Mutual Exclusion: Classical Algorithms for Locks

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Motivation

- **Problem: ensure that a data structure is maintained consistently**
  - **—avoid conflicting accesses to shared data (data races)**
    - **read/write conflicts**
    - **write/write conflicts**
- **Locks guarantee consistency by providing exclusion**
  - **—acquire lock before manipulating the shared data**
  - **—release lock when finished manipulating the shared data**

Problems with Locks

- **Conceptual**
  - —coarse-grained: poor scalability
  - —fine-grained: hard to write
- **Semantic**
  - —deadlock
  - —priority inversion
- **Performance**
  - —intolerance of page faults and preemption

Alternatives to Locks

- **Transactional memory (TM)**
  - **support arbitrary atomic actions on multi-word shared data**

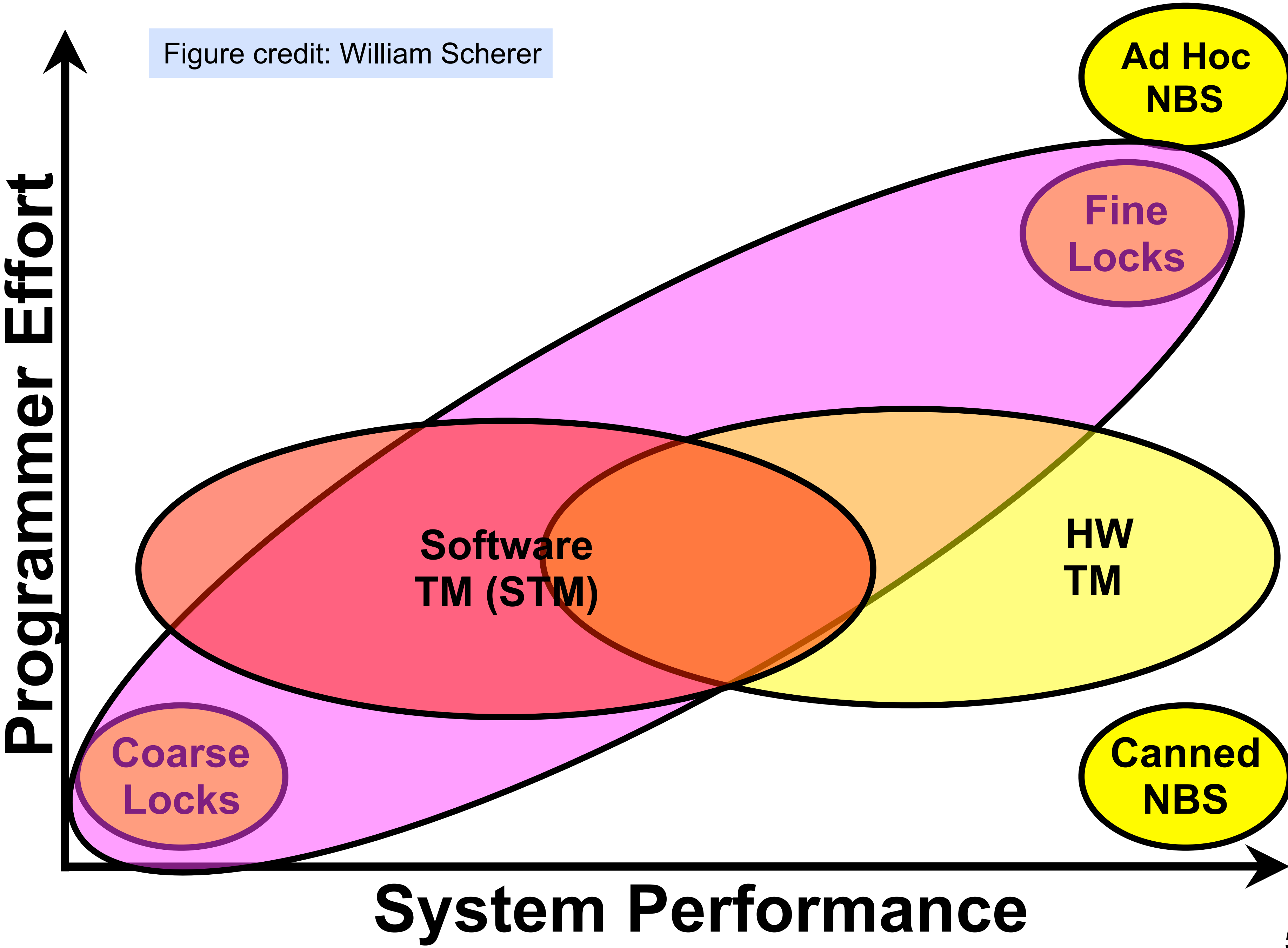
```
atomic (entries > 0) {
  node *first = head; head = head->next;
  entries--; return first;
}
```

    - **transactions that don't conflict run uninterrupted in parallel**
    - **transactions that conflict abort and retry**
      - **benefit: no need for programmer to worry about deadlock!**
      - **cost: repeated aborts can waste resources and hurt performance**

  + **easy to use, well-understood metaphor**
  – **high overhead in software; HTM on Blue Gene/Q, Intel Haswell, IBM Power8**
  ± **subject of much active research**

- **Ad hoc non-blocking synchronization (NBS)**
  + **thread failure/delay cannot prevent progress**
  + **can be faster than locks (stacks, queues)**
  – **difficult to write: every new algorithm is a publishable result**
  + **can be "canned" in libraries (e.g. java.util.concurrent's ConcurrentLinkedQueue)**

Synchronization Landscape



Properties of Good Lock Algorithms

- **Mutual exclusion (*safety* property)**
  - critical sections of different threads do not overlap
    - cannot guarantee integrity of computation without this property
- **No deadlock**
  - if some thread *attempts* to acquire the lock, then some thread *will* acquire the lock
- **No starvation**
  - every thread that attempts to acquire the lock eventually succeeds
    - implies no deadlock

Notes

- **Deadlock-free locks do not imply a deadlock-free program**
  - e.g., can create circular wait involving a pair of "good" locks
- **Starvation freedom is desirable, but not essential**
  - practical locks: many permit starvation, although it is unlikely to occur
- **Without a real-time guarantee, starvation freedom is weak property**

Topics for Today

Classical locking algorithms using load and store

- **Steps toward a two-thread solution**
  - **—two partial solutions and their properties**
- **Peterson's algorithm: a two-thread solution**
- **Tree lock for n threads**
- **Lamport's bakery lock for n threads**

Classical Lock Algorithms

- Use atomic load and store only, no stronger atomic primitives
- Not used in practice
  - —locks based on stronger atomic primitives are more efficient
- Why study classical lock algorithms?
  - —understand the principles underlying synchronization
    - – ubiquitous in parallel programs
  - —appreciate their subtlety
  - —understand the motivation for atomic operations in hardware

Toward a Classical Lock for Two Threads

- First, consider two inadequate but interesting lock algorithms
  - —use load and store only
- Assumptions
  - —only two threads
  - —each thread has a unique value of `self_threadid` $\in$ {0,1}

Lock1

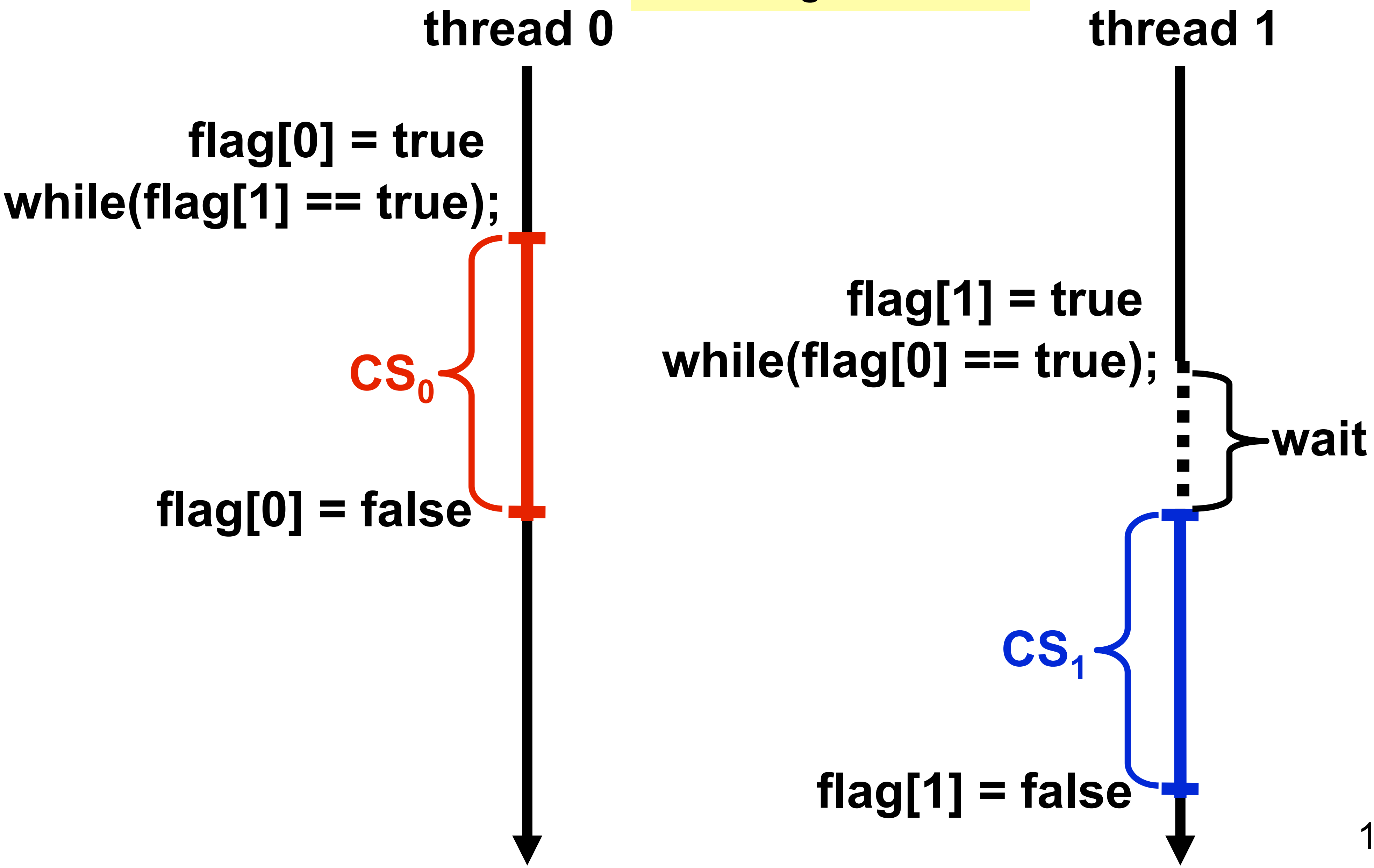
```
class Lock1: public Lock {
  private:
    volatile bool flag[2];
  public:
    void acquire() {
      int other_threadid = 1 - self_threadid;
      flag[self_threadid] = true;
      while (flag[other_threadid] == true);
    }
    void release() {
      flag[self_threadid] = false;
    }
}
```

set my flag

wait until other flag is false

Using Lock1

assume that initially both flags are false



Lock1 Provides Mutual Exclusion

Proof

- **Suppose not. Then ∃ j, k ∈ integers**

$$CS_0^j \not\rightarrow CS_1^k \quad \textbf{and} \quad CS_1^k \not\rightarrow CS_0^j$$

- **Consider each thread's acquire before its j^th^ (k^th^) critical section**

write$_0$(flag[0] = *true*) → read$_0$(flag[1] == *false*) → CS$_0$ **(1)**

write$_1$(flag[1] = *true*) → read$_1$(flag[0] == *false*) → CS$_1$ **(2)**

- **However, once flag[1] == *true*, it remains *true* while thread 1 in CS$_1$**
- **So (1) could not hold unless**

read$_0$(flag[1] == *false*) → write$_1$(flag[1] = *true*) **(3)**

- **From (1), (2), and (3)**

write$_0$(flag[0] = *true*) → read$_0$(flag[1] == *false*) → **(4)**

write$_1$(flag[1] = *true*) → read$_1$(flag[0] == *false*)

- **By (4) write$_0$(flag[0] = *true*) → read$_1$(flag[0] == *false*): a contradiction**

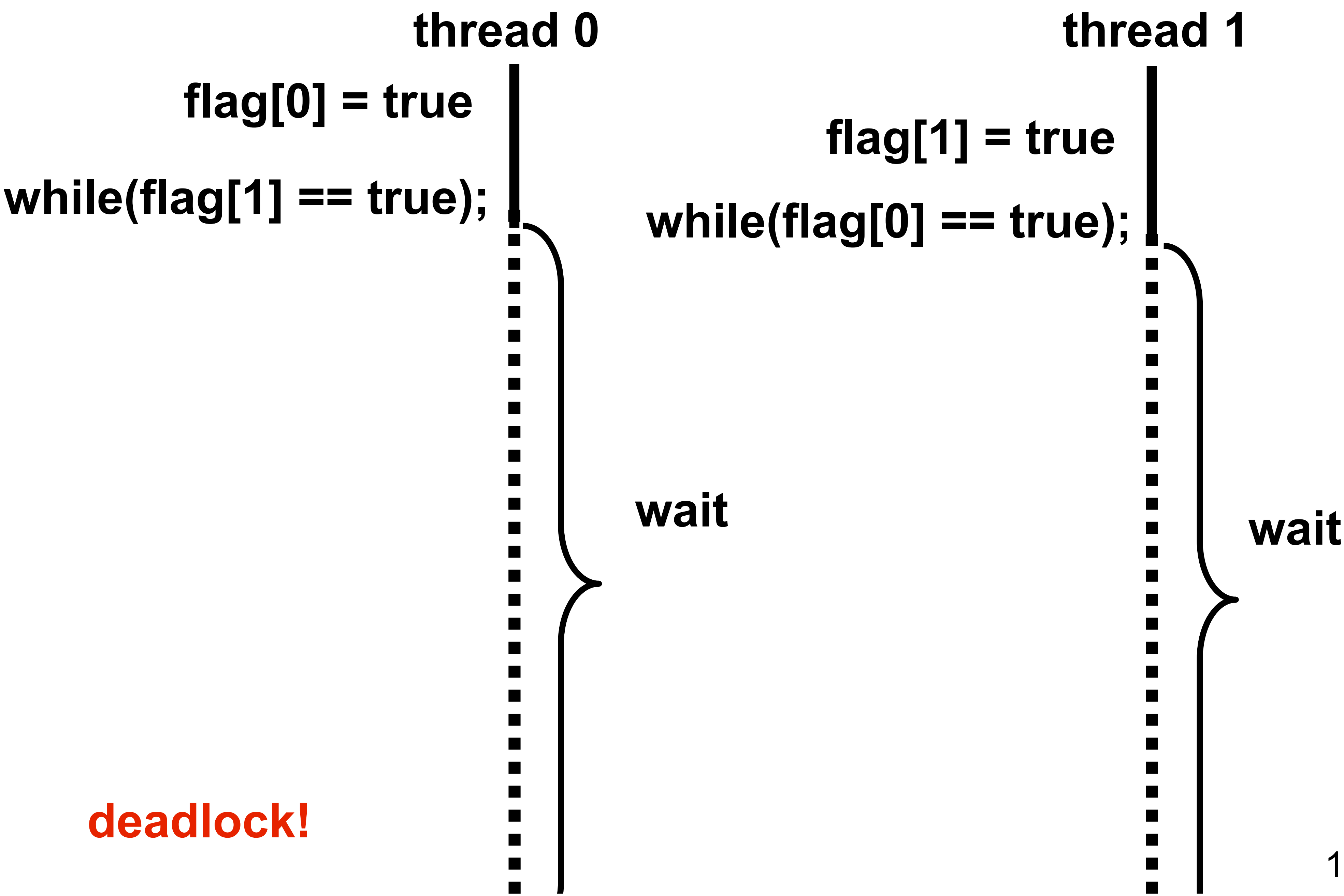
Lock1

```
class Lock1: public Lock {
  private:
    volatile bool flag[2];
  public:
    void acquire() {
      int other_threadid = 1 - self_threadid;
      flag[self_threadid] = true;
      while (flag[other_threadid] == true);
    }
    void release() {
      flag[self_threadid] = false;
    }
}
```

set my flag

wait until other flag is false

Using Lock1


thread 0
flag[0] = true
while(flag[1] == true);
wait
thread 1
flag[1] = true
while(flag[0] == true);
wait
deadlock!

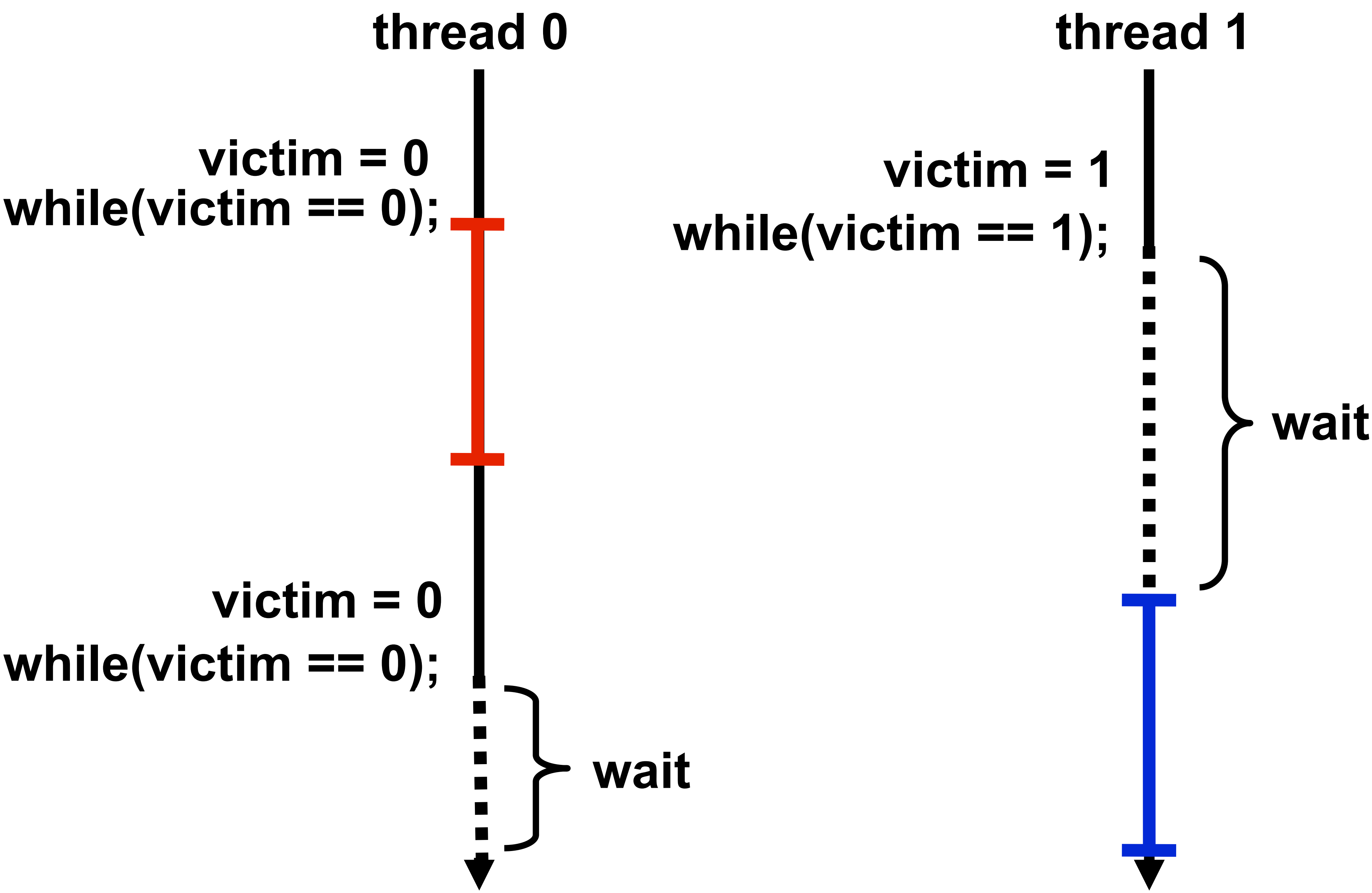
Summary of Lock1 Properties

- **Lock1 guarantees mutual exclusion**
- **Works if one thread completes its acquire before the other**
- **Deadlock if both threads write flags before either reads**
- **Since it admits deadlock, Lock1 is inadequate**

Lock2

```
class Lock2: public Lock {
  private:
    volatile int victim;
  public:
    void acquire() {
      victim = self_threadid;
      while (victim == self_threadid); // busy wait
    }
    void release() { }
}
```

Using Lock2



thread 0
thread 1
victim = 0
while(victim == 0);
victim = 1
while(victim == 1);
wait
victim = 0
while(victim == 0);
wait

Lock2 Provides Mutual Exclusion

Proof

- Suppose not. Then ∃ j, k ∈ integers

$$CS_0^j \not\to CS_1^k \quad \text{and} \quad CS_1^k \not\to CS_0^j$$

- Consider each thread's acquire before its j^{th} (k^{th}) critical section

  $write_0(victim = 0) \to read_0(victim \mathrel{!=} 0) \to CS_0$ (1)

  $write_1(victim = 1) \to read_1(victim \mathrel{!=} 1) \to CS_1$ (2)

- For thread 0 to enter the critical section, thread 1 must assign victim = 1

  $write_0(victim = 0) \to write_1(victim = 1) \to read_0(victim \mathrel{!=} 0) \to CS_0$ (3)

- Once $write_1(victim = 1)$ occurs, victim does not change
- Therefore, thread 1 cannot $read_1(victim \mathrel{!=} 1)$ and enter CS_1
- Contradiction!

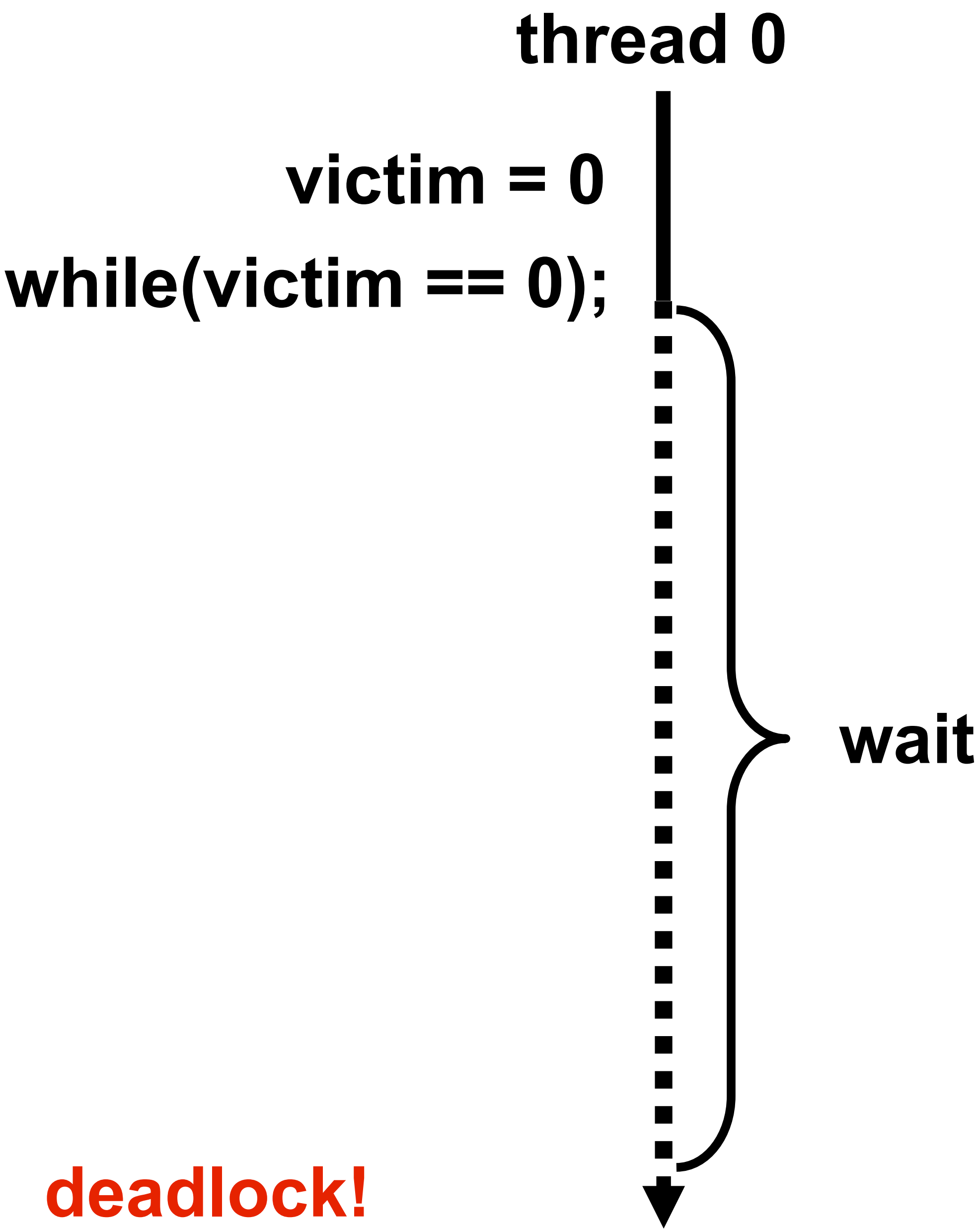
Lock2 protocol

```
void acquire() {
  victim = self_threadid;
  while (victim == self_threadid); // busy wait
}
```

Lock2

```
class Lock2: public Lock {
  private:
    volatile int victim;
  public:
    void acquire() {
      victim = self_threadid;
      while (victim == self_threadid); // busy wait
    }
    void release() { }
}
```

Using Lock2



deadlock!

Summary of Lock2 Properties

- **Guarantees mutual exclusion**
- **If two threads run concurrently: acquire succeeds for one**
- **Deadlock if one thread runs before the other**
- **Since it admits deadlock, Lock2 is inadequate**

Combining the Ideas

Lock1 and Lock2 complement each other

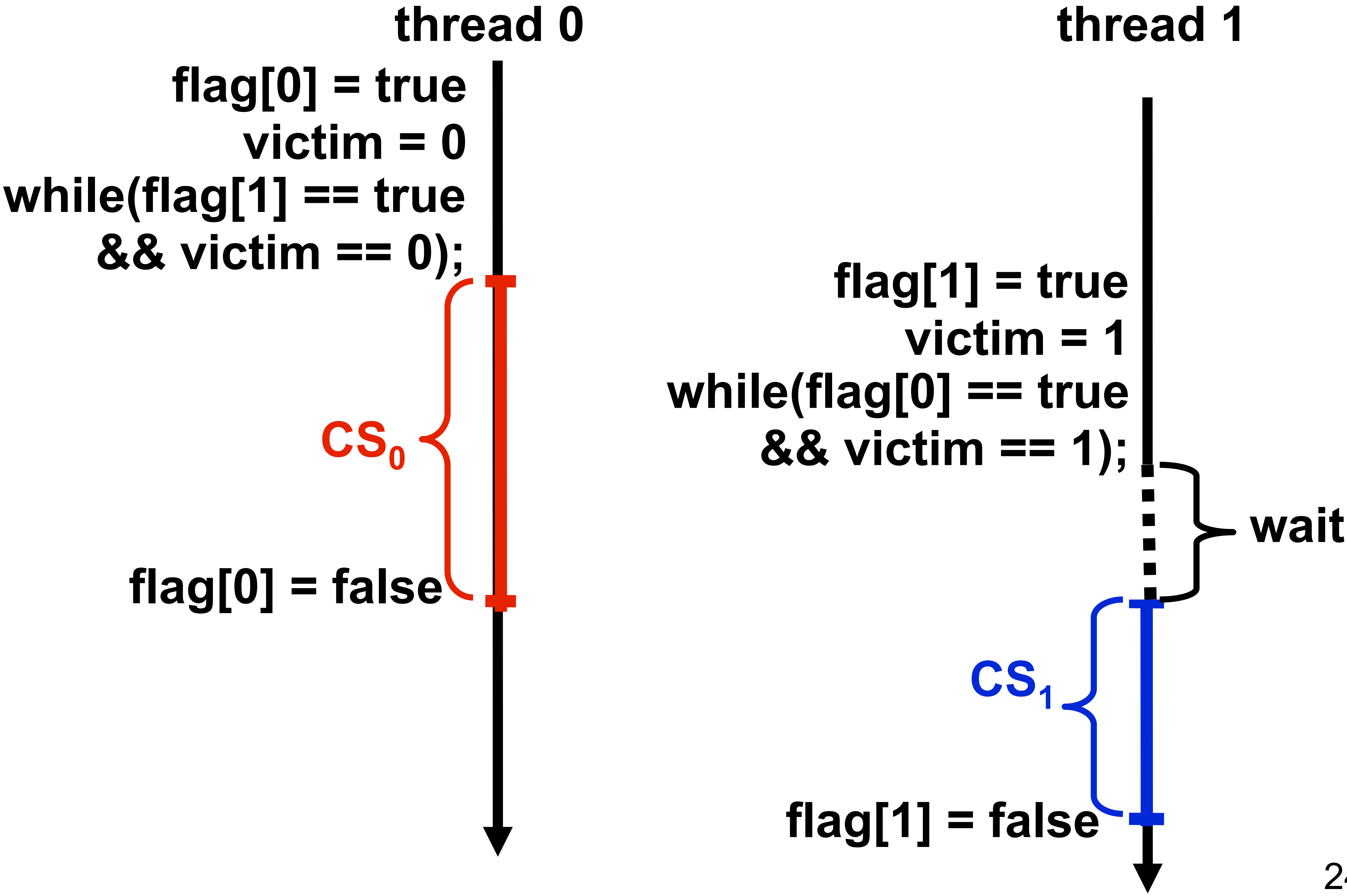
- **Each succeeds under conditions that causes the other to fail**
  - —Lock1 succeeds when CS attempts do not overlap
  - —Lock2 succeeds when CS attempts do overlap
- **Design a lock protocol that leverages the strengths of both…**

Peterson's Algorithm: 2-way Mutual Exclusion

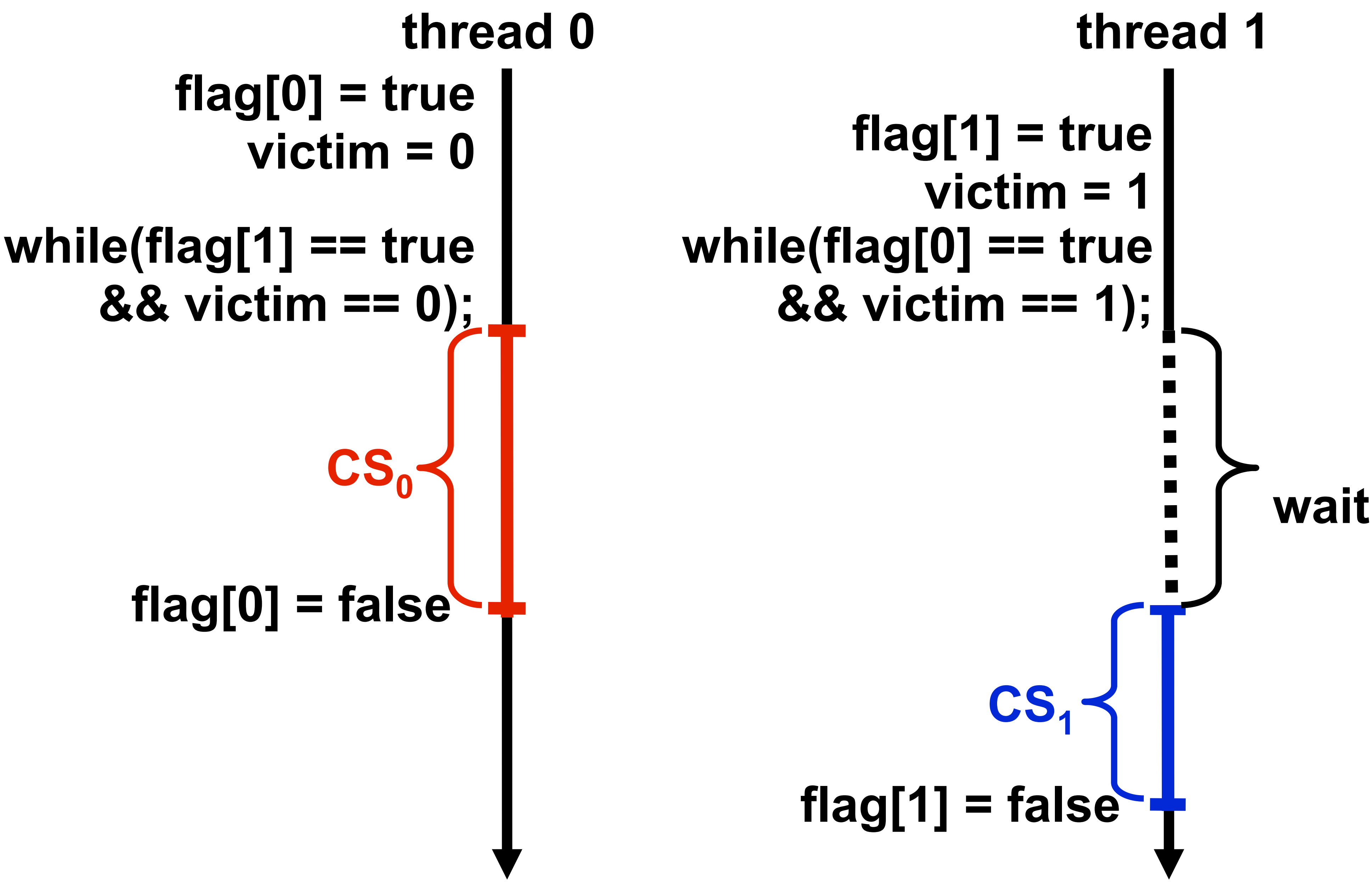
```
class Peterson: public Lock {
  private:
    volatile bool flag[2];
    volatile int victim;
  public:
    void acquire() {
      int other_threadid = 1 - self_threadid;
      flag[self_threadid] = true;    // I'm interested
      victim = self_threadid         // you go first
      while (flag[other_threadid] == true &&
             victim == self_threadid);
    }
    void release() {
      flag[self_threadid] = false;
    }
```

Gary Peterson. Myths about the Mutual Exclusion Problem.
Information Processing Letters, 12(3):115-116, 1981.

Peterson’s Lock: Serialized Acquires



Peterson's Lock: Concurrent Acquires



Peterson's Algorithm Provides Mutual Exclusion

- Suppose not. Then ∃ j, k ∈ integers

$$CS_0^j \not\rightarrow CS_1^k \quad \text{and} \quad CS_1^k \not\rightarrow CS_0^j$$

- Consider each thread's lock op before its j^{th} (k^{th}) critical section

  $write_0(flag[0] = \mathit{true}) \rightarrow write_0(victim = 0) \rightarrow$
  $read_0(flag[1] == false)$ or $read_0(victim \mathrel{!}= 0) \rightarrow CS_0$ (1)

  $write_1(flag[1] = \mathit{true}) \rightarrow write_1(victim = 1) \rightarrow$
  $read_1(flag[0] == false)$ or $read_1(victim \mathrel{!}= 1) \rightarrow CS_1$ (2)

- Without loss of generality, assume thread 0 was the last to write victim

  $write_1(victim = 1) \rightarrow write_0(victim = 0)$ (3)

- From (1) , (2), and (3), thread 0 must read victim == 0 in (1)
- Since thread 0 nevertheless enters its CS, it must have read flag[1]==false
- From (1), it must be the case that $write_0(victim = 0) \rightarrow read_0(flag[1] == \mathit{false})$
- From (1), (2), and (3) and transitivity,

  $write_1(flag[1] = \mathit{true}) \rightarrow write_1(victim = 1) \rightarrow$ (4)
  $write_0(victim = 0) \rightarrow read_0(flag[1] == false)$

- From (4), it follows that $write_1(flag[1] = \mathit{true}) \rightarrow read_0(flag[1] == \mathit{false})$
- Contradiction!

Peterson's Algorithm is Starvation-Free

- Suppose not: WLG, suppose that thread 0 waits forever in acquire
  - —it must be executing the while statement
    - waiting until flag[1] == false or victim != 0
- What is thread 1 doing while thread 0 fails to make progress?
  - —perhaps outside the critical section
    - flag[1] == true only if thread 1 is awaiting or in the critical section

      contradiction!
  - —perhaps entering and leaving the critical section
    - if so, thread 1 will set victim to 1 when it tries to re-enter the CS
    - once it is set to 1, it will not change
    - thus, thread 0 must eventually return from acquire

      contradiction!
  - —waiting in acquire as well
    - waiting for flag[0] == false or victim == 0
    - victim cannot be both 1 and 0, thus both threads cannot wait

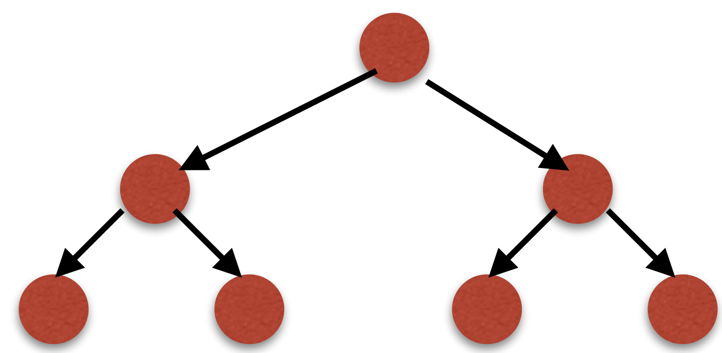
      contradiction!
- Corollary: Peterson's lock is deadlock-free as well

From 2-way to N-way Mutual Exclusion

- **Peterson’s lock provides 2-way mutual exclusion**
- **How can we generalize to N-way mutual exclusion, N > 2?**
- **Several strategies that are generalizations of Peterson’s lock**

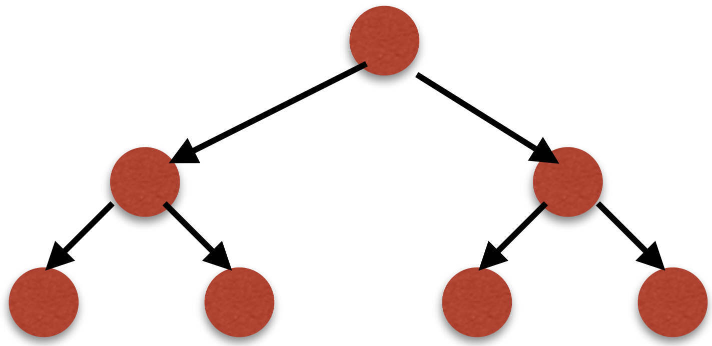
An N-way Lock as a Tree of Peterson Locks

- For a lock involving N threads, construct a balanced binary tree with N/2 leaves. Assume $N = 2^k$
- Each thread uses Peterson's lock to compete against another thread in a leaf node of the tree



- When a thread acquires a lock, it moves up the tree to compete for the parent lock
- When a thread acquires the root lock, it may enter the critical section
- When a thread exits the critical section, it releases locks along the path from the root to its leaf

Properties of Tree of Peterson Locks



- O(N) space
  - —if $N = 2^k$, there are 2^{k-1} leaves and N-1 nodes in total
- lg N steps to acquire or release the lock

Lamport’s N-way Bakery Algorithm

```
class LamportBakery: public Lock {
  private:
    volatile bool flag[N]; volatile Label label[N];
  public:
    void acquire() {
     int i = self_threadid;
     flag[i] = true;
     label[i] = max(label[0], …, label[N-1]) + 1;
     while (exists k != i such that
       flag[k] && <label[k],k> <L <label[i],i> );
    }
    void release() {
      flag[self_threadid] = 0;
    }
}
```

lexicographic ordering of <label, thread_id> tuples; thread id is used in tuple to break labeling ties

Bakery Algorithm Intuition

- **Data structure components**
  - —flag[A] = Boolean that indicate whether A wants to enter the CS
  - —label[A] = integer that indicates the thread's turn to enter the bakery
- **Protocol operation**
  - —when a thread tries to acquire the lock, it generates a new label
    - reads all other thread labels in some arbitrary order
    - generates a label greater than the largest it read
    - notes:
      - if 2 threads select labels concurrently, they may get the same
  - —algorithm uses lexicographical order on pairs of (label, thread_id)
    - (label[j], j) $<_L$ (label[k],k)
      - iff (label[j] < label[k]) || ((label[j] == label[k]) && j < k)
  - —in the waiting phase
    - a thread repeatedly rereads the labels
    - waits until
      - no thread with its flag set has a smaller (label, thread_id) pair
- **Proofs: See Herlihy and Shavit manuscript (deadlock-free, FIFO, ME)**

Spin Lock Performance: Maximal Contention

- Peterson-Buhr is a tree of Peterson's 2-party locks using load/store
- Spinlock uses test-and-set
- MCS lock uses SWAP and CAS

Figure credit: Peter A. Buhr, David Dice and Wim H. Hesselink. High-performance N-thread software solutions for mutual exclusion. Concurrency and Computation: Practice and Experience, 2014.

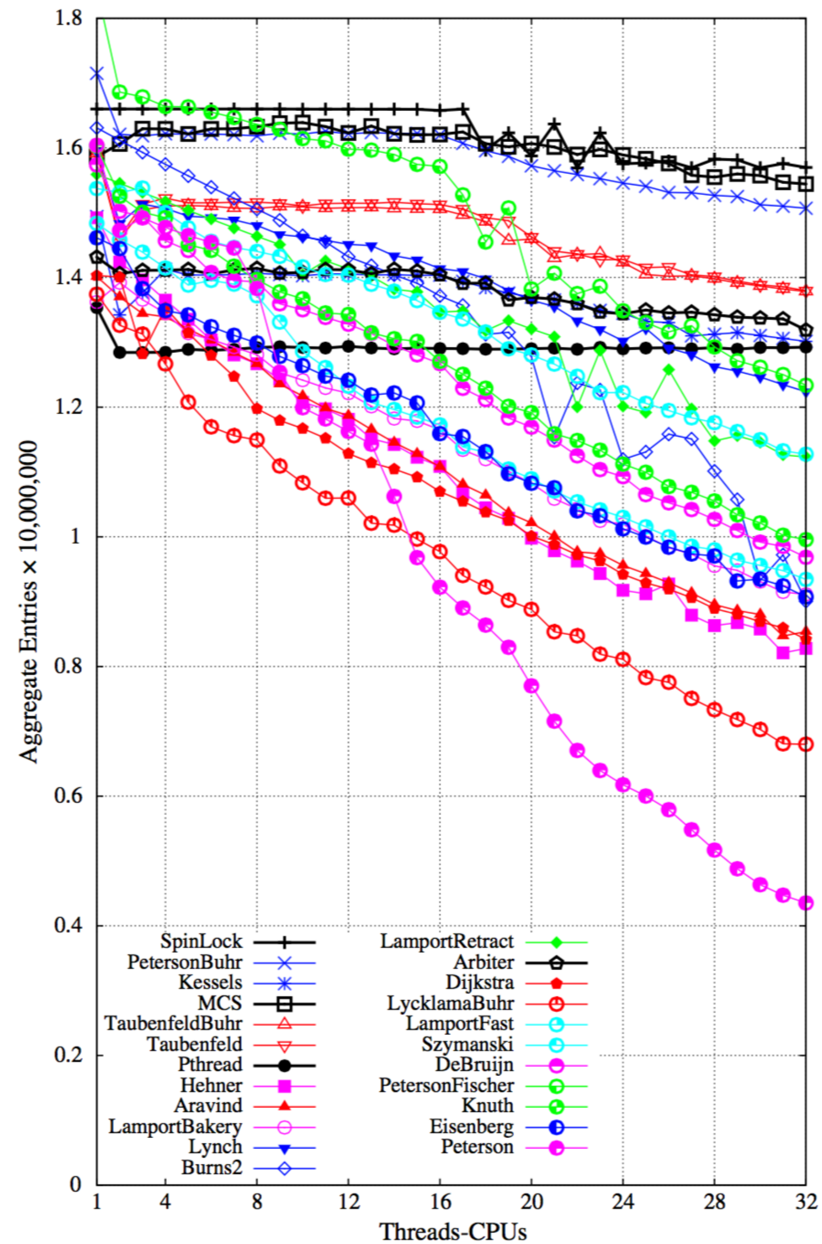


Figure 31. Critical section entry-counts, maximal contention: $N = 1..32$, SPARC, 20 s, measure of algorithm performance, where higher value is better.

Observations

- Bakery algorithm is concise, elegant and fair
- Why is it not practical?
  - —must read N distinct locations; N could be very large
  - —threads must be assigned unique ids between 0 and n-1
    - awkward for dynamic threads
  - —value of a label is monotonically increasing & unbounded
- Are locking algorithms based on load/store commonly used?
  - —no.
  - —minimum space O(N)
  - —uncontended acquisition latency is O(lg N)
- Atomic primitives enable locks with
  - —constant space
  - —constant time acquisition in the uncontended case
  - —maximum number of threads need not be known in advance

Spin Lock Performance: Minimal Contention

- Peterson-Buhr is a tree of Peterson's 2-party locks using load/store
- Spinlock uses test-and-set
- MCS lock uses SWAP and CAS

Figure credit: Peter A. Buhr, David Dice and Wim H. Hesselink. High-performance N-thread software solutions for mutual exclusion. Concurrency and Computation: Practice and Experience, 2014.

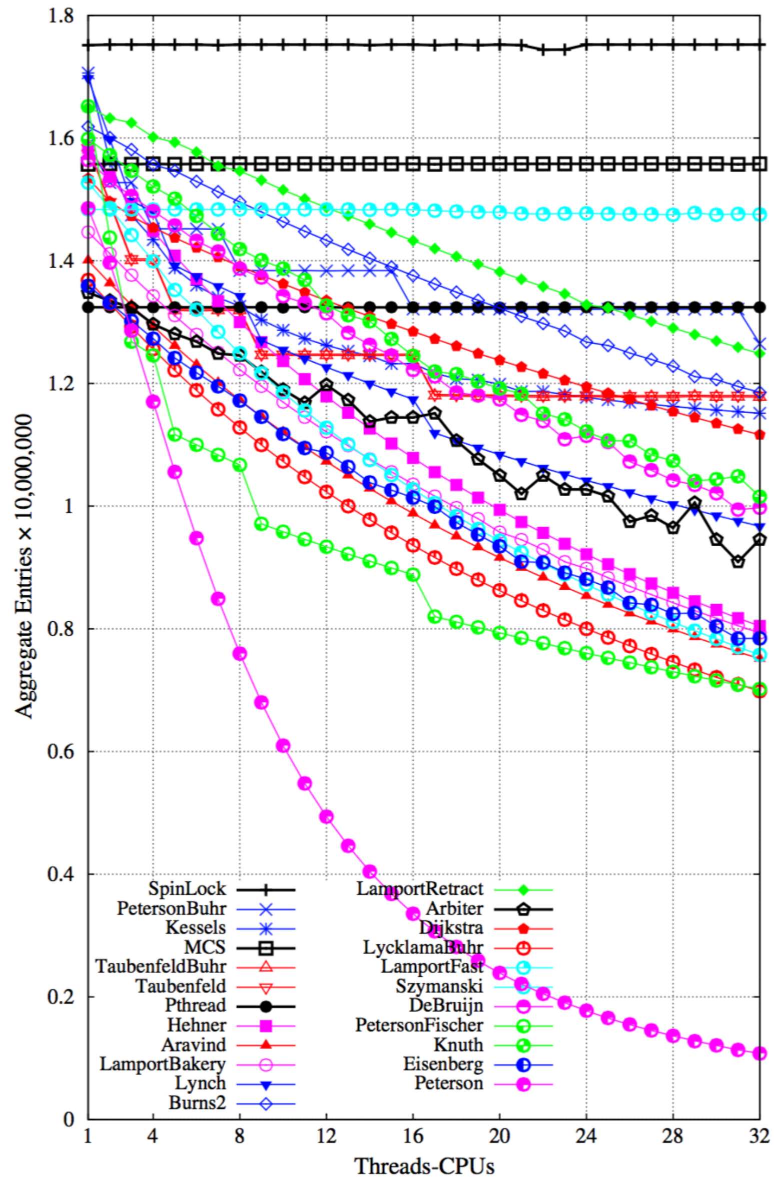


Figure 35. Critical section entry-counts, minimal contention: $N = 1..32$: SPARC, 20 s, measure of algorithm performance for zero contention, where higher value is better.

References

- **Maurice Herlihy and Nir Shavit. "Art of Multiprocessor Programming" Chapter 2 "Mutual Exclusion," Morgan Kaufmann, 2008.**
- **Gary Peterson. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, 12(3), 115-116, 1981. http://cs.nyu.edu/~lerner/spring12/Read03-MutualExclusion.pdf**