩的20%）

- **日期和时间：**9:00-11:00，10月21日，2019

- **地点：**2111室，E3-1号楼

- **范围：**本课程所学的一切内容

- **带黑色/蓝色笔和学生证**

- **学习指南**

- 你将被要求“解释”现象的原因。你的解释应该是一个非正式证明，但有一些空白。你的非正式证明应该清晰和精确。

- 你将被要求关于课堂上展示的代码的实现细节（例如锁和ART）。你可能会被要求手写一个实现。

- 你将被要求关于视频讲座。

- **问题或评论？**

# 期末考试 (12月16日, 占总成绩的20%)

- 日期和时间: 9:00-11:00, 12月16日, 201年 9
- 地点: 2111室, E3-1号楼
- Cov 平均: 期中考试之后你所学到的一切 m
- 带 使用黑色/蓝色笔和你的学生证
- Study guide
  - 你将被要求“解释”现象的原因。你的解释应该是一个非正式证明, 但有一些空白。你的非正式证明应该是清晰和精确的。
  - 你将被要求关于课堂上展示的代码的实现细节。你可能会被要求手写一个实现。
  - 你将被要求关于视频讲座。
- 问题或评论?

# 垃圾滑梯

# 由于线程交错导致的非确定性 (TODO)

- 例如并发计数器：多个线程递增一个共享位置：静态 `COUNTER = AtomicUsize::new(0);` // 线程A和线程B让 `c = COUNTER.load(); COUNTER.store(c + 1);`

- **问题：**由于不幸的调度导致的意外行为

- [COUNTER=0] A load, B load, A store, B store COUNTER=1]

- **解决方案：**通过原子读取和写入禁止这种调度

- “读改写”，例如交换、比较并交换、获取并添加

- // 线程A和线程B let `c =`

- `COUNTER.fetch_and_add(1);`

- [COUNTER=0] A fetch\_and\_add, B fetch\_and\_add [COUNTER= 2]

# 引入宽松行为的优化

● **重排序**：除非访问相同位置，任何两个加载/存储/原子操作指令可以重排序。

- 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)

● **合并**：如果访问相同位置，TODO: 不引入?? 两个加载/存储/原子操作指令 **可能** 可以合并。(仅当对顺序程序有意义时)

- 例如 `X=1; X=2; r=X -> X=2; r=X` (存储-存储, 存储-加载合并)
- 例如 `X=1; X.fetch_and_add(1) -> X=2` (存储-原子操作合并)

● **消除**：如果加载的结果未被使用，加载可能被消除。

- 例如 `r=X; -> nop` (加载消除) TODO: 不引入? ?

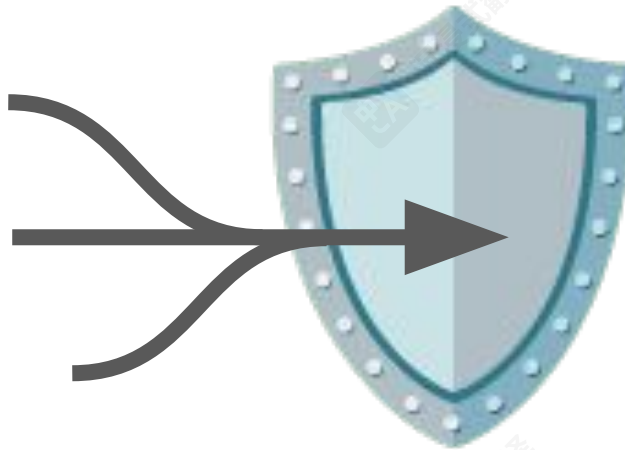
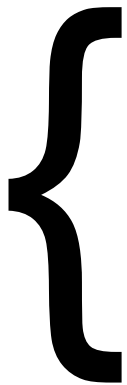
# 无阻塞性数据结构

- 关键思想：每个线程发布其正在进行的操作，  
“获胜”的线程通过完成其他线程发布的操作来帮助其他线程  
来帮助其他线程
- 帮助： <https://dl.acm.org/citation.cfm?id=102808>
- 扁平组合： <http://www.cs.bgu.ac.il/~hendlerd/papers/flat-combining.pdf>
- 示例： [一个与获取并添加一样快的无阻塞性队列](#)



# 指针/时代基础收集方法 (PEBR)

- 아이디어: PBR과 EBR의 <style id='19'>混合



- 最初是 EBR (快速), 内存 收集 不好 则 PBR로 淘汰 (收集 保证)

# Intermission: Crossbeam (TODO)

- [Crossbeam](#): 最广泛使用的Rus并发库
- 工具 (例如 缓存对齐)
- 指针值操作API
  - 已拥有: TODO
  - 原子: TODO
  - 共享: TODO
- 垃圾回收API (基于周期的回收)
  - 守卫: TODO
  - `defer_destroy()`: TODO
- 无锁数据结构
  - 栈, 队列, 工作窃取双端队列, 通道, ...

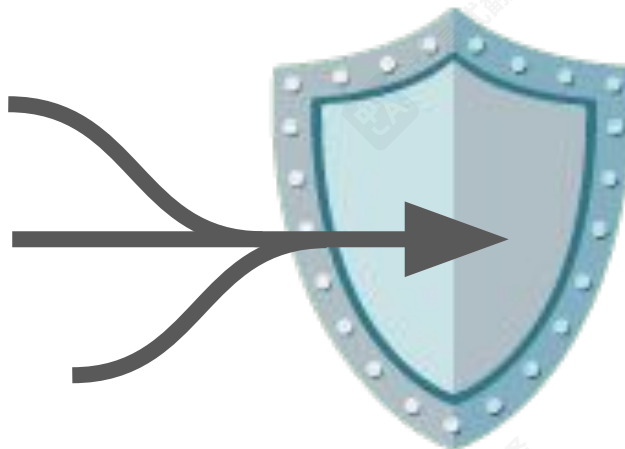
# 指针/时代基础回收 (PEBR)

- <https://cp.kaist.ac.kr/gc/>
- **想法: PBR 和 EBR 的混合**
  - EBR (快速) 首先使用, 当块未被回收时, 被弹出至 PBR (鲁棒)
- **Results**
  - 与 EBR 一样快 (85%-90% 吞吐量)
  - 与 PBR 一样鲁棒 (保证块被回收)
  - 便携, 通常适用于许多数据结构, 紧凑
    - **第一个sc 满足上述所有标准的主题 (即超级**
- 我们将学习PBR和EBR (作为背景), 以及PEBR

r)

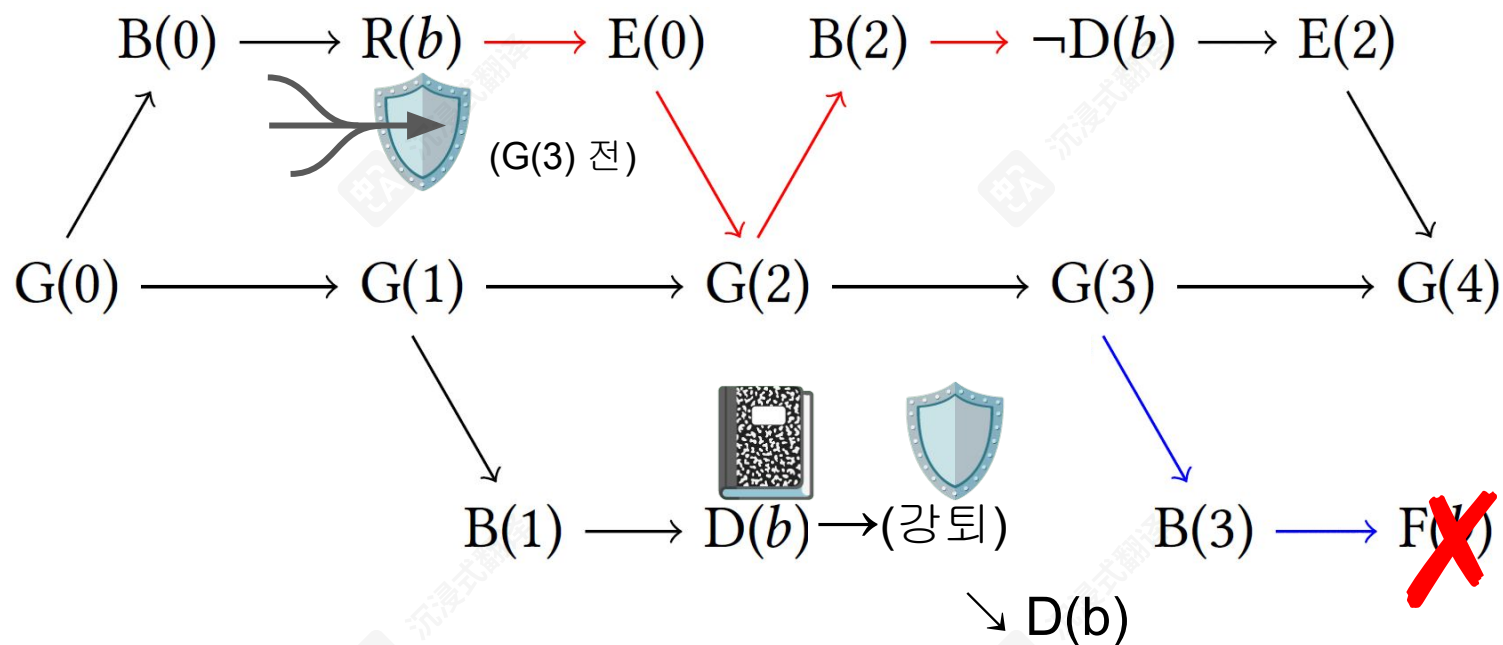
# 指针/时代基础收集方法 (PEBR)


- 아이디어: PBR과 EBR의 <style id='19'>混合



- 最初是 EBR (快速), 内存 收集 不好 如果 PBR则 淘汰 (收集 保证)

# 指针/时代基于收集方法：淘汰



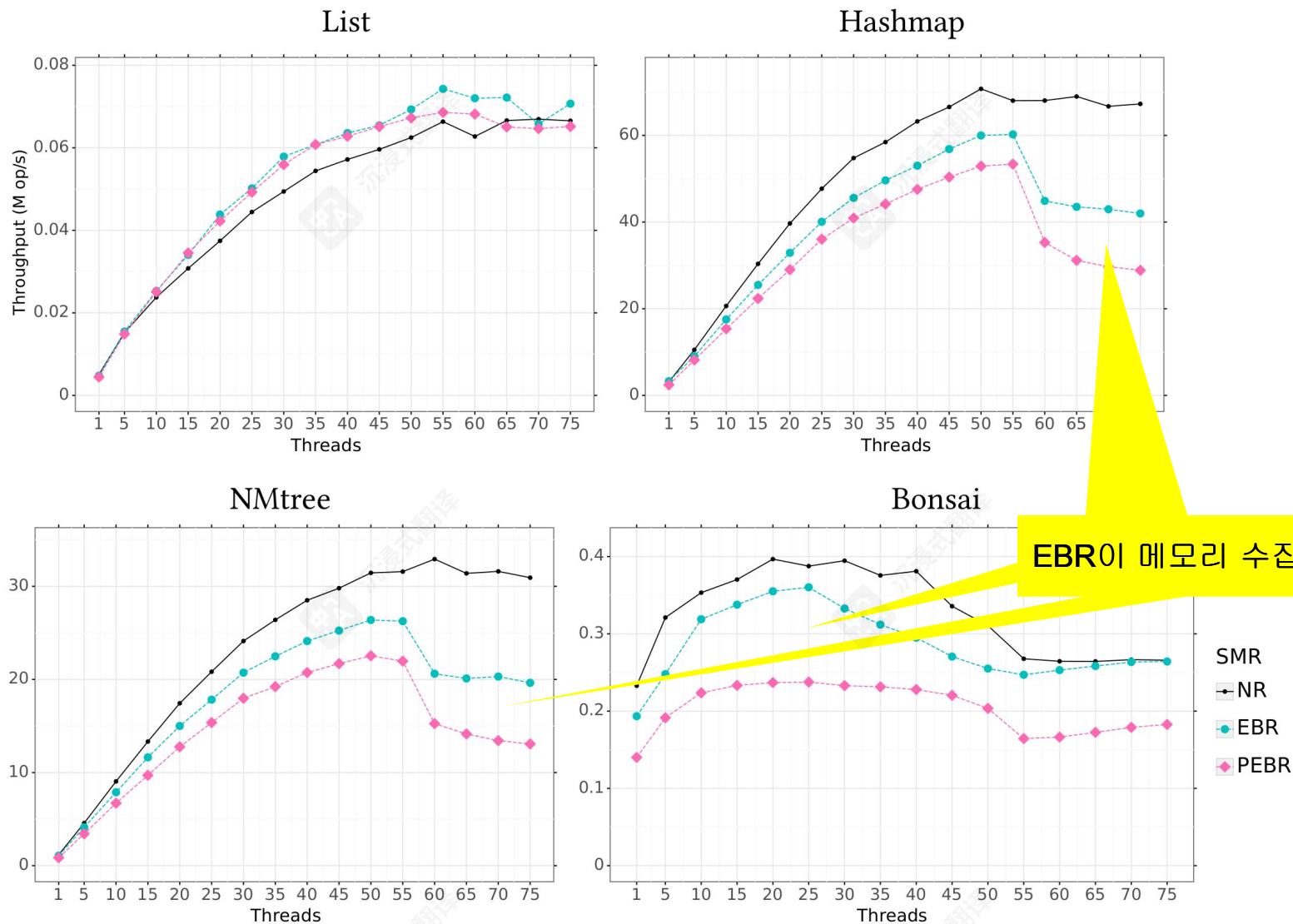
- EB基于R，集合过长时进行淘汰并开始新时代
- 淘汰时，集合中已访问的指针可再次访问， $G(3)$ 之后也不应释放
- 访问过的指针（在线程内）记录下来，淘汰时保护 

# 포인터/시대 기반 수집기법 장단점

- 优点 1: 快速 内存 收集 保证 (淘汰 通过)
- 优点 2: 淘汰 过程是 无锁 可移植性 较高
- 优点 3: 指针 需要 长时间 持有的情况 故意淘汰 可能
- 缺点 1: 访问的 指针 记录 运行 成本 (10-15%)
- 缺点 2: 编程 PBR/EBR相比 稍微 复杂

优点 1能 覆盖所有 缺点 并且 还有剩余

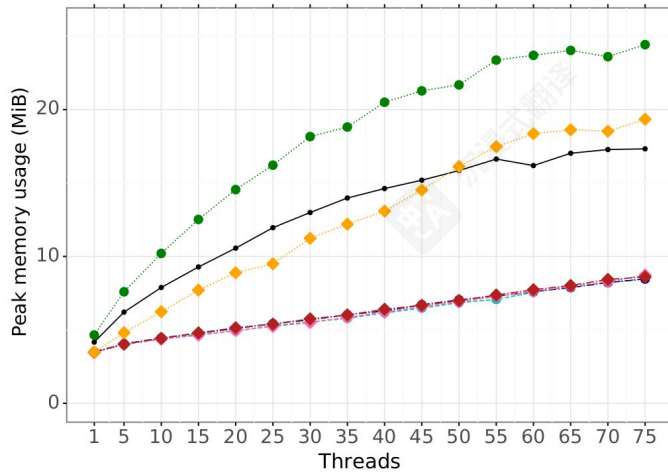
# 指针/时代基于收集方法结果 1: 吞吐量



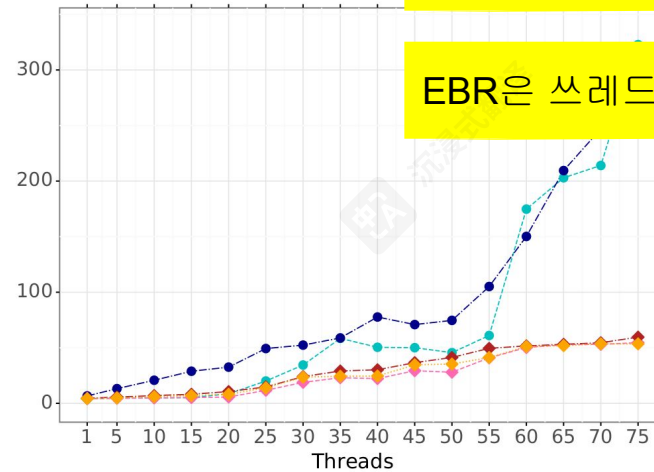
# 포인터/시대 기반 수집기법 결과 2: 메모리 사용량

Throughput이 적어 방향성 파악 실패

List



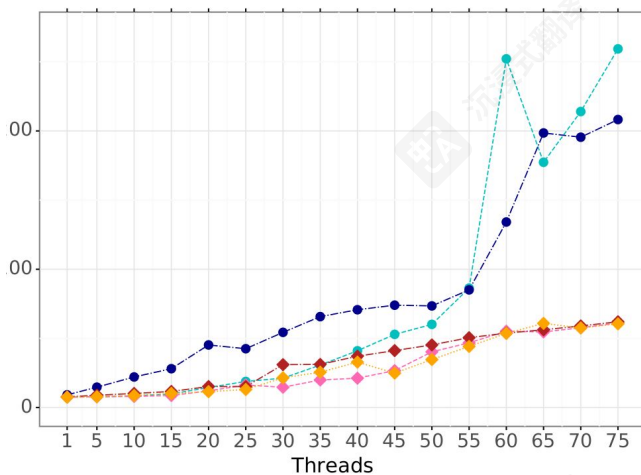
Hashmap



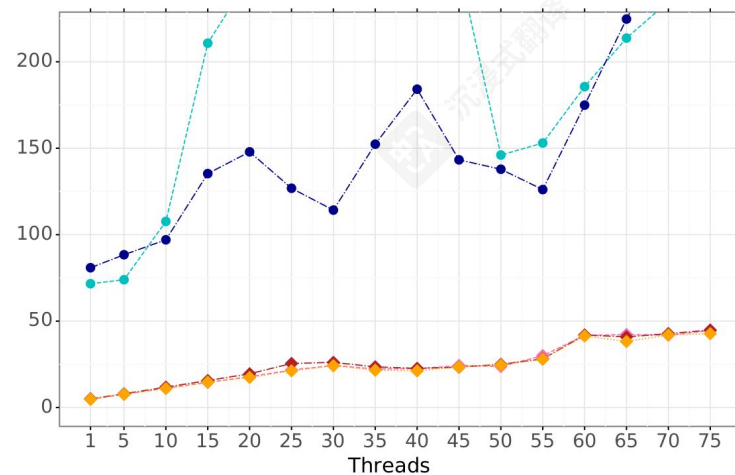
EBR, stalled는 메모리수집 아예 안함

EBR은 스레드 많으면 수집 느려짐

NMtree



Bonsai



SMR, interf.

- NR
- EBR
- EBR, 10ms
- EBR, stalled
- PEBR
- PEBR, 10ms
- PEBR, stalled



# HP中的同步：永远不会发生使用后释放

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

## ● 案例1：T2的P2发生在T1的R2之前

- P1对PL的写入对C2可见
- 要回收B，T1的C2应该确认B已从PL中删除（T2完成对B的访问后，然后从PL中删除B）
- T2对B的访问发生在T1对B的回收之前

## ● 案例2：T1的R2发生在T2的P2之前

- P3的验证失败，因为它无法从内存中读取B
- T2无法访问B

# 使用所有权分析共享可变状态

- **应用到并发：**适用于分析线程间的同步
  - 例如，如果线程 A 拥有一个缓冲区，线程 B 不能访问它
- **其所有权类型对于短信服务来说是一个非常实用的抽象**
  - **静态证明对短信服务的访问是安全的**
  - 不仅适用于顺序程序，也适用于并发程序
- **Key idea 1: disall默认借用共享可变操作 (SMOs)**
  - 资源要么是独占的，要么是不可变的（但不可同时两者）
  - **独占：**资源由其指定的 **所有者** 或其独占的 **借用者** 读取/写入；或
  - **不可变：**资源由其共享借用者读取
- **关键思想 2：以受控方式允许 SMOs**
  - 动机：SMOs 是必要且不可避免的（例如在并发中）
  - 机制：通过 **不安全** 块实现 **内部可变性**

# Rust和并发编程

- 使用SMOs实现
  - 并发编程中的关键要素
- 几乎不使用SMOs的接口
  - 为了简化并发编程
- 例如：使用非SMO接口的并发栈
  - 函数 `Stack::push(&self, 值: T);`
  - 函数 `Stack::pop(self) -> option<T>;`
- 使用编译时生命周期作为临界区生命周期的静态验证（稍后）

# Bonus: Rust的所有权类型基础

## ● 层3：所有权类型

- 计算检查器 / 数学“库”用于证明对SMS的访问安全

## ● 层2：生命周期逻辑

- 用于证明对生命周期和借用安全的数学库

## ● 层1：并发分离逻辑（CSL）

- 用于证明安全性的数学证明系统
- 所有权类型检查器作为 CSL 中安全性证明的“引理”
- 我们稍后会看到更多...

# Q: why are you so narcissistic...?

- 这门课程只涉及我的个人工作。
- 原因 1: 这是我能最好地教授的材料。
  - 我了解这项工作的来龙去脉, 包括陷阱和关键点。
- 原因 2: 我想关于并发交付一个连贯的叙述。
  - 没有现有的连贯叙述。
  - 我需要建立自己的体系 (语义学、设计模式、库、...)。
  - 这个课程: 第一次尝试