

第七章 并发程序测试与分析

蔡彦

计算机科学国家重点实验室
中国科学院软件研究所

yancai@ios.ac.cn

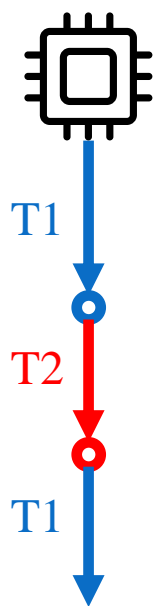
大纲

- 1) 并发执行简介
- 2) 偏序关系
- 3) 精准并发测试
- 4) 最新技术与趋势

1. 并发程序及其发展现状
2. 基本概念
 - 基础概念
 - 并发缺陷分类
3. 并发程序测试方法
 - 基本动态测试
 - 随机延迟扰动
4. 并发缺陷检测方法
5. 静态方法
 - 数据流分析
 - 符号执行
 - 模型检验
6. 动态方法
 - LockSet
 - 约束求解
 - 偏序方法-流式算法
 - 偏序方法-图算法
7. 最新技术与研究趋势
 - ToccRace
 - Eagle
8. 并发测试与分析技术总结

1. 并发程序及其发展现状

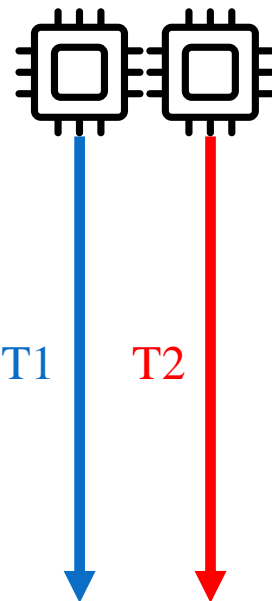
并发程序及其发展现状



并发（早期）

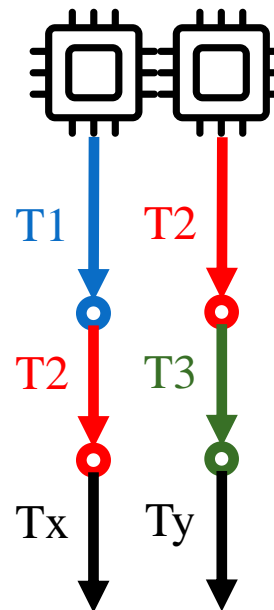
单个处理机上，同时有多个任务处于运行中状态，但同一时刻只有一个任务在处理

VS



并行

多个任务分别分布到多个不同的处理机上，同一时刻多个任务可以并行处理



并发（现在）

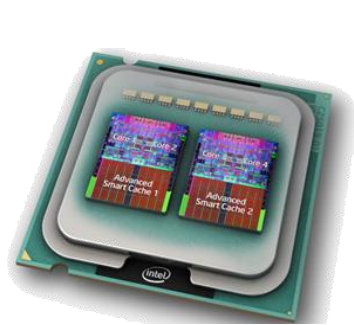
同时有多个任务处于运行中状态，但不再关注在硬件中的分布状态

本课程：同时有多个任务处于运行中状态，且任务以线程形式存在。

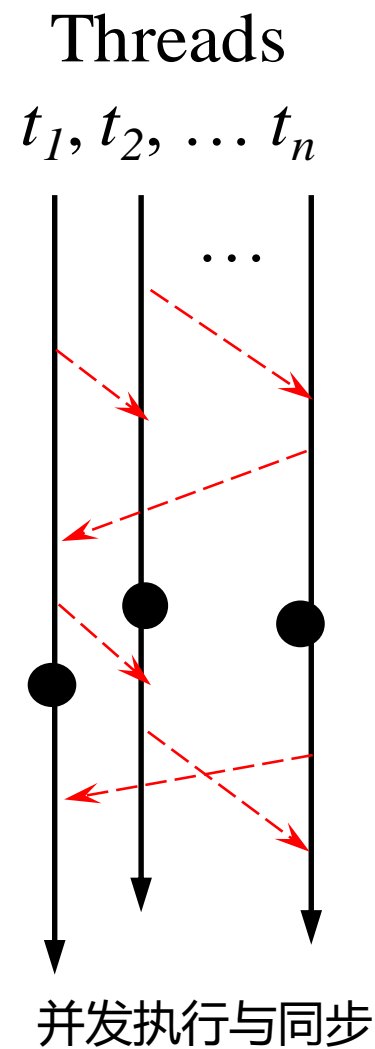
并发程序及其发展现状

➤ 并发程序的基本特点:

- 共享系统资源: 并发程序的各个线程共享系统中的各种资源, 如CPU、内存、I/O设备等。
- 共享数据: 并发程序中存在共享变量, 一个线程对共享变量的修改对于其他线程是立即可见的。
- 任务同步: 通过同步操作协调多个线程, 共同完成工作。
- 并发执行: 在同步操作之外, 各个线程可以并发执行, 互不干扰。
- 原子性: 并发程序中存在原子操作, 即一个指令序列, 要么全部执行成功, 要么全部不执行, 不会出现部分执行的状态。



•Thread 1
•Thread 2
•Thread 3
•Thread 4



并发程序及其发展现状

➤ 并发程序可以充分利用多核处理器的能力，提高系统的吞吐量和响应速度，充分利用硬件资源，提升系统性能，被广泛应用到计算机的各个领域：

- 数据库及文件系统：多个用户（或进程）可以同时同一数据（文件）进行读写操作，提升数据处理效率，同时通过同步等操作保证数据的一致性及可靠性。
- 网络通信：以并发方式进行数据传输和处理，通过同步保证数据的一致性和可靠性。
- Web应用：多用户同时访问和操作网页，通过并发保证响应的准确性和可靠性。
- 云计算：通过并发方式进行任务管理，提升IO以及数据处理的效率等。
- 人工智能：多线程计算加速、异步处理等。



并发程序及其发展现状

- 并发程序违反线性思维逻辑，因此编程难度极大，并发程序的正确性主要依靠同步操作维持，同步操作出现问题，就有可能引发并发缺陷，导致：
 - 破坏数据一致性
 - 引发系统崩溃
 - 导致程序漏洞，引发恶意攻击
- 并发程序测试与分析面临的主要问题：**状态空间爆炸**。同步操作之外，各个线程独立执行，线程之间的交错顺序随机，而并发缺陷仅在特定的交错顺序下才能触发。给定一个有 n 个线程 k 个指令的程序，不同的线程交错顺序数量：

$$\frac{(n \times k)!}{(k!)^n} \geq (n!)^k$$



Nasdaq's Facebook Glitch



Northeast blackout of 2003



Therac-25

并发程序及其发展现状

- 并发测试及分析工具的评测指标:

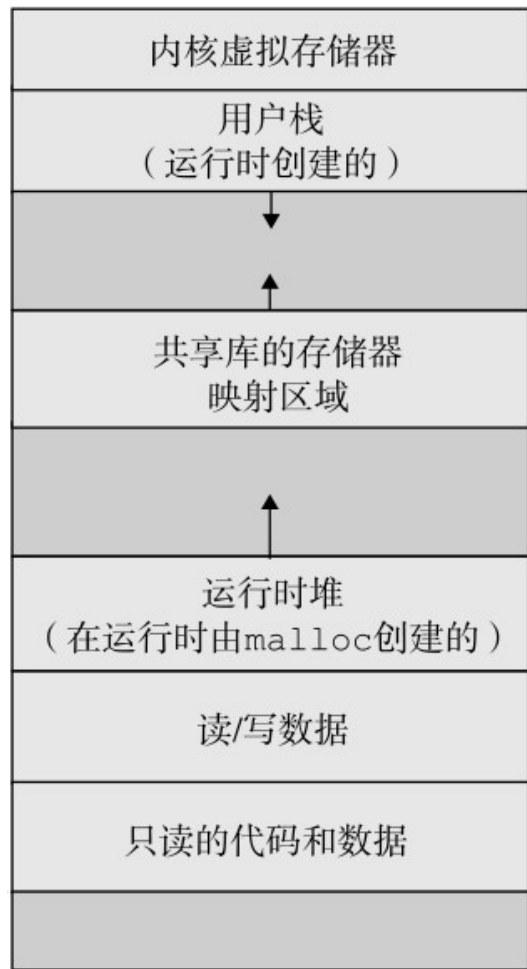
- 缺陷覆盖率
- 执行效率
- 准确度
- 可扩展性



2. 基本概念

基本概念

- 线程模型：线程模型定义了计算机程序中对线程进行调度和管理机制，包括线程的创建、调度、同步和通信以及终止等。
- 不同的编程语言可能采取不同的线程模型，即使是相同的语言内部，也存在多种不同的线程模型，本课程以**POSIX标准**为基础开展。
- Pthread是一个符合POSIX标准的跨平台C/C++线程库，主要包含以下特性：
 - 线程操作主要包括：创建、结束、同步、调度、数据管理和进程交互
 - 在同一进程中的所有线程共享同一虚拟地址空间
 - 同进程中的线程共享：用户及组ID、共享变量、文件描述符、信号及处理函数等
 - 除共享资源外，每个线程独占：线程ID（Tid）、寄存器、栈等



进程地址空间

基本概念

- 线程：操作系统能够进行运算调度的最小单位，每个线程有唯一的线程标号（Tid），用Tx表示标号为x的线程。
- 事件：最小的指令执行单元，多数情况下，一个事件对应一条指令，事件之间按照发生顺序依次编号，事件有全局编号(Gid)和线程内编号(Eid)，用(x,y)表示线程x中第y个事件。
- 线程操作：
 - 线程创建：除初始线程外，每个线程均由父线程创建，每个线程依附于一个函数，函数执行完成，则线程执行结束。
 - 线程结束：除线程函数执行完成外，线程可通过调用Pthread接口主动结束，也可通过调用Pthread接口命令其他线程退出。主线程运行结束时，所有子线程强制结束。
 - 线程等待：一个线程可通过调用pthread_join函数等待目标线程结束，等待过程中原线程阻塞，目标线程结束后继续执行。

	t ₁	t ₂
1	pt_fork(t ₂)	
2		lock(l) → 事件(2, 0)
3		p = 0
4		unlock(l)
	lock(l)	
	p = p + 1	
	unlock(l)	
		pt_exit()
	pt_join(t ₂)	

基本概念

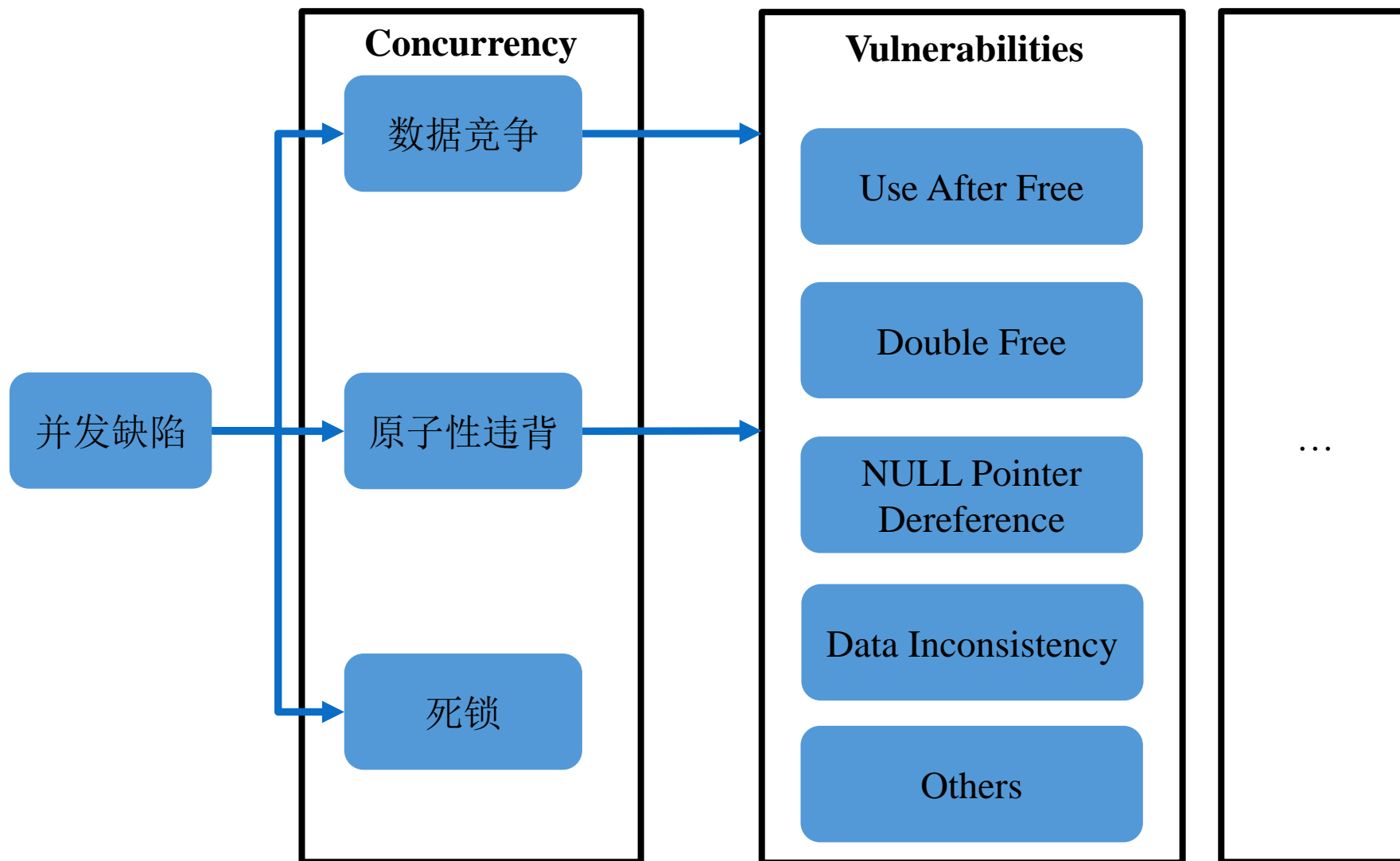
- 线程间同步：
 - 锁：互斥锁、读写锁等，操作包括初始化、加锁、解锁和销毁。
 - 信号量：本质上是计数器，操作包括初始化、等待(wait)、通知(signal)、广播(broadcast)和销毁。
 - 原子操作：一个指令序列，要么全部执行成功，要么全部不执行，不会出现部分执行的状态。
 - Barrier：用于同步多个线程的执行进度，线程到达Barrier所在位置时会阻塞，直至其他所有目标线程都到达Barrier，然后才会继续执行，操作包括初始化、等待(wait)和销毁。
- 线程间通信：
 - 共享变量及内存：由于多个线程都可以访问共享变量及内存，因此可以通过对共享变量和内存进行读写来实现线程间通信，主要操作包括分配、读、写和释放。
 - 网络通信等：线程可通过网络套接字等形式通过网络协议实现通信。

	t ₁	t ₂
1	pt_fork(t ₂)	
2		lock(l) → 事件(2, 0)
3		p = 0
4		unlock(l)
	lock(l)	
	p = p + 1	
	unlock(l)	
		pt_exit()
	pt_join(t ₂)	

基本概念

➤ 常见并发缺陷：


- 数据竞争
- 死锁
- 原子性违背



基本概念

- 数据竞争：多个线程同时访问共享数据，且其中至少一个操作为写操作。
- 左图中，line 2和4构成对指针p的数据竞争，在右图的线程交错中，会导致UAF
- 引起的危害：
 - 破坏数据一致性
 - 进一步发展为UAF、DF等漏洞

	t ₁	t ₂
1	p = malloc()	p[0] = 0
2		
3	free(p)	
4	p = null	

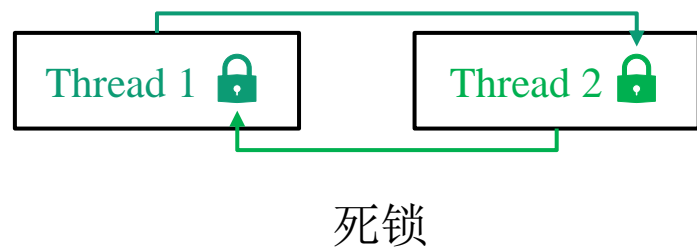


	t ₁	t ₂
1	p = malloc()	
2	free(p)	
3	p = null	
4		p[0] = 0

if(p!=NULL) p[0]=0;

基本概念

- 死锁：一组线程中，每个线程都持有其他线程所等待的资源，且这种等待关系形成闭环，导致这些线程被阻塞。
- 右图中，线程1持有锁1，等待锁2，线程2持有锁2，等待锁1，构成死锁
- 引起的危害：程序挂起，服务不可用。



	t_1	t_2
1	lock(1)	
2		lock(2)
3	lock(2)	lock(1)
4	unlock(2)	unlock(1)
5	unlock(1)	unlock(2)

基本概念

- 原子性违背：一个必须原子性执行的指令序列，其涉及的变量，在执行过程中被其他线程访问。
- 程序期待左侧指令 $a = a + 1$ 按照原子执行，实际执行中，读取到 a 的值后，写入新值之前， a 被修改，导致数据不一致，触发原子性违背
- 引起的危害：
 - 破坏数据一致性
 - 引发其他并发缺陷
 - 引发异常行为甚至系统崩溃

	t_1	t_2
1		$a = 0$
2		$a = 1$
3	$a = a + 1$	

Val(a) is 2.

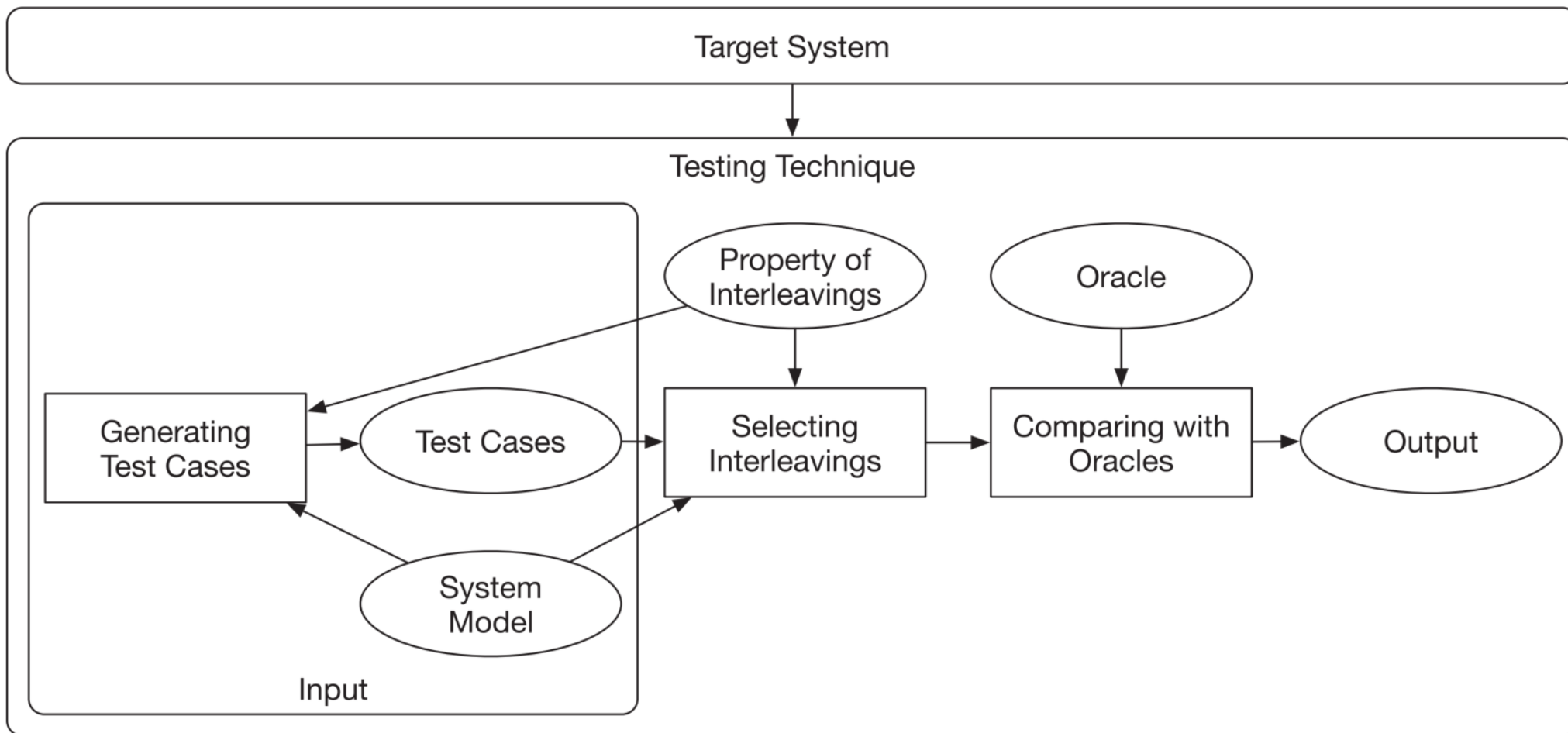


	t_1	t_2
1		$a = 0$
2	read(a): reg1 = 0	
3		$a = 1$
4	$a = \text{reg1} + 1$	

Val(a) is 1.

3. 并发程序测试方法

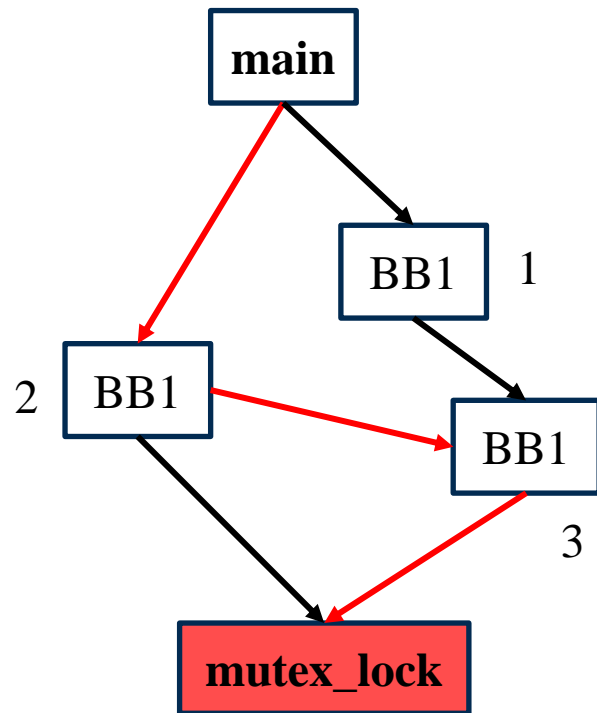
并发程序测试方法



一个通用的并发程序测试框架

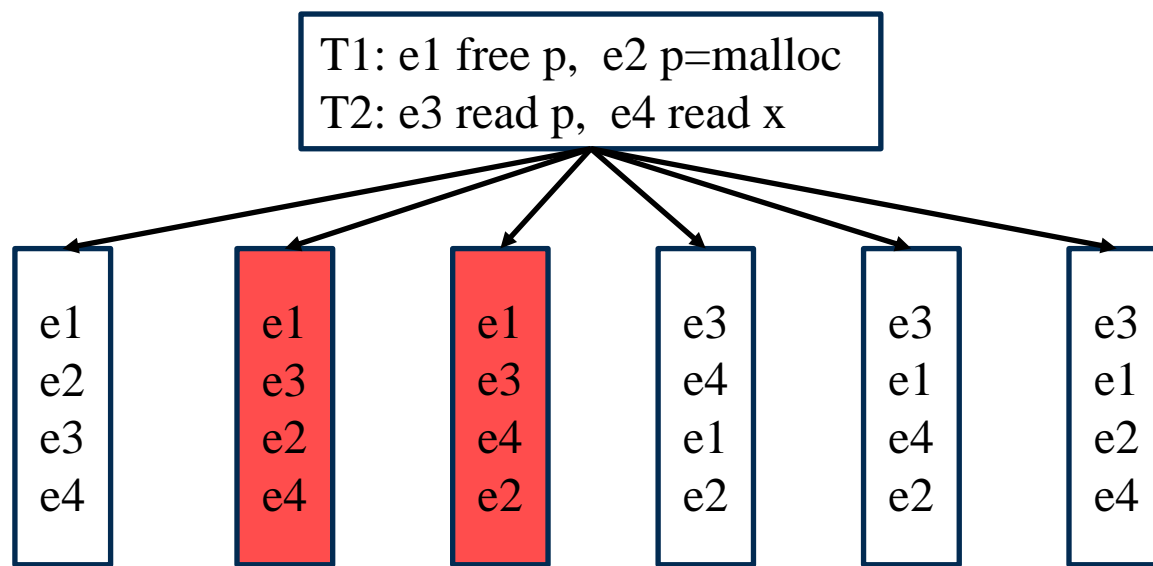
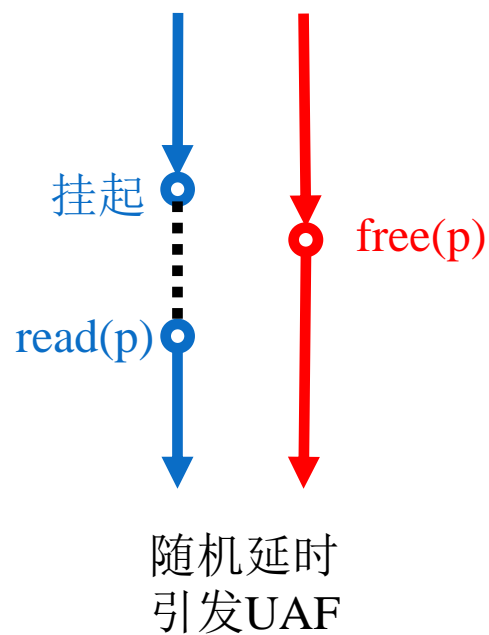
并发程序测试方法

- 基本动态（模糊）测试：通过结合一定的线程调度方法、并发缺陷检测方法，引导模糊测试，触发并发缺陷。关键点在于输入质量评估。
- 常见技术：ConFuzz、RaceFuzzer、DeadlockFuzzer等。
- ConFuzz：
 - 从线程相关函数调用点开始，逆向追踪所有从main函数到达该调用点的所有路径
 - 对路径上的基本块添加评分（评分依据为距离线程函数调用点的距离）
 - 给定输入，追踪其执行轨迹，根据轨迹上的基本块评分，结合已有的代码覆盖情况，给输入一个综合评分
 - 选择综合评分高的输入作为种子，再进行变异生成新输入
- 优点：有一定的导向性，可以实际触发缺陷，结果准确。
- 缺点：线程间的交错具有随机性，即使是相同的执行路径，多次执行的结果可能也不一样，会导致大量的漏报。



并发程序测试方法

- 随机延时扰动：在并发程序中的敏感位置添加延迟函数，延迟时长随机，敏感位置一般包括访问共享变量、加锁、解锁等。完全依靠随机，效果一般。
- 调度枚举方法：尽可能枚举更多的线程交错顺序。面临状态空间爆炸问题，一般在一个切片范围内进行枚举，但并发缺陷一般较为稀疏，即使切片，检测效率也不高，同时切片引入了大量的漏报。



调度枚举

4. 并发缺陷检测方法

并发缺陷检测方法

- 静态检测技术：不执行程序，而是在源代码（或者中间表示代码）基础上直接进行分析，通过收集程序的**数据流、控制流**信息，对程序的执行序列进行状态模拟或预测，判断是否出现违法的执行序列，从而检测并发缺陷。现有静态检测技术主要受制于两个方面：**分析精度以及执行效率**。
- 动态检测技术：通过分析程序的运行时信息进行缺陷检测，不同的动态方法有不同的特性。
 - LockSet：Eraser等
 - 约束求解：Said、RVPredict、RDIT、Dirk
 - 偏序方法
 - 流式算法：HB、SHB、CP、WCP、SyncP等
 - 图算法：M2、SeqCheck、ToccRace、Eagle等



5. 静态方法

静态方法-数据流分析

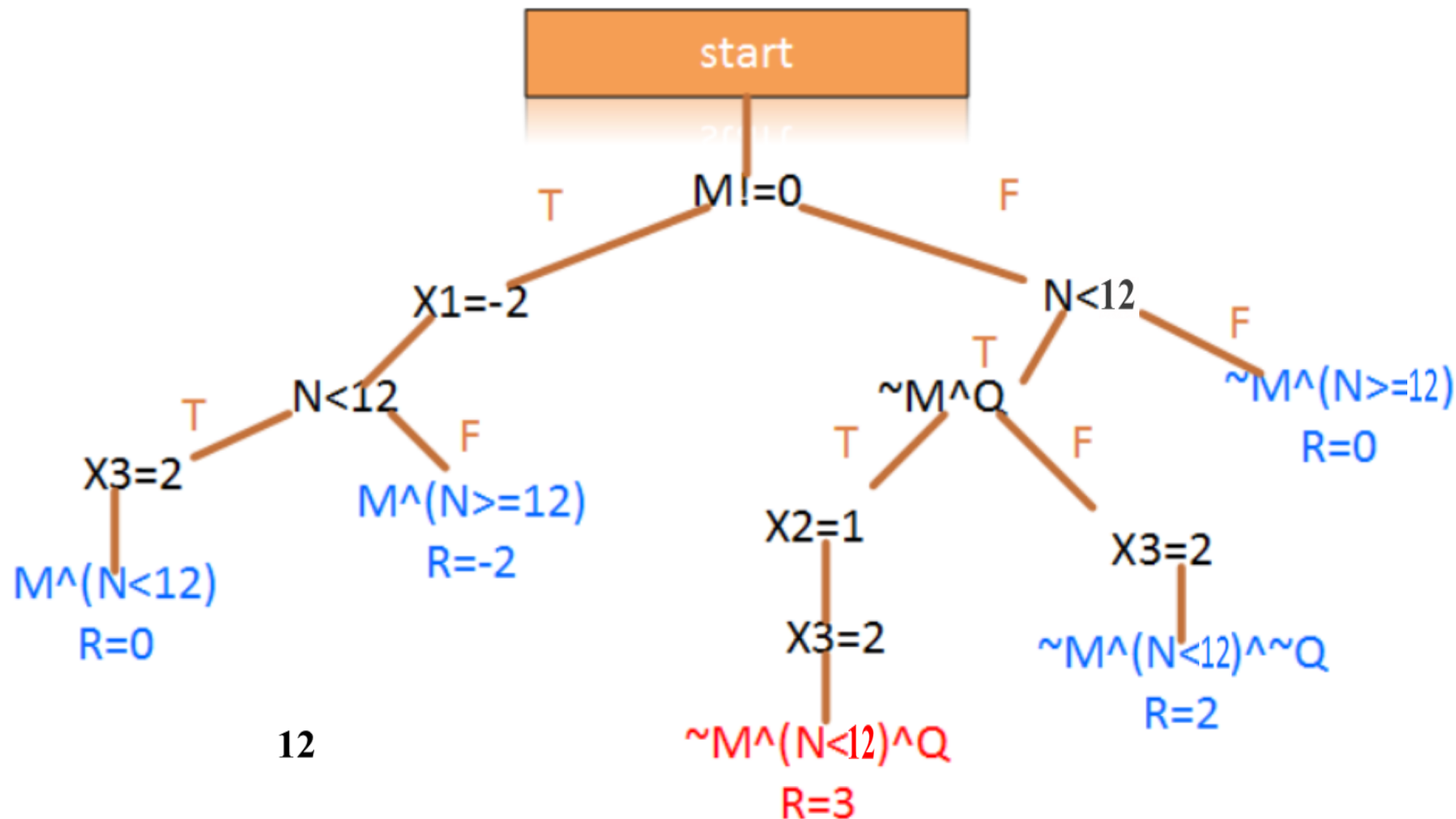
- 数据流分析技术：通常在程序的控制流图基础上，针对敏感操作的代码（如锁操作）进行分析，一般采用过程间分析，记录程序不同路径上的数据流信息、锁相关信息等，并根据要检测的目标缺陷的模式进行匹配，检测并发缺陷，常见工具有RacerX、LockLint等。
- RacerX：
 - 为所有函数构建控制流图，然后根据函数调用关系链接成一个全局控制流图
 - 图中没有调用者的结点为根结点
 - 从根节点开始执行DFS(深度优先搜索)，遍历所有基本块
 - 遍历过程中，针对锁操作，记录锁的依赖关系，例如当前持有锁a，正在获取锁b，则记录a→b
 - 上述遍历完成后，给定需要考虑的线程数量，根据锁的依赖关系计算是否存在环路，检测死锁
- 优势：扩展性好，执行效率较高，可以覆盖大量缺陷
- 缺点：精度低，会产生大量误报

静态方法-符号执行

- 符号执行技术：利用符号表示程序状态，以及状态之间的转移，在此基础上进行形式化的语义分析。
 - 符号化：利用符号表示变量，具体而言，维护一个名为符号状态的全局变量，记录变量到符号表达式的映射。
 - 获取控制流：分析程序的指令，确定程序的控制流。
 - 状态维护：沿着控制流遍历程序指令，遍历过程中，维护两个全局变量：
 - 符号状态
 - 符号化路径约束：记录当前路径下，符号之间满足的约束
 - 约束求解：给定安全约束，判定路径可行性以及是否满足安全约束
- 并发缺陷检测工具：Whoop、CONTESSA等
- 优势：扩展性好，精度较高
- 缺点：执行效率低，面临状态空间爆炸问题，一般情况下会添加限制，限定分析范围（例如循环次数、函数深度），由此带来了大量的漏报

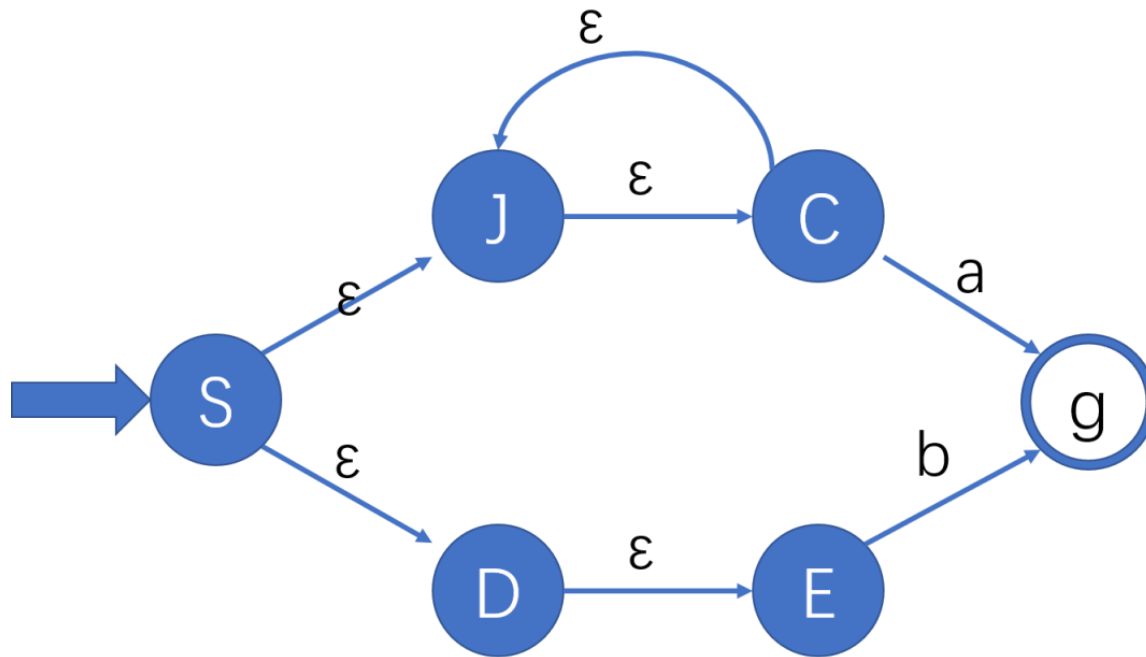
静态方法-符号执行

```
1 int m=M, n=N, q=Q;
2 int x1=0,x2=0,x3=0;
3 if(m!=0)
4 {
5     x1=-2;
6 }
7 if(n<12)
8 {
9     if(!m && q)
10    {
11        x2=1;
12    }
13    x3=2;
14 }
15 assert(x1+x2+x3!=3)
```



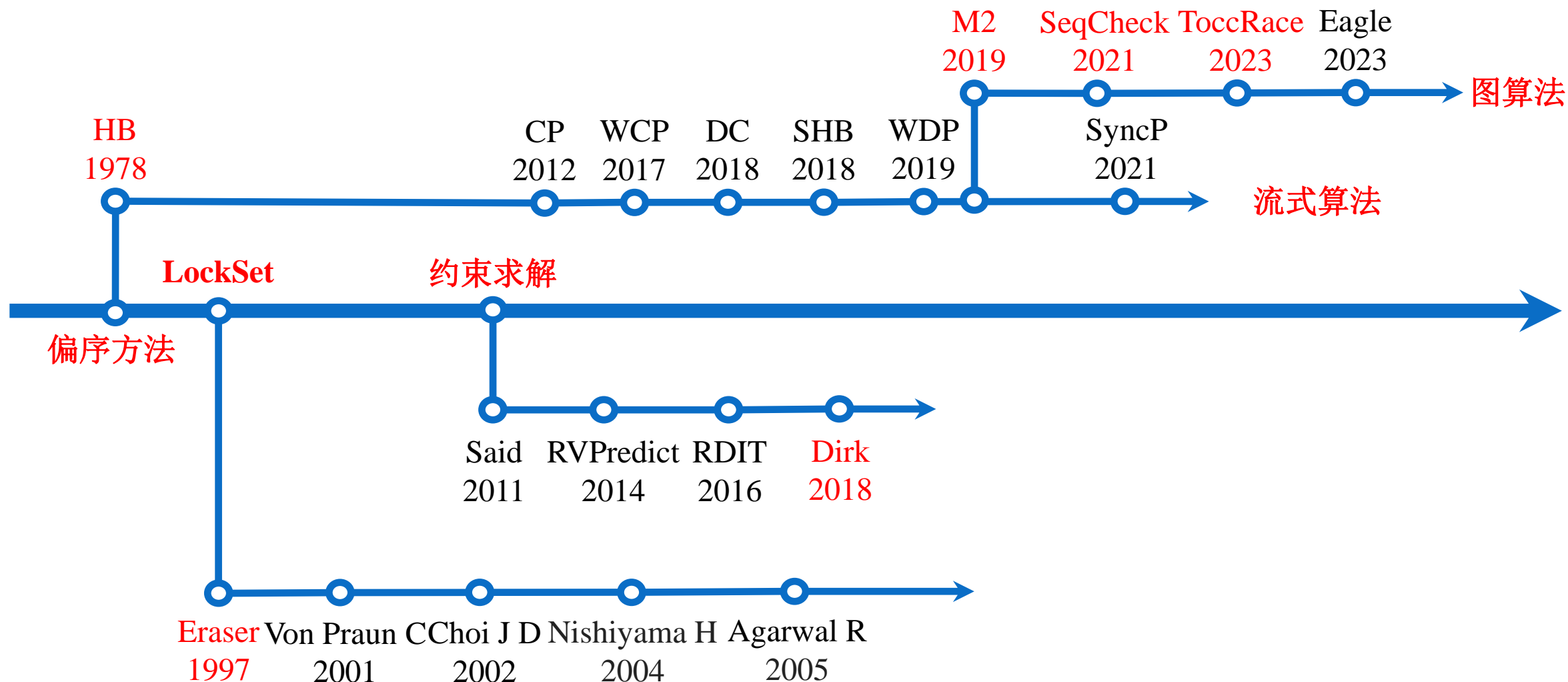
静态方法-模型检验

- 模型检验技术：利用状态迁移系统（例如自动机）表示程序的行为，用模态逻辑公式描述系统性质，之后执行检验算法，验证程序是否满足要求的性质。
- 并发缺陷检测工具：RCMC、ZING、TCBMC等
- 优势：扩展性好，精度高，在理论情况下不存在误报和漏报。
- 缺点：执行效率低，面临状态空间爆炸问题，复杂系统抽象难度高。



6. 动态方法（第一部分）

并发缺陷检测-动态方法



动态方法-LockSet

- 1997年由Savage S, Burrows M, Nelson G等人提出：Eraser: A dynamic data race detector for multithreaded programs[J].
- 基本思想：在多线程程序中，我们一般都会使用锁对共享变量进行保护，如果一个共享变量在程序多线程执行过程中能够始终被一个或多个锁保护的话，那么在该共享变量上肯定不会发生数据竞争。反之，则有可能发生数据竞争。
- 算法思路：对目标程序进行插桩，在程序运行时实时维护共享内存的锁集，通过锁集是否为空判断是否可能引发并发缺陷。

动态方法-LockSet

➤ 朴素的LockSet算法:

- 对每个线程 t , 维护一个 $\text{locks_held}(t)$, 记录 t 当前持有的锁的集合（持有是指加锁之后还未解锁）
- 对每一个共享变量 v , 维护一个锁集 $C(v)$, 记录着所有对 v 进行访问的操作共同持有的锁的集合
- 初始状态下, $C(v)$ 包含所有的锁, $\text{locks_held}(t)$ 为空
- 当线程 t 对锁 l_1 加锁时, $\text{locks_held}(t) = \text{locks_held}(t) \cup \{l_1\}$; 对锁 l_1 解锁时, 将 l_1 从 $\text{locks_held}(t)$ 中移除
- 当线程 t 对共享变量进行访问时（读或者写）, 更新 $C(v) = C(v) \cap \text{locks_held}(t)$
- 如果执行过程中, $C(v)$ 变成空集, 就认为此处存在并发缺陷（数据竞争）

动态方法-LockSet

共享变量a，锁 l_1 和 l_2

初始: $C(a) = \{l_1, l_2\}$, $locks_held(t_1) = locks_held(t_2) = \{\}$

	t_1	t_2	
$locks_held(t_1) = \{l_1\}$	1 $lock(l_1)$		
$C(a) = C(a) \cap locks_held(t_1) = \{l_1\}$	2 $a = 0$		
$locks_held(t_1) = \{\}$	3 $unlock(l_1)$		
	4	4 $lock(l_2)$	$locks_held(t_2) = \{l_2\}$
	5	5 $a = 1$	$C(a) = C(a) \cap locks_held(t_2) = \{\}$
	6	6 $unlock(l_2)$	$locks_held(t_2) = \{\}$

RACE!



数据竞争: line 2, line 5

动态方法-LockSet

共享变量a, b, 锁 l_1 和 l_2

初始: $C(a) = C(b) = \{l_1, l_2\}$, $locks_held(t_1) = locks_held(t_2) = \{\}$

	t_1	t_2	
$locks_held(t_1) = \{l_1\}$	1 $lock(l_1)$		
$C(a) = C(a) \cap locks_held(t_1) = \{l_1\}$	2 $a = 0$		
$locks_held(t_1) = \{\}$	3 $unlock(l_1)$		
	4	$lock(l_2)$	$locks_held(t_2) = \{l_2\}$
	5	$a = 1$	$C(a) = C(a) \cap locks_held(t_2) = \{\}$
	6	$unlock(l_2)$	$locks_held(t_2) = \{\}$
$C(b) = C(b) \cap locks_held(t_1) = \{\}$	7 $b = 0$		

RACE!



误报: 只有一个线程进行过访问!

动态方法-LockSet

共享变量a, 锁 l_1 和 l_2

初始: $C(a) = \{l_1, l_2\}$, $locks_held(t_1) = locks_held(t_2) = \{\}$

	t_1	t_2
$locks_held(t_1) = \{l_1\}$	1 $lock(l_1)$	
$C(a) = C(a) \cap locks_held(t_1) = \{l_1\}$	2 $x = a + 1$	
$locks_held(t_1) = \{\}$	3 $unlock(l_1)$	
	4	$lock(l_2)$ $locks_held(t_2) = \{l_2\}$
	5	$y = a + 1$ $C(a) = C(a) \cap locks_held(t_2) = \{\}$
	6	$unlock(l_2)$ $locks_held(t_2) = \{\}$

误报: 只有读操作!



动态方法-LockSet

共享变量a, 读写锁 l_1

初始: $C(a) = \{l_1\}$, $locks_held(t_1) = locks_held(t_2) = \{\}$

	t_1	t_2
$locks_held(t_1) = \{l_1\}$	1 $rdlock(l_1)$	
$C(a) = C(a) \cap locks_held(t_1) = \{l_1\}$	2 $x = a + 1$	
$locks_held(t_1) = \{\}$	3 $unlock(l_1)$	
	4	$rdlock(l_1)$ $locks_held(t_2) = \{l_1\}$
	5	$a = 1$ $C(a) = C(a) \cap locks_held(t_2) = \{l_1\}$
	6	$unlock(l_1)$ $locks_held(t_2) = \{\}$

漏报: 写操作只持有读锁!

动态方法-LockSet

- 朴素的LockSet约束无法处理的情况：
 - **Initialization:** 共享变量初始化之后，其他线程未访问共享变量，此时即使没有锁保护，也不会产生数据竞争。
 - **Read-shared data:** 共享变量只有在初始化的时候被写访问过，其他后续线程对这个共享变量只有读操作，那么也不会产生数据竞争（数据竞争至少要有有一个写操作）。
 - **Read-Write locks:** 读写锁（允许多个线程获取读锁，只允许一个线程获得写锁；获得了读锁之后，写锁将会被阻塞，直到所有获得读锁线程释放；而获得写锁之后，所有读锁将会被阻塞，直到获得写锁的线程释放），这时候允许多个线程获得读锁并对共享变量进行访问。

动态方法-LockSet

➤ Improved LockSet:

➤ 为共享变量添加状态描述，采用了状态图来描述共享变量生命周期的变化以及根据状态图实施并发缺陷检测。

➤ 状态转移:

➤ 共享变量初始状态为Virgin

➤ 初始化（写操作）之后转换为Exclusive

➤ 当前状态为Exclusive:

➤ 如果遇到第一个线程的读写，转移为Exclusive

➤ 如果遇到其他线程的读，转移为Shared

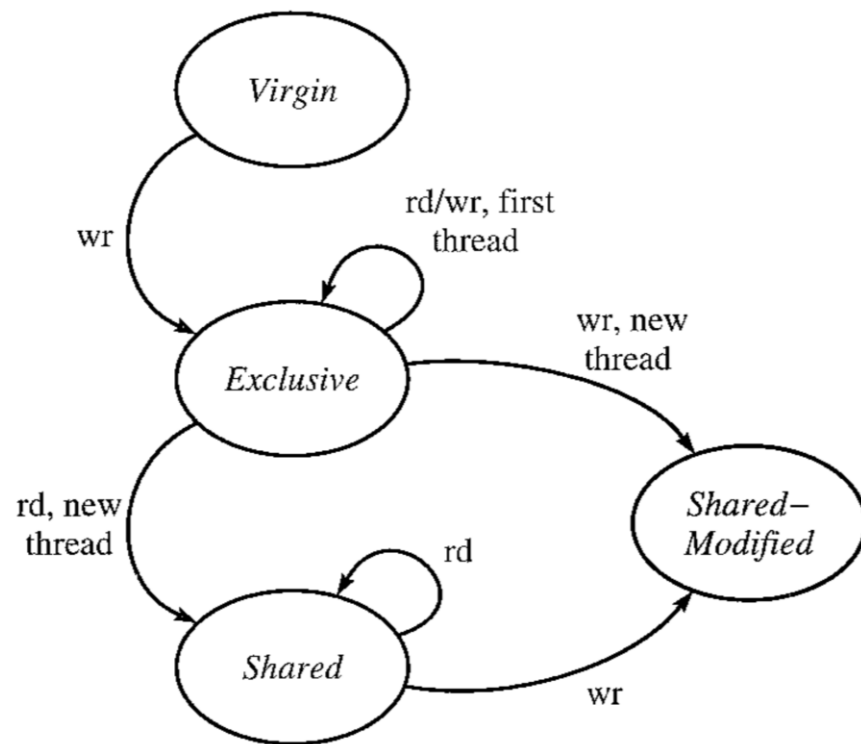
➤ 如果遇到其他线程的写，转移为Shared-Modified

➤ 当前状态为Shared:

➤ 如果遇到读，转移为Shared

➤ 如果遇到写，转移为Shared-Modified

➤ 只有处于Shared-Modified状态的变量才有可能有数据竞争。



动态方法-LockSet

- Improved LockSet:

- 除 $C(v)$ 外，为每个线程 t 维护两个锁集：

- $locks_held(t)$: 记录线程 t 当前持有的锁的集合（无论是以写模式持有还是以读模式持有）

- $write_locks(t)$: 记录线程 t 当前以写模式持有的锁的集合

- 锁集计算：

- 初始状态下， $C(v)$ 包含所有的锁， $locks_held(t)$ 为空， $write_locks(t)$ 为空

- 当线程 t 对锁 l_1 加锁时， $locks_held(t) = locks_held(t) \cup \{l_1\}$ ；对锁 l_1 解锁时，将 l_1 从 $locks_held(t)$ 中移除

- 当线程 t 对锁 l_1 以写模式加锁时， $write_locks(t) = write_locks(t) \cup \{l_1\}$ ；对锁 l_1 解锁时，将 l_1 从 $write_locks(t)$ 中移除

- 当线程 t 对共享变量进行读访问时，更新 $C(v) = C(v) \cap locks_held(t)$

- 当线程 t 对共享变量进行写访问时，更新 $C(v) = C(v) \cap write_locks(t)$

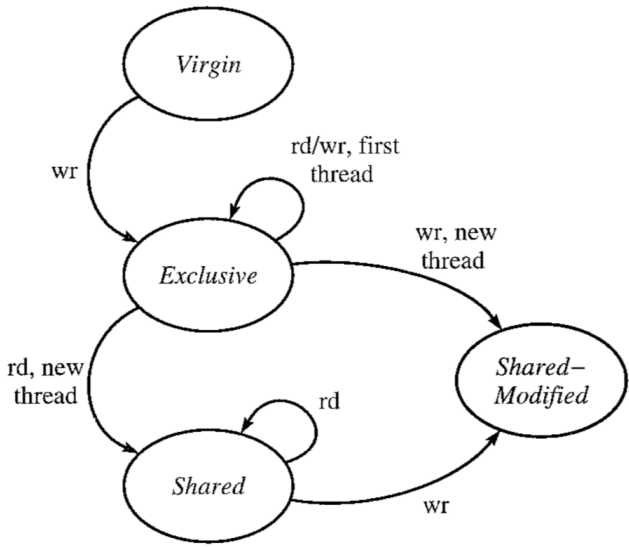
- 如果执行过程中， $C(v)$ 变成空集，就认为此处存在并发缺陷（数据竞争）

动态方法-LockSet

共享变量a, b已经初始化, 锁 l_1 和 l_2

初始: $C(a) = C(b) = \{l_1, l_2\}$, $S(a) = S(b) = \text{Exclusive}$

$\text{locks_held}(t_1) = \text{locks_held}(t_2) = \text{write_locks}(t_1) = \text{write_locks}(t_2) = \{\}$



$\text{locks_held}(t_1) = \{l_1\}$

$C(a) = \{l_1\}$, $S(a) = \text{Exclusive}$

$\text{locks_held}(t_1) = \{\}$

$C(b) = \{\}$, $S(b) = \text{Exclusive}$

b处于Exclusive状态, 不构成并发缺陷!

	t_1	t_2
1	$lock(l_1)$	
2	$a = 0$	
3	$unlock(l_1)$	
4		$lock(l_2)$
5		$a = 1$
6		$unlock(l_2)$
7	$b = 0$	

$\text{locks_held}(t_2) = \{l_2\}$

$C(a) = \{\}$, $S(a) = \text{Shared-Modified}$

$\text{locks_held}(t_2) = \{\}$

RACE!



动态方法-LockSet

共享变量a已经初始化，锁 l_1 和 l_2
初始：C(a) = { l_1, l_2 }, S(a) = Exclusive

locks_held(t_1) = locks_held(t_2) = write_locks(t_1) = write_locks(t_2) = {}

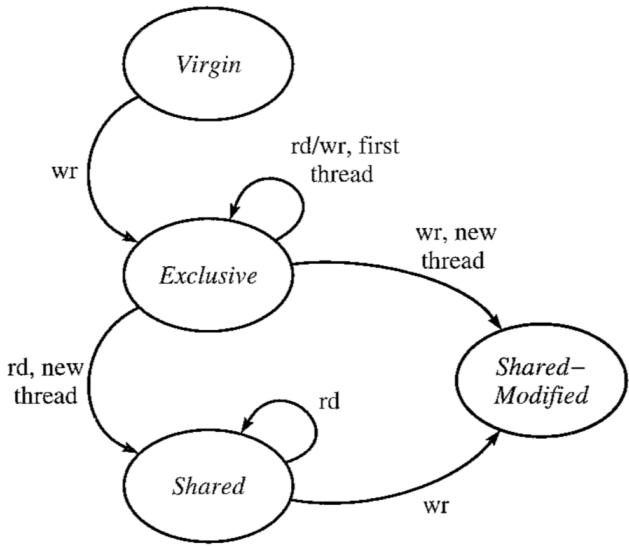
locks_held(t_1) = { l_1 }
C(a) = { l_1 }, S(a) = Exclusive
locks_held(t_1) = {}

	t_1	t_2
1	$lock(l_1)$	
2	$x = a + 1$	
3	$unlock(l_1)$	
4		$lock(l_2)$
5		$y = a + 1$
6		$unlock(l_2)$

locks_held(t_2) = { l_2 }

C(a) = {}, S(a) = Shared

locks_held(t_2) = {}



a处于Shared状态，
不构成并发缺陷

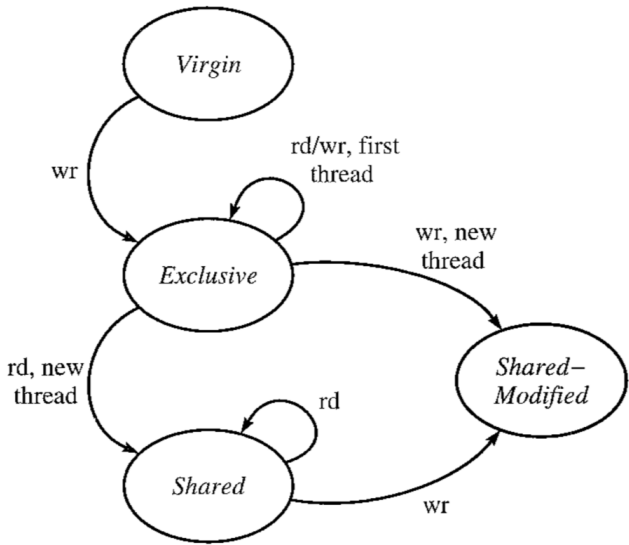


动态方法-LockSet

共享变量a已经初始化，读写锁 l_1

初始: $C(a) = \{l_1\}$, $S(a) = \text{Exclusive}$

$\text{locks_held}(t_1) = \text{locks_held}(t_2) = \text{write_locks}(t_1) = \text{write_locks}(t_2) = \{\}$



$\text{locks_held}(t_1) = \{l_1\}$

$C(a) = C(a) \cap \text{locks_held}(t_1) = \{l_1\}$

$\text{locks_held}(t_1) = \{\}$

	t_1	t_2
1	$rdlock(l_1)$	
2	$x = a + 1$	
3	$unlock(l_1)$	
4		$rdlock(l_1)$
5		$a = 1$
6		
7		$unlock(l_1)$

$\text{locks_held}(t_2) = \{l_1\}$

$C(a) = C(a) \cap \text{write_locks}(t_2) = \{\}$, $S(a) = \text{Shared-Modified}$

$\text{locks_held}(t_2) = \{\}$

RACE!



动态方法-LockSet

共享变量a已经初始化，锁 l_1 和 l_2
初始: $C(a) = \{l_1, l_2\}$, $S(a) = \text{Exclusive}$

$\text{locks_held}(t_1) = \text{locks_held}(t_2) = \text{write_locks}(t_1) = \text{write_locks}(t_2) = \{\}$

$\text{locks_held}(t_1) = \{l_1\}$

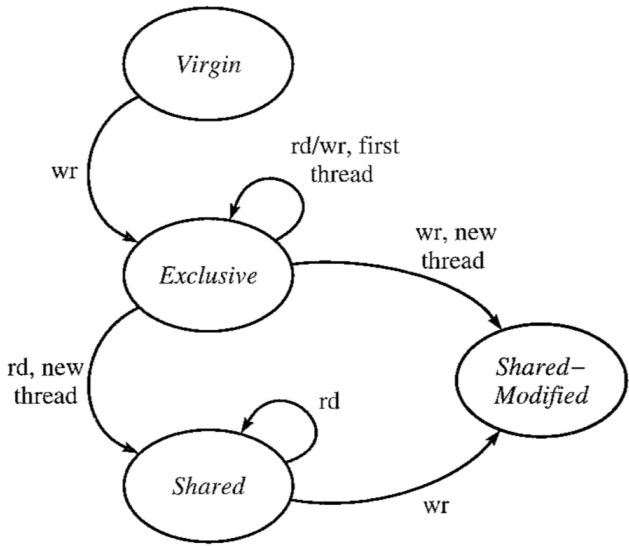
$\text{locks_held}(t_1) = \{\}$

	t_1	t_2
1	$lock(l_1)$	
2	$x = 1$	
3	$unlock(l_1)$	
4		$lock(l_2)$
5		$y = 1$
6		$unlock(l_2)$
7		$a = 1$

$\text{locks_held}(t_2) = \{l_2\}$

$\text{locks_held}(t_2) = \{\}$

$C(a) = C(a) \cap \text{locks_held}(t_2) = \{\}$, $S(a) = \text{Shared-Modified}$



误报: a只在第二个线程中进行了访问, 不构成并发缺陷



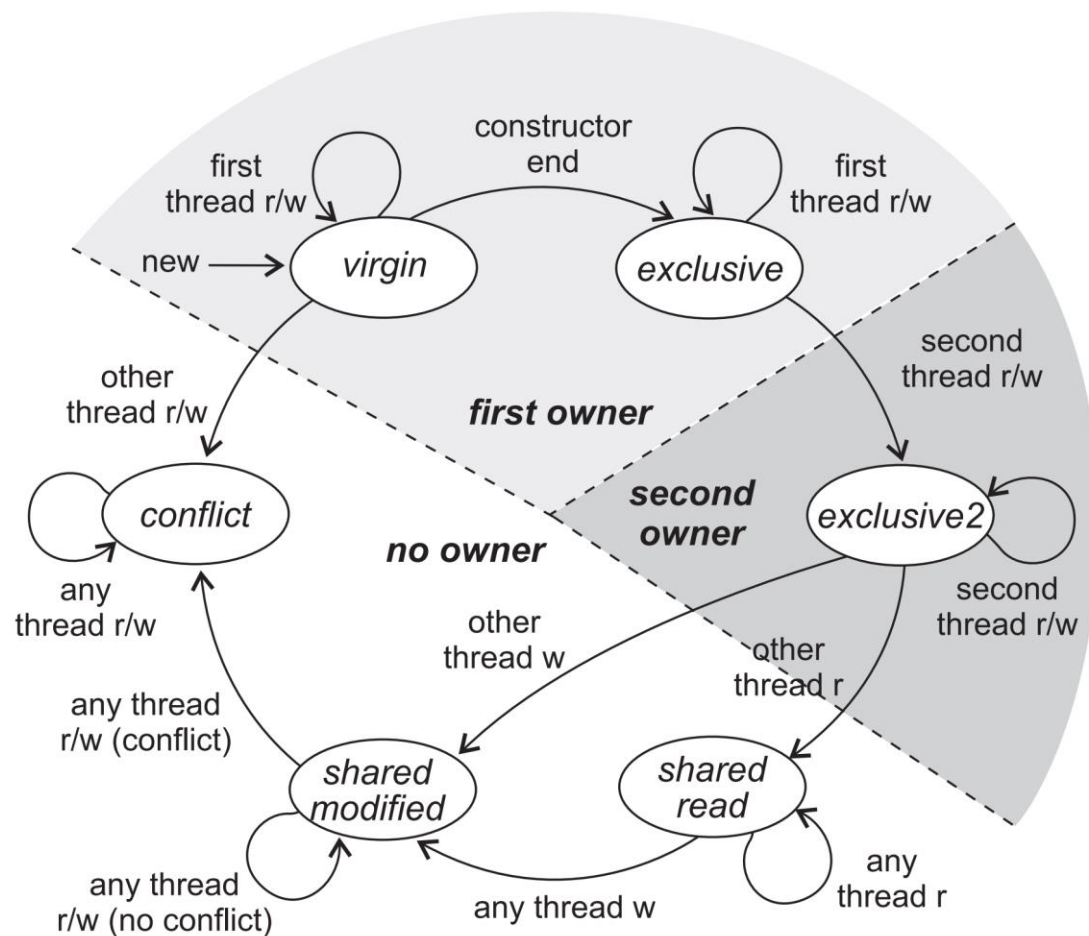
动态方法-LockSet

➤ 改进:

- 引入Ownership Model, 为每个共享变量记录一个Owner线程。

➤ 状态转移:

- **Virgin**: 考虑了在对象初始化构造过程中可能会有并发访问, 如果有的话就直接转到Conflict状态, 此时other thread指初始化线程之外的所有线程
- **Exclusive**: 与Eraser中相似
- **Virgin**和**Exclusive**都归属于first owner, 即初始化线程
- **Exclusive2**: ownership发生变迁, 在first owner之后, 第二个线程将会成为其second owner, 这一ownership transition不可逆, 即后续first owner再对变量进行访问时, 会被当做other thread



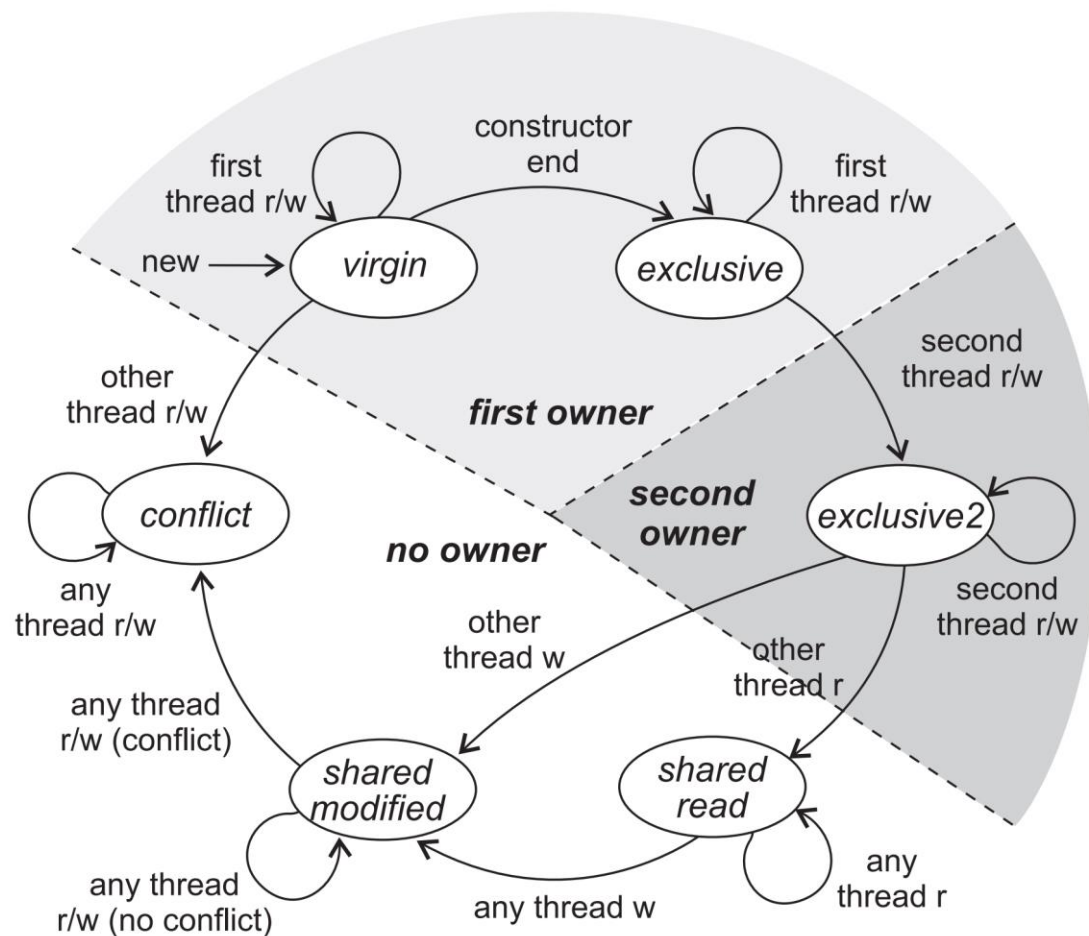
动态方法-LockSet

➤ 改进:

- 引入Ownership Model, 为每个共享变量记录一个Owner线程。

➤ 状态转移:

- Shared-Read: 与Eraser一致
- Shared-Modified: 与Eraser中相似
- Conflict: 当出现有冲突的访问时, 数据竞争发生, 变量状态转为Conflict



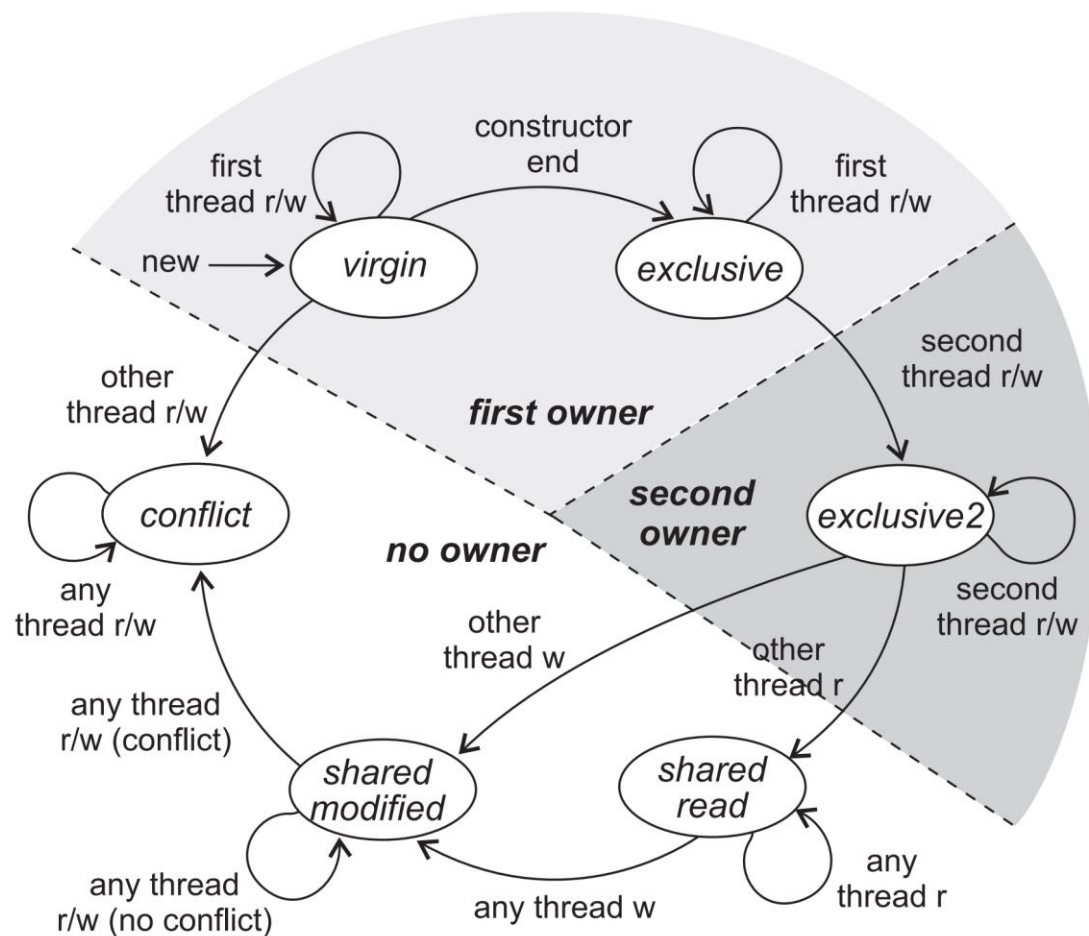
动态方法-LockSet

➤ 改进:

- 引入Ownership Model, 为每个共享变量记录一个Owner线程。

➤ Ownership转移:

- 为每个共享变量记录一个Owner线程 (threadID)
- 当一个线程访问共享变量v时, 如果threadID与当前线程id一致, 则不发生Ownership转移, 否则:
 - 如果原有owner thread为first thread且已经结束, 则当前线程成为first owner
 - 如果owner thread处于active状态, 则当前线程发送异步信号给owner thread, 表明想要获得ownership, 然后当前线程阻塞; owner thread选择一个线程转交ownership, 被转交的线程成为second owner, 阻塞线程继续执行
 - 如果owner thread阻塞 (sleep、wait), 则当前线程之间成为second owner



动态方法-LockSet

共享变量a已经初始化，锁 l_1 和 l_2

初始: $C(a) = \{l_1, l_2\}$, $S(a) = \text{Exclusive}$

$\text{locks_held}(t_1) = \text{locks_held}(t_2) = \text{write_locks}(t_1) = \text{write_locks}(t_2) = \{\}$

$\text{locks_held}(t_1) = \{l_1\}$

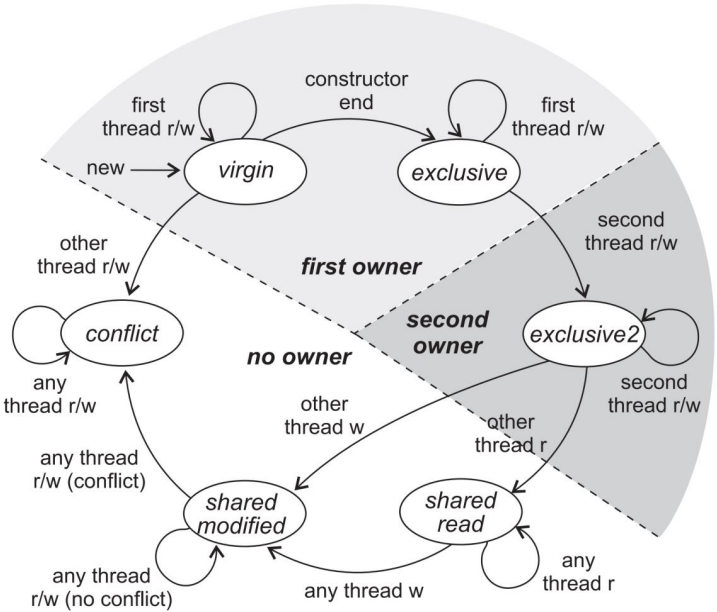
$\text{locks_held}(t_1) = \{\}$

	t_1	t_2
1	$lock(l_1)$	
2	$x = 1$	
3	$unlock(l_1)$	
4		$lock(l_2)$
5		$y = 1$
6		$unlock(l_2)$
7		$a = 1$

$\text{locks_held}(t_2) = \{l_2\}$

$\text{locks_held}(t_2) = \{\}$

$C(a) = C(a) \cap \text{locks_held}(t_2) = \{\}$, $S(a) = \text{Exclusive2}$



a处于Exclusive2状态，不构成并发缺陷



动态方法-LockSet

- 其他改进:

- 引入更多的状态, 完善状态转移过程, 包括共享变量的创建、初始化、删除等, 同时考虑barrier等操作带来的影响。

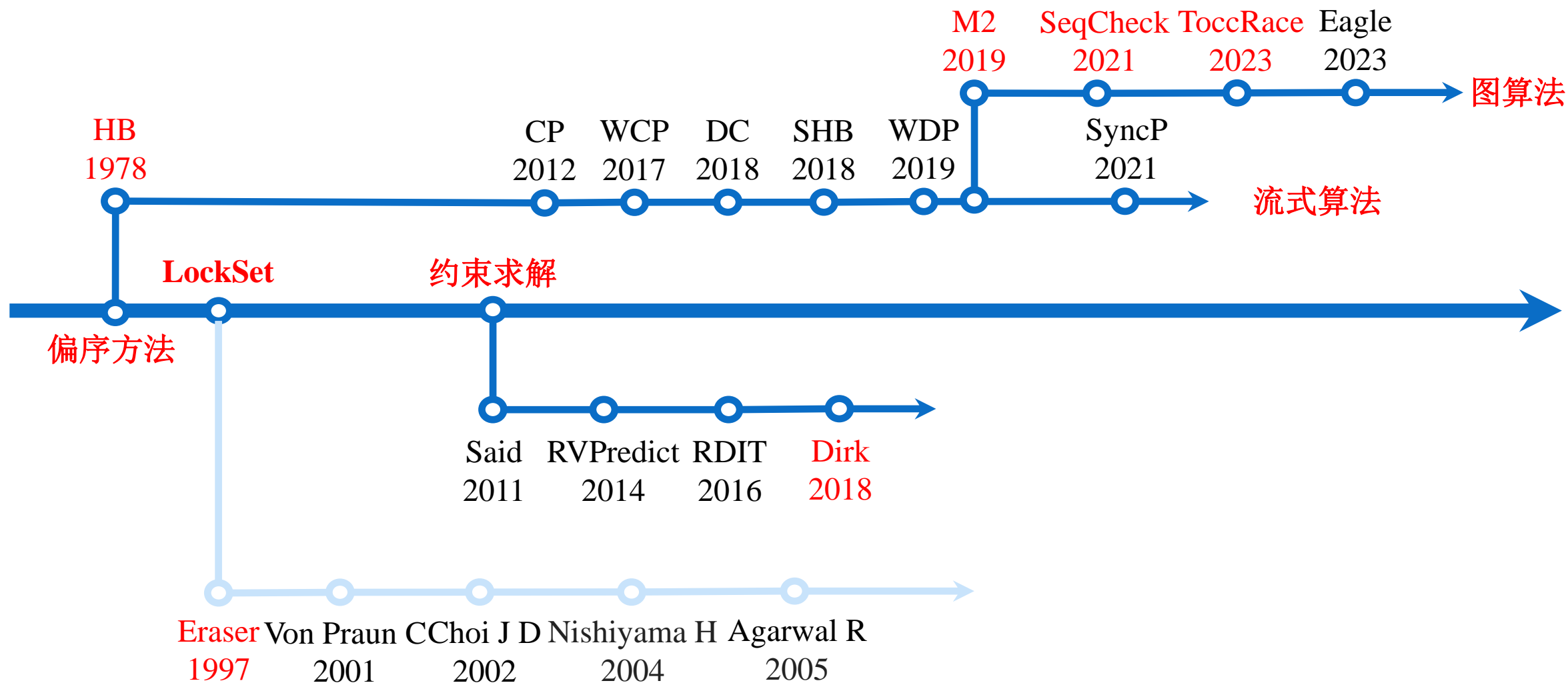
- 优势:

- 在程序运行时进行分析, 分析精度相对于静态方法有显著提高
- 实现难度低, 状态维护简单, 存在多种优化策略, 例如ownership
- 可扩展性强, 可以应用到大型应用中
- 缺陷覆盖率较高

- 不足:

- 严重依赖运行时的执行路径, 对于未执行到的代码无法开展分析
- 精度较低, 存在大量误报
- 依赖插桩, 运行时检测, 给应用程序带来了比较大的运行时开销

并发缺陷检测-动态方法



动态方法-基础概念

- 执行轨迹 (Execution Trace, 简称trace)：一个多线程程序的一次执行对应的操作的序列，其中每一个操作称为一个事件。
- 事件：一个事件被表征为 (tid, eid, type, var, val)
 - tid表示该事件发生在线程号为tid的线程中
 - eid表示该事件在线程tid中的事件的序号
 - type表示该事件的类型，可取值包括{fork, join, read, write, acquire, release, branch}等
 - var表示该事件的目标对象，可以是共享变量、锁、线程等
 - val表示依附于该操作的特定值，例如读事件读到的值
- 给定一个事件 $e = (tid, eid, type, var, val)$ ，其属性值x用e.x表示
- 此外，每个事件e有一个全局序号，用e.idx表示
- 用 t_i 表示线程号为i的线程

	t_1	t_2
e_1	$rd(a)$	
e_2	$wr(p)$	
e_3		$rd(p)$
e_4		$rd(p[0])$
e_5		br
e_6	$rd(b)$	
e_7	$wr(b[0])$	
e_8	$rd(b)$	
e_9	$wr(p)$	
e_{10}		$rd(p)$
e_{11}		$wr(p[0])$

Trace

动态方法-基础概念

事件	简写	含义
(tid, eid, fork, var, val)	fork(val)	创建线程号为val的子线程
(tid, eid, join, var, val)	join(val)	等待线程号为val的线程执行结束
(tid, eid, read, var, val)	read(var, val)	读事件读取共享变量var，获得值val
(tid, eid, write, var, val)	write(var, val)	写事件向共享变量var写入值val
(tid, eid, acq, var, val)	acq(var)	获取锁var
(tid, eid, rel, var, val)	rel(var)	释放锁var
(tid, eid, branch, var, val)	branch	发生分支事件

动态方法-基础概念

- 投影 (projection): 将trace σ 中所有属于线程 t 的事件按照顺序排列到一个新的序列中, 称新的序列为 σ 在 t 上的投影, 记作 σ_t
- 读写关系: 给定一个读事件 e_w 和一个写事件 e_r , 如果 e_r 读到的值是由 e_w 写入的, 则称 e_r 读到了 e_w 。对于给定读事件 e , 用 $obs_\sigma(e)$ 表示 e 读到的写事件, 可简写为 $obs(e)$ 。例如 $obs_\sigma(e_3) = e_2$ 。
- 锁配对关系: 给定一个acquired/release事件 e , 用 $match_\sigma(e)$ 表示和 e 配对的release/acquire事件, 可简写为 $match(e)$ 。
- 冲突事件对: 给定从属不同线程的两个内存事件 e_1 和 e_2 , 如果 $e_1.var = e_2.var$, 且两个事件中至少一个为写事件, 则称 (e_1, e_2) 为冲突事件对, 简称冲突对。数据竞争必然由冲突事件对引起。例如, (e_2, e_3) , (e_3, e_9) 。

	t_1	t_2
e_1	$rd(a)$	
e_2	$wr(p)$	
e_3		$rd(p)$
e_4		$rd(p[0])$
e_5		br
e_6	$rd(b)$	
e_7	$wr(b[0])$	
e_8	$rd(b)$	
e_9	$wr(p)$	
e_{10}		$rd(p)$
e_{11}		$wr(p[0])$

Trace

动态方法-基础概念

- 偏序：特殊的序关系，在一个集合上定义了元素之间的相对顺序。偏序关系可以描述为一种二元关系，通常用符号“ \leq ”表示，满足以下三个性质：
 - 自反性：多于所有元素 a ，都有 $a \leq a$
 - 反对称：如果 $a \leq b$ 且 $b \leq a$ ，则 $a = b$
 - 传递性：如果 $a \leq b$ 且 $b \leq c$ ，则 $a \leq c$
- 全序关系：定义了一个集合中任意两个元素之间的顺序关系，是反对称、传递且完全的（即任意两个元素之间都有序）。

动态方法-基础概念

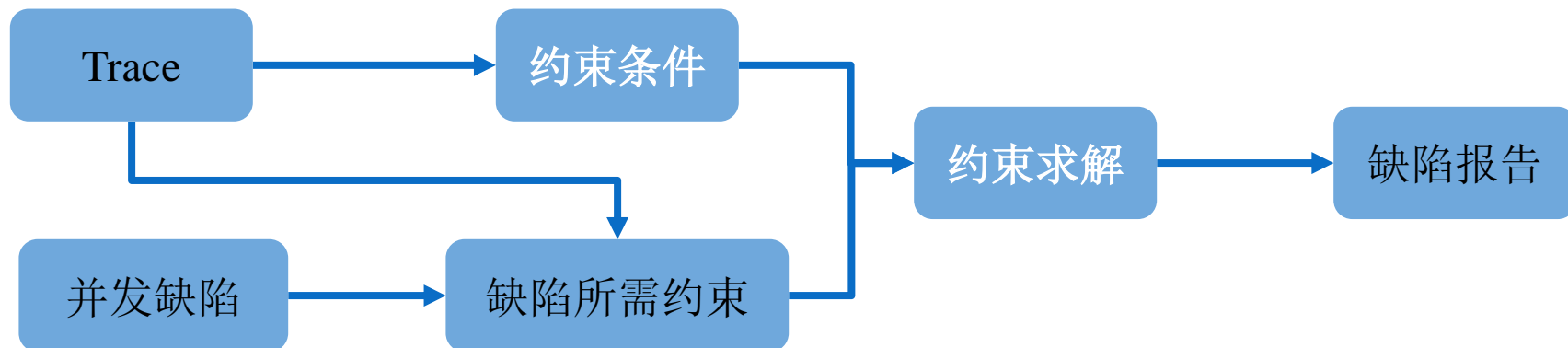
- 并发中一些基础的偏序：给定一个trace σ
 - Trace Order: 如果 e_i 在 σ 中发生在 e_j 前，则 $e_i <_{\sigma} e_j$ 。例如 $e_1 <_{\sigma} e_3$
 - Program Order (Thread Order): 如果 $e_i.tid = e_j.tid$ ，且 e_i 在 σ 中发生在 e_j 前，则 $e_i <_{PO} e_j$ 。例如 $e_1 <_{PO} e_2$ 。
 - Sync Order: 如果 $e_i = (tid1, eid1, fork, -, tid2)$ ，且 e_j 为线程 $tid2$ 中第一个事件，则 $e_i <_{SO} e_j$ ；如果 $e_i = (tid1, eid1, join, -, tid2)$ ，且 e_j 为线程 $tid2$ 中最后一个事件，则 $e_j <_{SO} e_i$ 。
- 可行trace (feasible trace): 如果一个事件序列，可以对应程序P的一次实际执行，则称该事件序列是P的一个可行trace (feasible trace)。

	t_1	t_2
e_1	$rd(a)$	
e_2	$wr(p)$	
e_3		$rd(p)$
e_4		$rd(p[0])$
e_5		br
e_6	$rd(b)$	
e_7	$wr(b[0])$	
e_8	$rd(b)$	
e_9	$wr(p)$	
e_{10}		$rd(p)$
e_{11}		$wr(p[0])$

Trace

动态方法-约束求解

- 针对潜在的数据竞争等并发缺陷，可以使用约束求解方法尝试生成实际的执行序列，通常与其他。
- 基本思想：给定程序的一个执行轨迹，对其中的部分事件的顺序进行调整，可以得到的不同的线程交错序列，在特定条件下，调整得到的线程交错序列是实际可行的。
- 算法思路：给定一个程序的执行轨迹
 - 首先分析得出其中可能的并发缺陷有哪些
 - 然后对执行轨迹中的数据流、控制流关系等进行分析，提取出限制条件，保证满足该条件的线程交错序列是可以实际发生的
 - 将以上提取出的限制条件转化为约束，然后执行约束求解



动态方法-约束求解

- 序列一致线性化 (sequentially consistent linearization, 简称SCL) : 给定程序P的一个事件序列 ρ 和一个实际执行序列 σ , 如果满足一下条件, 则称 ρ 为程序P的一个SCL。
 - 满足Trace Order、Program Order和Sync Order
 - **Write-Read Consistency**: 读事件 e 获取到的值, 必然来自于 ρ 中 e 之前的最近的对同一共享变量的写事件, 且 e 在 ρ 中读到的值与 σ 中相同
 - Synchronization Consistency: ρ 不违反同步事件 (线程创建、等待、锁、信号量等) 的语义
- 一个SCL是一个可行trace。
- Write-Read Consistency保证了:
 - ρ 的控制流与 σ 完全一致: 因为控制流只受读事件和随机数的影响, 在限定 ρ 中的随机数与 σ 保持一致的情况下, 控制流只受读事件影响
 - 保证了读写语义的正确性: 读事件获取的值必须是最新的值
- SCL实际上就是限制了序列的控制流和同步条件不变, 在上述前提下允许事件的顺序发生一定的变化

动态方法-约束求解

➤ 序列一致线性化(sequentially consistent linearization, 简称SCL) : 给定程序P的一个事件序列 π , 如果满足一下条件, 则称 π 为程序P的一个SCL。

➤ 满足Program Order和Sync Order

➤ Write-Read Consistency: 读事件 e 获取到的值, 必然来自于 π 中 e 之前的最近的对同一共享变量的写事件

➤ Synchronization Consistency: π 不违反同步事件(线程创建、等待、锁、信号量等)的语义

$$\alpha_{\pi} \equiv \left(\bigwedge_{t=1}^T \left(o_{e_1^t.idx} < \dots < o_{e_n^t.idx} \right) \wedge \bigwedge_{e \in FORK} \left(o_{e.idx} < o_{(t_{e.val}).first.idx} \right) \wedge \bigwedge_{e \in JOIN} \left(o_{(t_{e.val}).last.idx} < o_{e.idx} \right) \right)$$

条件一抽象出的约束 (first和last分别指给定线程的第一个和最后一个事件)

动态方法-约束求解

- Thread Immediate Write Predecessor (tiwp) : 给定一个属于线程t的读事件e, e的tiwp, 用e.tiwp表示, 指满足以下条件的事件:
 - e.tiwp 属于线程t的写事件
 - e.tiwp.var = e.var 即e.tiwp与e的目标变量相同
 - 不存在事件 e' , 使得 $e.tiwp \leq_{PO} e' \leq_{PO} e$, 且 $e'.var = e.var$
- Predecessor Write Set (pws) : 给定一个读事件e, e的pws是一个事件集合, 集合满足以下条件:
 - 任意写事件 e' , 当且仅当 $e'.var = e.var$, 且 (1) $e'.tid \neq e.tid$ 或者 (2) $e'.tid = e.tid \wedge e' = e.tiwp$, 则 $e' \in e.pws$
- Predecessor Write Set of Same Value (pwsv) : pws的一个子集, 在pws的基础上进一步要求 $e'.val = e.val$ 。

动态方法-约束求解

- 序列一致线性化 (sequentially consistent linearization, 简称SCL) : 给定程序P的一个事件序列 π , 如果满足一下条件, 则称 π 为程序P的一个SCL.
 - 满足Program Order和Sync Order
 - **Write-Read Consistency**: 读事件 e 获取到的值, 必然来自于 π 中 e 之前的最近的对同一共享变量的写事件
 - Synchronization Consistency: π 不违反同步事件 (线程创建、等待、锁、信号量等) 的语义

$$\beta_{\pi} \equiv \bigwedge_{e \in \pi \wedge e.type = read} \left(\left(\overset{\text{e没有对应的写事件, 值来自初始化的值}}{(e.tiwp = null) \wedge (e.val = e.var.init)} \wedge \bigwedge_{e1 \in e.pws} (o_{e.idx} < o_{e1.idx}) \right) \vee \bigvee_{e1 \in e.pws} \left((o_{e1.idx} < o_{e.idx}) \wedge \bigwedge_{e2 \in e.pws \wedge e2 \neq e1} (o_{e.idx} < o_{e2.idx} \vee o_{e2.idx} < o_{e1.idx}) \right) \right)$$

条件二抽象出的约束

动态方法-约束求解

$e_0 : (1, \text{fork}, -, 2)$
 $e_1 : (1, \text{write}, x, 1)$
 $e_2 : (1, \text{acquire}, o, -)$
 $e_3 : (1, \text{write}, x, 0)$
 $e_4 : (1, \text{wait}, o, -)$
 $e_5 : (2, \text{acquire}, o, -)$

$e_6 : (2, \text{read}, x, 0)$
 $e_7 : (2, \text{notifyAll}, o, -)$
 $e_8 : (2, \text{release}, o, -)$
 $e_9 : (2, \text{read}, x, 0)$
 $e_{10} : (1, \text{release}, o, -)$

partial order:

$\alpha_1 : o_0 < o_1 < o_2 < o_3 < o_4 < o_{10}$

$\alpha_2 : o_5 < o_6 < o_7 < o_8 < o_9$

$\alpha_3 : o_0 < o_5$

write-read consistency:

$\beta : (o_6 < o_1 \vee o_3 < o_6)$

$\wedge (o_9 < o_1 \vee o_3 < o_9)$

An execution with initial value $x = 0$.

动态方法-约束求解

- 序列一致线性化 (sequentially consistent linearization, 简称SCL) : 给定程序P的一个事件序列 π , 如果满足一下条件, 则称 π 为程序P的一个SCL.
 - 满足Program Order和Sync Order
 - Write-Read Consistency: 读事件 e 获取到的值, 必然来自于 π 中 e 之前的最近的对同一共享变量的写事件
 - Synchronization Consistency: π 不违反同步事件 (线程创建、等待、锁、信号量等) 的语义

$$\gamma_e \equiv \bigwedge_{v \in e.\text{assume}} \left(\left(v_e.av = v.iv \wedge v_e.first \wedge \bigwedge_{e_1 \in v_e.pws} o_{e.idx} < o_{e_1.idx} \right) \vee \bigvee_{e_1 \in v_e.pws} \left(\left(o_{e.idx} < o_{e_1.idx} \right) \wedge \bigwedge_{e_2 \in v_e.pws \wedge e_2 \neq e_1} \left(o_{e.idx} < o_{e_2.idx} \vee o_{e_2.idx} < o_{e_1.idx} \right) \right) \right)$$

条件三抽象出的约束

不再展开, 参考 Said, M., Wang, C., Yang, Z. and Sakallah, K., 2011, April. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods Symposium* (pp. 313-327). Berlin, Heidelberg: Springer Berlin Heidelberg.

动态方法-约束求解

➤ 为并发缺陷构建约束

- Potential Data Race Set (PDR): 给定原始执行序列 σ , PDR为所有可能的数据竞争对的集合。
- 对数据竞争, 构建以下约束 (即所有满足该约束的执行序列, 其中的事件对 (e_1, e_2) 构成数据竞争), 其中 e'_i 和 e''_i 分别是在线程 $e.tid$ 中紧挨着发生在 e 之前和之后的事件

$$\rho_{\pi} \equiv \bigvee_{(e_1, e_2) \in PDR} ((o_{e_1'.idx} < o_{e_2.idx} < o_{e_1''.idx}) \wedge (o_{e_2'.idx} < o_{e_1.idx} < o_{e_2''.idx}))$$

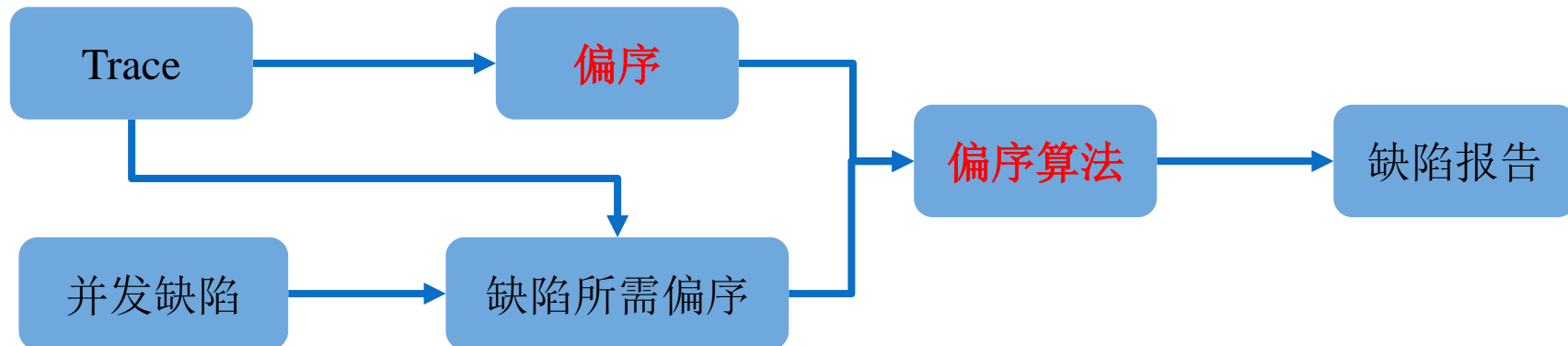
满足数据竞争的约束

动态方法-约束求解

- 约束构建完整后，就可以采用约束求解器（如SMT）对以上约束进行求解。
- 方案改进：
 - 放松约束条件：现有约束可以保证结果sound，但是约束过强，会有漏报，可以结合程序语义放宽约束限制（Dirk等）
 - 优化约束求解器：通用的约束求解器无法对并发程序模型进行高效求解，可以面向并发程序做针对性优化
- 约束求解的优势：
 - 准确度高：可以保证结果sound（即不存在误报）
 - 缺陷覆盖率较高：在已有执行序列的基础上探索周围的线程交错空间，在有一定数量执行序列的情况下，可以稳定地探索较大的线程交错空间，覆盖更多缺陷
 - 可扩展性较强：可以牺牲一定的缺陷覆盖率，拓展到大型应用程序上
- 约束求解的不足：
 - 面临状态空间爆炸问题，执行效率较低，通常限定约束求解上下文的深度来提升效率，但损失了覆盖率
 - 推测出的执行序列必须与原始执行序列有相同的控制流，限制了覆盖率

动态方法-偏序方法

- 要保证一个事件序列是一个可行trace，只需保证其中的部分事件之间的顺序关系，这种顺序关系可以用偏序表示。
- 基本思想：给定程序的一个执行轨迹，对其中的部分事件的顺序进行调整，可以得到的不同的线程交错序列，在特定条件下，调整得到的线程交错序列是实际可行的。（与约束求解相同）
- 算法思路：给定一个程序的执行轨迹
 - 首先提取偏序关系：不同的算法通常限定不同的偏序关系，偏序关系的强弱直接影响缺陷覆盖率
 - 然后采用特定的算法依据偏序关系进行计算
 - 触发并发缺陷需要满足一定的偏序条件，计算过程中判断是否违反这些偏序条件

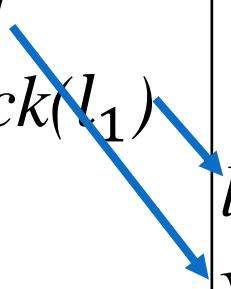


动态方法-流式算法

➤ Happen-Before关系：由图灵奖得主Leslie Lamport在1978年提出。用 \rightarrow 表示，指满足以下条件的最小的传递性关系：

- 如果事件 e_1 和 e_2 属于同一个线程，且 e_1 发生在前，则 $e_1 \rightarrow e_2$ 。
- 如果事件 e_1 是一个消息的发送者，而 e_2 是该消息的接受者，则 $e_1 \rightarrow e_2$ 。（此处的消息指广义的消息，在并发程序中，包含线程创建、等待、共享变量读写、锁等事件）

	t_1	t_2
1	$lock(l_1)$	
2	$a = 1$	
3	$unlock(l_1)$	
4		$lock(l_1)$
5		$y = a + 1$
6		$unlock(l_1)$



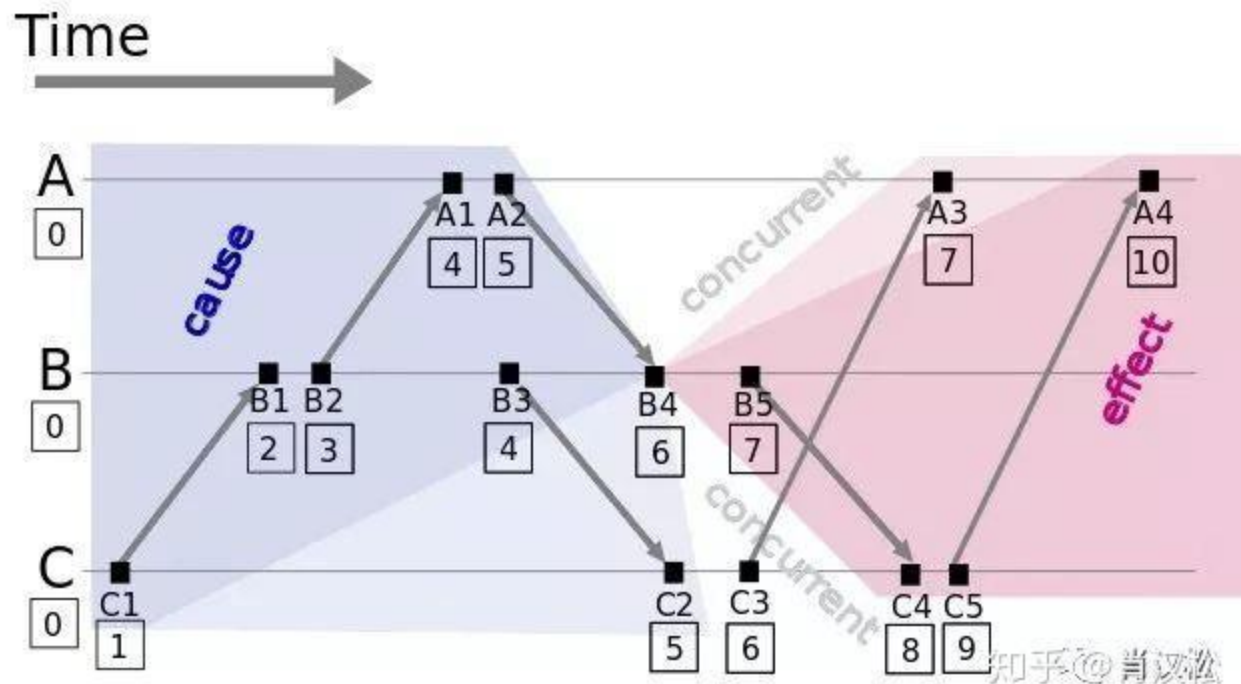
HB关系：（忽略同线程）

$13 \rightarrow 14$

$12 \rightarrow 15$

动态方法-流式算法

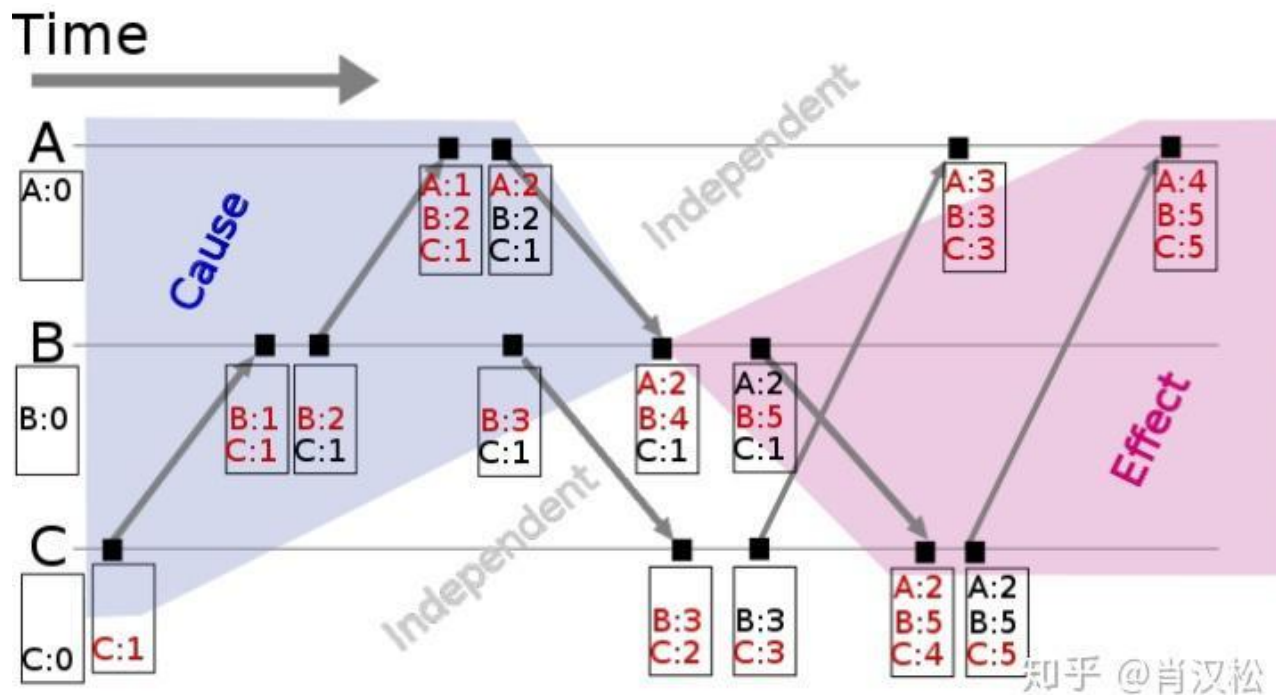
- Lamport clock: 每个线程都用一个integer来模拟自己的时间，对于本地事件或者消息的发送和接收按照以下规则来更新时间：
 - 每一次发生本地事件，该线程的时间+=1
 - 每一次发送消息，发送的线程的时间先+=1，然后再发送。发送的消息中包含它的时间
 - 每一次接收消息，接收的线程先获取消息中带有时间，再和它自己的local时间对比，取最大值后，再+=1作为自己新的local时间



动态方法-流式算法

➤ Vector Clock:

- 更新方法和Lamport clock类似
- 每个线程需要维护一个vector来记录它眼中每一个线程（包括它自己）的时间。
- 发送消息时，整个vector都要包含在消息里。
- 接收消息时，把自己的local时间+=1，然后对于vector中的其他每一个项，取本地时钟和消息中时钟的最大值。



动态方法-流式算法

- 基于HB和向量时钟进行数据竞争检测：Djit+算法
- 初始化
 - 对于每个线程 t ，给定一个 VC_t ，其中 $VC_t[i]$ 初始为1；对于同步对象（锁） l ，同样初始化一个 VC_l （ $VC_l[i]$ 全为0）
 - 对每个共享变量 v ，初始化两个VC，一个是 WVC_v ，另一个是 RVC_v ，分别表示最近对该变量进行写/读操作的线程的时间戳，全部初始化为0
- 状态转移
 - 线程 t 遇到Acquire l ： $VC_t[i] = \max(VC_t[i], VC_l[i])$
 - 线程 t 遇到Release l ： $VC_t[t] += 1$ ， $VC_l[i] = \max(VC_t[i], VC_l[i])$
 - 线程 t 遇到共享变量的 v 读写：如果是读操作， $RVC_v[t] = VC_t[t]$ ；否则， $WVC_v[t] = VC_t[t]$
- 数据竞争检测
 - 线程 t 遇到读 v ：遍历 t 以外的线程，如果线程 i 满足 $WVC_t[i] \geq VC_v[i]$ ，则汇报一处数据竞争
 - 线程 t 遇到写 v ：遍历 t 以外的线程，如果线程 i 满足 $WVC_t[i] \geq VC_v[i]$ 或者 $RVC_t[i] \geq VC_v[i]$ ，则汇报一处数据竞争

动态方法-流式算法

共享变量a, 锁 l_1 和 l_2

初始: $VC_1 = VC_2 = \langle 1, 1 \rangle$

$RVC_a = WVC_a = VC_{l_1} = VC_{l_2} = \langle 0, 0 \rangle$

$VC_1 = \langle 1, 1 \rangle$

$WVC_a = \langle 1, 0 \rangle, RVC_a = \langle 0, 0 \rangle$

$VC_1 = \langle 2, 1 \rangle$

	t_1	t_2
1	$lock(l_1)$	
2	$a = 0$	
3	$unlock(l_1)$	
4		$lock(l_2)$
5		$a = 1$
6		
7		$unlock(l_2)$

$VC_2 = \langle 1, 1 \rangle$

$WVC_a = \langle 1, 1 \rangle, RVC_a = \langle 0, 0 \rangle$

$WVC_a[1] = 1 \geq VC_2[1] = 1$

$VC_2 = \langle 1, 2 \rangle$

RACE!



数据竞争: line 2, line 5

动态方法-流式算法

➤ 方案改进:

- 放宽偏序条件: 采用比HB关系更弱的偏序关系, 例如WCP、SHB等
- 与LockSet方法结合
- 算法效率提升: FastTrack、Loft

➤ 流式算法的优势:

- 运行效率高: 时间复杂度通常是线性的
- 准确度高: 可以保证结果sound (即不存在误报)
- 可扩展性较强: 可以拓展到大型应用程序上

➤ 流式算法的不足:

- 缺陷覆盖率较低: 向量时钟算法执行过程中会固定执行前缀, 引入了大量不必要的偏序, 限制了缺陷覆盖率
- 必须在现有执行的相同的控制流下检测并发缺陷, 限制了缺陷覆盖率