Mutual Exclusion: Classical Algorithms for Locks

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@cs.rice.edu

Motivation

Ensure that a block of code manipulating a data structure is executed by only one thread at a time

- **Why? avoid conflicting accesses to shared data (data races)**
  - —read/write conflicts
  - —write/write conflicts
- **Approach: critical section**
- **Mechanism: lock**
  - —methods
    - – acquire
    - – release
- **Usage**
  - —acquire lock to enter the critical section
  - —release lock to leave the critical section

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**
- **Filter lock: an n-thread solution**
- **Lamport’s bakery lock**

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 algorithms?
  - —understand the principles underlying synchronization
    - – subtle
    - – such issues are ubiquitous in parallel programs

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}

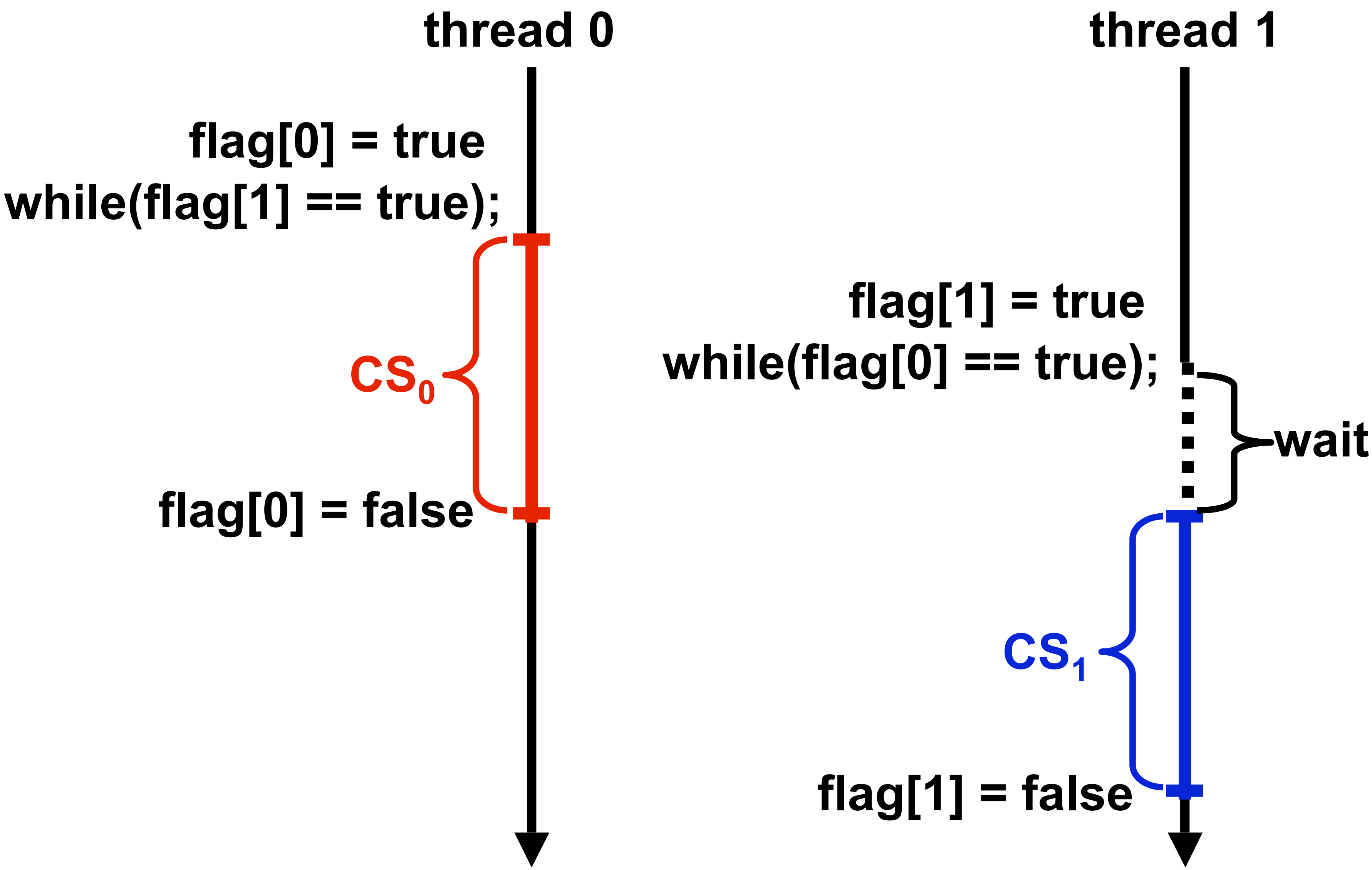
Lock1

```
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



thread 0
thread 1
flag[0] = true
while(flag[1] == true);
CS_0
flag[0] = false
flag[1] = true
while(flag[0] == true);
wait
CS_1
flag[1] = false

Lock1 Provides Mutual Exclusion

Proof

- 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 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)

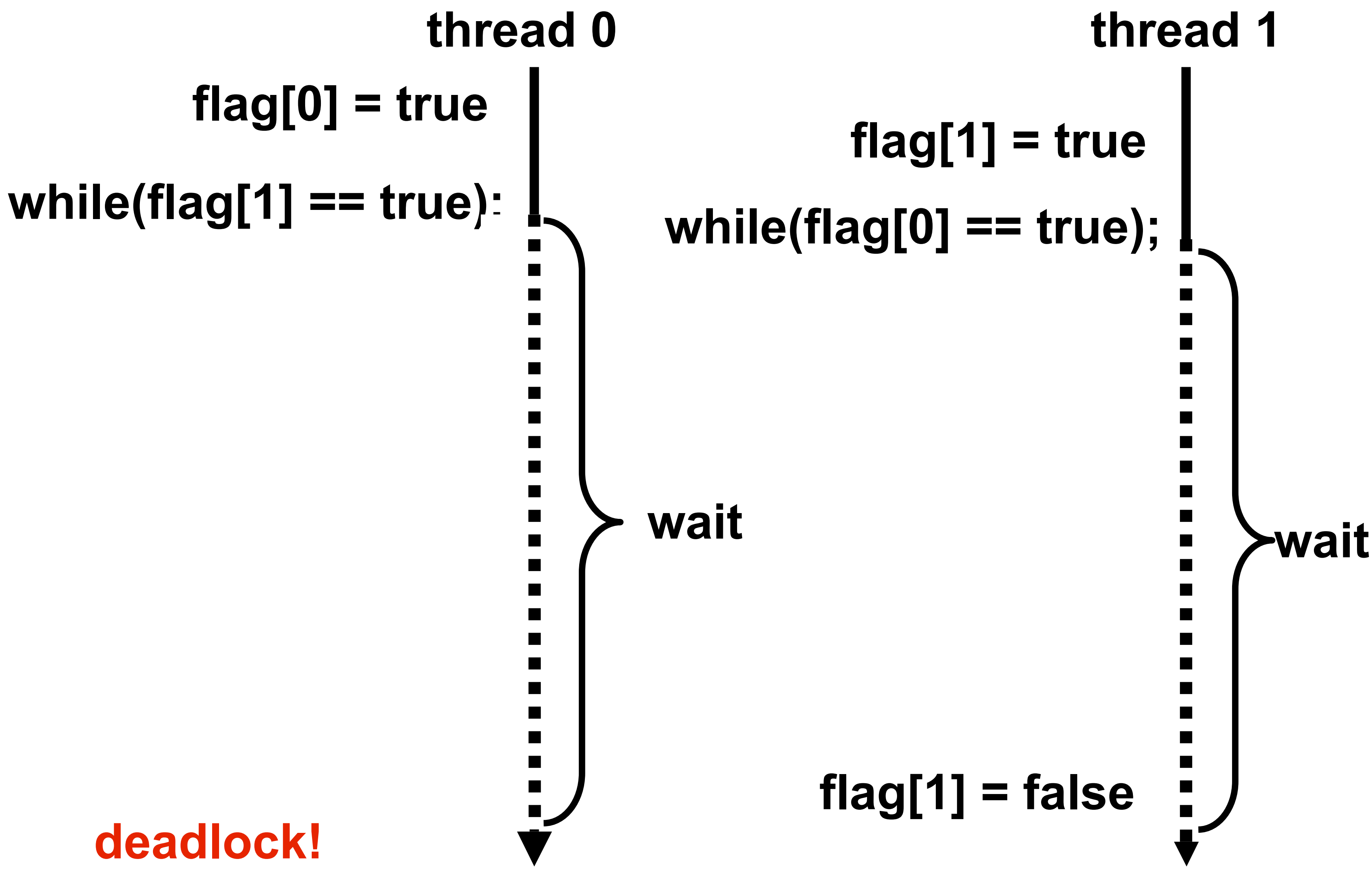
- From (1), (2), and (3)

  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

Using Lock1


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

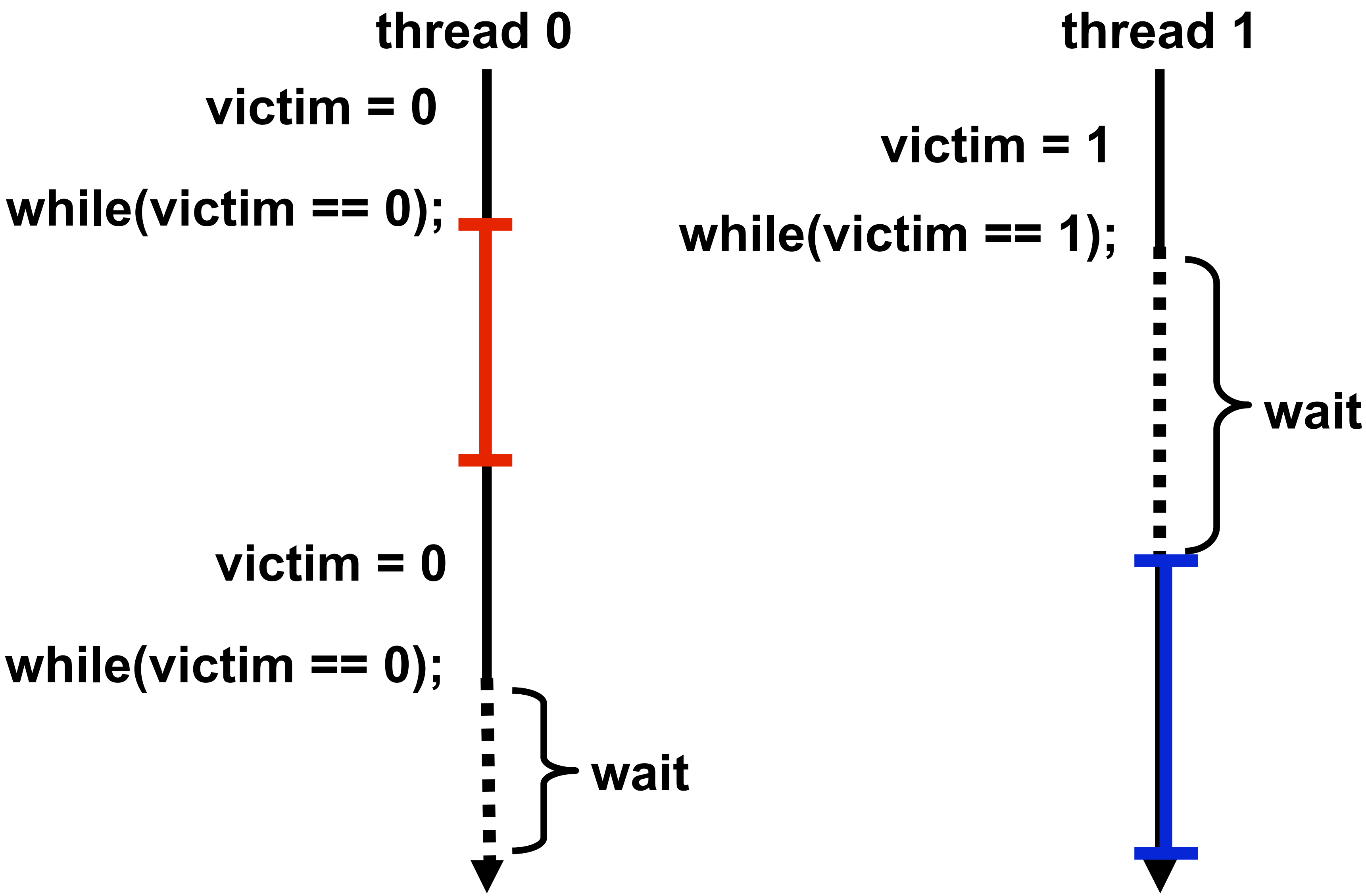
Summary of Lock1 Properties

- If one thread executes acquire before the other, works fine
  - —Lock1 provides mutual exclusion
- However, Lock1 is inadequate
  - —if both threads write flags before either reads → deadlock

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

Lock2 Provides Mutual Exclusion

Proof

- 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 acquire before its j^th^ (k^th^) critical section

  $write_0(victim = 0) \rightarrow read_0(victim == 1) \rightarrow CS_0$ (1)

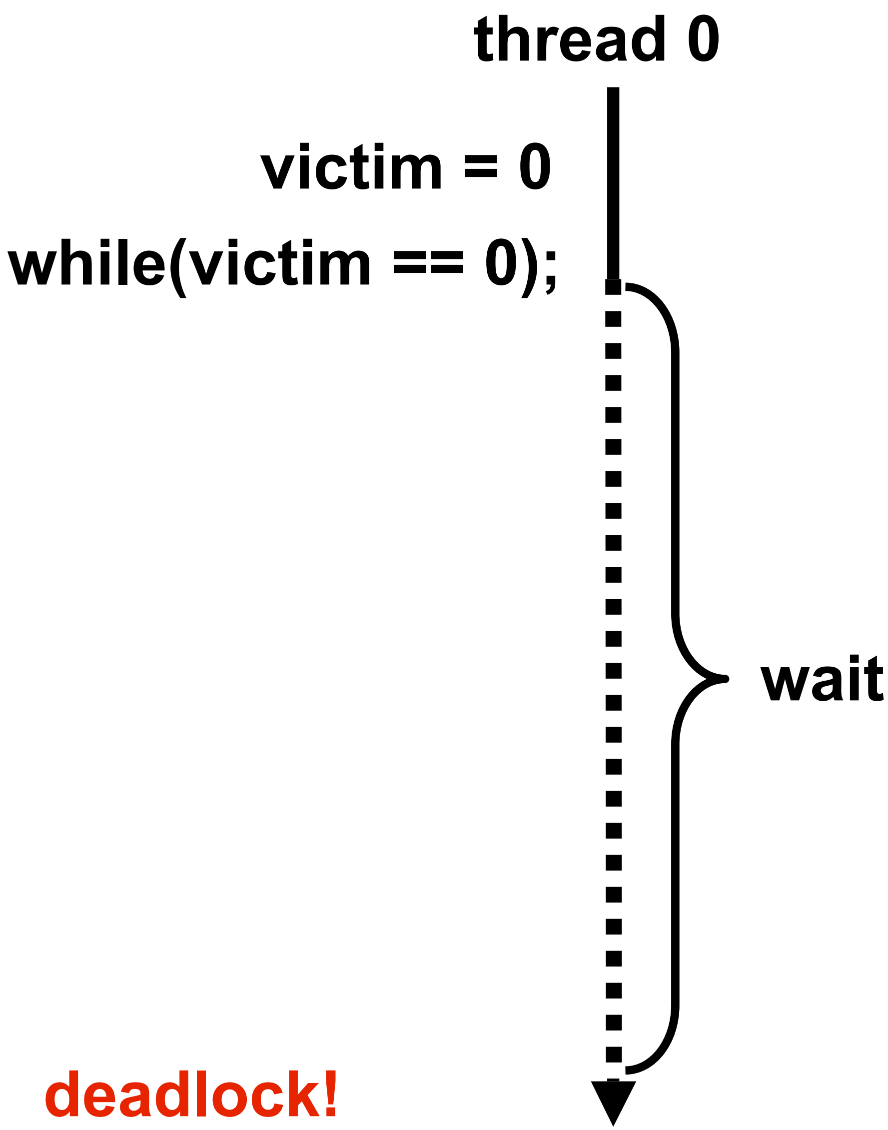
  $write_1(victim = 1) \rightarrow read_1(victim == 0) \rightarrow CS_1$ (2)

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

  $write_0(victim = 0) \rightarrow write_1(victim = 1) \rightarrow read_0(victim == 1)$ (3)

- Once $write_1(victim = 1)$ occurs, victim does not change
- Therefore, thread 1 cannot $read_1(victim == 0)$ and enter its CS
- Contradiction!

Using Lock2



Summary of Lock2 Properties

- If the two threads run concurrently, acquire succeeds for one
  - —provides mutual exclusion
- However, Lock2 is inadequate
  - —if one thread runs before the other, it will deadlock

Combining the Ideas

Lock1 and Lock2 complement each other

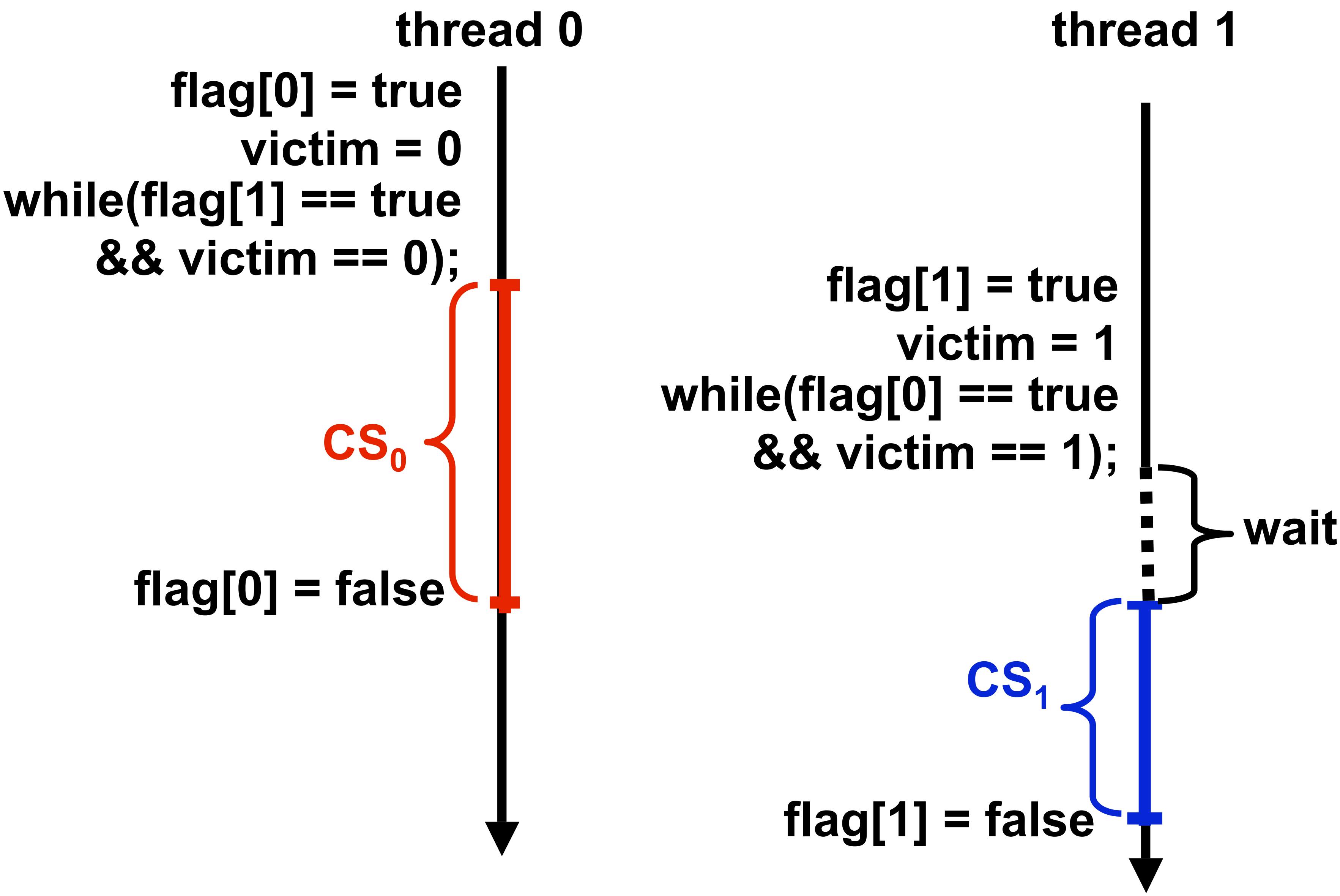
- **Each succeeds under conditions that causes the other to fail**
  - **Lock1 succeeds when CS attempts do not overlap**
  - **Lock2 succeeds when CS attempts do overlap**
- **Design a lock protocol that leverages the strengths of both…**

Peterson's Algorithm: 2-way Mutual Exclusion

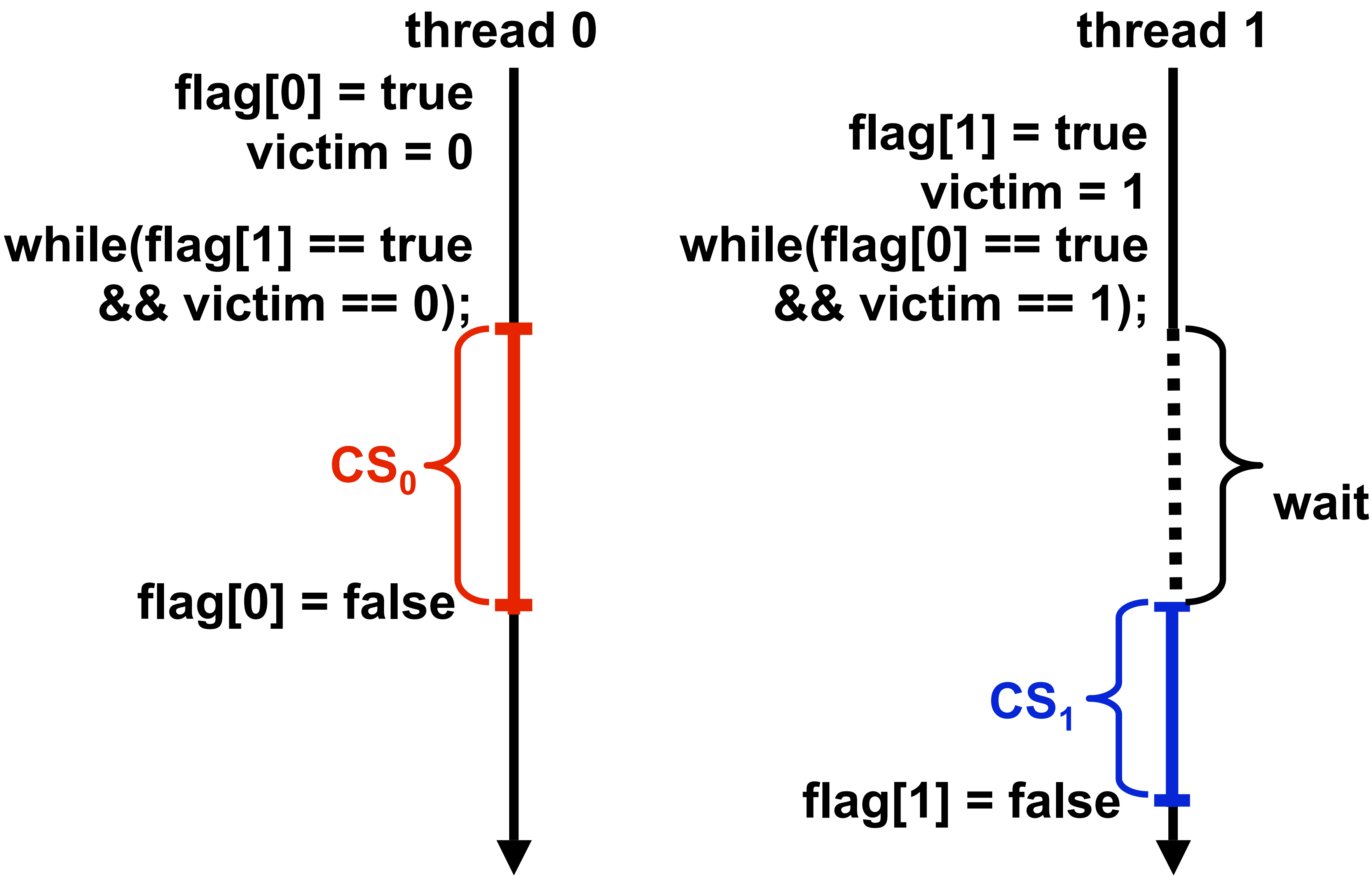
```
class Peterson: public Lock {
  private:
    volatile bool flag[2];
    volatile int victim;
  public:
    void acquire() {
      int other_threadid = 1 - self_threadid;
      flag[self_threadid] = true;    // I'm interested
      victim = self_threadid         // you go first
      while (flag[other_threadid] == 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] = *true*) → write_0(victim = 0) →
  read_0(flag[1] == *false*) → read_0(victim == 1) → CS_0 (1)

  write_1(flag[1] = *true*) → write_1(victim = 1) →
  read_1(flag[0] == *false*) → read_1(victim == 0) → CS_1 (2)

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

  write_1(victim = 1) → write_0(victim = 0) (3)

- Equation (3) implies that thread 0 reads 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) → read_0(flag[1] == *false*)

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

  write_1(flag[1] = *true*) → write_1(victim = 1) → (4)
  write_0(victim = 0) → read_0(flag[1] == *false*)

- From (4), it follows that write_1(flag[1] = *true*) → read_0(flag[1] == *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 == 1
- **What is thread 1 doing while thread 0 fails to make progress?**
  - —perhaps entering or 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?
- Filter lock: direct generalization of Peterson’s lock

Filter Lock

```
class Filter: public Lock {
  private:
    volatile int level[N]; volatile int victim[N-1];
  public:
    void acquire() {
      for (int j = 1; j < N; j++) {
        level [self_threadid] = j;
        victim [j] = self_threadid;
        // wait while conflicts exist
        while (sameOrHigher(self_threadid,j) &&
               victim[j] == self_threadid);
      }
    }
    bool sameOrHigher(int i, int j) {
      for(int k = 0; k < N; k++)
        if (k != i && level[k] >= j) return true;
      return false;
    }
    void release() {
      level[self_threadid] = 0;
    }
}
```

Understanding the Filter Lock

- Peterson's lock used two-element Boolean `flag` array
- Filter lock generalization: an N-element integer `level` array
  - —value of level[k] = highest level thread k is interested in entering
  - —each thread must pass through N-1 levels of exclusion
- Each level has it's own `victim` flag to filter out 1 thread, excluding it from the next level
  - —natural generalization of victim variable in Peterson's algorithm
- Properties of levels
  - —at least one thread trying to enter level k succeeds
  - —if more than one thread is trying to enter level k, then at least one is blocked
- For proofs, see Herlihy and Shavit's manuscript

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)<<(label[i],i));
    }
    void release() {
      flag[self_threadid] = 0;
    }
}
```

Bakery Algorithm Intuition

- **Data structure components**
  - —flag[A] = Boolean indicating 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) << (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)**

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**
- **Is there a more clever lock using only atomic load/store that avoids these problems?**
  - **—No. Any deadlock-free algorithm requires reading or writing at least N distinct locations in the worst case.**
  - **—See Herlihy and Shavit manuscript for the proof.**

References

- Maurice Herlihy and Nir Shavit. "Multiprocessor Synchronization and Concurrent Data Structures." Chapter 3 "Mutual Exclusion." Draft manuscript, 2005.
- Gary Peterson. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, 12(3), 115-116, 1981.