PURDUE UNIVERSITY

# Principles of Concurrency

Week 2

Mutual Exclusion

# Coordination



- In a shared-memory system, threads communicate by updating and observing changes to shared state
- These observations and changes are known as *events*
- Intuitively, a thread should be able to observe all the events that *happened-before* it
  - Assumptions:
    - events are instantaneous
    - events are discrete i.e, no two events can happen "at the same time"
- For the purposes of reasoning about threads, we can also assume that a thread is comprised of a sequence of events
- The interleaved sequence capturing the events of all threads is known as a *trace*

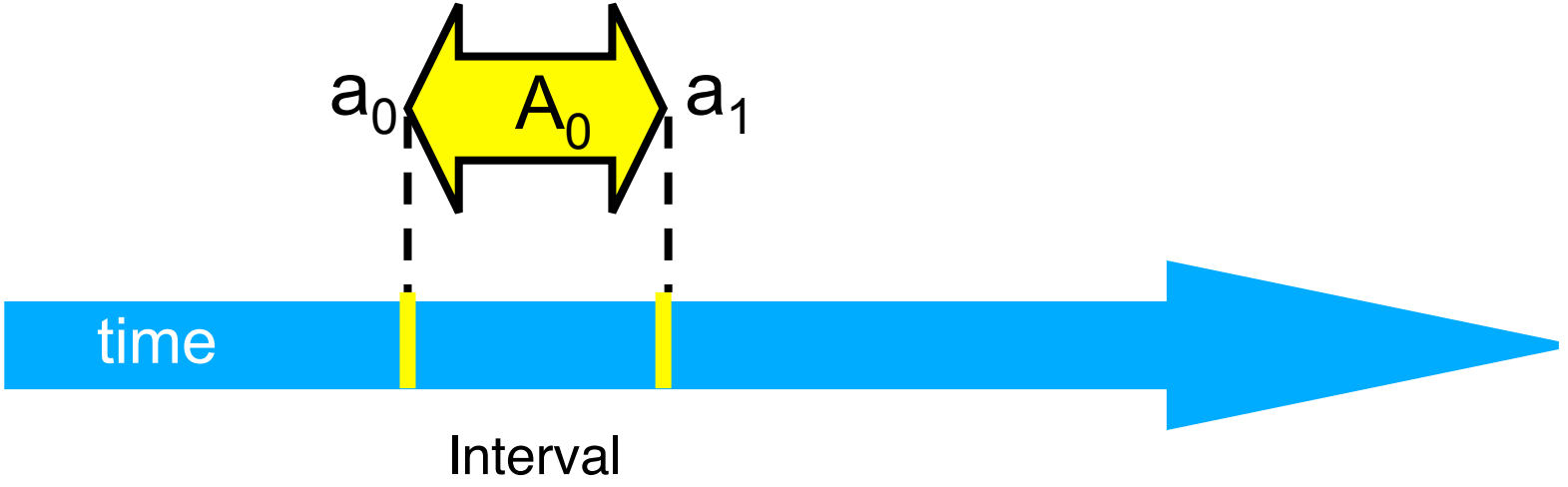
# Coordination



We assume a fixed set of events defined by a concurrent programming language:

- Read the value of a shared variable
- Write a value to a shared variable
- Invoke a method
- Return from a method

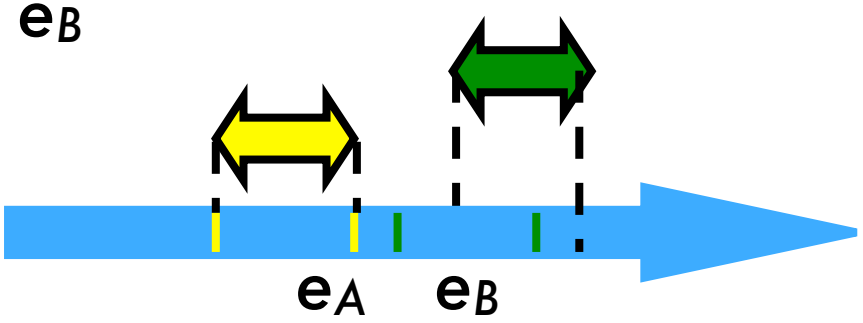
All of these events are treated as atomic actions, represented as distinct elements in a trace



# Happens-Before



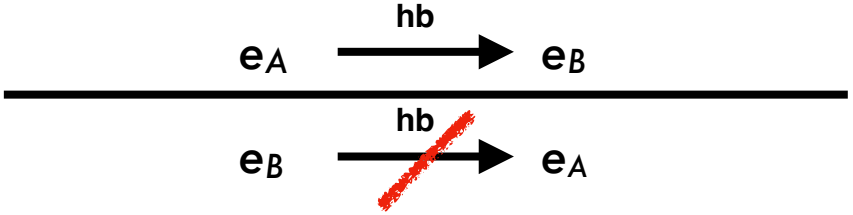
- If the end event $e_A$ of an interval $A_T$ by thread T occurs before the start event $e_B$ of an interval $B_{T'}$ by thread T', then $e_A$ happens-before $e_B$



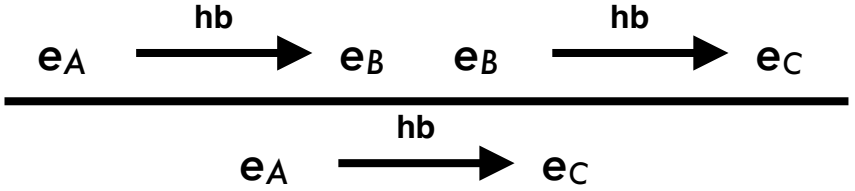

$e_A \xrightarrow{hb} e_B$

The happens-before relation is:

- irreflexive: $e_A \not\xrightarrow{hb} e_A$
- anti-symmetric




- transitive:



# Mutual Exclusion



Must enforce a happens-before order on all accesses to a shared object

- All events over the object (i.e., the interval comprising all accesses) must be related under hb

Example: a shared counter (assume temp is a thread-local register)

```
fun increment() = {
        temp = value
        value = temp + 1
}
```

Need to enforce that the read and write of a value happen atomically (i.e., are indivisible). Why?

# Locking



## One obvious approach: Use locks

```
fun increment() = {
    lock.acquire()
    temp = value
    value = temp + 1
    lock.release()
}
```

*critical section:*

lock is also a shared resource.

A happens-before edge is established between the `acquire` and the latest `release` in a trace

Clearly enforces mutual exclusion since the semantics of locks ensures that all accesses to value by different threads are totally ordered under happens-before

The solution is also *deadlock-free*:

Every acquire is matched with a release

The solution is *not starvation-free*:

No guarantee that a thread will ever successfully acquire the lock

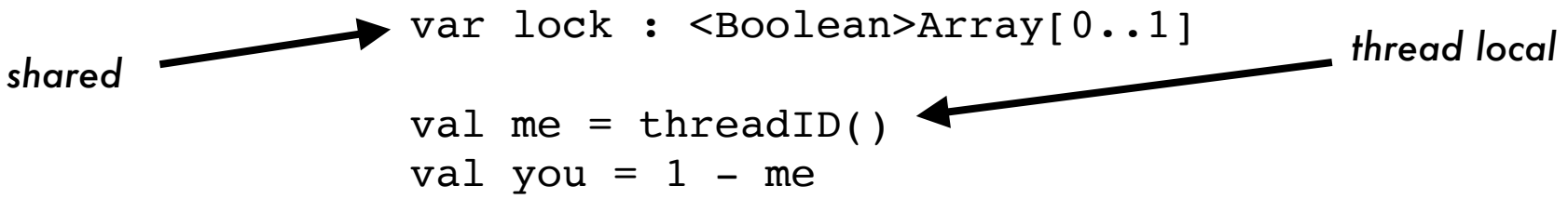
# Implementing a lock



Can enforce locking using hardware

- But, this may be expensive

Can we implement a mutual exclusion lock as a protocol?




```
fun acquire () = {
  lock[me] = true
  while (lock[you]) {}
```

```
fun release () = {
     lock[me] = false
```

- Satisfies mutual exclusion
- What about deadlock-freedom? Starvation-freedom?

# Another variant …



```
var turn : int

fun acquire () = {
   turn = me
   while (turn == me) {}
}

fun release () = {}
```

If thread i is in critical section, then turn = j. (Turn represents a request by a thread to get access to CS.)

Suppose two threads concurrently attempt to acquire access to CS. The second thread to update turn allows the first thread to get through. When this thread subsequently attempts to reacquire the lock, the (waiting) second thread is allowed in.

Is mutual exclusion satisfied?
What about deadlock-freedom? Starvation-freedom?

# Peterson's Algorithm



```
var lock : <Boolean>Array[0..1]
var turn : Int

val me = threadID()
val you = 1 - me
```

```
fun acquire () = {
   lock[me] = true
   turn = me
   while (lock[you] && turn = me) {}
```

```
fun release () = {
   lock[me] = false
}
```

A thread is blocked if:

- the other thread's lock flag is true
- if it has requested access to the CS

- A single thread can enter the CS because the other lock (lock[you]) would be false
- If two threads concurrently attempt to enter the CS, their lock bits would both be set, but turn would be set to one or the other (but not both)

# Bakery Algorithm



- Previous approaches work for two threads.
- Generalize these methods for n threads.

*Intuition*:

- Customers entering a bakery collect a number from a machine.
- Numbers are unique and monotonically increasing.

A global counter displays the current number being served. Customers, other than the one being served, wait in a queue. The counter is incremented when the current customer is finished.

In practice, cannot prevent two threads (customers) from getting the same number, so use the thread ID to establish a priority. (Lower thread IDs have higher priority.)

# Bakery Algorithm



```
var flag : <Boolean>Array[0..n-1]
var count : <Int>Array[0..n-1]
```

*initially, all false*

initially, all 0

Thread i executes the following:

```
fun acquire() = {
  flag[i] = true
  count[i] = max(count[0],…,count[n-1])+1
  while (exists (k != i) s.t.. flag[k] &&
                              (count[k], k) << (count[i],i)) {}
}
```

*Get an increasing entry number - this is racy!*

*Wait until earlier, higher priority, threads wanting to enter CS have completed*

```
fun release () = {
    flag[i] = false
}
```

*No longer interested in remaining in critical section*

# Bakery Algorithm



- Labels represent timestamps
- How do we ensure timestamps never overflow? Can we reuse them?
- Satisfies mutual exclusion? Deadlock-freedom? Starvation-freedom?
- Drawbacks:
  - To enter a lock, requires reading at least $2n$ distinct variables (representing the flag and counts of all other $n$ threads