

## 01. MUTUAL EXCLUSION

### properties of a mutex algorithm

- mutual exclusion** → no more than one process in the critical section
- progress** → if one or more process wants to enter and if no one is in the critical section, then one of them can eventually enter the critical section
- no starvation** → if a process wants to enter, it eventually can always enter
  - no starvation implies progress!
- ! if a process is in the CS, we always assume that it will eventually exit the CS
  - finite number of instructions in the CS

### Peterson's Algorithm

```
Shared bool wantCS[0] = false, bool wantCS[1] = false, int turn = 0;

Process 0
RequestCS(0) {
    wantCS[0] = true;
    turn = 1;
    while (wantCS[1] == true && turn == 1) {};
}
ReleaseCS(0) {
    wantCS[0] = false;
}

Process 1
RequestCS(1) {
    wantCS[1] = true;
    turn = 0;
    while (wantCS[0] == true && turn == 0) {};
}
ReleaseCS(1) {
    wantCS[1] = false;
}
```

### proof

- mutual exclusion:** proof by contradiction
  - case 1 - turn == 0 when P0 and P1 are both in CS
    - then P0 executed turn=1 before P1 executed turn=0
    - hence wantCS[0]==false as seen by P1
    - but wantCS[0] set to true by P0
  - case 2 - turn ==1. symmetric
- progress:** proof by contradiction
  - suppose both want to enter but none can enter ⇒ wantCS must be true for both
  - case 1: turn==0. then P0 can enter
  - case 2: symmetric
- no starvation:** proof by contradiction
  - case 1: P0 is waiting, then wantCS[1]==true and turn=1
    - P1 in critical region - eventually it exits and sets wantCS[1] to false
    - (what if P1 wants to enter again immediately? then P1 will wait first because wantCS[0]==true and it has set turn==0)
  - case 2: P1 is waiting. symmetric

## Lamport's Bakery Algorithm

• for  $n$  processes

- get a number first (weak guarantee)
- get served when all lower numbers have been served (sufficient for mutex)

• 2 shared arrays of  $n$  elements

- boolean choosing[i] = false ⇒ is process  $i$  trying to get a number
- int number[i] = 0 ⇒ the number gotten by process  $i$
- number[i] > 0: process wants to enter CS and that is the queue number

```
ReleaseCS(int myid) {
    number[myid] = 0;
}

// a utility function
boolean Smaller(int number1, int id1, int number2, int id2) {
    if (number1 < number2) return true;
    if (number1 == number2) {
        if (id1 < id2) return true; else return false;
    }
    if (number1 > number2) return false;
}

RequestCS(int myid) {
    choosing[myid] = true;
    for (int j = 0; j < n; j++) {
        if (number[j] > number[myid]) number[myid] = number[j];
    }
    number[myid]++;
    choosing[myid] = false;
}

get a number
wait for people ahead of me
for (int j = 0; j < n; j++) {
    while (choosing[j] == true);
    while (number[j] != 0 && Smaller(number[j], j, number[myid], myid));
}
```

## Hardware Solutions

- disable interrupts** - to prevent context switch
  - do not allow context switch in the critical section
- special machine-level instructions: **TestAndSet** executed atomically
  - when you design CPU, you want all instructions to roughly be the same complexity so that your pipelines don't have bubbles in it
  - degrades performance

```
boolean TestAndSet(Boolean openDoor, boolean newValue) {
    boolean tmp = openDoor.getValue();
    openDoor.setValue(newValue);
    return tmp;
}
```

} Executed atomically

```
shared Boolean variable openDoor initialized to true;
RequestCS(process_id) {
    while (TestAndSet(openDoor, false) == false) {};
}
ReleaseCS(process_id) { openDoor.setValue(true); }
```

## Proof: Lamport's Bakery Algorithm

▪ **Progress:** Proof by contradiction.  
 Consider any set of processes that wants to enter the CS but no one can make progress. Each process is guaranteed to get a queue #. Let process  $i$  be the one with the smallest queue number. Consider where process  $i$  can be blocked:

Case 1:  
 Process  $j$  will eventually set choosing[j] to false  
 Process  $j$  will then block (otherwise there is progress already!)  
 Case 2: Impossible since process  $i$  has the smallest queue number

**Mutual exclusion:** Suppose  $i$  and  $k$  both in critical section.

At T1, process  $i$  is here  
 choosing[myid] = true;  
 for (int j = 0; j < n; j++)  
 if (number[j] > number[myid])  
 number[myid] = number[j];  
 number[myid]++;  
 choosing[myid] = false;

for (int j = 0; j < n; j++) {  
 while (choosing[j] == true);  
 while (number[j] != 0 && Smaller(number[j], j, number[myid], myid));  
} At T1, process  $k$  is here

Case 2: At T2, process  $i$  is here  
 At T1, process  $i$  is here  
 choosing[myid] = true;  
 for (int j = 0; j < n; j++)  
 if (number[j] > number[myid])  
 number[myid] = number[j];  
 number[myid]++;  
 choosing[myid] = false;

At T2, process  $k$  is here  
 for (int j = 0; j < n; j++) {  
 while (choosing[j] == true);  
 while (number[j] != 0 && Smaller(number[j], j, number[myid], myid));  
} At T1, process  $k$  is here

Now continue and consider the time T2 when process  $k$  invoked "while (choosing[i] == true)" and passed that statement.

We want to see where process  $i$  is at T2. Since choosing[i] == false, process  $i$  must either have finished choosing its queue number or have not started choosing:

- Case 1: process  $i$  has finished choosing and has executed choosing[i] == false; Impossible since T2 < T1.
- Case 2: process  $i$  has not started choosing and has not executed choosing[i] == true. But then number[i] will be larger than number[k]. Contradiction.

## 02. SYNCHRONISATION PRIMITIVES

- solves **busy wait problem** (wastes CPU cycles)
- synchronisation primitives: OS-level APIs that the program may call

### Semaphores

2 variables for each semaphore

- boolean value := true
- queue (of blocked processes) := empty

**2 APIs, executed atomically**

P() - wait

- if value == false, add self to queue and block
- can context switch to some other process

V() - signal

- set value = true

- wake up one arbitrary process in queue

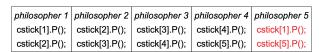
**semaphore for mutex:**

- RequestCS() { P(); }
- ReleaseCS() { V(); }

### dining philosophers

- one semaphore for each chopstick

- waits-for graph has a cycle  $\Rightarrow$  **deadlock**
  - avoid cycles in WFG / have a total ordering

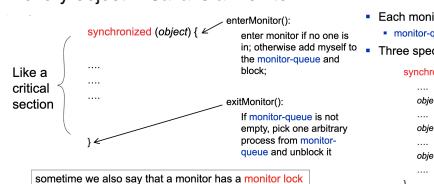


- Avoid cycles (or have a total ordering of the chopsticks)



### Monitor

- ✓ higher-level/easier to use than semaphore
- every object in Java is a monitor



### Hoare-style

- notify() immediately switches from caller to a waiting thread
- doesn't use notifyAll()

#### First kind: Hoare-style Monitor

Process 0	Process 1
synchronized (object) { if (x != 1) object.wait();}	
	synchronized (object) { x=1; object.notify();}
* assert(x == 1); // x must be 1 x=2; }	// needs to acquire monitor lock // x may not be 1 here }

process 0 takes over the execution

### Java-style

- notify() places a waiter on the ready thread but signaller continues inside the monitor

#### Second kind: Java-style Monitor

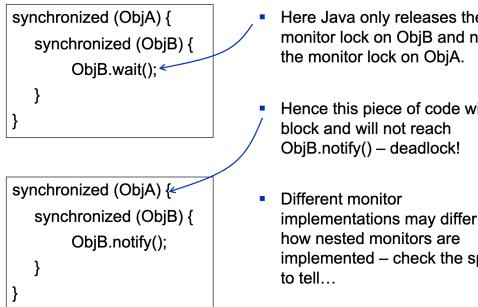
Process 0	Process 1
synchronized (object) { if (x != 1) object.wait();}	
	synchronized (object) { x=1; object.notify();}
* assert(x == 1); // x must be 1 x=2; }	// needs to acquire monitor lock // x may not be 1 here }

- P0 must acquire monitor lock
  - use (while x!=1) to ensure x==1

### nested monitor

- wait() only releases the *immediate* monitor lock

#### Nested Monitor in Java



- Here Java only releases the monitor lock on ObjB and not the monitor lock on ObjA.

- Hence this piece of code will block and will not reach ObjB.notify() – deadlock!

- Different monitor implementations may differ in how nested monitors are implemented – check the spec to tell...

### producer-consumer problem

- circular buffer of size  $n$
- single producer, single consumer
  - producer places item to the end of the buffer if buffer is not full
  - consumer removes item from the head of the buffer if not empty

object sharedBuffer;

```
void produce() {
    synchronized (sharedBuffer) {
        if (sharedBuffer is full)
            sharedBuffer.wait();
        add an item to sharedBuffer;
        if (sharedBuffer *was* empty)
            sharedBuffer.notify();
    }
}

void consume() {
    synchronized (sharedBuffer) {
        if (sharedBuffer is empty)
            sharedBuffer.wait();
        remove item from sharedBuffer;
        if (sharedBuffer *was* full)
            sharedBuffer.notify();
    }
}
```

### reader-writer problem

- multiple readers and writers accessing a file
  - writer must have exclusive access
  - readers may simultaneously access the file

int numReader, numWriter; Object object;

```
void writeFile() {
    synchronized (object) {
        while (numReader > 0 || numWriter > 0)
            object.wait();
        numWriter = 1;
    }
    // write to file;
    synchronized (object) {
        numWriter = 0;
        object.notify();
    }
}

void readFile() {
    synchronized (object) {
        while (numWriter > 0)
            object.wait();
        numReader++;
    }
    // read from file;
    synchronized (object) {
        numReader--;
        object.notify();
    }
}
```

you can prove that it must be a writer who is notified

### reader-writer (without starvation)

- maintain an explicit queue

```
Vector queue; // shared among all threads
int numReader; int numWriter;

// code for writer entry
Writer w = new Writer(myname);
synchronized (queue) {
    if (numReader > 0 || numWriter > 0) {
        w.okToGo = false;
        queue.add(w);
    } else {
        w.okToGo = true;
        numWriter++;
    }
}
synchronized (w) { if (!w.okToGo) w.wait(); }

// code for writer exit
synchronized (queue) {
    numWriter--;
    if (queue is not empty) {
        remove a single writer or a batch of readers from queue
        for each request removed do {
            numberWriter++ or numberReader++;
        synchronized (request) {
            request.okToGo = true;
            request.notify();
        }
    }
}
}

// code for reader entry
Reader r = new Reader(myname);
synchronized (queue) {
    if ((numWriter > 0) || !queue.isEmpty()) {
        r.okToGo = false;
        queue.add(r);
    } else {
        r.okToGo = true;
        numReader++;
    }
}
synchronized (r) { if (!r.okToGo) r.wait(); }

// code for reader exit
synchronized (queue) {
    numReader--;
    if (numReader > 0) exit(); // I am not the last reader
    if (queue is not empty) {
        remove a single writer or a batch of readers from queue
        for each request removed do {
            numWriter++ or numReader++;
        synchronized (request) {
            request.okToGo = true;
            request.notify();
        }
    }
}
}
```

## barber-shop problem

### Customer

```

Vector customQueue; // shared data among all threads

synchronized (numberChair) {
    if (numberChair > 0) numberChair--;
    else return; // leave // this releases monitor lock
}

// middle part (see next slide)

synchronized (numChair) {
    numberChair++;
}

```

### Customer – Middle part

```

// middle part
Customer myself = new Customer(mynname);
myself.done = false;
synchronized (customQueue) {
    customQueue.add(myself); // can also simulate chair here
    customQueue.notify();
}

synchronized (myself) {
    if (!myself.done) myself.wait(); // no need to use while
}

```

### Barber

```

while (true) {
    Customer current;
    synchronized (customerQueue) {
        if (customerQueue.isEmpty()) customerQueue.wait();
        // no need to use "while" here because only one barber
        current = customerQueue.removeFirst();
    }

    // hair cut the current customer

    synchronized (current) {
        current.done = true;
        // this flag helps to avoid needing nested monitor
        current.notify();
    }
}

```

## 03. CONSISTENCY CONDITIONS

### Formalizing a Parallel System

- **Operation:** A single invocation/response pair of a single method of a single shared object by a process
  - e being an operation
  - proc(e): The invoking process
  - obj(e): The object
  - inv(e): Invocation event (start time)
  - resp(e): Reply event (finish time)

wall clock time /  
physical time /  
real time

- Two invocation events are the same if invoker, invokee, parameters are the same – inv(p, read, X)
- Two response events are the same if invoker, invokee, response are the same – resp(p, read, X, 1)

- **consistency** → specifies what behaviour is allowed when a shared object is accessed by multiple processes
  - “consistent” = satisfies the specification
- **history H** → a sequence of invocations and responses ordered by wall clock time
  - for any invocation  $H$ , the corresponding response must be in  $H$
  - each execution of a parallel system corresponds to a history and vice versa
- **sequential** → an invocation is always *immediately* followed by its response
  - no interleaving (else is **concurrent**)

Sequential:

inv(p, read, X) resp(p, read, X, 0) inv(q, write, X, 1) resp(q, write, X, OK)

concurrent:

inv(p, read, X) inv(q, write, X, 1) resp(p, read, X, 0) resp(q, write, X, OK)

- a history  $H$  is **legal** → if all responses satisfies the sequential semantics of the data type

- **sequential semantics** → the semantics you would get if there is *only one* process accessing that data type
- possible for sequential history to not be legal
  - e.g. x=0, P1 writes 1 to x, P2 reads 0 from x (if it were the same thread it would have been the same value)
- process  $p$ 's **process subhistory** of  $H$ ,  $H|p$  → the subsequence of all events of  $p$ 
  - process subhistory is always sequential
- object  $o$ 's **object subhistory** of  $H$ ,  $H|o$  → the subsequence of all events of  $o$
- two histories are **equivalent** → if they have the exact same set of events
  - same events ⇒ implies all responses are the same
  - may be different ordering of events (only care about responses)
- **process/program order** → a partial order among all events
  - within the same process, process order is the same as execution order
  - no other additional orderings
- **sequential consistency** → equivalent to some legal sequential history that preserves process order
  - (*Lamport's definition*) results are same as in some sequential order & preserves program order

### Linearisability

- stronger than sequential consistency
- **external order** → a history  $H$  induces the “ $<$ ” partial order among operations
  - $o_1 < o_2 \iff$  the response of  $o_1$  appears in  $H$  before the invocation of  $o_2$ 
    - aka  $o_1$  finishes before  $o_2$  starts
  - preserves external order ⇒ preserves program order
- **linearisability** → sequentially consistent (with some legal sequential history  $S$ ) and  $S$  preserves the external order in  $H$ 
  - (*alternate definition*) The execution is equivalent to some execution such that each operation happens instantaneously at some point between the invocation and response
    - for every operation in the execution, you can find a linearisation point

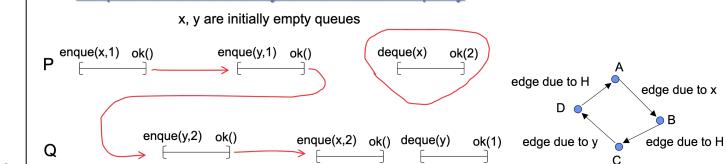
between the invocation and response event

- linearisable ⇒ sequentially consistent

### local property

- linearisability is a local property
  - $H$  is linearisable ⇔ for any object  $x$ ,  $H|x$  is linearisable
  - useful because you can reason about objects instead
- sequential consistency is not a local property
  - $H$  may not be sequentially consistent, but  $H|x$  and  $H|y$  can be sequentially consistent

### Sequential Consistency is Not Local Property

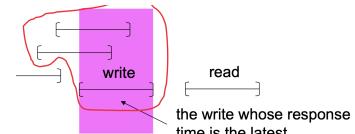


### proof: linearisability is a local property

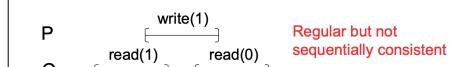
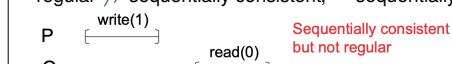
- using a directed graph: directed edge from  $o_1 \rightarrow o_2$  if
  - $o_1$  and  $o_2$  are on the same object  $x$  and  $o_1$  is before  $o_2$  when linearising  $H|x$
  - $o_1 \rightarrow o_2$  due to obj)
  - $o_1 < o_2$  in external order ( $o_1 \rightarrow o_2$  due to  $H$ )
- any topological sorting of the graph gives us a legal sequential history  $S$
- any cycle must be composed of
  - edges to some object  $x$  ( $\approx 1$  edge since  $H|x$  is equivalent  $S$  with a total order)
  - edges due to some  $H$  ( $\approx 1$  edge since partial order induced by  $H$  is transitive)
  - edges due to some object  $y$  ( $\approx 1$  edge)
  - edges due to some  $H$  ( $\approx 1$  edge)

### Consistency definitions for registers

- **register** → ADT: a single value that can be read and written
- register is **atomic** → if the implementation always ensures linearisability of the history
- register is **sequentially consistent** → if the implementation always ensures sequential consistency of the history
- register is **regular** → when a read
  - not overlap with any write, the read returns the value written by one of the most recent writes
  - overlaps with one or more writes, the read returns the value written by one of the most recent writes OR the value written by one of the overlapping writes



- register is **safe** → if the implementation always ensures that
  - when a read does not overlap with any write, it returns the value written by one of the most recent writes
  - when a read overlaps with one or more writes, it can return anything
- ! atomic ⇒ regular ⇒ safe
- regular ≠ sequentially consistent; sequentially consistent ≠ regular



## 4. MODELS & CLOCKS

- process can perform 3 kinds of atomic events/actions

- local computation**
- send a single message to a single process
- receive a single message from a single process

### communication model

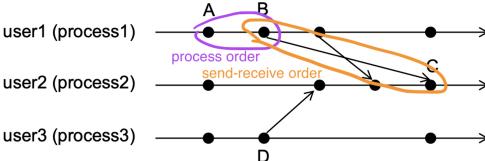
- point-to-point (to send to multiple processes: multiple send events)
- error-free, infinite buffer
- potentially out of order

### software clocks

- capture event ordering that are visible to users who do not have physical clocks
- allows a protocol to infer ordering among events

### visible orderings

- process order** → if A and B are on the same process, I can tell that A is before B
- send-receive order** → the send event must be before the receive event
- transitivity** → if  $A < B$  and  $B < C$ , then  $A < C$

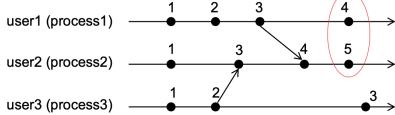


- happened-before relation** (denoted  $e \rightarrow f$ ) captures the ordering that is visible to users when there is no physical clock
  - partial order among events
- concurrent-with relation** (denoted  $e \parallel f$ ) if  $\neg(e \rightarrow f) \wedge \neg(f \rightarrow e)$

### Logical Clocks

- each event has a single integer as its logical clock value
- each process has a local counter  $C$
- protocol
  - increment  $C$  at each **local computation** and **send event**
  - send event**: attaches the logical clock value  $V$  to the message
  - receive event**:  $C = \max(C, V) + 1$

- if event  $s$  happens before event  $t \Rightarrow C_s < C_t$
- $C_s < C_t \not\Rightarrow s$  happens before  $t$

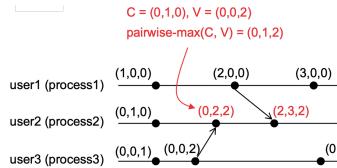


- for total order, extend with process number
  - $s.c$  denotes the value of  $c$  in state  $s$ ,  $s.p$  indicates the process it belongs to
  - the timestamp of any event is a tuple  $(s.c, s.p)$
  - $(s.c, s.p) < (t.c, t.p) \iff (s.c < t.c) \vee ((s.c = t.c) \wedge (s.p < t.p))$

### Vector Clocks

- event  $s$  happens-before event  $t \iff C_s < C_t$
- each event has a vector of  $n$  integers as its vector clock value
  - $v1 = v2$  if all  $n$  fields are the same
  - $v1 \leq v2$  if every field in  $v1$  is less than or equal to the corresponding field in  $v2$ 
    - e.g.  $(3,1,5) \leq (4,1,7)$
  - $v1 < v2$  if  $v1 \leq v2$  and  $v1 \neq v2$  ( $<$  is NOT a total order here)
- each process  $i$  has a local vector  $C$
- protocol
  - increment  $C[i]$  at each **local computation** and **send event**
    - $i^{th}$  entry is the principle entry
  - process  $i$  is the only process that can create new values for the  $i^{th}$  entry

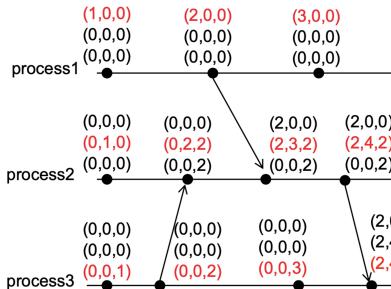
- send event**: vector clock value  $V$  is attached to the message
- receive event**:  $C = \text{pairwise-max}(C, V)$ ;  $C++$



### Matrix Clocks

- each event has 1 vector clock for each process
  - the  $i^{th}$  process on vector  $i$  is called process  $i$ 's principle vector
- protocol

- for **principal vector**  $C$  on process  $i$ ,
  - increment  $C[i]$  at each **local computation** and **send event**
  - send event**: all  $n$  vectors are attached to the message
  - receive event**:  $C = \text{pairwise-max}(C, V)$ ;  $C[i]++$ 
    - where  $V$  is the principle vector of the sender
- for **non-principal vector**  $C$  on process  $i$ 
  - receive event**:  $C = \text{pairwise-max}(C, V)$ ;

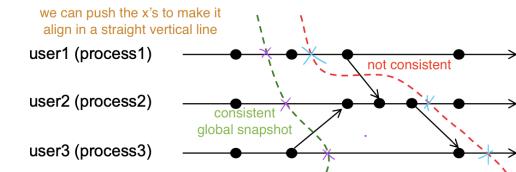


## 5. GLOBAL SNAPSHOT

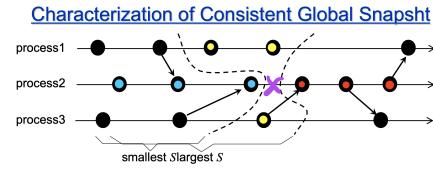
- captures a snapshot of local states on  $n$  processes such that the global snapshot could have happened sometime in the past (user cannot tell the difference)

### Consistent Snapshot

- consistent snapshot** → a snapshot of local states on  $n$  processes such that the global snapshot could have happened sometime in the past
  - can have outgoing ( $L \rightarrow R$ ) arrows, but can't have incoming ( $R \rightarrow L$ ) arrows



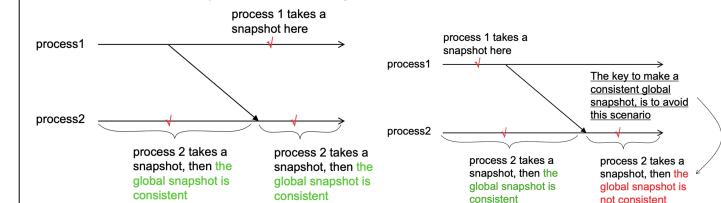
- global snapshot** → a set of events such that if  $e_2$  is in the set and  $e_1$  is before  $e_2$  in process order, then  $e_1$  must be in the set
  - a collection of local snapshots
- consistent local snapshot** → a global snapshot s.t. if  $e_2$  is in the set and  $e_1$  is before  $e_2$  in send-receive order, then  $e_1$  must be in the set
  - aka global snapshot + any receive event in the set has its corresponding send event
  - transitive relations implied



- There are other valid choices for  $S$
- A key characterization: 3 kinds of events
  - Must be in  $S$ : Those happened before blue
  - Cannot be  $S$ : Those happened after red
  - May or may not be  $S$ : Other events

### capturing a CGS

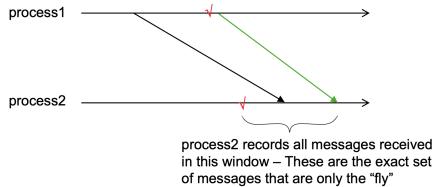
- communication model
  - no message loss
  - communication channels are unidirectional (model bidirectional channels as 2 unidirectional channels)
  - FIFO delivery on each channel
- ensuring FIFO
  - each process maintains a message number counter for each channel and stamps each message sent
  - receiver will only deliver messages in order



### Chandy & Lamport's Protocol

- each process is either
  - red - has taken local snapshot
  - white - has not taken local snapshot
- protocol initiated by a single process by turning itself from white to red

- once a process turns red, immediately send out Marker messages to all other processes
- upon receiving Marker, process turns red
- total  $n * (n - 1)$  Marker messages
- on-the-fly** messages: sent before sender's local snapshot, received after receiver's local snapshot



## 06. MESSAGE ORDERING

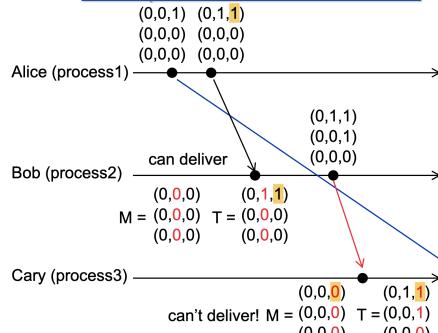
### Causal Order

- causal order** → if  $s_1$  happened before  $s_2$ , and  $r_1$  and  $r_2$  are on the same process, then  $r_1$  must be before  $r_2$ 
  - $s_1 \rightarrow s_2 \Rightarrow \neg(r_2 \prec r_1)$
- FIFO** → any 2 messages from process  $P_i$  to  $P_j$  are received in the same order as they are sent
  - $s_i \prec s_j \Rightarrow \neg(r_j \prec r_i)$
- $e \prec f$  denotes  $e$  occurred before  $f$  in the same process
- $s_i \rightsquigarrow r_i$  denotes  $s_i$  is the send event corresponding to receive event  $r_i$

### protocol: ensure causal order

- each process maintains a  $n$  by  $n$  matrix  $M$ 
  - $M[i, j] =$  # of messages sent from  $i$  to  $j$ , as known by local (current) process
- when process  $i$  sends a message to process  $j$ ,
  - on process  $i$ :  $M[i, j]++$
  - piggyback  $M$  on the message
- when process  $j$  (with local matrix  $T$ ) receives a message from process  $i$  with matrix  $T$  piggybacked,
  - set  $M = \text{pairwise-max}(M, T)$  if  $\begin{cases} T[k, j] \leq M[k, j] & \text{for all } k \neq i \\ T[i, j] = M[i, j] + 1 & \end{cases}$
  - intuition:  $M[i, j]$  on process  $j$  takes on consecutive values
  - if the entry is  $> 1$  larger than the local entry, it means that there is another message in propagation
    - we only care about column  $j$  -  $[i, j]$  should have a difference of 1
  - else, delay the message

### Example Run for the Protocol



- $M$  never decreases!
- for broadcast messages, same protocol (modelled as  $n$  point-to-point messages)

### Total Ordering of Broadcast Messages

- broadcast** → sent to all (including the sender itself)
- total ordering** → all messages delivered to all processes in exactly the same order (aka **atomic broadcast**)
  - i.e. if every message is assigned a number, the number has to be consistent across all users
  - total ordering only applies to broadcast messages
- total ordering  $\neq$  causal ordering
  - causal ordering  $\neq$  total ordering

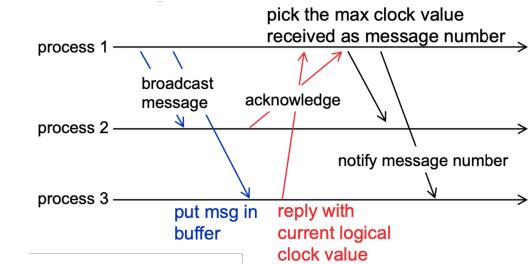
### Coordinator protocol

- a special process is assigned as the **coordinator**
- to broadcast a message:
  - send a message to the coordinator
  - coordinator assigns a seq # to the message

- coordinator forwards the message to all processes with the sequence number
  - messages delivered according to seq# order
- problem: coordinator has too much control

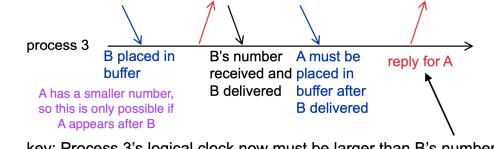
### Skeen's Algorithm

- each process maintains
  - logical clock
  - message buffer for undelivered messages
- a message in the buffer is delivered/removed if
  - all messages in the buffer have been assigned numbers
  - this message has the smallest number
- protocol
  - process broadcasts a message
  - receiving processes put the message in buffer and reply (ACK) with their current logical clock value
  - sending process picks the max clock value as message number and notifies (broadcasts) message number



### correctness proof

- claim: all messages will be assigned message numbers
- claim: all messages will be delivered
- claim: if message  $A$  has number smaller than  $B$ , then  $B$  is delivered after  $A$



- Suppose  $A$  is delivered on process 3 after  $B$ .
- Then  $A$  must have been placed in buffer after  $B$  was delivered.
- $A$  must have a number larger than  $B$  – Contradiction.

## 07. LEADER ELECTION

- leader election trivially solves mutual exclusion and total order broadcast

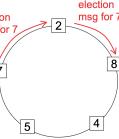
### Leader Election on a Ring

#### anonymous ring

- anonymous ring** → no unique identifiers
  - all must run the same algorithm (otherwise the algo itself is the unique ID)
  - node can only send messages to its neighbours
- leader election impossible using deterministic algorithms
  - same: initial state, state at each step, final state, algorithm on each node

#### Chang-Roberts Algorithm

- idea: largest ID is the leader
  - each node has a unique ID
  - nodes only send messages clockwise
- protocol:
  - node sends an election message with its own ID clockwise
  - a node forwards an election message if the ID is larger than its own ID
    - otherwise discard
  - a node becomes a leader if it sees its own election message



- performance**
  - best case:  $2n - 1$  messages
  - worst case:  $\frac{n(n+1)}{2}$

#### average case: $O(n \log n)$

- taken over all possible orderings of nodes, each ordering having the same probability
  - $(n - 1)!$  total orderings of the IDs
  - let  $x_k$  = number of messages caused by node  $k$ 's election message
  - we want to find  $E[x_k]$  for all  $k$  from 1 to  $n$
- by linearity of expectation,  $E[\sum x_k] = \sum E[x_k]$
- $E[x_k] = \sum_{i=1}^k (i \cdot Pr[x_k = i])$
- $Pr[x_k = 1] = Pr[\text{next node has larger ID than } k] = \frac{n-k}{n-1}$
- $E[x_k] \leq E[y] = \frac{1}{p} = \frac{n-1}{n-k}$  where  $y$  is a random variable denoting the number of lottery tickets we need to buy until winning the lottery (of probability  $p$ ) for the first time
- $\sum_{k=1}^n E[X_k] = n + \sum_{k=1}^{n-1} E[X_k] < n + \sum_{k=1}^{n-1} \frac{n-1}{n-k} = n + (n-1) \sum_{k=1}^{n-1} \frac{1}{k} = n + (n-1)O(\log n) = O(n \log n)$

### Leader Election on a General Graph

#### $n$ is known

- complete graph
  - each node sends its ID to all other nodes
  - wait until you receive  $n$  IDs - biggest ID wins
- any connected graph
  - flood your ID to all other nodes
  - ask neighbours to recursively forward ID to other neighbours
  - wait until you receive  $n$  IDs - biggest ID wins

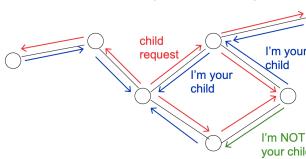
#### $n$ is unknown

- complete graph:  $n$  must be known
- any connected graph: use an auxiliary protocol to calculate  $n$ 
  - initiated by any node that wants to know  $n$
  - establish a spanning tree starting from the initiator

#### spanning tree to calculate $n$

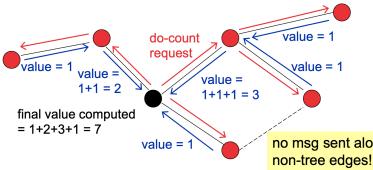
- goal: each node knows its parents and children
- it's fine if multiple nodes initiate this process - you're just counting  $n$
- 1. construct spanning tree:** initiator sends child requests
  - if node has not ACKed, sends ACK and send child requests to its neighbours

- if node has already ACKed, reject the request



#### 2. count nodes: initiator sends do-count request

- recursive: children will respond with  $1 + \text{the number of children they have}$



## 08. DISTRIBUTED CONSENSUS

### goal

- termination** → all nodes (that have not failed) eventually decide
- agreement** → all nodes that decide should decide on the same value
  - if a node agrees then crashes, still satisfy the agreement
- validity** → if all nodes have the same initial input, that value should be the only possible decision value
  - otherwise can decide on anything (but still satisfy Agreement)

### v0. no failures

- trivial

### v1. Node crash failures

#### model

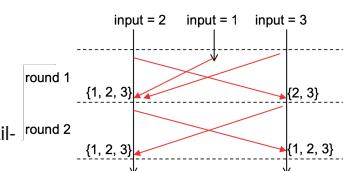
- failure model
  - ✗ node **crash failures**
    - node runs the algo, but stops executing at some arbitrary point in time
    - ✓ communication channels are reliable
- timing model
  - ✓ communication channels are **synchronous**
    - message delay has a *known* upper bound ✗
    - node processing delay has a *known* upper bound *y* (given as an input)

### synchronous systems and rounds

- each round, every process:
  - does some local computation (local processing delay)
  - sends one message to every other process (message propagation delay)
  - receives one message from every other process
- round duration** = clock error + msg propagation delay + local processing delay
  - assume each process has a physical clock with some bounded clock error
- start a new round every ROUND\_DURATION seconds
  - according to local clock
- each message has a round number attached to it
  - a message sent in a round must be received by the end of that round on the receiver
  - if receiver receives a message before it even starts the round: buffer the message until round starts

### protocol

- protocol: at each round, keep forwarding the values received
  - each process sends its input to all others
  - pick the min (or max)



#### $f + 1$ rounds needed for $f$ failures

- 1 round will be failure-free
- lower bound  $\Omega(f)$

#### $f$ must be an input to the protocol

- user indicates maximum number of failures to be tolerated

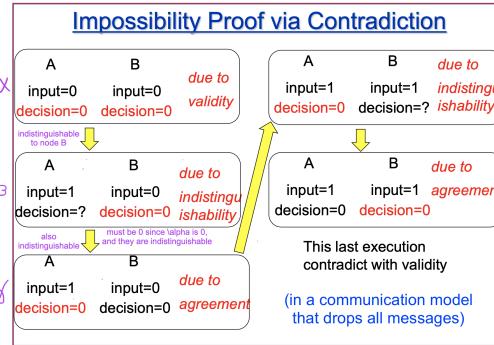
### agreement proof

- with  $f + 1$  rounds and  $f$  failures, there must be at least one good round
  - claim: At the end of any good round  $r$ , all non-faulty nodes during round  $r$  have the same  $S$
  - claim: Suppose  $r$  is a good round. The value of  $S$  on any non-faulty nodes does not change during any round after  $r$ .
  - claim: All nonfaulty processes at round  $f+1$  will have the same  $S$

## v2. Link failures (Coordinated Attack)

### model

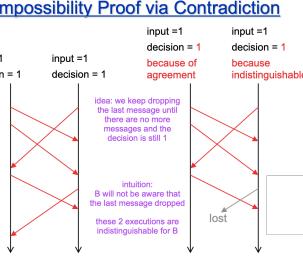
- failure model
  - ✓ nodes do not fail
  - ✗ communication channels may fail - drop arbitrary (unbounded) # of msgs
- timing model
  - ✓ synchronous
- goal: termination/agreement/validity
  - impossible to achieve these using a deterministic algorithm
    - cos communication channel can drop all messages
    - execution  $\alpha$  is **indistinguishable** from execution  $\beta \rightarrow$  if the node sees the same messages and inputs in both execution



### v2.1. Weakened goal

still impossible using deterministic algo

- if all nodes start with 0, the decision = 0
- if all nodes start with 1 and no message is lost during execution, decision = 1
- otherwise, any consensus

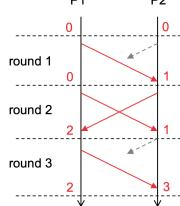


## v2.2. Limited disagreement (small error)

- goal
  - termination
  - agreement: all nodes decide on the same value with probability  $1 - \epsilon$
  - validity
    - if all nodes start with 0, decision = 0
    - if all nodes start with 1 and no message is lost throughout, decision = 1
    - else: anything
- adversary maximises error probability
  - ✓ set inputs of the processes
  - ✓ cause message losses
  - ✗ does not know the outcome of any randomisation

### randomised algorithm

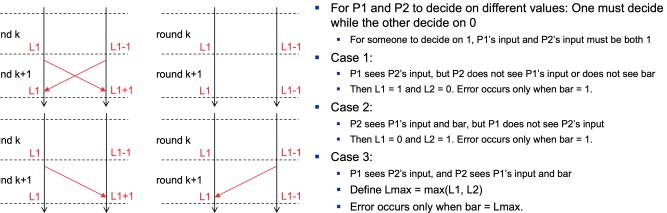
- 2 processes (P1, P2), predetermined number ( $r$ ) of rounds
  - adversary determines which messages are lost before seeing random choices
- protocol
  - P1 picks a random integer  $bar \in [1..r]$
  - each P1, P2 maintains a **level** variable L1, L2
    - L1 and L2 differ by at most 1
  - each round: send each other messages
    - attach input,  $bar$  and **level** to each message
    - P1: set L1 = L2+1
      - (symmetric) same for P2
    - ! L1/L2 never decreases!



- decision rule: after  $r$  rounds, P1/P2 decide on 1  $\iff$ 
  - it knows its input and the other process input are both 1
  - it knows bar (always true for P1)
  - its **level**  $\geq bar$

- error probability  $\epsilon = \frac{1}{r}$

### Inductive Proof for the Lemma



### When does error occur?

- For P1 and P2 to decide on different values: One must decide on 1 while the other decide on 0
  - For someone to decide on 1, P1's input and P2's input must be both 1
  - Case 1:
    - P1 sees P2's input, but P2 does not see P1's input or does not see bar
      - Then L1 = 1 and L2 = 0. Error occurs only when bar = 1.
  - Case 2:
    - P2 sees P1's input and bar, but P1 does not see P2's input
      - Then L1 = 0 and L2 = 1. Error occurs only when bar = 1.
  - Case 3:
    - P1 sees P2's input, and P2 sees P1's input and bar
      - Define Lmax = max(L1, L2)
    - Error occurs only when bar = Lmax.

## v3. Node crash failures + Asynchronous

### model

- failure model
  - ✓ reliable channel
  - ✗ node crash failures
- timing model
  - asynchronous**  $\rightarrow$  process delay and message delay are *finite but unbounded*
  - can no longer define a round
- goal: termination/agreement/validity
- FLP theorem** (Fischer, Lynch, Paterson)  $\rightarrow$  the distributed consensus problem under asynchronous communication model is impossible to solve even with a single node crash failure
  - fundamentally, because the protocol cannot accurately detect node failure

### formalisms of FLP theorem

- global state of a system = all process states + message system state
  - message system captures on-the-fly messages:
    - {(p, m)} message  $m$  on the fly to process  $p$
  - all messages are distinct
  - sending and receiving operations are adding and removing content from the message system, and changing process local state
- each step given a global state is fully described by p's receiving m
  - (p, m) is an event
    - events are inputs to the state machine, that cause state transitions
    - event  $e$  can be applied to the global state G if either  $m == null$  or  $(p, m)$  is in the message system
  - execution of any protocol is an *infinite sequence of events*
    - process fail = *finite* number of steps
- schedule  $\sigma$  is a sequence of events that captures the execution of a protocol
  - $G' = \sigma(G)$  means we apply  $\sigma$  to G to get G'
  - must make sure  $\sigma$  can be applied to G (aka G' is **reachable** from G if  $\exists \sigma$  such that  $G' = \sigma(G)$ )
- messages have unbounded but finite delay - every message is eventually delivered

## v4. Node Byzantine failures

### model

- failure model
  - ✓ reliable channels
  - ✗ **byzantine failures**  $\rightarrow$  nodes can behave unexpectedly and arbitrarily
    - must consider the worst case where nodes intentionally break the protocol
- timing model
  - ✓ synchronous
- goal: termination/agreement/validity for *non-faulty nodes* only
  - agreement: bad nodes can't be forced to decide

- validity:** if all the good nodes have the same initial input, that value should be the only possible decision value

- byzantine consensus threshold** for  $n$  processes,  $f$  possible byzantine failures
  - if  $n \leq f$ , then the byzantine consensus problem cannot be solved

### protocol for $n \geq 4f + 1$

- intuition
  - rotating coordinator:* process  $i$  is the coordinator for phase  $i$
  - coordinator sends a proposal to all processes
  - a phase is a **deciding phase** if the coordinator is nonfaulty
    - agreement after a phase with a good coordinator
    - if the coordinator is non faulty, all processes see the proposal
- protocol
  - $f + 1$  phases (cos at most  $f$  bad nodes)
    - each phase is 2 rounds
    - set local value = input
    - each phase
      - round 1: all-to-all broadcast
        - send your local value to all processes
      - round 2: coordinator round
        - set proposal to majority ( $> n/2$ )
        - send the proposal to everyone
      - decide whether to listen to the coordinator
        - if overwhelming majority ( $> \frac{n}{2} + f$ ), set local value to majority
        - else, set local value = coordinator's proposal

### proof

- Lemma 1: if all good processes have the same local value at the beginning of a given phase, then this remains true at the end of a phase
  - since we have  $n - f$  good processes and  $n - f > \frac{n}{2} + f$ , they will be the overwhelming majority
- Lemma 2: if the coordinator in a phase is good, agreement will be achieved at the end of that phase
  - case 1: coordinator sees (and proposes) a majority =  $x$ 
    - then  $x$  appears ( $>n/2$ ) times, of which  $>n/2-f$  must be from good processes
      - thus other processes will see  $x$  ( $>n/2-f$ ) times
      - so no other value can be overwhelming majority
  - case 2: coordinator receives equal distribution
    - then it's impossible for anyone to see an overwhelming majority
    - for coordinator, no value  $x$  appears ( $>n/2$ ) times
    - thus for other processes, impossible for any  $y \neq x$  to appear ( $>n/2+f$ ) times
- termination: after  $f+1$  phases
- validity: follows from lemma 1
- agreement:
  - with  $f+1$  phases, at least one is a deciding phase
  - by Lemma 2, all good processes will agree at deciding phase
  - by Lemma 1, after a deciding phase, additional phases will not disrupt the agreement achieved

### Summary

Failure Model and Timing Model	Consensus Protocol
Ver 0: No node or link failures	Trivial – all-to-all broadcast
Ver 1: Node crash failures; Channels are reliable; Synchronous;	( $f+1$ )-round protocol can tolerate $f$ crash failures
Ver 2: No node failures; Channels may drop messages (the coordinated attack problem)	Impossible without error
Ver 3: Node crash failures; Channels are reliable; Asynchronous;	Randomized algorithm with $1/r$ error prob
Ver 4: Node Byzantine failures; Channels are reliable; Synchronous; (the Byzantine Generals problem)	Impossible (the FLP theorem)
Ver 5: Node Byzantine failures; Channels are unreliable; Asynchronous;	If $n \leq 3f$ , impossible.
Ver 6: Node Byzantine failures; Channels are reliable; Synchronous; (the Byzantine Generals problem)	If $n \geq 4f + 1$ , we have a $(2f+2)$ -round protocol.
	How about $3f+1 \leq n \leq 4f$ ?

## 10. SELF-STABILISATION

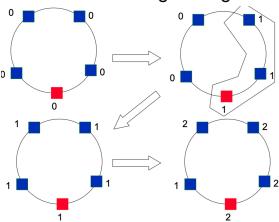
- state of a distributed system is either legal or illegal
  - based on application semantics
- self-stabilising** → if
  - starting from any (legal or illegal) state, the protocol will eventually reach a legal state if there are no more faults
  - once in a legal state, it will only transit to a legal state unless there are faults
  - typically runs in the background and never stops

### Rotating Privilege Problem

- a ring of  $n$  processes
  - each process can only communicate with neighbours
- at any time, only one node may have the privilege
- privilege is like a token

#### algorithm

- each process  $i$  has local integer variable  $V_i$  where  $0 \leq V_i \leq k$ ,  $k \geq n$
- one red process and some blue processes (allocate before running the algorithm)
- red process**
  - retrieve value  $L$  of clockwise neighbour
  - if  $V == L$ : (has privilege)
    - increment  $V++$  with mod  $k$
- blue process**
  - retrieve value  $L$  of clockwise neighbour
  - if  $V \neq L$ : (has privilege)
    - set  $V = L$



#### legal states

- Lemma: there are only 2 legal states
  - all  $n$  values same ⇒ red process privilege
  - only 2 different values forming 2 consecutive bands, with one band starting from the red process ⇒ blue process privilege

### Self-Stabilising Spanning Tree

- given  $n$  processes connected by an undirected graph and one special process P1, construct a spanning tree rooted at P1
- each process maintains 2 variables: **parent** and **dist** (to root,  $\geq 0$ )
  - faults: wrong value of these variables

#### algorithm

- P1 repeatedly executes 'dist=0; parent=null;'
- all other nodes periodically execute
  - retrieve dist from all neighbours
    - set own  $dist = 1 + \min(\text{neighbour dists})$
  - set own parent = neighbour with smallest dist
    - break tie: based on ID (e.g. smaller ID)

#### proof

- phase** → minimum time period where each process has executed its code at least once (can be more than once)
- assume that the topology doesn't change
  - i.e. no additional faults - we only need to reason about self-stabilising algos if there are no additional faults
- let
  - $A_i$  be the **level** of process  $i$  (length of the shortest path from  $i$  to root)
    - $A_i$  will not change
  - $dist_i$  be the value of dist on  $i$ 
    - $dist_i$  may change
- properties of levels of the nodes
  - a node at level  $X$  has at least one neighbour in level  $X - 1$

- if there is a neighbour with level  $< X - 1$ , then the node can have a smaller level  $< X$
- a node at level  $X$  can only have neighbours in level  $X - 1, X, X + 1$ 
  - for  $X-1$ : same proof as (1)
  - if there is a neighbour with  $> X+1$ , then that node can be  $X+1$  by going through the node instead
- lemma: at the end of phase  $r$ ,
  - any process  $i$  whose  $A_i \leq r - 1$ , has  $dist_i = A_i$ 
    - if level is  $< r$ , the distance value must become correct (i.e. dist=level)
  - any process  $i$  whose  $A_i \geq r$ , has  $dist_i \geq r$ 
    - for the remaining processes, the distance value is  $\geq r$  and may still be incorrect
- proof:** by induction
  - base: holds for  $r = 1$
  - assume lemma holds at phase  $r$  and consider phase  $r + 1$

#### common self-stabilisation proof technique

- Step 1: Prove that the  $t$  actions will not roll back what is already achieved so far by phase  $r$  (no backward move)
- Step 2: Prove that at some point, each node will achieve more (forward move)
- Step 3: Prove that the  $t$  actions will not roll back the effects of the forward move after the forward move happens (no backward move after the forward move)