Mutual exclusion

Processes contend to enter their *critical section*.

A process (allowed to be) in its critical section is called *privileged*.

For each computation we require:

Mutual exclusion: Always at most one process is privileged.

Starvation-freeness: If a process p tries to enter its critical section, and no process stays privileged forever, then p eventually becomes privileged.

Applications: Distributed shared memory, replicated data, atomic commit.

Mutual exclusion

Processes contend to enter their *critical section*.

A process (allowed to be) in its critical section is called *privileged*.

For each computation we require:

Mutual exclusion: Always at most one process is privileged.

Starvation-freeness: If a process p tries to enter its critical section, and no process stays privileged forever, then p eventually becomes privileged.

Applications: Distributed shared memory, replicated data, atomic commit.

Mutual exclusion

Processes contend to enter their *critical section*.

A process (allowed to be) in its critical section is called *privileged*.

For each computation we require:

Mutual exclusion: Always at most one process is privileged.

Starvation-freeness: If a process p tries to enter its critical section, and no process stays privileged forever, then p eventually becomes privileged.

Applications: Distributed shared memory, replicated data, atomic commit.

Question

Why is the mutual exclusion algorithm below for two threads flawed ?

A multi-reader/multi-writer register initially has the value -1.

If thread 0 (or 1) wants to enter its critical section,
it spins on the register until it is -1.

Then thread 0 (or 1) writes the value 0 (or 1) into the register.

Thread 0 (or 1) checks whether the value of the register is 0 (or 1).

If not, it returns to spinning on the register until it is -1.

If so, it enters its critical section.

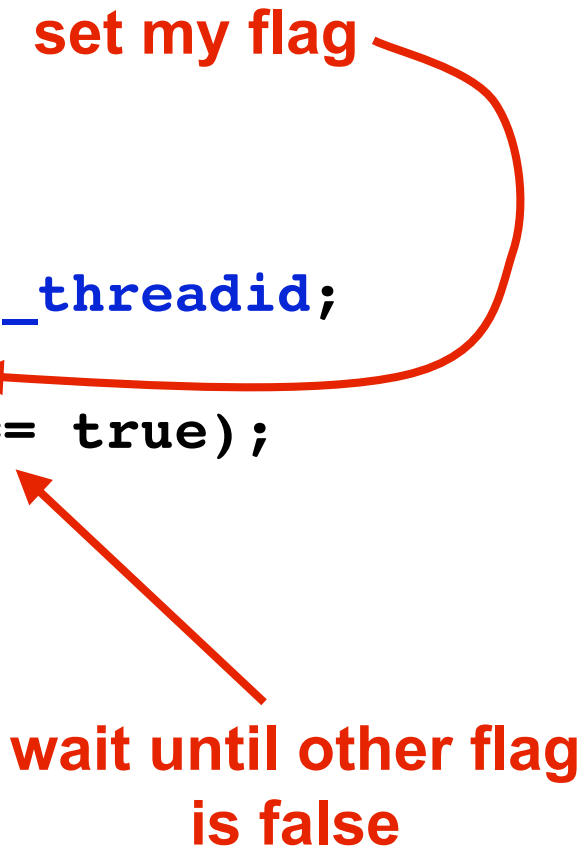
When a thread exits its critical section, it writes -1 into the register.

Toward a Classical Lock for Two Threads

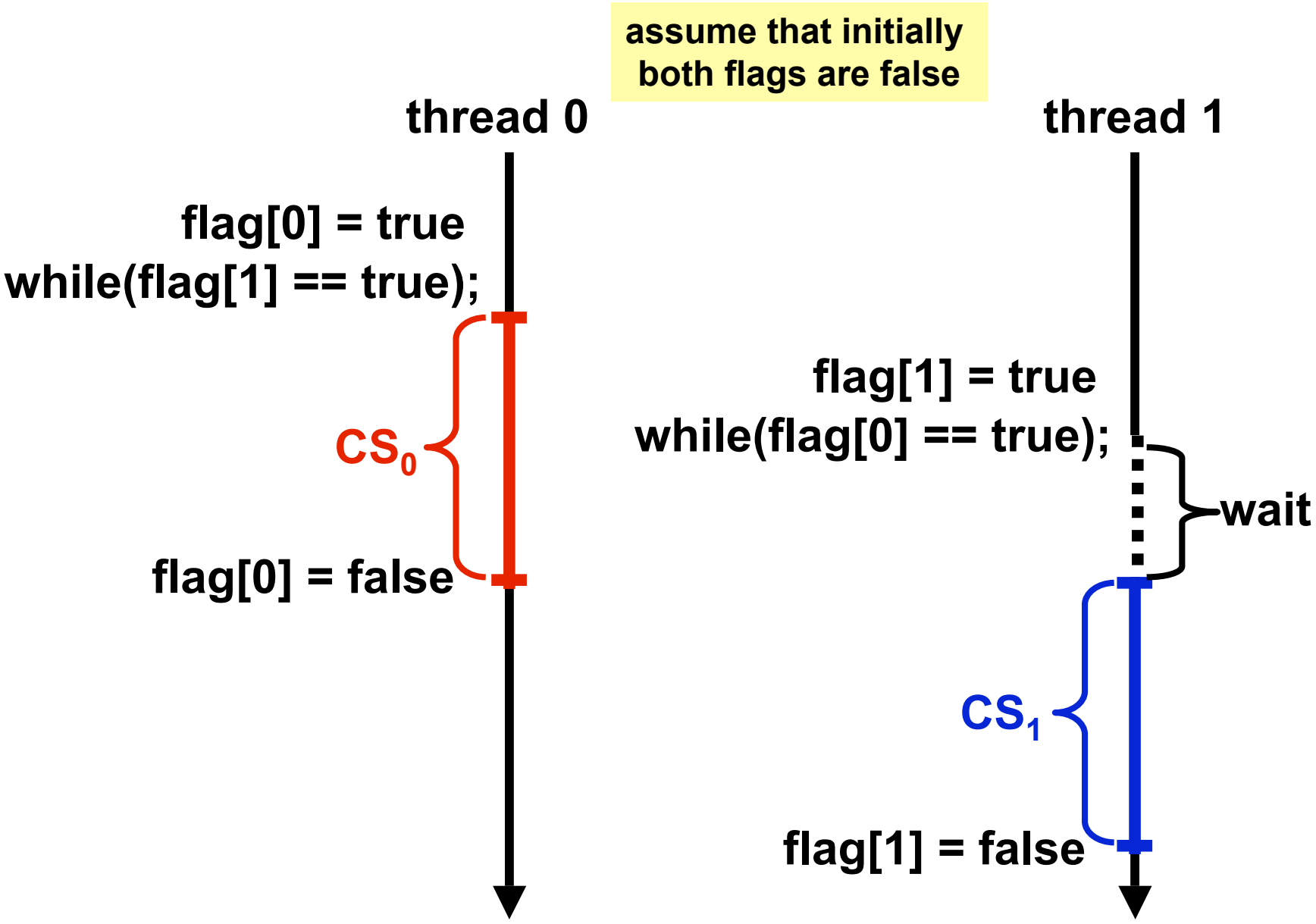
- First, consider two inadequate but interesting lock algorithms
  - —use multi-reader/multi-writer registers
- Assumptions
  - —only two threads
  - —each thread has a unique value of self_threadid ∈ {0,1}

Lock1

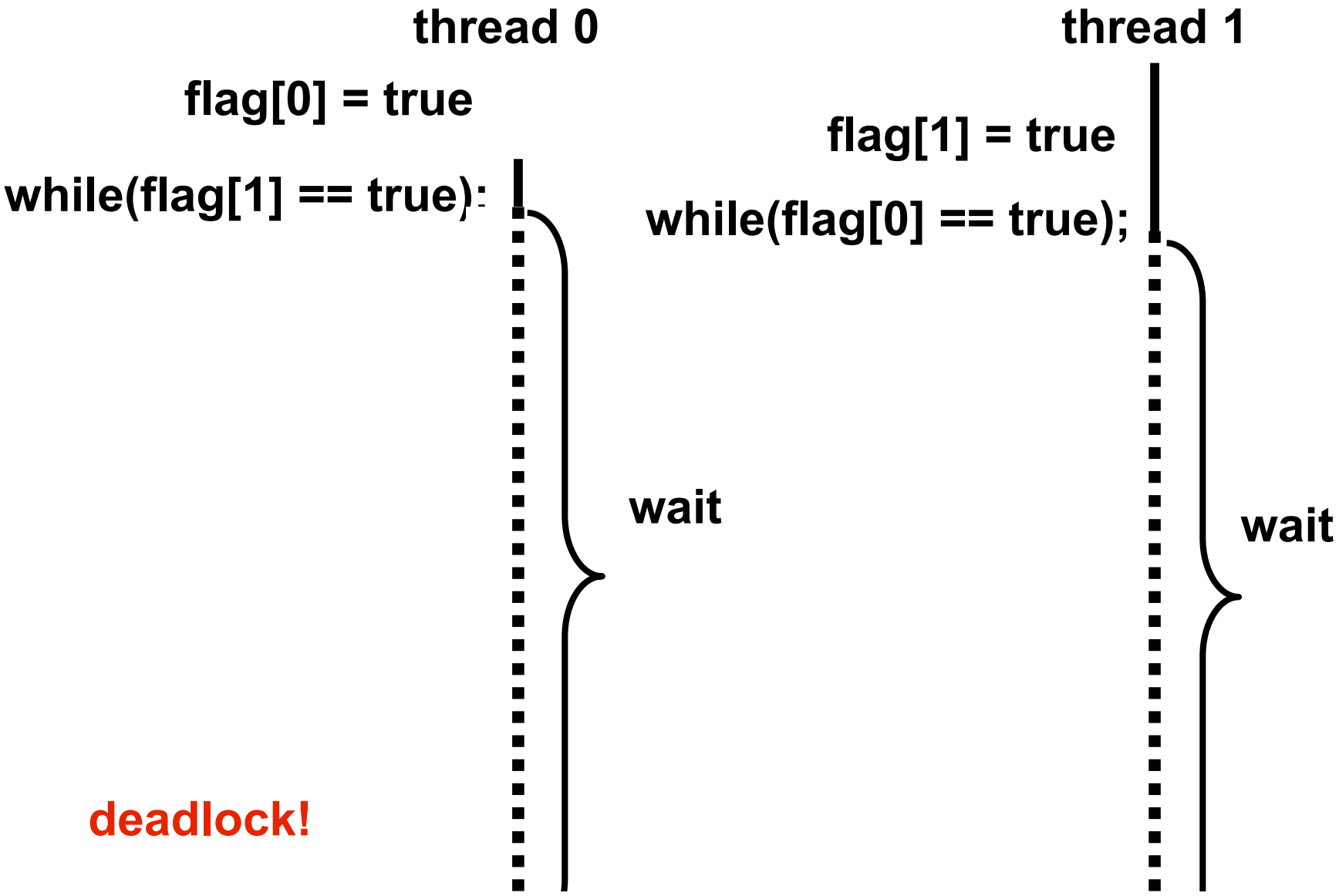
```
class Lock1: public Lock {
  private:
    volatile bool flag[2];
  public:
    void acquire() {
      int other_threadid = 1 - self_threadid;
      flag[self_threadid] = true;
      while (flag[other_threadid] == true);
    }
    void release() {
      flag[self_threadid] = false;
    }
}
```



Using Lock1


assume that initially
both flags are false
thread 0
thread 1
flag[0] = true
while(flag[1] == true);
CS0
flag[0] = false
flag[1] = true
while(flag[0] == true);
wait
CS1
flag[1] = false

Using Lock1


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

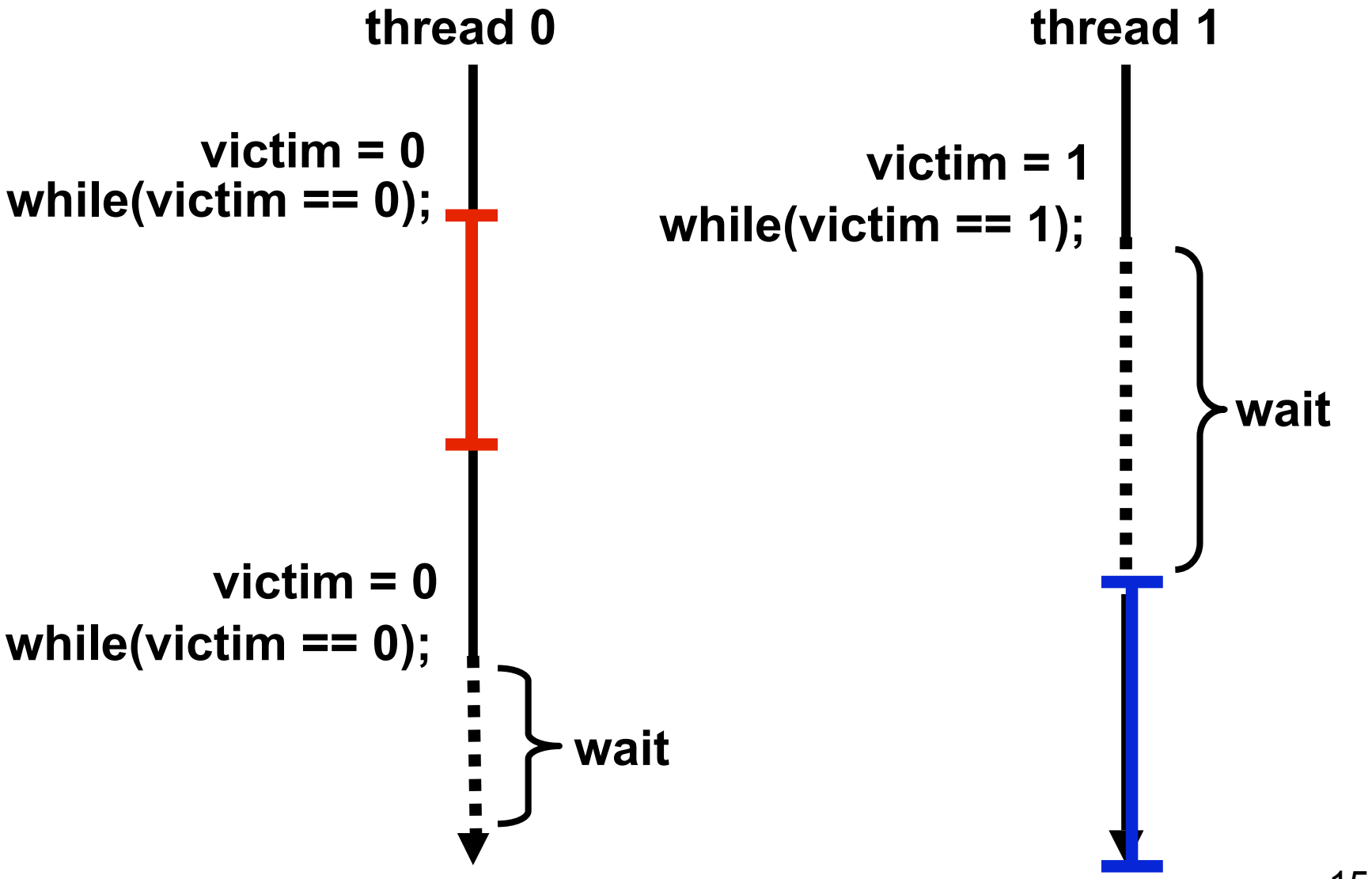
Summary of Lock1 Properties

- 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

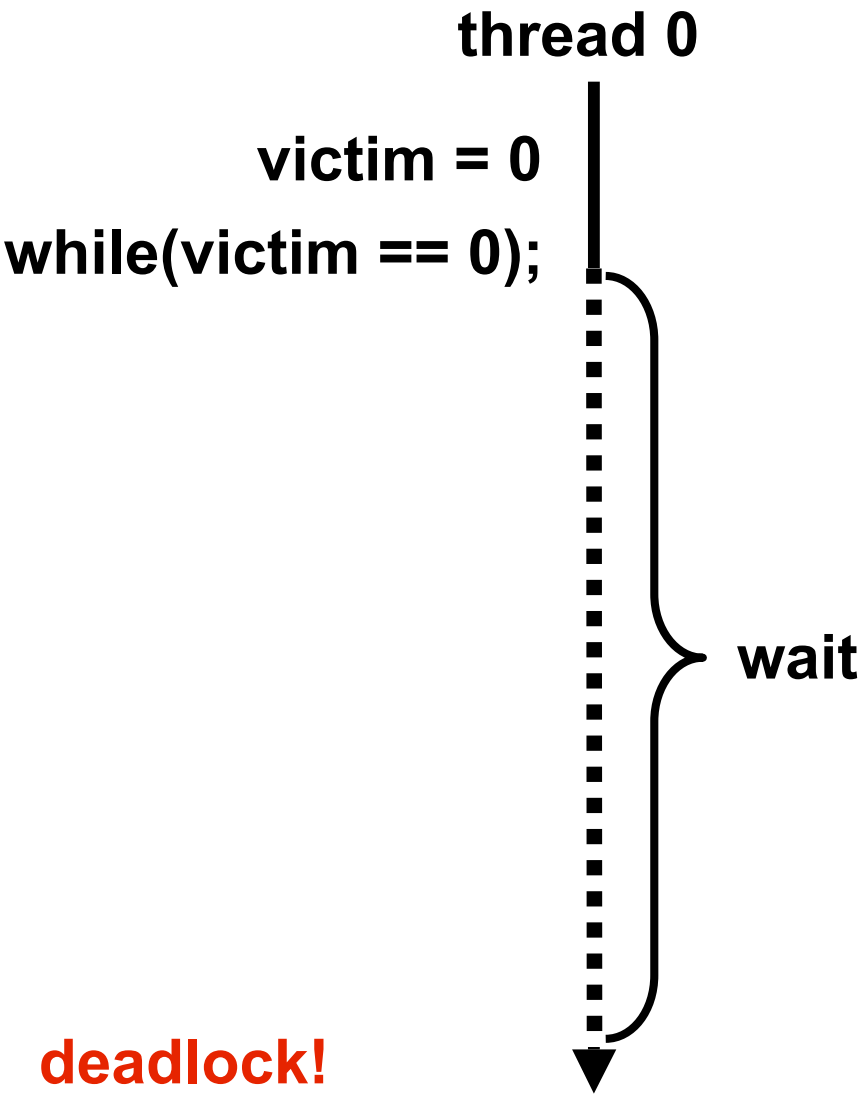
Lock2

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

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

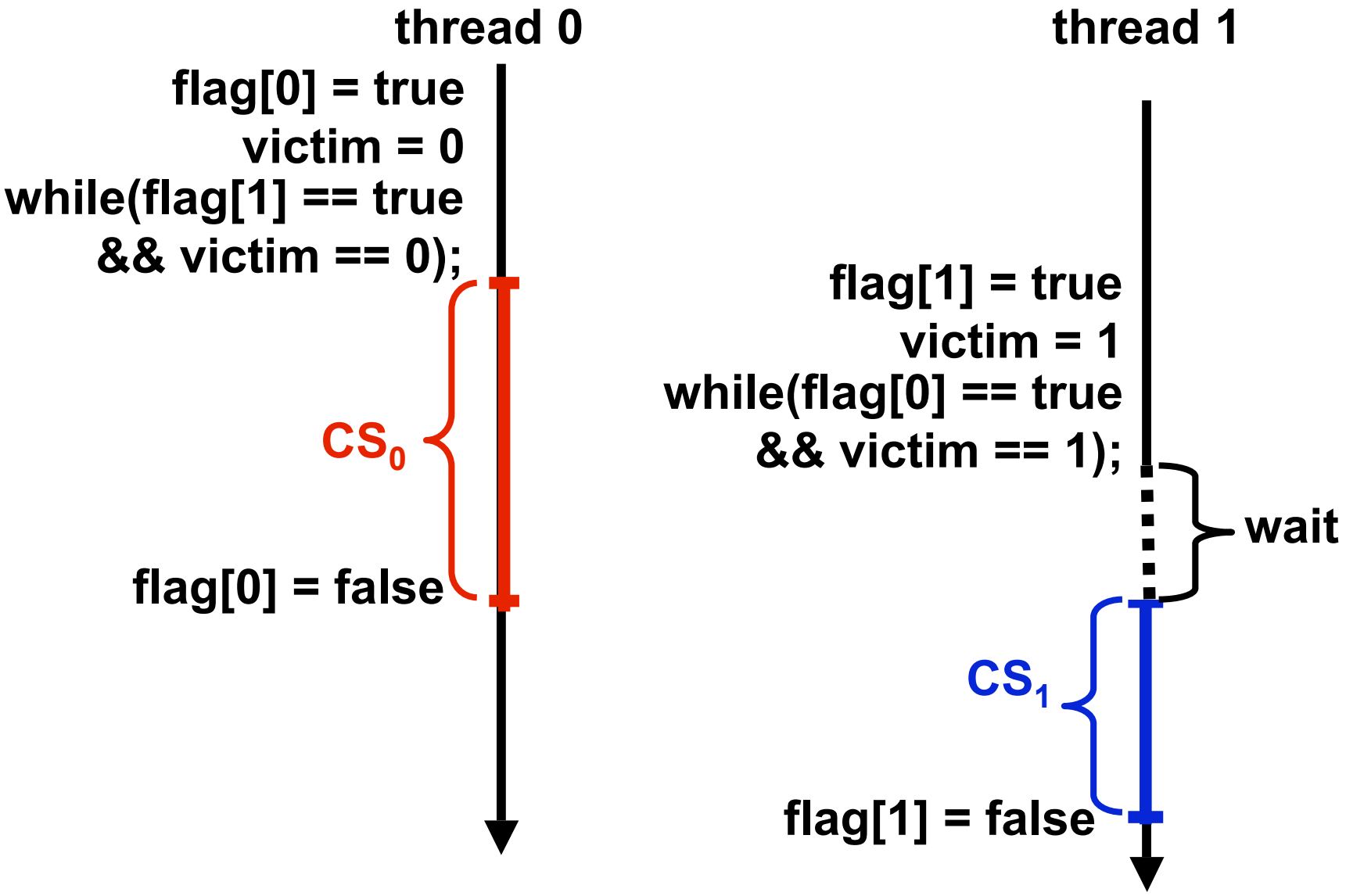
- **Each succeeds under conditions that causes the other to fail**
  - **—Lock1 succeeds when CS attempts do not overlap**
  - **—Lock2 succeeds when CS attempts do overlap**
- **Design a lock protocol that leverages the strengths of both…**

Peterson's Algorithm: 2-way Mutual Exclusion

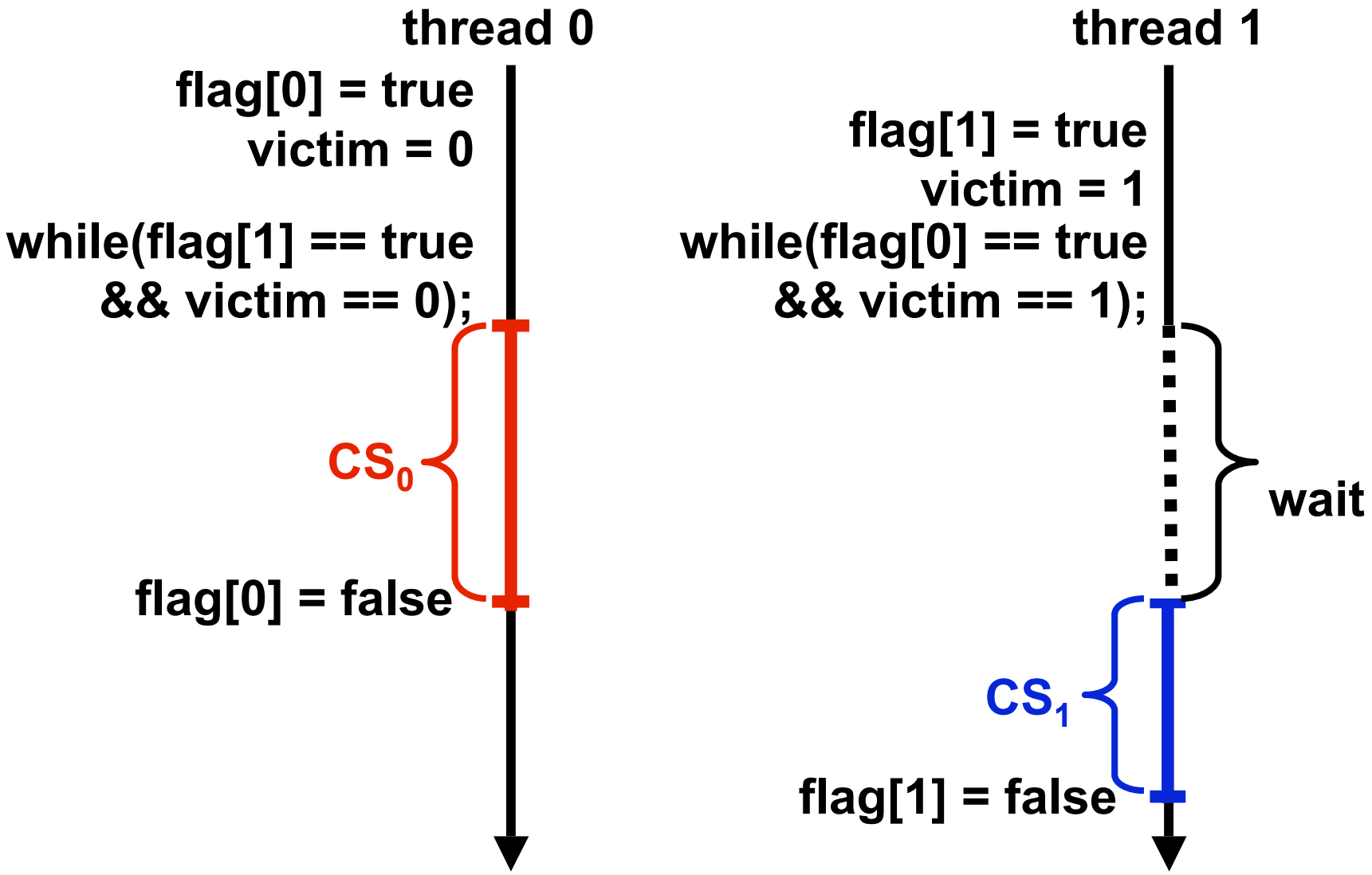
```
class Peterson: public Lock {
  private:
    volatile bool flag[2];
    volatile int victim;
  public:
    void acquire() {
      int other_threadid = 1 - self_threadid;
      flag[self_threadid] = true;    // I'm interested
      victim = self_threadid         // you go first
      while (flag[other_threadid]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



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

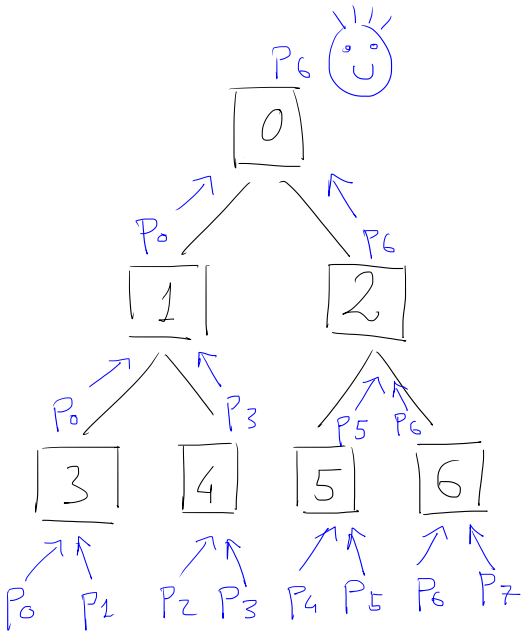
- Peterson's lock used two-element Boolean `flag` array
- Filter lock generalization: an N-element integer `level` array
  - —value of level[k] = highest level thread k is interested in entering
  - —each thread must pass through N-1 levels of exclusion
- Each level has it's own `victim` flag to filter 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

TOURNAMENT TREE

at each node
a run of
Peterson Alg.

⇓

a winner
is promoted
to the parent



```
flag_n[b] ← true;      wait_n ← b;
while flag_n[1 − b] = true and wait_n = b do
    {};
end while
if n = 0 then
    enter critical section;
    exit critical section;
else
    perform procedure Peterson(⌈n/2⌉− 1, (n + 1) mod 2);
end if
flag_n[b] ← false;
```

Lower bound on the number of registers

Theorem: At least n read/write registers are needed to solve deadlock-free mutual exclusion for n threads.

Proof (for $n = 2$): Given threads A, B, and one multi-reader/multi-writer register R.

Before A or B can enter its critical section, it must write to R.

Bring A and B in a position where they are about to write to R, after which they perform reads, and may enter their critical section.

Let A write to R first, perform reads, and enter its critical section.

The subsequent write by B obliterates the value A wrote to R, so B can no longer tell that A is in its critical section.

B also performs reads and enters its critical section.

Lower bound on the number of registers

Theorem: At least n read/write registers are needed to solve deadlock-free mutual exclusion for n threads.

Proof (for $n = 2$): Given threads A, B, and one multi-reader/multi-writer register R.

Before A or B can enter its critical section, it must write to R.

Bring A and B in a position where they are about to write to R, after which they perform reads, and may enter their critical section.

Let A write to R first, perform reads, and enter its critical section.

The subsequent write by B obliterates the value A wrote to R, so B can no longer tell that A is in its critical section.

B also performs reads and enters its critical section.

Lower bound on the number of registers

Theorem: At least n read/write registers are needed to solve deadlock-free mutual exclusion for n threads.

Proof (for $n = 2$): Given threads A, B, and one multi-reader/multi-writer register R.

Before A or B can enter its critical section, it must write to R.

Bring A and B in a position where they are about to write to R, after which they perform reads, and may enter their critical section.

Let A write to R first, perform reads, and enter its critical section.

The subsequent write by B obliterates the value A wrote to R, so B can no longer tell that A is in its critical section.

B also performs reads and enters its critical section.

Question

How does this proof idea carry over to general n?

Question

How does this proof idea carry over to general n?

Answer: With only $n-1$ registers, two threads must share a register to signal to other threads that they have entered their critical section.

Then the scenario from the previous slide applies.

Mutual exclusion with message passing

Mutual exclusion algorithms with *message passing* are generally based on one of the following paradigms.

- Leader election: A process that wants to become privileged sends a request to the leader.
- Token passing: The process holding the token is privileged.
- Logical clock: Requests to enter a critical section are prioritized by means of logical time stamps.
- Quorum: To become privileged, a process needs permission from a quorum of processes.

  Each pair of quorums has a non-empty intersection.

Ricart-Agrawala algorithm

When a process p_i wants to access its critical section, it sends $request(ts_i, i)$ to all other processes, with ts_i its logical time stamp.

When p_j receives this request, it sends permission to p_i as soon as :

- p_j isn't privileged, and
- p_j doesn't have a pending request with time stamp ts_j where $(ts_j, j) < (ts_i, i)$ (lexicographical order).

p_i enters its critical section when it has received permission from all other processes.

When p_i exits its critical section, it sends permission to . . . ?

Ricart-Agrawala algorithm

When a process p_i wants to access its critical section, it sends $request(ts_i, i)$ to all other processes, with ts_i its logical time stamp.

When p_j receives this request, it sends permission to p_i as soon as :

- p_j isn't privileged, and
- p_j doesn't have a pending request with time stamp ts_j where $(ts_j, j) < (ts_i, i)$ (lexicographical order).

p_i enters its critical section when it has received permission from all other processes.

When p_i exits its critical section, it sends permission to . . . ?

Ricart-Agrawala algorithm

When a process p_i wants to access its critical section, it sends $request(ts_i, i)$ to all other processes, with ts_i its logical time stamp.

When p_j receives this request, it sends permission to p_i as soon as :

- p_j isn't privileged, and
- p_j doesn't have a pending request with time stamp ts_j where $(ts_j, j) < (ts_i, i)$ (lexicographical order).

p_i enters its critical section when it has received permission from all other processes.

When p_i exits its critical section, it sends permission to . . . ?

Ricart-Agrawala algorithm

When a process p_i wants to access its critical section, it sends $request(ts_i, i)$ to all other processes, with ts_i its logical time stamp.

When p_j receives this request, it sends permission to p_i as soon as :

- p_j isn't privileged, and
- p_j doesn't have a pending request with time stamp ts_j where $(ts_j, j) < (ts_i, i)$ (lexicographical order).

p_i enters its critical section when it has received permission from all other processes.

When p_i exits its critical section, it sends permission to ... ?

Ricart-Agrawala algorithm

When a process p_i wants to access its critical section, it sends $request(ts_i, i)$ to all other processes, with ts_i its logical time stamp.

When p_j receives this request, it sends permission to p_i as soon as :

- p_j isn't privileged, and
- p_j doesn't have a pending request with time stamp ts_j where $(ts_j, j) < (ts_i, i)$ (lexicographical order).

p_i enters its critical section when it has received permission from all other processes.

When p_i exits its critical section, it sends permission to all pending requests.

Ricart-Agrawala algorithm - Example 1

$N = 2$, and p_0 and p_1 both are at logical time 0.

p_1 sends $request(0, 1)$ to p_0.

When p_0 receives this message, it sends permission to p_1, setting the time at p_0 to 2.

p_0 sends $request(2, 0)$ to p_1.

When p_1 receives this message, it doesn't send permission to p_0, because $(0, 1) < (2, 0)$.

p_1 receives permission from p_0, and enters its critical section.

Ricart-Agrawala algorithm - Example 2

$N = 2$, and p_0 and p_1 both are at logical time 0.

p_1 sends $request(0, 1)$ to p_0, and p_0 sends $request(0, 0)$ to p_1.

When p_0 receives the request from p_1,
it doesn't send permission to p_1, because $(0, 0) < (0, 1)$.

When p_1 receives the request from p_0,
it sends permission to p_0, because $(0, 0) < (0, 1)$.

p_0 and p_1 both set their logical time to 2.

p_0 receives permission from p_1, and enters its critical section.

Ricart-Agrawala algorithm - Correctness

Mutual exclusion: When p sends permission to q:

- p isn't privileged; and
- p won't get permission from q to enter its critical section until q has entered and left its critical section.

  (Because p's pending or future request is larger than q's current request.)

Starvation-freeness:

Ricart-Agrawala algorithm - Correctness

Mutual exclusion: When p sends permission to q:

- p isn't privileged; and
- p won't get permission from q to enter its critical section until q has entered and left its critical section.

  (Because p's pending or future request is larger than q's current request.)

Starvation-freeness:

Ricart-Agrawala algorithm - Correctness

Mutual exclusion: When p sends permission to q:

- p isn't privileged; and
- p won't get permission from q to enter its critical section until q has entered and left its critical section.

  (Because p's pending or future request is larger than q's current request.)

Starvation-freeness: Each request will eventually become the smallest request in the network.

Ricart-Agrawala algorithm - Optimization

Drawback: High message overhead, because requests must be sent to all other processes.

Carvalho-Roucairol optimization: After a process q has exited its critical section, q only needs to send requests to the processes that q has sent permission to since this exit.

Suppose q is waiting for permissions and didn't send a request to p.

If p sends a request to q that is smaller than q's request, then q *. . . does what ?*

This optimization is correct since for each pair of distinct processes, at least one must ask permission from the other.

Ricart-Agrawala algorithm - Optimization

Drawback: High message overhead, because requests must be sent to all other processes.

Carvalho-Roucairol optimization: After a process q has exited its critical section, q only needs to send requests to the processes that q has sent permission to since this exit.

Suppose q is waiting for permissions and didn't send a request to p.

If p sends a request to q that is smaller than q's request, then q *. . . does what ?*

This optimization is correct since for each pair of distinct processes, at least one must ask permission from the other.

Ricart-Agrawala algorithm - Optimization

Drawback: High message overhead, because requests must be sent to all other processes.

Carvalho-Roucairol optimization: After a process q has exited its critical section, q only needs to send requests to the processes that q has sent permission to since this exit.

Suppose q is waiting for permissions and didn't send a request to p.

If p sends a request to q that is smaller than q's request, then q *. . . does what ?*

This optimization is correct since for each pair of distinct processes, at least one must ask permission from the other.

Ricart-Agrawala algorithm - Optimization

Drawback: High message overhead, because requests must be sent to all other processes.

Carvalho-Roucairol optimization: After a process q has exited its critical section, q only needs to send requests to the processes that q has sent permission to since this exit.

Suppose q is waiting for permissions and didn't send a request to p.

If p sends a request to q that is smaller than q's request, then q sends both permission and a request to p.

This optimization is correct since for each pair of distinct processes, at least one must ask permission from the other.

Ricart-Agrawala algorithm - Optimization

Drawback: High message overhead, because requests must be sent to all other processes.

Carvalho-Roucairol optimization: After a process q has exited its critical section, q only needs to send requests to the processes that q has sent permission to since this exit.

Suppose q is waiting for permissions and didn't send a request to p.

If p sends a request to q that is smaller than q's request, then q sends both permission and a request to p.

This optimization is correct since for each pair of distinct processes, at least one must ask permission from the other.

Question

Let first p_0 and then p_1 become privileged.

Next they want to become privileged again.

Which scenario's are possible, if the Carvalho-Roucairol optimization is employed ?

Question

Let first p_0 and then p_1 become privileged.

Next they want to become privileged again.

Which scenario's are possible, if the Carvalho-Roucairol optimization is employed ?

Answer: p_0 needs permission from p_1, but not vice versa.

If p_0's request reaches p_1 before it wants to become privileged again, then p_1 sends permission and later a request to p_0.

Else p_1 enters its critical section, and answers p_0's request only after exiting the critical section.

Agrawal-El Abbadi algorithm (mutual exclusion with crashes)

To enter a critical section, permission from a quorum is required.

For simplicity we assume that $N = 2^k - 1$, for some $k > 1$.

The processes are structured in a *binary tree* of depth $k - 1$.

A *quorum* consists of all processes on a path from the root to a leaf.

If a *non-leaf* p has crashed (or is unresponsive), permission is asked from all processes on two paths instead: from each child of p to a leaf.

Agrawal-El Abbadi algorithm (mutual exclusion with crashes)

To enter a critical section, permission from a quorum is required.

For simplicity we assume that $N = 2^k - 1$, for some $k > 1$.

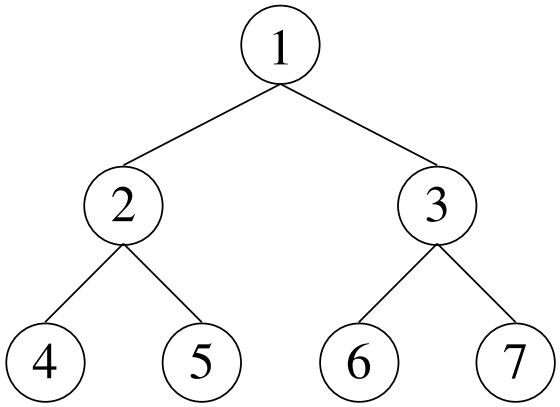
The processes are structured in a *binary tree* of depth $k - 1$.

A *quorum* consists of all processes on a path from the root to a leaf.

If a *non-leaf* p has crashed (or is unresponsive), permission is asked from all processes on two paths instead: from each child of p to a leaf.

Agrawal-El Abbadi algorithm - Example

Example: Let $N = 7$.



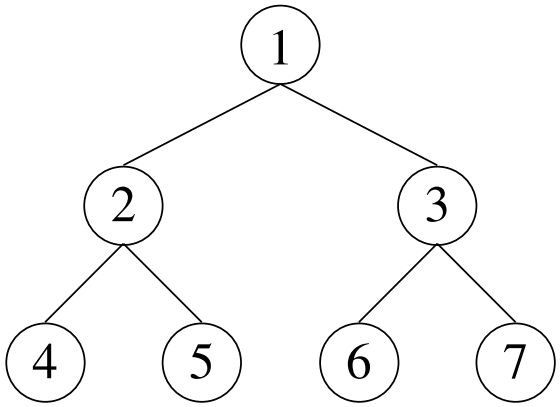
Possible quorums are:

- $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 3, 6\}$, $\{1, 3, 7\}$
- if 1 crashed: $\{2, 4, 3, 6\}$, $\{2, 5, 3, 6\}$, $\{2, 4, 3, 7\}$, $\{2, 5, 3, 7\}$
- if 2 crashed: $\{1, 4, 5\}$ (and $\{1, 3, 6\}$, $\{1, 3, 7\}$)
- if 3 crashed: $\{1, 6, 7\}$ (and $\{1, 2, 4\}$, $\{1, 2, 5\}$)

Question: What are the quorums if 1,2 crashed? And if 1,2,3 crashed? And if 1,2,4 crashed?

Agrawal-El Abbadi algorithm - Example

Example: Let $N = 7$.



Possible quorums are:

- $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 3, 6\}$, $\{1, 3, 7\}$
- if 1 crashed: $\{2, 4, 3, 6\}$, $\{2, 5, 3, 6\}$, $\{2, 4, 3, 7\}$, $\{2, 5, 3, 7\}$
- if 2 crashed: $\{1, 4, 5\}$ (and $\{1, 3, 6\}$, $\{1, 3, 7\}$)
- if 3 crashed: $\{1, 6, 7\}$ (and $\{1, 2, 4\}$, $\{1, 2, 5\}$)

Question: What are the quorums if 1,2 crashed? And if 1,2,3 crashed? And if 1,2,4 crashed?

Agrawal-El Abbadi algorithm

A process p that wants to enter its critical section, places the root of the tree in a queue.

p repeatedly tries to get permission from the head r of its queue.

If successful, r is removed from p's queue.

If r is a non-leaf, *one* of r's children is appended to p's queue.

If non-leaf r has crashed, it is removed from p's queue, and *both* of r's children are appended at the end of the queue (in a fixed order, to avoid deadlocks).

If leaf r has crashed, p *aborts* its attempt to become privileged.

When p's queue becomes empty, it enters its critical section.

After exiting its critical section, p informs all processes in the quorum that their permission to p can be withdrawn.

Agrawal-El Abbadi algorithm

A process p that wants to enter its critical section, places the root of the tree in a queue.

p repeatedly tries to get permission from the head r of its queue.

If successful, r is removed from p's queue.

If r is a non-leaf, *one* of r's children is appended to p's queue.

If non-leaf r has crashed, it is removed from p's queue, and *both* of r's children are appended at the end of the queue (in a fixed order, to avoid deadlocks).

If leaf r has crashed, p *aborts* its attempt to become privileged.

When p's queue becomes empty, it enters its critical section.

After exiting its critical section, p informs all processes in the quorum that their permission to p can be withdrawn.

Agrawal-El Abbadi algorithm

A process p that wants to enter its critical section, places the root of the tree in a queue.

p repeatedly tries to get permission from the head r of its queue.

If successful, r is removed from p's queue.

If r is a non-leaf, *one* of r's children is appended to p's queue.

If non-leaf r has crashed, it is removed from p's queue, and *both* of r's children are appended at the end of the queue (in a fixed order, to avoid deadlocks).

If leaf r has crashed, p *aborts* its attempt to become privileged.

When p's queue becomes empty, it enters its critical section.

After exiting its critical section, p informs all processes in the quorum that their permission to p can be withdrawn.

Agrawal-El Abbadi algorithm

A process p that wants to enter its critical section, places the root of the tree in a queue.

p repeatedly tries to get permission from the head r of its queue.

If successful, r is removed from p's queue.

If r is a non-leaf, *one* of r's children is appended to p's queue.

If non-leaf r has crashed, it is removed from p's queue, and *both* of r's children are appended at the end of the queue (in a fixed order, to avoid deadlocks).

If leaf r has crashed, p *aborts* its attempt to become privileged.

When p's queue becomes empty, it enters its critical section.

After exiting its critical section, p informs all processes in the quorum that their permission to p can be withdrawn.

Agrawal-El Abbadi algorithm

A process p that wants to enter its critical section, places the root of the tree in a queue.

p repeatedly tries to get permission from the head r of its queue.

If successful, r is removed from p's queue.

If r is a non-leaf, *one* of r's children is appended to p's queue.

If non-leaf r has crashed, it is removed from p's queue, and *both* of r's children are appended at the end of the queue (in a fixed order, to avoid deadlocks).

If leaf r has crashed, p *aborts* its attempt to become privileged.

When p's queue becomes empty, it enters its critical section.

After exiting its critical section, p informs all processes in the quorum that their permission to p can be withdrawn.

Agrawal-El Abbadi algorithm

A process p that wants to enter its critical section, places the root of the tree in a queue.

p repeatedly tries to get permission from the head r of its queue.

If successful, r is removed from p's queue.

If r is a non-leaf, *one* of r's children is appended to p's queue.

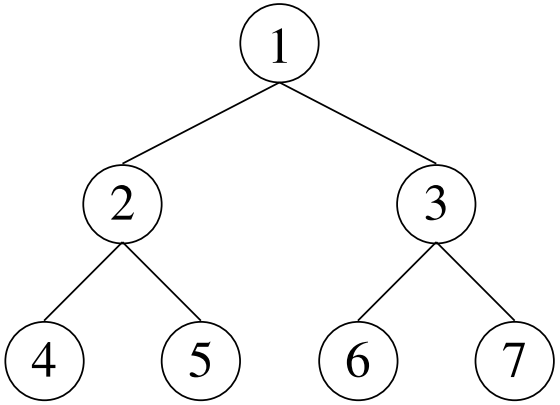
If non-leaf r has crashed, it is removed from p's queue, and *both* of r's children are appended at the end of the queue (in a fixed order, to avoid deadlocks).

If leaf r has crashed, p *aborts* its attempt to become privileged.

When p's queue becomes empty, it enters its critical section.

After exiting its critical section, p informs all processes in the quorum that their permission to p can be withdrawn.

Agrawal-El Abbadi algorithm - Example



p and q concurrently want to enter their critical section.

p gets permission from 1, and wants permission from 3.

1 crashes, and q now wants permission from 2 and 3.

q gets permission from 2, and appends 4 to its queue.

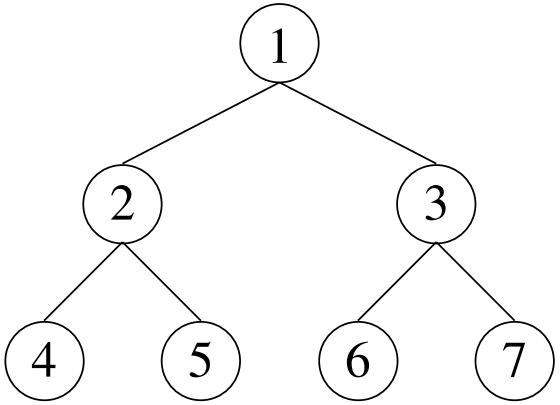
q obtains permission from 3, and appends 7 to its queue.

3 crashes, and p now wants permission from 6 and 7.

q gets permission from 4, and now wants permission from 7.

p gets permission from both 6 and 7, and enters its critical section.

Agrawal-El Abbadi algorithm - Example



p and q concurrently want to enter their critical section.

p gets permission from 1, and wants permission from 3.

1 crashes, and q now wants permission from 2 and 3.

q gets permission from 2, and appends 4 to its queue.

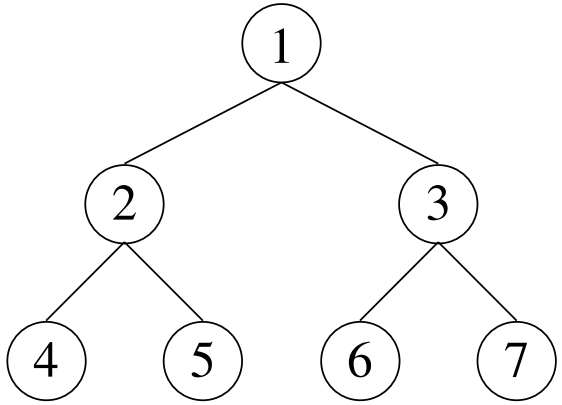
q obtains permission from 3, and appends 7 to its queue.

3 crashes, and p now wants permission from 6 and 7.

q gets permission from 4, and now wants permission from 7.

p gets permission from both 6 and 7, and enters its critical section.

Agrawal-El Abbadi algorithm - Example



p and q concurrently want to enter their critical section.

p gets permission from 1, and wants permission from 3.

1 crashes, and q now wants permission from 2 and 3.

q gets permission from 2, and appends 4 to its queue.

q obtains permission from 3, and appends 7 to its queue.

3 crashes, and p now wants permission from 6 and 7.

q gets permission from 4, and now wants permission from 7.

p gets permission from both 6 and 7, and enters its critical section.

Agrawal-El Abbadi algorithm - Mutual exclusion

We prove, by induction on depth k, that each pair of quorums has a non-empty intersection, and so mutual exclusion is guaranteed.

A quorum with 1 contains a quorum in one of the subtrees below 1, while a quorum without 1 contains a quorum in both subtrees below 1.

- If two quorums both contain 1, we are done.
- If two quorums both don't contain 1, then by induction they have elements in common in the two subtrees below process 1.
- Suppose quorum Q contains 1, while quorum Q' doesn't.

  Then Q contains a quorum in one of the subtrees below 1, and Q' also contains a quorum in this subtree.

  By induction, they have an element in common in this subtree.

Agrawal-El Abbadi algorithm - Mutual exclusion

We prove, by induction on depth k, that each pair of quorums has a non-empty intersection, and so mutual exclusion is guaranteed.

A quorum with 1 contains a quorum in one of the subtrees below 1, while a quorum without 1 contains a quorum in both subtrees below 1.

- If two quorums both contain 1, we are done.
- If two quorums both don't contain 1, then by induction they have elements in common in the two subtrees below process 1.
- Suppose quorum Q contains 1, while quorum Q' doesn't.

  Then Q contains a quorum in one of the subtrees below 1, and Q' also contains a quorum in this subtree.

  By induction, they have an element in common in this subtree.

Agrawal-El Abbadi algorithm - Deadlock-freeness

In case of a crashed process, let its left child be put before its right child in the queue of a process that wants to become privileged.

Let a process p at depth d be greater than any process

- at a depth $> d$ in the binary tree, or
- at depth d and more to the right than p in the binary tree.

A process with permission from r, never needs permission from a $q < r$.

This guarantees that, in case some leave is responsive, eventually some process will become privileged.

Agrawal-El Abbadi algorithm - Deadlock-freeness

In case of a crashed process, let its left child be put before its right child in the queue of a process that wants to become privileged.

Let a process p at depth d be greater than any process

- at a depth $> d$ in the binary tree, or
- at depth d and more to the right than p in the binary tree.

A process with permission from r, never needs permission from a $q < r$.

This guarantees that, in case some leave is responsive, eventually some process will become privileged.