Mutual Exclusion

Mutual Exclusion. No more than 1 process in critical section.
Progress. If 1 or more process wants to enter and if no one is in the critical section, then one of them can **eventually** enter. Process in CS will always assume that it will eventually exit the CS.
No starvation. If process wants to enter, it **eventually** can always enter. Implies progress; if an algo has

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

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

ReleaseCS(0) {                     ReleaseCS(1) {
   wantCS[0] = false;                 wantCS[1] = false;
}                                  }
```

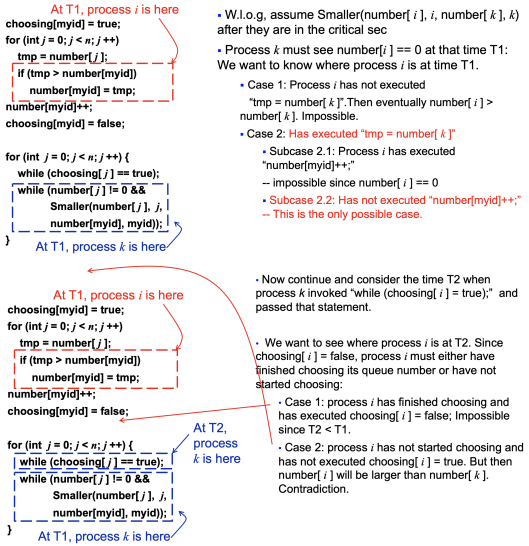
Petersons Algorithm. Works for 2 processes. If its my turn and I want to enter the CS, I can.
N-process mutual exclusion. To use Petersons algorithm for N processes, we can construct a binary tree where each node has either 2 or 0 children. Each node is a logical CS and runs an instance of Petersons algorithm. To enter CS, process need to enter all logical CS on path from itself to root.
Mutual Exclusion Proof. Proof by contradiction. Suppose both P0 and P1 are in CS when turn is 0. This means that turn = 1 was set before this. Furthermore, P0 would have wanted to enter CS. But how can this be since P1 sees wantCS[0] as false. Case for turn == 1 is symmetric.
Progress Proof. Consider value of turn when both P0 and P1 are waiting. This means that wantCS must be true for both. When turn == 0, P0 enters. Symmetric for turn == 1. Need to proof that nobody in CS and 1 process wants to enter and nobody in CS and 2 processes wants to enter.
No starvation Proof. Show what happens when both are waiting. If P0 is stuck, this means either P1 is stuck or in CS.
P1 in CS. P1 in CS then it will eventually exit and update wantCS[1] to be false. If P1 enters again immediately, then set turn to 0 and causes P0 to fall through. Symmetric for when P0 in CS and P1 waiting.
Lamport's Bakery Algorithm. For n processes, get a queue number and get served when all people with lower numbers have been served. Breaking ties with id. 2 shared arrays to indicate whether a process is choosing and the number obtained by a process.

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

// a utility function   for breaking using id.
boolean Smaller(int number1, int id1, int number2, int id2) {
   if (number1 < number 2) return true;
   if (number1 == number2) {
        if (id1 < id2) return true; else return false;
   }
   if (number 1 > number2) return false;
}
```

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

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

Progress Proof. Suppose every process wants to enter but no one can. Each process guaranteed a queue number (this step is non-blocking). Suppose process i has the smallest queue number. It can be blocked if another process j is choosing but this will eventually be set to false and process j would get block. This means i can make progress since its the smallest (contradiction). Another case of not entering is because i queue number is bigger but impossible since its already the smallest.
Mutual exclusion proof.

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



- W.l.o.g, assume Smaller(number[i], i, number[k], k) after they are in the critical sec
- Process k must see number[i] == 0 at that time T1: We want to know where process i is at time T1.
  - Case 1: Process i has not executed "tmp = number[k]".Then eventually number[i] > number[k]. Impossible.
  - Case 2: Has executed "tmp = number[k]"
    - Subcase 2.1: Process i has executed "number[myid]++;" -- impossible since number[i] == 0
    - Subcase 2.2: Has not executed "number[myid]++;" -- This is the only possible case.
- 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.

Hardware solutions. Disable interrupts and special machine-level instructions.
Disable interrupts. Don't allow context switch in CS.
Special machine level instructions.

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

Executed atomically

In CPU design, want all instructions to be roughly the same complexity so that pipelines wont have bubbles. Degrades performance as well.
Dekker's algorithm.

Dekker's algorithm – Mutual Exclusion

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
CRITICAL SECTION                                CRITICAL SECTION
turn = 1                                        turn = 0
wantCS[0] = false                               wantCS[1] = false
```

Prove by contradiction.
Case 1: turn = 0 when P0 and P1 are in critical section.
Then P1 last executed "turn = 0" after P0 last executed "turn = 1". Hence P1 must have seen turn = 0 when it checked turn's value. P1 must have seen wantCS[0] = false when it checks wantCS[0]'s value, since otherwise with turn = 0, P1 will get stuck and can never get into the critical section. Hence P1 must have executed "while (wantCS[0])" before P0 executes its first statement. Then P0 will see wantCS[1] being true, and will not pass its outer while loop and will not enter the critical section.

Dekker's algorithm – Progress

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
```

Prove by contradiction.
Case 1: turn = 0 when neither of them can enter critical section (will the value of turn change?).
On P1: wantCS[1] will become and remain false since turn = 0.
So P0 will enter critical section after wantCS[1] becomes false. Contradiction.

Case 2: turn = 1 when neither of them can enter critical section. Symmetric…

Dekker's algorithm – No starvation

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
```

Prove by contradiction: Suppose P0 (the case for P1 is similar) tries to enter CS and is blocked forever starting from time T1. P0 never modifies turn after T1, while P1 can only set turn to be 0 (when it exits the critical section).
Case 1: turn is always 0 (i.e., 0 at T1 and later potentially set to 0 again by P1).
Case 2: turn is always 1 (i.e., 1 at T1 and never set to 0 by P1)
Case 3: turn is 1 at T1, and later 0 (set by P1) forever.
In any of these cases, the value of turn "stabilizes" eventually (i.e., will not flip-flop forever). Hence we only need to consider fixed value for turn.

Dekker's algorithm – No starvation

Case 1 and 3: Eventually turn = 0.
Let T2 (where T2 ≥ T1) be the time starting from which turn is always 0.

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
```

On P0: With P0 being blocked and with turn always being 0 since T2, wantCS[0] will remain true forever since some time T3.
There can be many choices for T3 -- we choose a T3 such that T3 ≥ T2.

Dekker's algorithm – No starvation

Case 1 and 3: Eventually turn = 0.
From previous slide: turn = 0 and wantCS[0] = true forever since T3.

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
```

We want to show that wantCS[1] = false forever since some time T4 ≥ T3.
We check where P1 is at time T3.
- Case 1: At T3, P1 is somewhere in the above code segment. Then P1 must reach "while (turn == 0)" some time after T3, and the claim must hold at that point of time.
- Case 2: At T3, P1 is not in the above code segment, but will enter the above code some time in the future. Proof is the same as the previous case.
- Case 3: At T3, P1 is not in the above code segment, and will never enter it either. Then either wantCS[1] is false at T3, or wantCS[1] will be changed to false when P1 releases CS at time T4. Furthermore, wantCS[1] will remain false after that.

Dekker's algorithm – No starvation

Case 1 and 3: Eventually turn = 0.
From previous slide: turn = 0 and wantCS[0] = true and wantCS[1] = false forever since T4.

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
```

We know that P0 is still blocked at time T4, where T4 ≥ T3 ≥ T2 ≥ T1.
Regardless of where P0 is at time T4, given that turn = 0 and wantCS[1] = false, P0 will no longer block. Contradiction.

We are done with Case 1 and 3, next move on to Case 2…

Case 2: turn is always 1 since T1.
Let T2 (where T2 < T1) to be the time when turn was last assigned a value of 1 --- by P0 or by initialization. Now the value of turn is always 1 since T2.
If T2 was initialization time…

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
CRITICAL SECTION                                CRITICAL SECTION
turn = 1;                                       turn = 0;
wantCS[0] = false;                              wantCS[1] = false;
```

We claim that P1 never ever sets wantCS[1] to be true, since initialization time.
- Note that wantCS[1] only becomes true when P1 tries to enter the critical section. If P1 ever tries to enter, then P1 must be able to eventually enter. Otherwise since P0 is forever blocked after T1, P1's not being able to enter would violate the progress property (which we already proved). If P1 enters, P1 will eventually exit and set turn to be 0, contradicting with turn always being 1 since initialization time.

Now since wantCS[1] is always false since initialization time, P0 would not block at time T1 (where T1 > T2). Contradiction.

Case 2: turn is always 1 since T1.
Let T2 (where T2 < T1) to be the time when turn is last assigned a value of 1 --- by P0 or by initialization.
If T2 was the last time when P0 releases CS and sets turn to be 1…

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
CRITICAL SECTION                                CRITICAL SECTION
turn = 1;                                       turn = 0;
wantCS[0] = false;                              wantCS[1] = false;
```

A, B, C

We want to find out where P1 can potentially be, at time T2…

At T2, P1 must not be at A:
- Otherwise P1 will later change turn to 0, which contracts with the fact that turn is always 1 after T2.

Case 2: turn is always 1 since T1.
Let T2 (where T2 < T1) to be the time when turn is last assigned a value of 1 --- by P0 or by initialization.
If T2 was the last time when P0 releases CS and sets turn to be 1…

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
CRITICAL SECTION                                CRITICAL SECTION
turn = 1;                                       turn = 0;
wantCS[0] = false;                              wantCS[1] = false;
```

A, B, C

At T2, P1 must not be at B: Prove by contradiction and assume P1 is at B at time T2.
- P0 and P1 are releasing CS at T2. Consider when they requested CS this round…
- Let S0 be when P0 passes "while(wantCS[1])": P0 sees wantCS[1] being false at time S0
- Let S1 be when P1 passes "while(wantCS[0])": P1 sees wantCS[0] being false at time S1
- If S0 ≤ S1 < T2: Between S0 and T2, wantCS[0] is always true. Contradict with wantCS[0] being false at S1.
- If S1 ≤ S0 < T2: Between S1 and T2, wantCS[1] is always true. Contradict with wantCS[1] being false at S0.

Case 2: turn is always 1 since T1.
Let T2 (where T2 < T1) to be the time when turn is last assigned a value of 1 --- by P0 or by initialization.
If T2 was the last time when P0 releases CS and sets turn to be 1…

```
P0                                              P1
wantCS[0] = true;                               wantCS[1] = true;
while (wantCS[1]) {                             while (wantCS[0]) {
  if (turn == 1) {                                if (turn == 0) {
    wantCS[0] = false;                              wantCS[1] = false;
    while (turn == 1) ;                             while (turn == 0) ;
    wantCS[0] = true;                               wantCS[1] = true;
  }}                                              }}
CRITICAL SECTION                                CRITICAL SECTION
turn = 1;                                       turn = 0;
wantCS[0] = false;                              wantCS[1] = false;
```

A, B, C

Hence at T2, P1 must be at C!
- In turn, at T2, wantCS[1] must be false.
- After T2, wantCS[1] will remain false forever: To see why, note that wantCS[1] only becomes true when P1 tries to enter the critical section. If P1 is trying to enter, then by the earlier progress property (which we already proved), either P0 or P1 will be able to enter. If P0 enters, it contradicts to P0 being blocked forever. If P1 enters, P1 will eventually exit and set turn to be 0, contradicting with turn always being 1.

Hence when P0 later (anytime after T2) tries to request CS, P0 will never block. This contradict with the fact that P0 is blocked forever starting from T1 where T1 > T2.

Synchronisation Problems

```
P():
    if (value == false) {
        add myself to queue
        and block;
    }
    value = false;

V():
    value = true;
    if (queue is not empty) {
        wake up one arbitrary
        process on the queue
    }
```

Executed atomically (e.g., interrupt disabled)

(if blocks, will context switch to some other process)

Executed atomically (e.g., interrupt disabled)

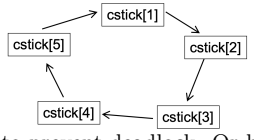
Does not use busy waiting. Internally has a boolean value that is set to true. And a queue of blocked processes that are initially empty (may not be FIFO). Possible to execute P() and V() out of order because they are kindof independent. Possible for a process to be stuck forever if there are other processes continually being added to the queue.

Semantics. Only 1 process is waken up, likely to be arbitrary but depends on implementation of queue.

Dining Philospher. 5 semaphores for chopsticks and philosphers are processes. To eat, must pick up left and right. When done, put down left and right. Neighbors cannot use the chopsticks while you're using them. Ensures that no 2 process fight with each other and proper use of resource. Semaphores are chopsticks.

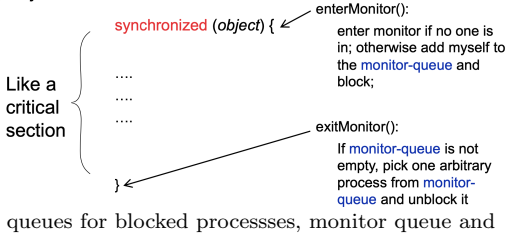
philosopher 1	philosopher 2	philosopher 3	philosopher 4	philosopher 5
cstick[1].P();	cstick[2].P();	cstick[3].P();	cstick[4].P();	cstick[5].P();
cstick[2].P();	cstick[3].P();	cstick[4].P();	cstick[5].P();	cstick[1].P();

- It is possible for all processes to block
  - Deadlock !



Avoid cycles to prevent deadlock. Or have total ordering.

Monitor.



2 queues for blocked processses, monitor queue and wait queue.

Object.wait. Add myself to wait-queue and exit monitor and then block myself. Done atomically.

Object.notify. If wait queue not empty, unblock 1 process from wait-queue.

Object.notifyAll. If wait-queue not empty, unblock all processes in wait-queue.

Hoare-style Monitor. If wait-queue is non-empty, unblock all processes in wait-queue.

Java-style Monitor. The process that called the notify continues execution. Placing the while loop means that it will keep checking the condition after it gets unblocked.

Conditional vs Loops. In monitors, a while loop will be checked again when that process is allowed to continue execution. For a conditional, it will fall through. The while loop has some nice guarantees in that it could allow a particular variable to hold some value throughout the execution.

Nested Monitors. Caling the wait on the inner monitor releases the lock on the inner object only. This means that wait() only releases the immediate lock.

```
synchronized (ObjA) {
    synchronized (ObjB) {
        ObjB.wait();
    }
}
```

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

```
synchronized (ObjA) {
    synchronized (ObjB) {
        ObjB.notify();
    }
}
```

- 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. Circular buffer with single producer and consumer. Producer put item at the end of buffer(if buffer not full) and consumer remove from the front (if buffer 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();
    }
}
```

Notification can be lost if no process is waiting.

Reader-writer. Multiple readers and writers accessing a file. Writer must have exclusive access but multiple readers can access the file at the same time. Flag ensures that notifications are never lost.

```
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.notifyAll();
    }
}

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

Starvation. To solve starvation on writers, maintain an explicit queue. Each monitor queue has at most 1 element.

```
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++;
    }
}
synchroized (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. Flag ensures that notifications are never lost.

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(myname);
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
}

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();
    }
}
```

PQR problem. Notify call meant for R. This problem will be solved if we replace while loop that checks numR == numP + numQ with an if instead since only 1 item is waiting.

```
int numP, numQ, numR; Object obj;
thread1:
    while (true) {
        print P;
        synchronized (obj) { numP++; obj.notify(); }
    }
thread2: similar as above
thread3:
    while (true) {
        synchronized (obj) {
            while (numR == numberP + numberQ) obj.wait();
        }
        print R;
        synchronized (obj) { numR++; }
    }
```

Consistency

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

Consistency. Specifies what behavior is allowed when a shared object is accesses by multiple processes.
History. Sequence of invocations and responses ordered by wall clock time. For any invocation in H, corresponding response is required to be in H. Each execution of a parallel system corresponds to a history and vice versa.
Sequential History. Any invocation is always immediately followed by its response. If there is interleaving, then its called concurrent.

Sequential:

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

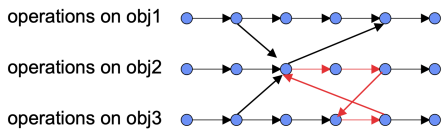
concurrent:

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

Legal Sequential History. All responses satisfies the sequential semantics of the data type. Possible for sequential history not to be legal.
Sequential Semantics. Semantics you would get if there is only 1 process accessing that data type.
Subhistory of a process. $H|p$ Subsequence of all events of the process. Thus a process subhistory is always sequential.
Subhistory of an object. $H|o$ Subsequence of all events of object.
Equivalence. 2 histories are equivalent if they have exactly the same set of events. Same events imply **all responses are the same**. Ordering can be different as the user only cares about responses.
Process Order. Partial order among all events. For any 2 events of the same **process**, process order is the same as execution order. No other additional orderings.
Sequential Consistency. A history that is equivalent to some legal sequential history that preserves process order. Focus on results only and require that program order be preserved.
External Order. o1 < o2 iff response of o1 appears in H before invocation of o2 (o1 finishes before o2 starts). Preserve external order implies preserving program order.
Linearisability Definition 1. Execution equivalent to some execution st each operation happens instantaneously at some point (linearisation point) between invocation and response.
Linearisability Definition 2. Equivalent to some legal seqeuential history S and S preserves external order in H. Partial order induced by H is a subset of partial order induced by S. In other words, if a response was before an invocation in the original history, it must still precede it in S.
Properties. Linearisable history must be sequentially consistent. But not necessarily the other way. Both definitions are the same.
Local Property. H is linearisable iff for any object x, $H|x$ is linearisable. Sequential consistency is not a local property.
Proof. Using definition 2. Construct a directed graph. Directed from o1 to o2 if either is satisfied:

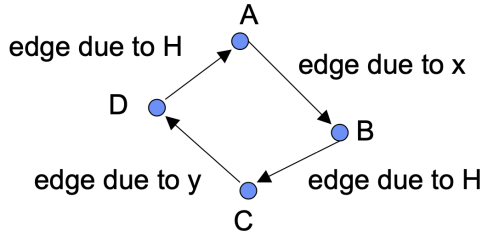
- o1 and o2 is on the same obj x and o1 is before o2 when linearising $H|x$ ($o1 \rightarrow o2$ due to object)
- o1 < o2 in external order (response of o1 appears before invocation of o2 in H), then $o1 \rightarrow o2$ due to H

Resulting graph is acyclic. So any topological sort of the graph gives a legal sequential history and the edges in the graph is a subset of the total order induced by S.
Legal + Legal = Legal. If $H|x$ and $H|y$ is linearisbale and legal for x, y respectively, interleaving the operations will not affect any behavior. Since all events in S are from H (including all invocation events and parameters). So values are already captured in the params of the invocation events.
Proof acyclic graph. Prove by contradiction.

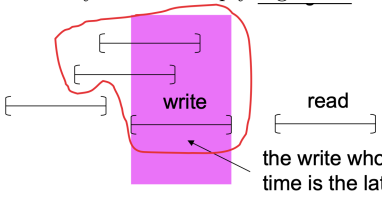


- To generalize, any cycle must be composed of:
  - Edges due to some object x, followed by
  - Edges due to H, followed by
  - Edges due to some object y, followed by
  - Edges due to H, followed by

Then $H|x$ is equivalent to a serial history S where S is a total order (we can directly draw an edge from start to end), similarly for partial order induced by H, because of transitivity.



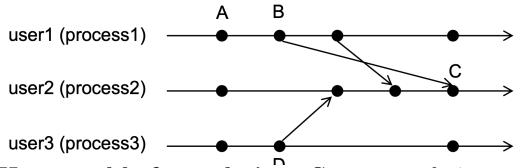
Then try to recreate the history from the cycle. From the image, its impossible for A to start before B finishes. The reason is that if B finishes before A, then B is before A in external order, which contradicts the edges from A to B. Contradiction here arises because we have to put the start of C to be before the end of D which is not possible.
Register. A kind of abstract data type - a single value that can be read and written.
Atomic register. Implementation ensures linearisability of history.
Regular register. When a read does not overlap with any write, the read returns the value written by 1 of the most recent writes. When a read overlaps, with ≥ 1, read returns value written by 1 of the most recent writes/1 of the overlapping writes. Atomic registers must be regular. **Does not imply sequential consistency**, similarly sequential consistency does not imply regular.



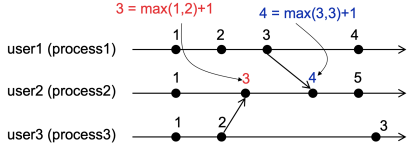
Safe registers. When a read does not overlap with any write, then it returns the value written by 1 of the most recent writes. When a read overlaps with ≥ 1 writes, return anything. Regular registers are safe. Does not imply sequential consistency and vice versa.

Clocks

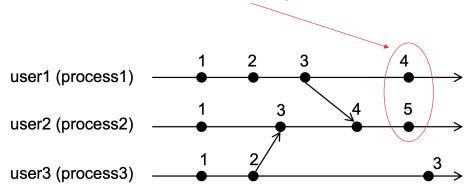
Process can receive 3 kinds of atomic actions/events. Local computation, send/receive single message to/from process.
Communication Model. Point-to-point. Error free, infinite buffer and potentially out of order.
Software Clocks. Capture event ordering that are visible to users who dont have physical clocks.
Visible ordering. Process order (A -¿ B), send-receive order (B -¿ C) and transitivity (A -¿ C)



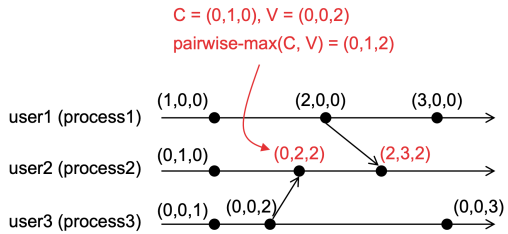
Happened-before relation. Captures ordering that is visible to users when there is no physical clock. Creates a partial order among events and preserves process order, send-receive order and transitivity.
Logical Clock. Each event has a single integer as its logical clokc value. Each process has a lock counter C. Increment C at each local computation and send event. When sending a message, logical clock value V is attached to the message. At each receive event C = max(C,V) + 1.



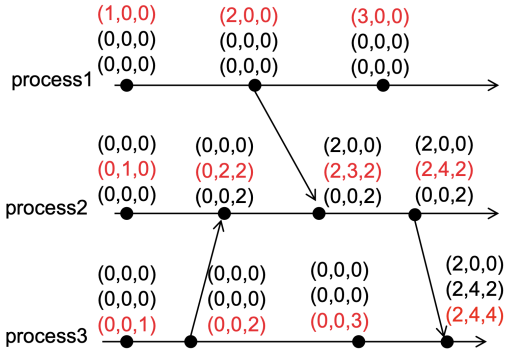
Properties. If event S happens before T, then the logical clock value of S is smaller than the logical clock value of T. The reverse may not be true.



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. Timestamp of any event is a tuple $(s.c, s.p)$. Then $(s.c, s.p) < (t.c, t.p) \Leftrightarrow (s.c < t.c) \vee ((s.c = t.c) \wedge (s.p < t.p))$
Vector clock. Event s happens before event t $\Leftrightarrow$ the vector clock value of s is smaller than that of t.
Partial order on clock value. Each event has a vector of n integers as its vector clock value. v1 = v2 if all the fields are the same. $v1 \leq v2$ if **every fields** in v1 $\leq$ to corresponding field in v2. $v1 < v2$ if $v1 \leq v2$ and $v1 \neq v2$. Partial order counter example: $[2, 0, 1], [1, 2, 0]$.
Protocol. Each process i has a local vector C. Increment $C[i]$ at each local computation and send event. When sending a message, vector clock V is attached to the message. At each receive event, C = pairwise-max(C,V). Then increase C[i] by 1.



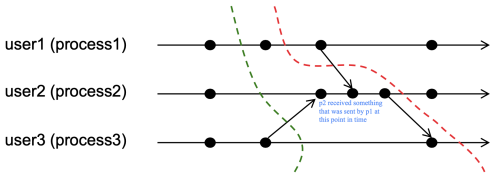
Properties Proof. Event s happened before t then vector clock value of s ¡ vector clock value of t. Enumerate all possible cases. Smaller than relation among vector clock values is transitive. Vector clock value of s ¡ vector clock value of t imples s happened before t. If s and t on the same process, done. If s is on p and t is on q, let VS be s vector clock and VT be t. $VS < VT \Rightarrow VS[p] \leq VT[p]$ Must be a sequence of message from p to q after s and before t.
Matrix Clock Protocol. Each event has n vector clocks, 1 for each process. $C[i]$ is the process i principle vector, where C is a matrix. Principle vector is the same as vector clock before. Non-principle vectors are just piggybacked on messages to update knowledge.
Protocol. Increment $C[i]$ at each local computation and send event. When sending a message, all n vectors are attached to the message. At each receive event, let V be the principle vector of the sender. C = pairwise-max(C,V). Then increase $C[i]$ by 1. For non-principle vector C on process i, suppose it corresponds to process j. At each receive event, let V be the vector corresponding to process j as in the received message. C = pairwise-max(C,V).



Global Snapshot

A snapshot of local states of n processes such that the global snapshot could have happened sometime in the past (users cannot tell the difference).

Consistent Snapshot. Snapshot of local states on n processes st global snapshot could have happened sometime in the past.

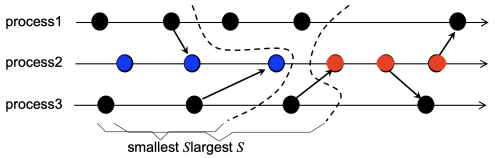


Green cut could have happened in the past if the processes had made progress faster/slower but not the red cut.

Global Snapshot. A set of events st if e2 is in theset and e1 is before e2 in **process order**, then e1 must be in the set.

Consistent Global Snapshot. A global snapshot st if e2 is in the set and e1 is before e2 in send-receive order, then e1 must be in the set. Transitive relation implied in global snapshot definition. In other words, if a receive event is in the set, corresponding send event must be in the set.

Existence. For any positive integer m, there is a consistent global snapshot S st $\begin{cases} e_i \in S & \text{for } i \leq m \\ e_i \notin S & \text{for } i > m \end{cases}$



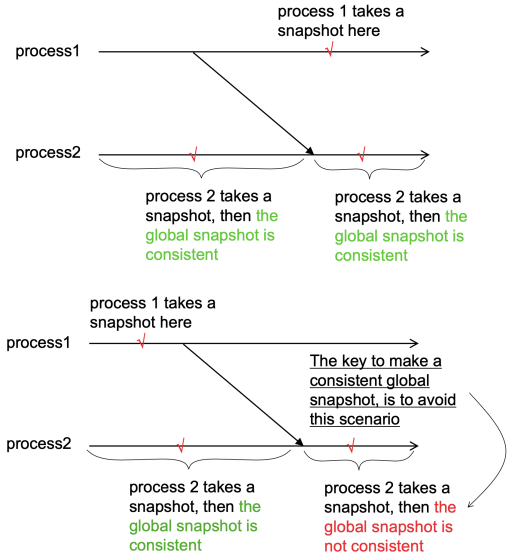
3 kinds of events:

- Blue events must be in S
- Red events cannot be in S
- Other events that may/may not be in S

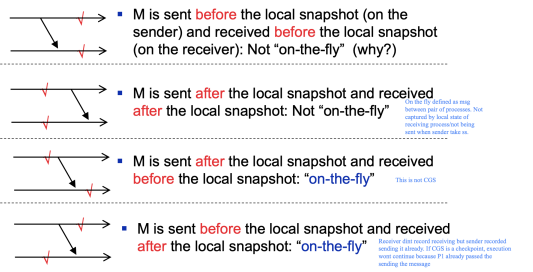
Capturing CGS. Look at communication model.

Communication Model. No message loss, unidirectional and FIFO.

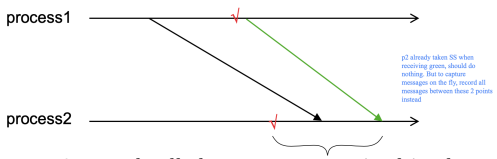
Ensuring FIFO. Each process maintains a message number counter for each channel and stamps each message sent. Receiver only deliver messages in order.



Chandy Lamport. Each process either red(has taken local SS) or white (hasn't taken). Initiated by single process by turning itself from white to red. Once a process turns red, send out market messages to all other processes. Upon receiving marker, process turns Red. There will be a total of $n(n+1)$ market messages. **On-the-fly.**



On the fly messages are messages that are sent before sender's local snapshot and received after receiver's local snapshot.



Process2 records all the messages received in the window from its local snapshot until it received the marker message. **Property.** If G and H are both CGS, then $G \cap H$ and $G \cup H$ are also CGS.

Proving tip. To show CGS, need to show that its a global snapshot (process order definition) then show that it is in CGS (send-receive order).

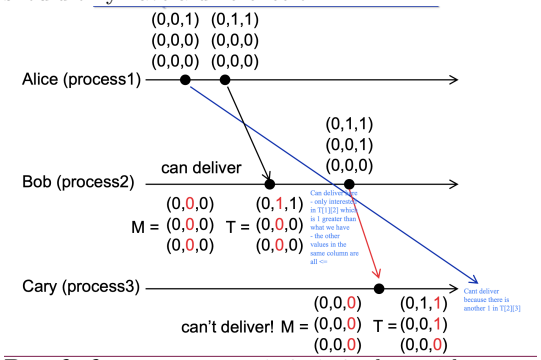
Message Ordering

Causal Order. If s1 happened before s2, and r1 and r2 are on the same process, then r1 must be before r2.

Protocol. Each process maintains a $n \times n$ matrix M. $M[i][j]$ is the number of messages sent from i to j as known by process i. If process i sends a message to process j, then on process i, $M[i][j]++$. Piggyback M on the message.

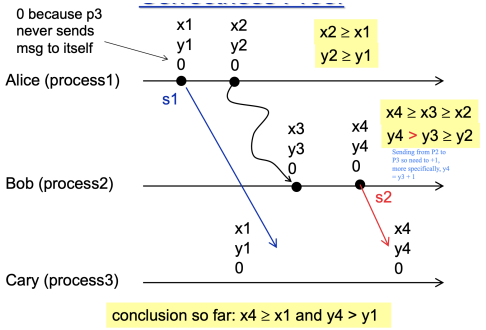
Receiver. Local matrix on j is M, piggybacked matrix from i is T.

Delivery conditions. Deliver the message and set M to be pairwise max of M, T if $\begin{cases} T[k][j] \leq M[k][j] & \text{for all} k \neq i \\ T[i][j] = M[i][j] + 1 \end{cases}$ otherwise delay message. $M[i][j]$ on process j takes consecutive values. If the entry value > 1 larger than local, then there is another message in propagaation. Column j should only have a difference of 1.



Proof of correctness. s1,s2,r1,r2 where s1 happen before s2. Need to prove both r1 and r2 eventually delivered and r2 not before r1.

Proof r2 not before r1.



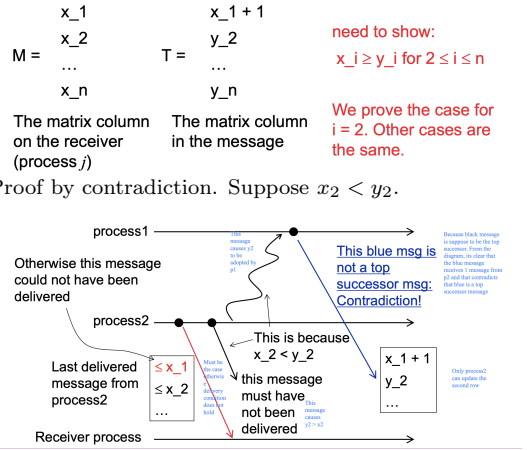
conclusion so far: x4 ≥ x1 and y4 > y1

Proof by contradiction. Assume red delivered before blue. Then after delivering red, 3rd column of M on process3 will contain x4,y4,0. Since **M never decreases**, blue cannot be delivered anymore.

Proof r1 r2 eventually delivered. At any given time, there must be 1 message that can be delivered to the process. Consider a receiver process j and its corresponding column in M, $x_1, \ldots, x_i, \ldots, x_n$

Successor message. Non-empty set of all undelivered messsages. For each sender i in the set, there is an undelivered message whose matrix T has $x_i + 1$ as a value in its column. Set of successor messages is non-empty, at most 1 message from any given sender and at most $n-1$ messages in total.

Top successor message. There must be a send event s that does not have any other send events that happened before s. Possible to have multiple such s events. Any top successor message can be delivered.



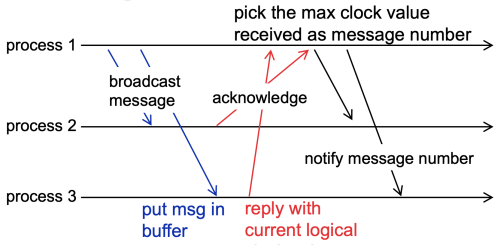
Proof by contradiction. Suppose $x_2 < y_2$.

Broadcast. Every message sent to everyone including itself. Modeled as n point-to-point messages.

Total Ordering. All messages delivered to all process in exactly the same order (atomic broadcast). Only applies to broadcast messages. Total order does not imply causal order and vice versa.

Coordinator. Special process. To broadcast message, send message to coordinator. Coordinator assigns sequence number to message and forwards the message to all processes with the sequence number. Messages delivered according to seqeunce number order. Problem is that coordinator has too much control.

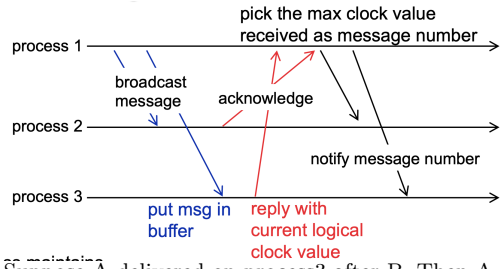
Skeens Algorithm. Each process maintain logical clock and message buffer for undelivered messages.



Buffer messages. Delivered/removed if all messages in the buffer have been assigned numbers or this message has the smallest number.

Correctness proof. All messages will be assigned message numbers and delivered.

If A has smaller number than B, then B delivere

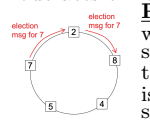


Suppose A delivered on process3 after B. Then A must have been place in buffer after B was delivered. This means that A must have a larger number than B - contradiction.

Leader Election

Trivially solves mutual exclusion and total order broadcast. **Anonymous ring.** No unique identifiers, node can only send messages to its neighbors. Impossible using deterministic algorithms. Same initial state, algo, step and final state.

Chang-Roberts Algorithm. Each node have unique id. Nodes only send messages clockwise. Each node acts on its own.



Protocol. Node send election messgae with its own id clockwise. Election message is forwarded if id in messages larger than own message. Otherwise message is discarded. Node becomes leader if it sees its own election message.

Performance. For distributed systems, communication is bottleneck. Performance described as message complexity. With n nodes on the ring, whats the total number of messages incurred by protocol.

Best/Worse case. Best case: $2n-1$. Worse cast: $\frac{n(n+1)}{2}$

Average performance. $O(n\ logn)$ Average is taken over all possible ordering of the nodes on the ring and each ordering has the same probability. Total ordering excluding isomorphic ones $(n-1)!$. x_k is the number of messages caused by election message from node k. Want to find $E[\sum x_k]$ for k summed from 1 to n. Then by linearity of expectation, $E[\sum x_k] = \sum E[x_k]$

Probability trick. $Pr[x=1] = \frac{n-k}{n-1}$. Since x_k is the number of messages caused by node k, total $n-1$ nodes other than k, of which $n-k$ nodes have id larger than k. Then $Pr[x_k = 2 | x_k > 1] = \frac{n-k}{n-2}$.

Simplify $p = \frac{n-k}{n-1}$ and upper bound everything by p. Let y be rv denoting number of tickets we buy until we win for the first time, assuming probability of winning is q, then $Pr[y=(i+1)|y>i] = q$. Then $E[y] = \frac{1}{q}$. So $E[x_k] \le E[y] = \frac{1}{q} = \frac{n-1}{n-k}$

$$\sum_{k=1}^{n} E[x_k] = n + \sum_{k=1}^{n-1} E[x_k]$$

$$< n + \sum_{k=1}^{n-1} \frac{n-1}{n-k}$$

$$= n + (n-1)\sum_{k=1}^{n-1} \frac{1}{k}$$

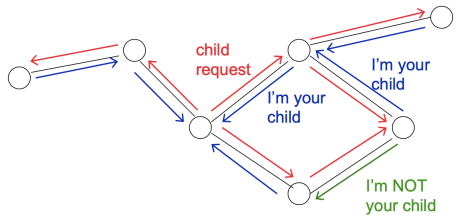
$$= n + (n-1)O(log\ n) = O(n\ log\ n)$$

General graph (known n). Complete graph or any connected graph.

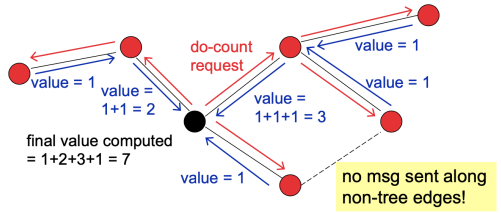
Complete graph. Each node sends its id to all other nodes. Wait until you receive n ids. Biggest id wins.

Any connected graph. Flood your id to all other nodes. Wait until you receive n ids. Biggest id wins.

Spanning Tree Construction to count n. No centralised coordinator, possible for multiple nodes to initiate this. Goal of the protocol is that each node knows its parent and children. Each node sends child request. If already a child of another node, reject request.



Counting nodes. Initiator sends a do-count request. Recursively, children will respond with 1 + number of children they have.



Spanning tree purpose. Broadcast (root doesnt need to send out n messages), aggregations, calculating maxmin.

Impossibility for unknown ring size. Modelling a randomised algorithm (a deterministic algorithm with 1 random input).

Proof. Assume there is an algorithm to solve the problem. Terminate with probability 1 implies that there at least exists some bit string input that terminates after r (constant) phases in ring 1. In ring 2, same bit string may be input for P1 and P2. Ring 1 and 2 indistinguishable for P1, so P1 declares leader after r phases in ring 2 as well. Similarly for P2. In ring 2, there will be 2 leaders, contradiction.

For known ring size. Each node has phase number initialised to 1. Each message sent has phase number attached.

Phase 1. Each node pick random id and run Chang-roberts. Winner proceed to next phase, losers only forward messages in future phases. A node know its the winner if it sees its own id after exactly n hops (ring size). Stop if there is a single winner. By definition there will be a single winner if protocol stops.

Proof of termination. Define the ith good phase to be the phase that results in ith loser. Need exactly $n-1$ good phase. Let r be the phase number. x_i be the number of phases between the ith and $i+1$th good phase. As $r \to \infty$, $Pr[x_i > \frac{r}{n-1}] \to 0$

Distributed Consensus

Set of nodes want to agree on something. Each node has an input.

Version 0. No failures. Everyone send its input to everyone else. Take the majority of the n values (with tie-breaking favoring 0). Decide and done.

Failure Model. Nodes can experience crash failure. Communication channels are reliable.

Synchronous timing model. Message delay has known upper bound x and node processing delay has a known upper bound y. Implies possibility of accurate failure detection.

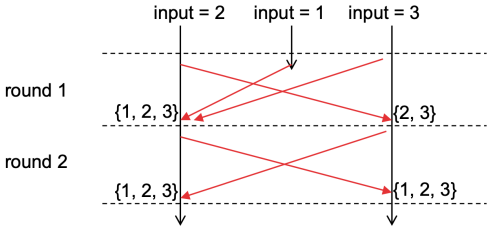
Goals. Termination - all nodes that have not failed eventually decides. **Agreement** - all nodes that decide should decide on the same value. **Validity** If all nodes have the same initial input, that value should be the only possible decision value. Otherwise nodes can decide on anything (except they still need to satisfy Agreement).

Version 1. Node crash failures and synchronous timing model.

Rounds. Under synchronous timing model, all processes proceed in inter-locked rounds. In each round, each process does some local computation, sends 1 message to every other process and receives 1 message from every other process.

Round implementation. Message propagation delay + local processing delay + clock error. Assume each process has a physical clock with some bounded clock error. Message sent in a round must be received by the end of that round on the receiver (may be earlier than that). Each message has a round number attached on it.

Protocol. Each process sends its input to all the others and pick min/max. If there are failures, process may send input to only a subset of the processes. Keep forwarding the values until all processes have received.



Need 1 extra round for each failure ($f+1$ rounds to tolerate f failures). f is input to protocol. With f crash failures, any consensus protocol will take $\Omega(f)$ rounds (lower bound).

Proof. Termination is obvious since there is no wait. Validity obvious since S will only have a single element.

Agreement proof. Prove that S is the same on all nodes when protocol terminates.

Nonfaulty. Nodes in round r that did not crash by the end of round r.

Good round. No node failure during that round. With $f+1$ rounds and f failures, at least 1 good round.

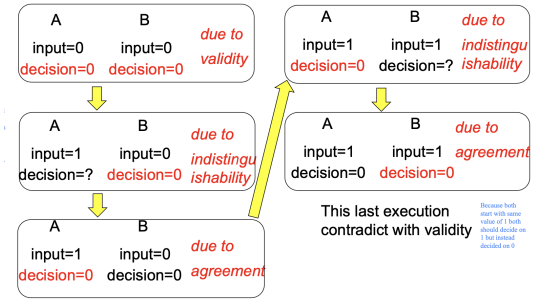
Claims.

- At end of any good round r, all non-faulty nodes during round r have the same S.
- Suppose r is a good round, and consider any round t after round r. During round t, value of S on any non-faulty nodes doesn't change.
- All non faulty processes at round $f+1$ have the same S.

Coordinated attack problem. Failure model nodes dont fail but communication channels might (may drop unbounded number of messages). Timing model is synchronous.

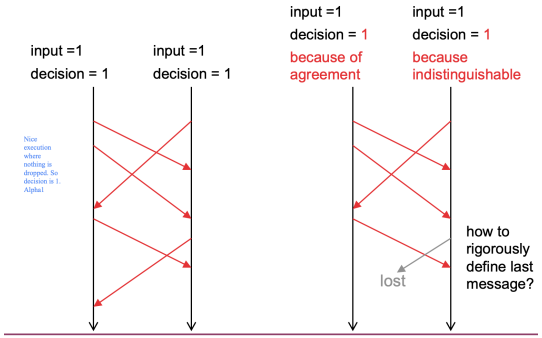
Achievability. Impossible to reach goal using deterministic algorithm since communication channel can drop all messages.

Indistinguishable. 2 executions are indistinguishable for some node if the node sees the same things (message and inputs) in both executions.



Contradiction arises beause with same inputs, should decide on 1 but decided on 0 instead. From the first execution to the next, channel drops all messages from B to A. So A cannot make a decision. Since B has the same things for the 2 executions, become indistinguishable.

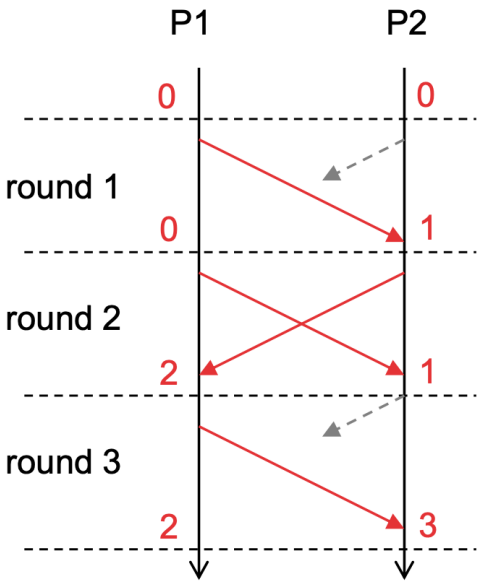
Weakened validity. Assuming input either 0/1. If all nodes start with 0, decision should be 0. If all nodes start with 1 and no message lost throughout execution decision should be 1. Otherwise allowed to decide on anything. Channel does not drop messages when all nodes start with 1. **Still impossible using deterministic algorithm**.



On the left is the nice execution where nothing is dropped, so decision is 1. On the right, if we drop the last message(sent time), there is no way of either process knowing. Indistinguishable for the right process since everything is the same. Keep repeating by dropping the last message for each execution until there are no more messages.

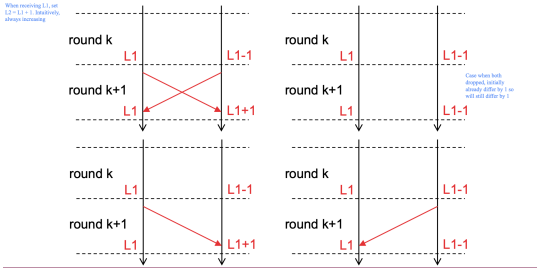
Adversay. Process can flip coins and adversary try to maximise P_e by setting inputs of process, causing message loss. Outcome of random flips unknown to adversary.

Randomised Algorithm. Consider 2 processes. Algorithm has predetermined number of rounds, r. Adversary determines which messages get lost before seeing random choise. P1 picks a random integer $bar \in [1, \dots r]$ at the start. Protocol allows P1 and P2 to each maintain a **level** variable (L1, L2) st level is influenced by adversary but L1, L2 differ by at most 1.



bar, input and current level attached on all messages. When receiving, update own level variable to be $level_{self} = level_{received} + 1$. **L1/L2** never decreases and differ by at most 1.

Inductive Proof for the Lemma



Decision rule. At the end of r rounds, P1 choose 1 iff P1 knows P1 and P2 inputs are 1 and L1 $\geq$ bar. Implies that P1 decide on 0 if doesnt see P2 input. P2 decides on 1 iff P2 knows P1 P2 inputs are both 1 and P2 knows bar and L2 $\geq$ bar. Implies that P2 decide on 0 if it doesnt see P1 input **or** doesnt see bar.

Error Occurance. For P1 and P2 to decide on different values, one must decide on 1 and the other decides on 0. Both inputs must be 1. All cases mutually exclusive and happens with probability $\frac{1}{r}$ 3 cases:

- P1 see P2 input but P2 dont see P1 input/bar. L1 = 1 and L2 = 0. Error occurs when bar = 1.
- P2 see P1 input and bar but P1 dont see P2 input. L1 = 0, L2 = 1. Error occurs only when bar = 1.
- P1 sees P2 input, P2 sees P1 input and bar. Then error occurs when bar = max of L1 and L2.

Correctness Proof. Termination is obvious since r rounds. Validity - if all rounds start with 0, decision always 0. If all nodes start with 1 and no messages lost throughout exection, decision should be 1. Since no message lost then L1 = L2 = r. Agreement happens with probability $1 - \frac{1}{r}$

Consensus with Node Crash - Async. Failure model same as v1. Asynchronous - process and message delay are finite but unbounded. Cannot define a round since if there is no message for a long time, unsure if sender failed or message delayed. Goals are still termination, agreement and validity.
FLP Theorem. Distributed consensus problem under async communication model is impossible to solve even with a single node crash failure. Fundamentally protocol is unable to accurately detect node failure.
Formalism. Goal is to abstract execution of any possible deterministic protocol. Each process has 2 variables, input $\in \{0, 1\}$ and decision $\in \{null, 0, 1\}$ decision is initially null and can be written exactly once.
Communication channel. Messages are on the fly.
Message system. Captures state of all communication channels. Is the set containing pairs of (p, m) where m is on the fly to process p. All messages are distinct.
Send. add a (dest, content) to the message system.
Receive. When invoked by process p. Remove some (p, content) from message system and return content **OR** leave message system unchanged and return null. No fifo ordering means no guarantee on message order. Not a special case of out-of-order. FLP holds when receive is blocking. Can use blocking to implement non-blocking.
Step. Takes system from 1 global state to another. By executing the following on process p. Receive a message m (can be null). Based on p local state and m, send an arbitrary but finite number of messages and change p local state to a new one.
Event. (p,m) describes p's receiving m. Events are input to state machine that cause state transitions. An event can be applied to global state G if either m is null or (p,m) in message system.
Execution. For any protocol, can be abstracted to be an **infinite** sequence of events. Each execution may be different. Can always make a protocol not to terminate. Each process must be able to handle null messages. Decisions are made when decision variable is set. Necessary to properly defined failed processes.
Schedule. σ is a sequence of events that captures execution of some protocol. σ can be applied to G if events can be applied to G in the order in σ. $G' = \sigma(G)$ means if we apply σ to G we will end up with G'. $\sigma(G)$ may or may not be applied to G. Schedule creates a total order on events.
Reachable. Given a consensus protocol A, a global state G2 is reachable from G1 if there is a schedule σ of A such that $G2 = \sigma(G1)$.
Async Systems. Processes have unbounded but finite delay. Nonfualty process takes infinite number of steps. A faulty one takes a finite number of steps. If we consider only finite sequences, cannot distinguish faulty and nonfaulty processes. **Messages have unbounded but finite delay**. Every message is eventually delivered. If there is a message (p,m) in the message system and p invokes receive multiple times, then the message system can only return null finite number of times. At most 1 faulty process.
Proof technique for FLP. Act as adversary to defeat consensus protocol. Scheduler can pick which messages to deliver and which process will take the next step (under constraints of async system). Goal is to prevent protocol from deciding.
Global states. G is **0-valent** if 0 is the only possible decision reachable from G. Processes in G may or may not yet decided on 0, but otherwise, they will eventually decide on 0. G is **1-valent** if 1 is the only possible decision reachable from G. G is **univalent** if G is either 0-valent or 1-valent. G is **bivalent** if its 0-valent and 1-valent.
Lemmas.

- For any protocol A, there exists a bivalent **initial** state. Prove by contradiction.
- Let $\sigma 1, \sigma 2$ be 2 schedules st the set of processes executing steps in $\sigma 1$ are disjoint from the set that executes $\sigma 2$. Then for any G that $\sigma 1, \sigma 2$ can both be applied, we have $\sigma 1(\sigma 2(G)) = \sigma 2(\sigma 1(G))$. Prove by induction.
- Let G be a global state, and $e = (p, m)$ is an event that can be applied to G. Let W be the set of global states that is reachable from G without applying e, then e can be applied to any state in W.
- Let G be a bivalent state, and $e = (p, m)$ is any event that can be applied to G. Let W be the set of global states that is reachable from G (G is in W) without applying e, and $V = e(W)$ to be the set of global states by applied e to the states in W. Then V contains a bivalent state.

Proof outline. Acting as scheduler, start with some initial bivalent state (**lemma 1**). Process takes step in RR fashion. Suppose its process p's turn. If message system contains no message for p, then let $e = (p, null)$. Otherwise consider the oldest message m destined to p and let $e = (p, m)$. Let G be the current state. Execute (p,m) if e(G) is bivalent. Otherwise find a finite length σ that doesnt contain e and $e(\sigma(G))$ is bivalent (**lemma 4**). Apply σ then apply e. The system will always be in a bivalent state. If process and message delays bounded, then consensus is solvable.
Consensus with Node Byzantine Failures. Timing model is **synchronous**. Nodes may experience byzantine failures (malicious nodes that can do anything and ignore guarantees), and reliable communication channel. Same goals apply for good nodes only.
Byzantine Consensus Threshold. If $n \leq 3f$ then byzantine consensus problem cannot be solved. n is the total number of processes and f is the number of possible failures.
Byzantine Consensus Intuition. Develop a protocol for $n \geq 4f + 1$. Idea of a rotating coordinator. Number the processes from 1 to n. Imagine a protocol with n phases and process i is the coordinator for phase i (only possible because we can define rounds).
Coordinator. Sends a proposal to all processes. Each phase has a coordinator round to do this. If coordinator is good, all processes sees the proposal (consensus). Phase is a **deciding phase** if coordinator is good.
Protocol. With at most f failures and $f + 1$ phases, at least 1 phase is a deciding one. After a deciding phase, all good processes have same value. Don't listen to coordinator if process sees alot of identical values from other proceses. Each phase will have a **all-to-all broadcast round**.

n processes; at most f failures; f+1 phases; each phase has two rounds

```
Code for Process i:
    V[1..n] = 0; V[i] = my input;
    for (k = 1; k ≤ f+1; k++) {   // (f+1) phases
round for all-to-all broadcast
        send V[i] to all processes;
        set V[1..n] to be the n values received;
        if (value x occurs (> n/2) times in V) proposal = x;
        else proposal = 0;
coordinator round
        if (k==i) send proposal to all; // I am coordinator
        receive coordinatorProposal from the coordinator
decide whether to listen to coordinator
        if (value y occurs (> n/2 + f) times in V) V[i] = y;
        else V[i] = coordinatorProposal;
    }
    decide on V[i];
```

send my value to all processes and set the values I get. If some value occurs more than half the time, then the proposed value is that. Otherwise, there is no proposal.

if Im the coordinator, send my proposal to everyone including myself. So everyone receives the proposal value from the previous step.

if some value occurs more than half the time + number of failures then we update that value. Otherwise we listen to coordinator.

Proof. **Lemma 1**: If all good processes P_i have $V[i] = y$ at the start of phase k, then this remains true at the end of phase k. There are $n - f$ good processes. This means that $n - f > \frac{n}{2} + f$, so they form majority. **Lemma 2**: If coordinator in phase k is good, then all nonfaulty processes P_i have the same $V[i]$ at the end of phase k. **Case 1** Coordinator has proposal value as x (must be unique to coordinator). On coordinator, x appears $> \frac{n}{2}$ times in V means that $> \frac{n}{2} - f$ must be from good processes. On other processes, x appears $> \frac{n}{2} - f$ times means its impossible for y to appear $> \frac{n}{2} + f$ times in V. **Case 2** Coordinator has proposal value as 0. On coordinator, no value x appears $> \frac{n}{2}$ times in V. So on other processes its impossible for y to appear $> \frac{n}{2} + f$ times in V.
Correctness. **Termination** obvious since $f + 1$ phases. **Validity** follows lemma 1. **Agreement** - with $f + 1$ phases, at least 1 is a deciding phase. Then by lemma 2, after deciding phase, all good process have same V[i]. From lemma 1, in following phases, V[i] on good processes dont change.

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 Randomized algorithm with 1/r error prob
Ver 3: Node crash failures; Channels are reliable; Asynchronous;	Impossible (the FLP theorem)
Ver 4: Node Byzantine failures; Channels are reliable; Synchronous; (the Byzantine Generals problem)	If n ≤ 3f, impossible. If n ≥ 4f + 1, we have a (2f+2)-round protocol. How about 3f+1 ≤ n ≤ 4f ? Solvable

Self-stabilisation

State of distributed system is either legal/illegal. Definition based on application semantics. Code on each process assumed to be correct all the time.
Self-stabilising. Starting from any (legal/illegal) state, protocol eventually reach legal state if there are no faults. Once the system is in a legal state, it will only transit to other legal states unless there are faults. Intuitively will always recover from faults and once recovered, will remain recovered forever. Self-stabilising algo typically runs in background and never stops.
Self-stabilising Spanning Tree. Given n processes connected by undirected graph and 1 special process p1, construct a spanning tree rooted at p1. Not all

processes can communicate with all processes directly. Can be used to compute shortest path.

Algorithm. Each process maintains 2 variables, parent (whose my parent) and dist (distance to root, never negative). Variables are continuously updated. At any given point of time, values of 2 variables can be wrong due to faults.

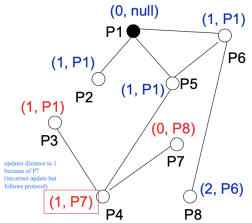
On P1 (executed periodically):

- dist = 0; parent = null;

On all other processes (executed periodically):

- Retrieve dist from all neighbors
- Set my own dist = 1 + (the smallest dist received)
- Set my own parent = my neighbor with the smallest dist (with tie breaking if needed)

Red values are initially incorrect values; blue values are values that have become correct



Comparison with leader election. Leader election builds spanning tree from scratch. Self-stabilising either builds from scratch or incrementally repairs.

Formality. A **phase** to be the min time period where each process has executed its code at least once. A **level** is A_i be the length of shortest path from process i to root. $dist_i$ be value of dist on process i. When topology does not change, A_i fixed but $dist_i$ may change.

Level properties. A node at level X has at least 1 neighbor in level X-1. A node at level X can only have neighbors in level X-1, X and X+1.

Correctness proof. **Lemma 1**: at end of phase 1, $dist_1 = 0$ and $dist_i \geq 1, \forall i \geq 2$. **Lemma 2**: At end of phase r, any process i whose $A_i \leq r - 1$ will have $dist_i = A_i$ and any process i whose $A_i \geq r$ will have $dist_i \geq r$

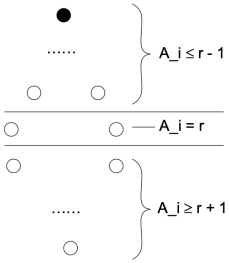
Technique for self-stabilisation proof. Process may take multiple actions in a phase and actions may be done in parallel - cannot assume a serialisation of all actions. Steps to proof:

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

Claim 1: The blue conditions hold *throughout* phase r+1.

Proof: Use an induction on t. If t = 0, trivial. Next assume the statement holds for t and consider action (t+1) by some node i. (Cannot assume that action (t+1) happens after the t actions.)

for nodes with	already know: phase r	want to show: phase r+1
A_i ≤ r-1	dist_i = A_i	dist_i = A_i
A_i = r	dist_i ≥ r	dist_i = r
A_i ≥ r+1	dist_i ≥ r	dist_i ≥ r+1

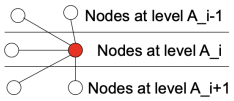


Proof (continued):

Case I: A_i ≤ r-1. Node i has at least one neighbor Y at level A_i-1. By the blue condition and by our inductive hypothesis, we know that Y's dist value is exactly A_i -1. Furthermore, node i only has neighbors at level A_i-1, A_i, and A_i+1. By the blue condition and by our inductive hypothesis, the dist value on any of these neighbors will not be smaller than A_i-1. Hence the min dist seen by node i is A_i-1. Hence node i will set its dist value to be A_i.

Case II: A_i ≥ r. Similar…

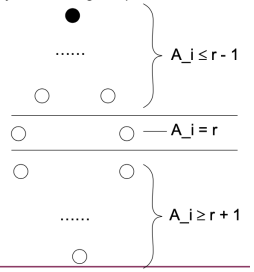
for nodes with	already know: phase r	want to show: phase r+1
A_i ≤ r-1	dist_i = A_i	dist_i = A_i
A_i = r	dist_i ≥ r	dist_i = A_i
A_i ≥ r+1	dist_i ≥ r	dist_i ≥ r+1



Claim 2: Each node i will satisfy the red conditions at some point of time during phase r+1.

Proof: By definition of a phase, node i takes at least one action in phase r+1. Consider three possible cases for node i: A_i ≤ r-1, A_i = r and A_i ≥ r. Also, leverage the fact the blue conditions always hold throughout phase r+1.

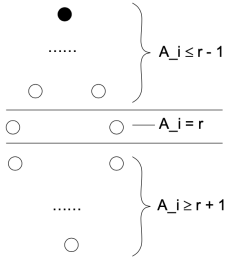
for nodes with	already know: phase r	want to show: phase r+1
A_i ≤ r-1	dist_i = A_i	dist_i = A_i
A_i = r	dist_i ≥ r	dist_i = A_i
A_i ≥ r+1	dist_i ≥ r	dist_i ≥ r+1



Claim 3: For each node after it first satisfies the red condition, it will continue to satisfy the red condition for the remainder of phase r+1.

Proof: Enumerate three cases, and also to leverage the fact that the blue conditions always hold throughout phase r+1.

for nodes with	already know: phase r	want to show: phase r+1
A_i ≤ r-1	dist_i = A_i	dist_i = A_i
A_i = r	dist_i ≥ r	dist_i = A_i
A_i ≥ r+1	dist_i ≥ r	dist_i ≥ r+1



Theorems. After H+1 phases, $dist_i = A_i$ on all processes. H is the length of the shortest path from the most far away process to the root. Directly follows from earlier lemma. After H+1 phases, dist and parent values on all processes are correct. Each process has a single parent pointer except the root. So the graph has n nodes and n-1 edges. Each process has a path to the root, thus the graph is connected. Hence its a spanning tree.