Lectures notes of Distributed Algorithm on Shared memory

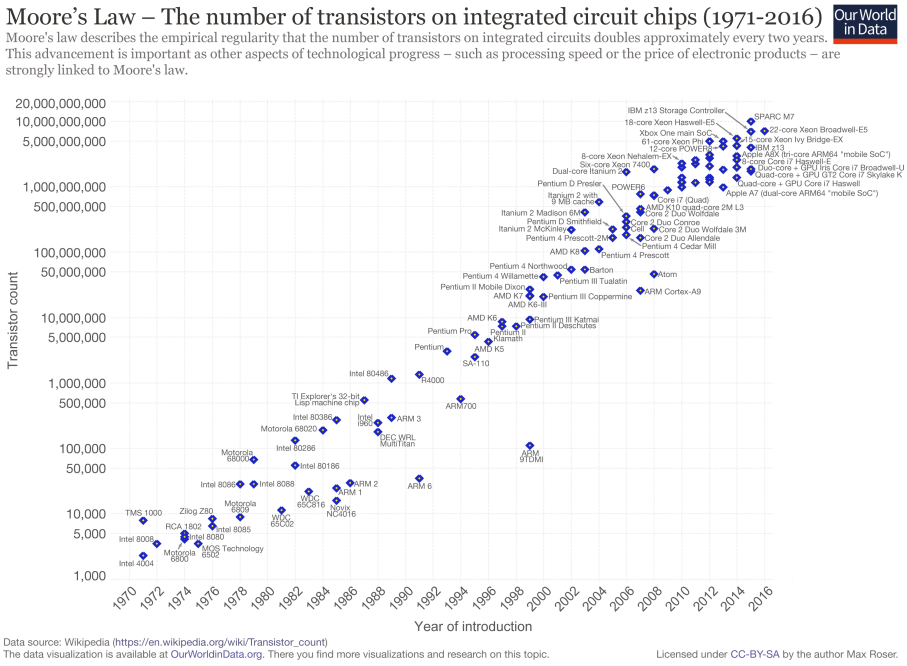
Garance Gourdel

September 2018

Contents

1 Introduction

Distributed algorithm is a crucial problem as some problems are just impossible to solve in sequential. Overall there are two important laws that were significant in the understanding of the importance of Distributed algorithm.
First, the Moore's law was that the number of transistors in a dense integrated circuit double every two year.



This improvement of the processors ended around the years 2003 2004. But the conclusion was that we could still add more cores to improve the performances.

Then comes the Amdahl's law, it says that if p is the fraction of work that can be done in parallel and n the number of processors, and considering that each processors needs time 1 to complete the job, the speed up is :

$$\frac{1}{1-p+\frac{p}{n}}$$

Which can be summed up as "the cost of synchronization is high.

Synchronization is the coordination of every program and it takes time, asynchronicity at the opposite is random scheduling.

Something that is also important to understand is that Distributed algorithm and parallelization aren't the same thing, one considers that there is a supervisor (parallel) and the other one does not. The main challenge in distributed computing is correctness, furthermore in distributed computing if a computer crashes, no computation can be done.

In conclusion, every system is now concurrent, every parallel program needs to synchronize and the cost of synchronization is high.

Synchronization, blocking and non blocking

2 Definition

Definition 2.1. Mutual exclusion No two processes are in their critical sections (CS) at the same time.

Definition 2.2. Deadlock-freedom At least one process eventually enters its CS. (Assuming no process blocks in CS or Entry section)

Definition 2.3. Starvation-freedom Every process eventually enters its CS. (Assuming no process blocks in CS or Entry section)

Those where originally implemented by reading an writing algorithm such as peterson's lock or Lamport's bakery algorithm, now it's currently handle in hardware (with mutex, semaphores).

3 Peterson Lock

Peterson's lock 2 processes

```
bool flag[0]   = false;
bool flag[1]   = false;
int turn;
```

```
P0:                                 P1:

flag[0] = true;                     flag[1] = true;
turn = 1;                           turn = 0;
while (flag[1] and turn==1)         while (flag[0] and turn==0)
{                                   {
          // busy wait                          // busy wait
}                                   }
// critical section                 // critical section
…                                   …
// end of critical section          // end of critical section
flag[0] = false;                    flag[1] = false;
```

Peterson's lock N processes

```
// initialization
level[0..N-1] = {-1};     // current level of processes 0...N-1
waiting[0..N-2] = {-1}; // the waiting process in each level
                          // 0...N-2

// code for process i that wishes to enter CS
for (m = 0; m < N-1; ++m) {
    level[i] = m;
    waiting[m] = i;
    while(waiting[m] == i &&(exists k ≠ i: level[k] ≥ m)) {
        // busy wait
    }
}
// critical section
level[i] = -1; // exit section
```

Bakery

```
// initialization
flag: array [1..N] of bool = {false};
label: array [1..N] of integer = {0}; //assume no bound

// code for process i that wishes to enter CS

flag[i] = true; //enter the "doorway"
label[i] = 1 + max(label[1], ..., label[N]); //pick a ticket
//leave the "doorway"
while (for some k ≠ i: flag[k] and (label[k],k)<<(label[i],i));
// wait until all processes "ahead" are served
…
// critical section
…
flag[i] = false; // exit section
```

Processes are served in the "ticket order": first-come-first-serve.

Bakery simplified

```
// initialization
flag: array [1..N] of bool = {false};
label: array [1..N] of integer = {0}; //assume no bound

// code for process i that wishes to enter CS
flag[i] = true; //enter the doorway
label[i] = 1 + max(label[1], ..., label[N]); //pick a ticket
flag[i] = false; //exit the doorway
for j=1 to N do {
          while (flag[j]); //wait until j is not in the doorway
          while (label[j]≠0 and (label[j],j)<<(label[i],i));
          // wait until j is not "ahead"
}
…
// critical section
…
label[i] = 0; // exit section
```

Ticket withdrawal is "protected" with flags: a very useful trick: works with "safe" (non-atomic) shared variables.

Black and white Bakery

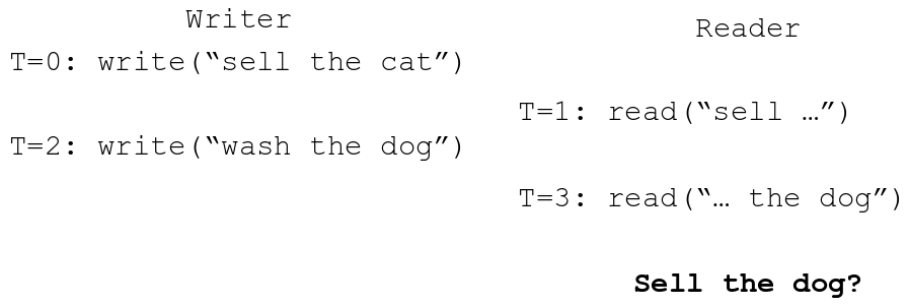
```
// initialization
color: {black,white};
flag: array [1..N] of bool = {false};
label[1..N]: array of type {0,…,N} = {0}  //bounded ticket numbers
mycolor[1..N]: array of type {black,white}

// code for process i that wishes to enter CS
flag[i] = true; //enter the "doorway"
mycolor[i] =color;
label[i] = 1 + max({label[j]| j=1,…,N: mycolor[i]=mycolor[j]});
flag[i] = false; //exit the "doorway"
for j=1 to N do
   while (flag[j]);
   if mycolor[j]=mycolor[i] then
      while (label[j]≠0 and (label[j],j)<<(label[i],i) and mycolor[j]=mycolor[i]  );
   else
      while (label[j]≠0 and mycolor[i]=color and mycolor[j] ≠ mycolor[i]);
// wait until all processes "ahead" of my color are served
…
// critical section
…
if mycolor[i]=black then color = white else color = black;
label[i] = 0; // exit section
```

Colored tickets implies bounded variables!

4 Synchronization tools

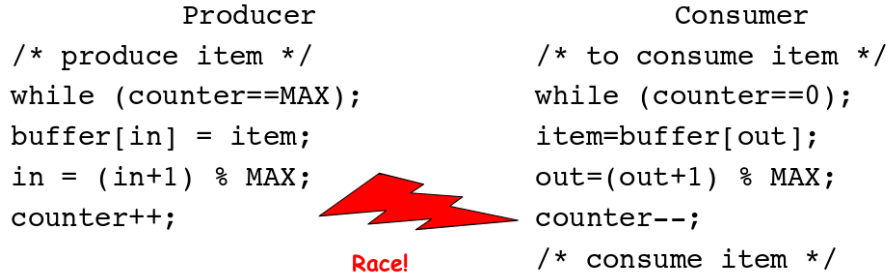
4.1 Readers-writers problem

We have a writer and a reader, the writer updates a file and the reads keeps itself up-to-date but read and write are non-atomic. Synchronisation is necessary to make sure the reader doesn't read inconsistent values.



4.2 Producer-consumer (bounded buffer) problem

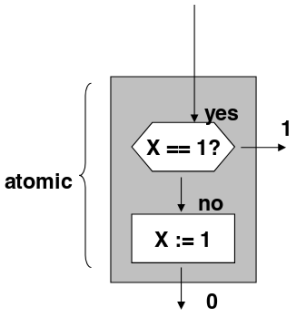
We have producers and consumers, Producers put items in the buffer (of bounded size), Consumers get items from the buffer. Every item is consumed, but no item is consumed twice, it is a very common situation : Client-server, multi-threaded web servers, pipes, ... Synchronisation is necessary to make sure items don't get lost and are not consumed twice.



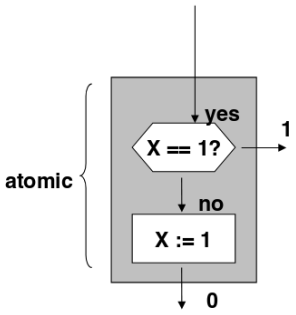
4.3 Busy-wait: Test and Set

Definition 4.1. Test and set $TAS(X)$ tests if $X = 1$, sets X to 1 if not, and returns the old value of X.

```
TAS(X):

atomic { if X == 1 return 1;
         X = 1;
         return 0;
```



But the problem is that the atomicity of the TAS is achieved by programs busy waiting, and for multiple producers and consumers we don't record the order of the requests.



shared X:=0

Producer	Consumer
while(counter==MAX);	while (counter==0);
. . .	. . .
buffer[in] = item;	item = buffer[out];
. . .	. . .
while TAS(X);	while TAS(X);
counter++;	counter--;
X:=0;	X:=0;
. . .	...

4.4 Semaphores and locks

4.4.1 Semaphores

Definition 4.2. Semaphores A semaphore S is an integer variable accessed (apart from initialization) with two atomic operations $P(S)$ and $V(S)$ (Stands for "passeren" (to pass) and "vrijgeven" (to release) in Dutch).
The value of S indicates the number of resource elements available (if positive), or the number of processes waiting to acquire a resource element (if negative).

```
Init(S,v){ S := v; }

P(S){
        while S<=0;     /* wait until a resource is available */
        S--;            /* pass to a resource */
}

V(S){
        S++;            /* release a resource */
}
```

The implementation of a semaphore S is a composite object:

- $S.counter$ the value of the semaphore.
- $S.wq$ the waiting queue, memorizing the processes having requested a resource element.

```
Init(S,R_nb) {
   S.counter=R_nb;
   S.wq=empty;
}
```

```
P(S) {
   S.counter--;
   if S.counter<0{
    put the process in S.wq and wait until
   READY;}
}
```

```
V(S) {
   S.counter++
   if S.counter>=0{
            mark 1st process in S.wq as
            READY;}
}
```

4.4.2 Locks

Definition 4.3. Lock A semaphore initialized to 1, is called a lock (or a mutex). When a process is in a critical section, no other process can come in.

shared semaphore S := 1

Producer	Consumer
`while (counter==MAX);`	`while (counter==0);`
`. . .`	`. . .`
`buffer[in] = item;`	`item = buffer[out];`
`. . .`	`. . .`
`P(S);`	`P(S);`
`counter++;`	`counter--;`
`V(S)`	`V(S);`
`. . .`	`...`

But we are, as mentioned before, still waiting until the buffer is ready.

4.4.3 Semaphores for producer-consumer

We can use 2 semaphores:

- empty: indicates empty slots in the buffer (to be used by the producer)
- full: indicates full slots in the buffer (to be read by the consumer)

`shared semaphores empty := MAX, full := 0;`

Producer	Consumer
`P(empty)`	`P(full);`
`buffer[in] = item;`	`item = buffer[out];`
`in = (in+1) % MAX;`	`out=(out+1) % MAX;`
`V(full)`	`V(empty);`

4.4.4 Problems of blocking synchronization

With semaphores/locks we can have different problems:

- Blocking: the progress of a process is conditional, it depends on other processes.
- Deadlock: no progress ever made

`X1:=1; X2:=1`

Process 1	Process 2
`...`	`...`
`P(X1)`	`P(X2)`
`P(X2)`	`P(X1)`
`critical section`	`critical section`
`V(X2)`	`V(X1)`
`V(X1)`	`V(X2)`
`...`	`...`

- Starvation: requests blocked in the waiting queue forever

Overall blocking synchronization can cause priority inversion (High-priority threads blocked), Program not being robust (Page faults, cache misses etc) and not composable. Considering those problems could we think of anything else ?

4.5 Nonblocking synchronization

In such algorithm a process makes progress, regardless of the other processes, for example:

shared buffer[MAX]:=empty; head:=0; tail:=0;

Producer `put(item)`	Consumer `get()`

`if (tail-head == MAX){`
`  return(full);`
`}`
`buffer[tail%MAX]=item;`
`tail++;`
`return(ok);`
	`if (tail-head == 0){`
	`  return(empty);`
	`}`
	`item=buffer[head%MAX];`
	`head++;`
	`return(item);`

However, it works for 2 processes but it's hard to tell why it works, and how could we do for multiple producers or consumers, how could be apply to other synchronization problems ? We will investigate more those questions in the next sections of the course

4.6 Transactional memory

In this model, we have mark sequences of instructions as an atomic transaction, e.g., the resulting producer code:

```
atomic {
        if (tail-head == MAX){
        return full;
        }
        items[tail%MAX]=item;
        tail++;
}
return ok;
```

Each transaction can be either committed or aborted:

- Committed transactions are serializable
- Let the transactional memory (TM) care about the conflicts
- Easy to program, but performance may be problematic

5 Correctness of algorithms: Safety and Liveness

Now if we want to continue in this path of analysing the system formally we have to define models (traceability, realism), devise abstractions for the system design (convenience, efficiency), and devise algorithms and determine complexity bounds.

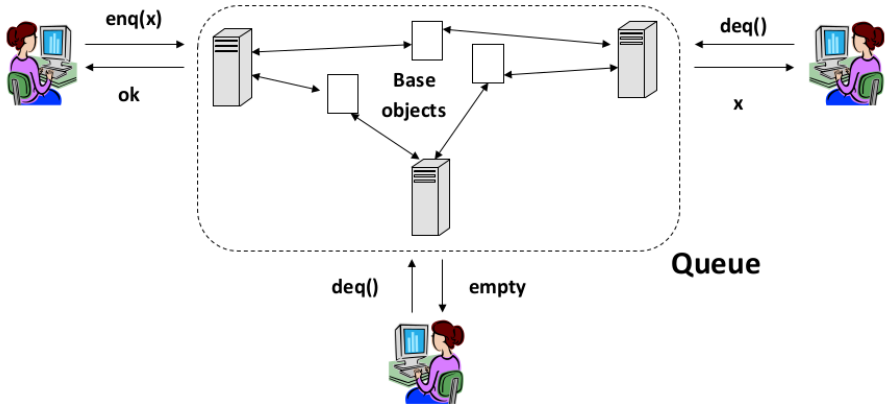
Definition 5.1. A **process** is an entity performing independent computation (this is an abstraction). The communication is made through message passing by **channel** and using a shared memory **objects**.

A process can be considered as automaton with states inputs outputs and a sequential specification.

An algorithm is a set of processes and can be either deterministic or randomized.

Processes communicate by applying operations on an receiving responses from *shared objects*. Then a shared object instantiate a state machine with states, operations, responses and a sequential specification. We will see several examples of that : TAS, CAS, LL/SC ...

We are looking to implement an object, using base object, we aim to create an illusion that an object O is available.



How can we define the correctness of an implementation ?

5.1 Intuition

Intuitively we define **Safety** as "noting bad ever happens". It can be violated in a finite execution, for example by producing a wrong output or sending an incorrect message. The question is "What the implementation is allowed o output ?"

Intuitively a **Liveness** property is "something good eventually happens. It can only be violated by an infinite execution, for example by never producing an expected output. And the question is under which condition the implementation outputs.

In our context, processes access abstractions by invoking operations. Operations are implemented by using sequences of accesses to base objects (for ex : a bounded-buffer using reads, writes, TAS, etc)

Definition 5.2. We say that a process is **correct** if it never fails (stop taking steps) in the middle of its operation.

Definition 5.3. A system **run** is a sequence of events, for example actions that processes may take. Σ is the event's alphabet, for example all possible actions. and $\Sigma^{*\cup\infty}$ is the set of all finite and infinite runs.

5.2 Formal definitions

Definition 5.4. A property P is a **safety** property if :

- P is **prefix-closed**: if σ is in P, then each prefix of σ is in P.
- P is **limit-closed**: for each infinite sequence of traces σ_0, σ_1, σ_2..., such that each σ_i is a prefix of σ_{i+1} and each σ_i is in P, the limit trace σ is in P.

Definition 5.5. P is a **liveness** property if every **finite** σ in Σ^* has an **extension** in P.

Note that it is enough to prove safety for all finite traces of an algorithm, and liveness for all infinite runs.

5.3 Some examples

We place our selves in a context where every processes **propose** values and **decide** on values. Determine whether the following are safety or liveness :

$$\Sigma = U_{i,v}\{propose_i(v), decide_i(v)\} U \{base\text{-}object\ accesses\}$$

And under the conditions that:

- Every decided value was previously proposed. Safety.
- No two processes decide differently. Safety.

- Every **correct** (taking infinity many steps) process eventually decides. Liveness.
- No two correct processes decide differently. Liveness, it can not be determine in finite time whether a process is correct.

5.4 Rules of thumb

Let P be a property:

- If every run that violates P is **infinite** $\rightarrow$ P is liveness.
- Every run that violates P has a finite prefix that violates $P \rightarrow P$ is safety.
- Otherwise, P is a mixture of safety and liveness.

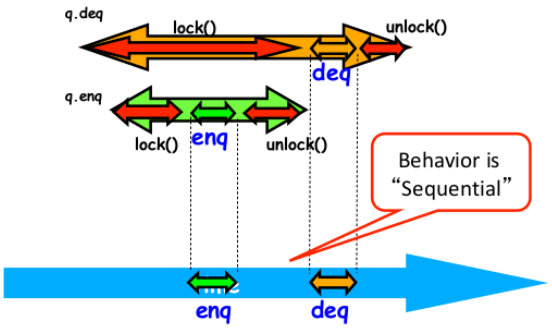
6 Linearisability

6.1 Example implementing a concurrent queue

We want to implement a concurrent FIFO queue but we are confronted with the fact that FIFO means strict temporal order, and concurrent implies an ambiguous temporal order.

By using a lock we immediately introduce a sequential behavior for the concurrency.

```
shared
     items[];
     tail, head := 0

deq()

  lock.lock();
    if (tail = head)
       x := empty;
    else
       x := items[head];
       head++;
  lock.unlock();
  return x;
```



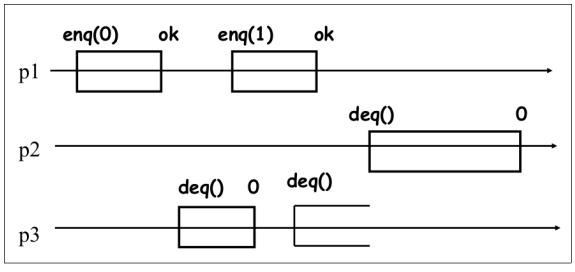
6.2 linearizability and histories

Linearizability is a Safety (and atomicity) property. Each operation should :

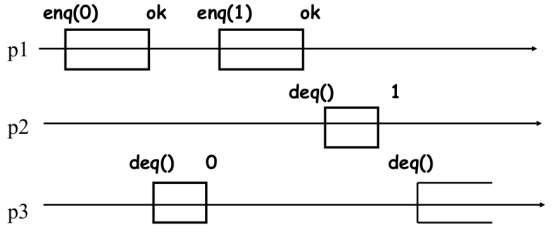
- "take effect"
- Instantaneously
- Between invocation and response events

On the thumb we can say that the history of a concurrent execution is correct if its "sequential equivalent" (it's history) is correct. Let us now define formally histories:

Definition 6.1. A **history** is a sequence of invocation and responses :
E.g. p1-enq(0), p2-deq(),p1-ok,p2-0,...
A history is **sequential** if every invocation s immediately followed by a corresponding response :
E.g., p1-enq(0), p1-ok, p2-deq(),p2-0,...
In particular a sequential history has no concurrent operations.



p1-enq(0); p1-ok; p3-deq(); p1-enq(1); p3-0; p3-deq(); p1-ok; p2-deq(); p2-0



p1-enq(0); p1-ok; p3-deq(); p3-0; p1-enq(1); p1-ok; p2-deq(); p2-1;p3-deq();

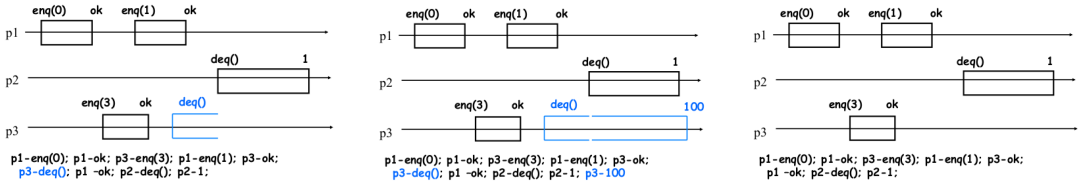
Definition 6.2. A sequential history is **legal** if it satisfies the sequential specification of the shared object.

This definition translate for (FIFO) queues as : every deq returns the firs not yet dequeued value. And of Read-write registers : every read returns the last written value.

6.3 completion

Definition 6.3. Let H be a history. AN operation op is **complete** in H if H contains both the invocation and the response of op.
A **completion** of H is a history H' that includes all complete operations of H and a subset of incomplete operations of H followed with the matching responses.

For the same history there are several completion possible :



Definition 6.4. Histories H and H' are **equivalent** if for all process p_i, $H|_{p_i} = H'|_{p_i}$.

Examples : $H = p_1 - enq(0); p_1 - ok; p_3 - deq(); p_3 - 3$ and $H' = p_1 - enq(0); p_3 - deq(); p_1 - ok; p_3 - 3$

Definition 6.5. linearizability/atomicity
A history H is **linearizable** if there exists a **sequential legal** history S such that:

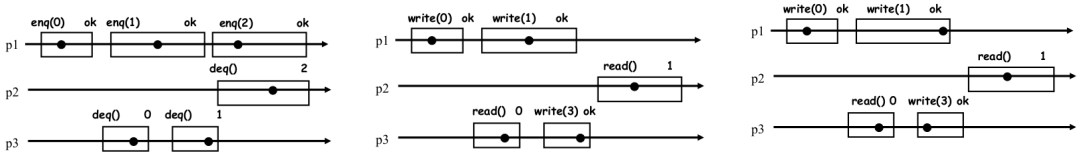
- S is **equivalent** to some completion of H.
- S preserves the **precedence relation** of H:

$$op1 \text{ precedes } op2 \text{ in } H \rightarrow op1 \text{ precedes } op2 \text{ in } S$$

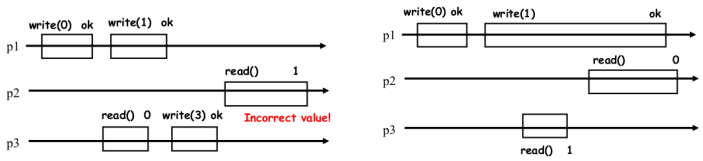
6.4 linearisation points

Definition 6.6. linearizable
An implementation is **linearizable** id every history it produces is linearizable.

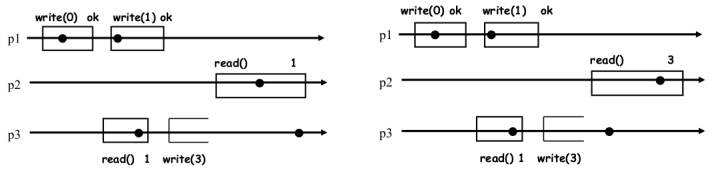
Informally we can view this as the fact that all operations (and some incomplete operation) in a history are seen as taking effect instantaneously at sometime between their invocation and responses. Operation are then ordered by their **linearization points** and must constitute a legal (sequential) history.
It here is are some examples of linearazitation, legal for 1 and 3 and illegal for 2 :



But some history can't be linearized at all :



And some can be linearized because they are incomplete :

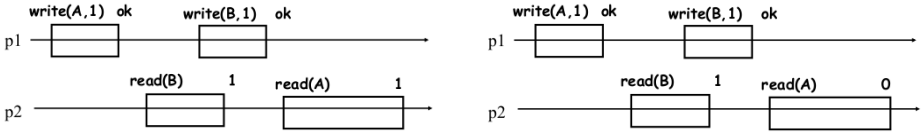


We can also define **Sequential consistency** as : a history H is **sequentially consistent** if there exists a **sequential legal** history S such that :

- S is **equivalent** to some completion of H.
- S preserves the **per(process order** of H:

$$p_i \text{ executes } op1 \text{ before } op2 \text{ in } H \rightarrow p_i \text{ executes } op1 \text{ before } op2 \text{ in } S$$

But it's far less interesting as **linearizability is compositional** : Any history on two linearizable objects A and B is a history of a linearizable $composition(A, B)$. For example A composition of two registers A and B is a two-field $register(A, B)$. Where as sequential consistency is not, a composition of sequential consistent objects is not always sequential consistent.



Every incomplete operation in a finite history can be independently : **Linearizability is nonblocking**.

Linearizability is a safety property :

- Prefix-closed: every prefix of a linearizable history is linearizable
- Limit-closed: the limit of a sequence of linearizable histories is linearizable

And (see section 5) implementation is linearizable if and only if all its finite histories are linearizable.

Here it is good to think again why we wish to design linearizable objects and not just use a lock. Using a lock is simple it is an automatic transformation of the sequential code, it is correct, we get linearizability for free. On the other hand In the best case we get starvation freedom "if the lock is "fair" and every process cooperates, every process makes progress" but it is not robust to failures/asynchrony and trying to improve that with Fine-grained locking is complicated and prone to deadlocks.

7 Liveness proprieties

7.1 definition

Let us know define several liveness properties :

- Deadlock-free:If every process cooperates (takes enough steps), some process makes progress.
- Starvation-free: If every process cooperates, every process makes progress.
- Lock-free (sometimes called non-blocking): Some active process makes progress.
- Wait-free: Every active process makes progress.
- Obstruction-free: Every process makes progress if it executes in isolation.

We can also sort them in a sort of "periodic" table of liveness properties:

	independent non-blocking	dependent non-blocking	dependent blocking
every process makes progress	wait-freedom	obstruction-freedom	starvation-freedom
some process makes progress	lock-freedom	?	deadlock-freedom

7.2 hierarchy

We can define relations between liveness properties as follows :

Definition 7.1. Property A is **stronger** than property B if every run satisfying A also satisfies B (A is a subset of B). A is **strictly stronger** than B if, additionally, some run in B does not satisfy A, i.e., A is a proper subset of B.

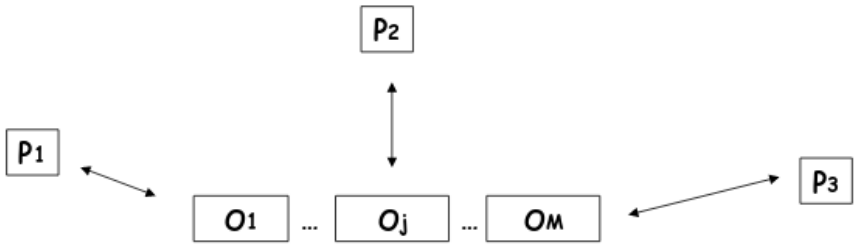
For example:

- WF is stronger than SF
  Every run that satisfies WF also satisfies SF: every correct process makes progress (regardless whether processes cooperate or not). WF is actually strictly stronger than SF.
- SF and OF are incomparable (none of them is stronger than the other)
  There is a run that satisfies SF but not OF: the run in which $p1$ is the only correct process but does not make progress. There is a run that satisfies OF but not SF: the run in which every process is correct but no process makes progress

One interesting thing about linearization, is that from a model that gives us no guarantees in case of failures can also be transform into a model that will give us guarantees.

8 Shared memory model

On this model every processes communicates by applying operations on and receiving responses from shared objects.
A shared object is a state machine, with states, operations, responses and sequential specification. For example **read-write registers**, TAS, CAS, LLSC, ...



8.1 Readwrite register

They can store values (in a value set V) an export two operations : read and write.

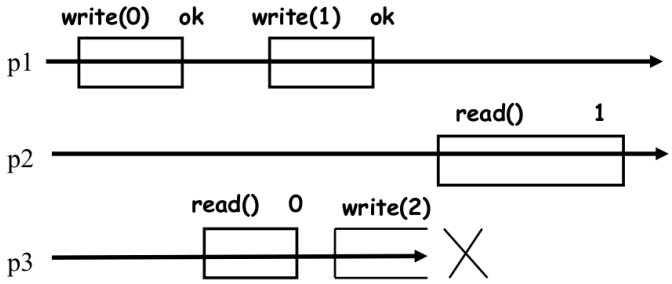
- Write takes an argument in V and returns ok.
- Read takes no arguments and returns a value in V.

Processes invoke operations on the shared objects and they may want some guarantees:

- **Liveness**: the operations eventually return something

- **Safety**: the operations never return anything incorrect

Let us note what objects we will consider in this class, all objects considered in this class are wait-free and we only consider well-formed runs: a process never invokes an operation before returning from the previous invocation.

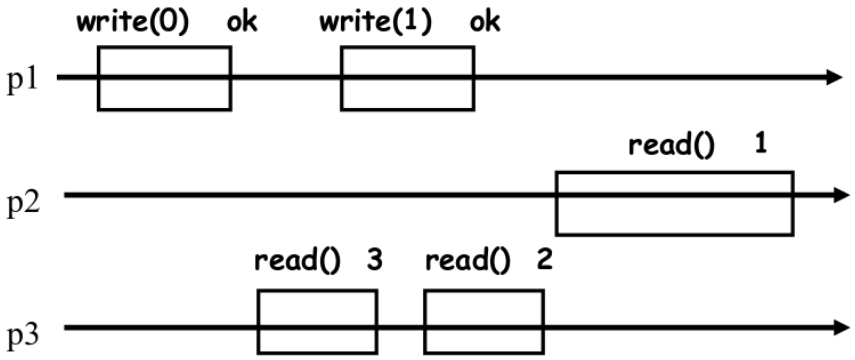


Definition 8.1.
- Operation $op1$ **precedes** operation $op2$ in a run R if the response of $op1$ precedes (in global time) the invocation of $op2$ in R.
- If neither $op1$ precedes $op2$ nor $op2$ precedes $op1$ than $op1$ and $op2$ are **concurrent**

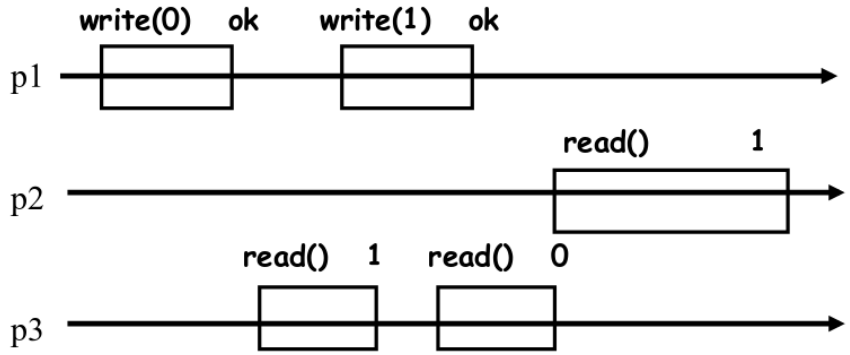
8.2 Safety criteria

As a reminder, we can say informally that safety implies that every read operation returns the "last" written value (the argument of the "last" write operation).

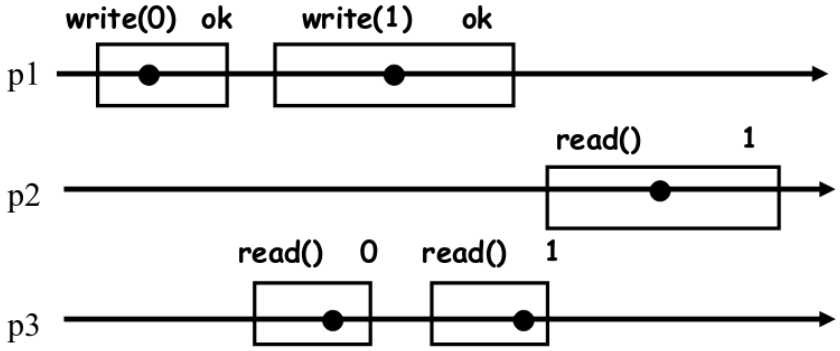
Definition 8.2. safe, regular and atomic registers **Safe registers** : every read that does not overlap with a write returns the last written value.
Regular registers : every read returns the last written value, or the concurrently written value.
Atomic registers : the operations can be totally ordered, preserving legality and precedence (linearisability). if read1 returns v, read2 returns v' and read1 precedes,read2, then write(v') cannot precede write(v)



Safe register.



Regular register: this is not linearisable.



Atomic register: this is linearisable.

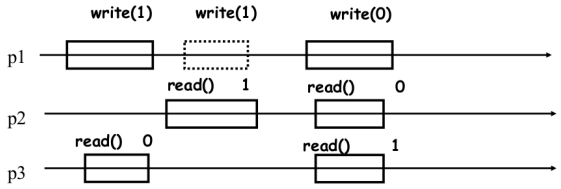
8.3 Transformation

Let us consider how we define our registers. There are values either binaries or multi-valued. The number of readers anr writers : from 1-writer 1-reader (1W1R) to multi-writer multi-reader (NWNR). And finally the safety criteria: from safe to atomic.

Theorem 1. 1W1R binary safe registers can be used to implement an NWNR multi-valued atomic registers!

8.3.1 1WNR safe to 1WNR regular

```
initially:
      shared 1WNR safe register R := 0
      lv := 0      \\ last written value

upon write(v)
      if v ≠ lv then
         lv := v
         R.write(v)
      return ok

upon read()
      return R.read()
```



8.3.2 1W1R regular to 1WNR regular

The registers can be either binary or multi-valued, but the proof is for binary.Let p1 be the only writer and 0 be the initial value.

```
initially:
  shared R[1..N] (1W1R binary regular registers) := 0^N
         // R[i] is written by p1 and read by pi

upon read()
  return R[i].read()

upon write(v)  // if i=1
  for all j do R[j].write(v)
  return ok
```

8.3.3 from binary to multi valued 1WNR

```
initially:
  shared array R[0,..M-1] of 1WNR registers := [1,0,…,0]

upon read()
  for j = 0 to M-1 do
         if R[j].read() = 1 then return j

upon write(v)  // if i=1
  R[v].write(1)
  for j=v-1 down to 0 do R[j].write(0)
  return ok
```

8.3.4 regular to atomic (1W1R)

Timestamps!

```
initially:
      shared 1W1R regular register R := 0
      local variables t := 0, x := 0

upon read()
      (t',x') := R.read()
      if t' > t  then t:=t'; x:=x';
      return(x)

upon write(v)   // if i=1
      t:=t+1
      R.write(t,v)
```

8.3.5 From 1W1R to 1WNR (multi-valued atomic)

```
shared:
  matrix RR[1..N][1..N] of 1W1R atomic registers := 0^(NxN)
  // for all i,j, RR[i][j] is read by pi and written by pj

  array WR[1..N] of 1W1R atomic registers := 0^N
  // for all i WR[i] is written by p1 and read by pi


upon write(v)  // code for p1
  ts:=ts+1
  for all j do WR[j].write([v,ts])
  return ok
```

```
upon read() // code for pi
   for all j=1,…,N  do (t[j],x[j]) := RR[i][j].read()
   (t[0],x[0]) := WR[i].read()
    (tmax,xmax) := highest(t,x)
    for all j do RR[j][i].write([tmax,xmax]);
    return(xmax)
```

9 Atomic snapshot

In the previous section, we saw that every registers, independently from their number of readers/writers, it's value set and safety property, all registers are (computationally) equivalent.

Definition 9.1. Atomic snapshot Its is a new object, reading multiple location atomatically, like a snapshot of all registers. Write to one, read to all.

We can define a sequential specification, each process p_i is provided with operation :

- $update_i(v)$ returns ok
- $snapshot_i()$ returns $[v_1, ..., v_n]$

In this sequential execution, for each $[v_1, ..., v_n]$ returned by $snapshot_i()$, v_j $(j = 1...N)$ is the argument of the last $update_j(.)$ (or the initial value if no such update)

Snapshot for free ?

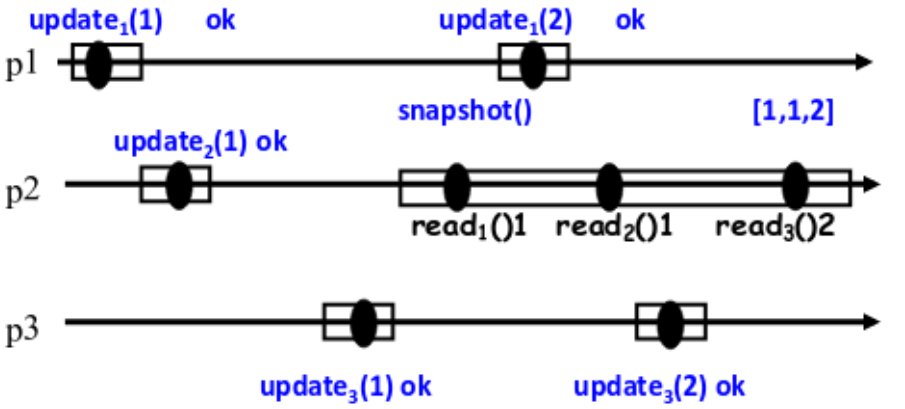
Code for process p_i:

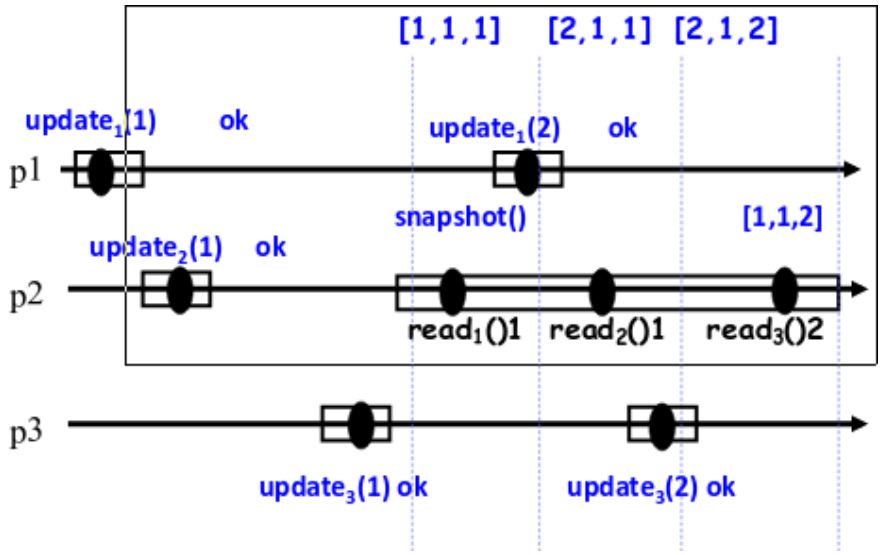
initially:
shared 1WNR *atomic* register $R_i := 0$

upon snapshot()
$[x_1,\ldots,x_N] := scan(R_1,\ldots,R_N)$ /*read $R_1,\ldots R_N$*/
return $[x_1,\ldots,x_N]$

upon update$_i$(v)
R_i.write(v)

Problem!





The difficulty comes from having 3 or more processes. But what about lock-free snapshots ? At least one correct process makes progress (completes infinitely many operations).

Code for process p_i (all written values, including the initial one, are unique, e.g., equipped with a sequence number) :

Initially:
shared 1W1R atomic register $R_i := 0$

upon snapshot()

$[x_1,...,x_N]$:= scan($R_1,...,R_N$)
repeat
$[y_1,...,y_N] := [x_1,...,x_N]$
$[x_1,...,x_N]$:= scan($R_1,...,R_N$)
until $[y_1,...,y_N] = [x_1,...,x_N]$
return $[x_1,...,x_N]$

upon update$_i$(v)

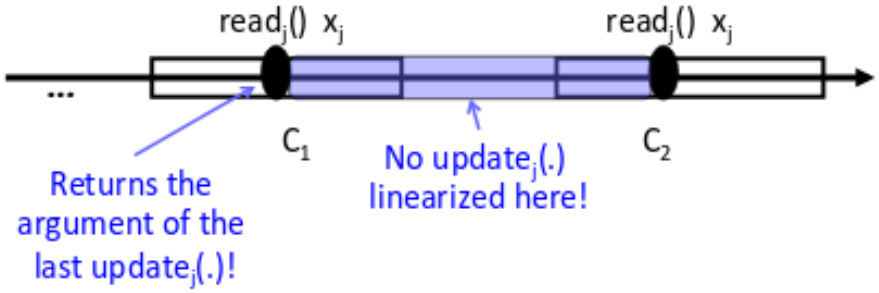
R_i.write(v)

This can be linearized by assigning a linearization point to each operation:

- $update_i(v)$ $R_i.write(v)$ if present, otherwise remove the op.
- $snapshot_i()$ if complete - any point between identical scans, otherwise remove the op.

It builds a **sequential history S** in the order of linearisation points. S is legal: every $snapshot_i()$ returns the last written value for every p_j.
proof : Suppose not: $snapshot_i()$ returns $[x_1, ..., x_N]$ and some x_j is not the the argument of the $lastupdate_j(v)$ in S preceding $snapshot_i()$.

Let C_1 and C_2 be two scans that returned $[x_1, ..., x_N]$, then :



Let's proove correctness: lock-freedom. Suppose a process p_i execution $snapshot_i()$ eventually runs in isolation (no process takes steps concurrently). Then all scan received by p_i are distinct, which means at least one process performs an update between. As there are finitely many processes, there is at least one process executes infinitely many updates.

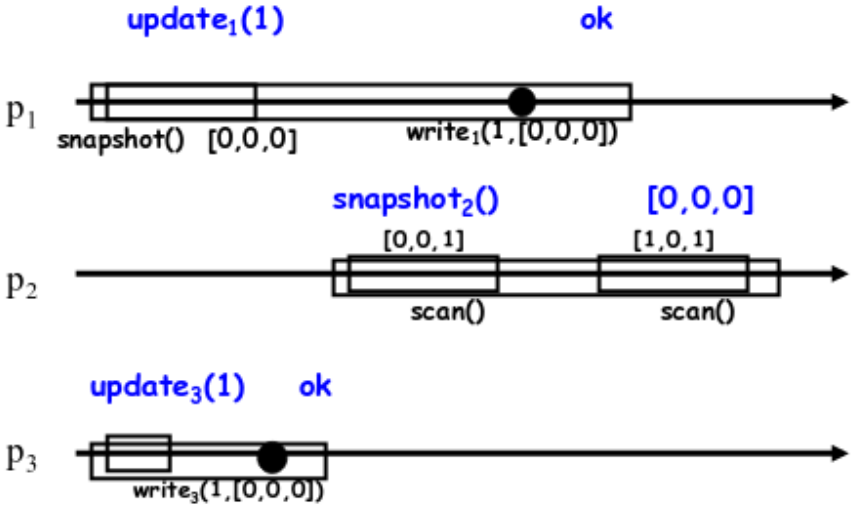
What can we do to make it wait free ? You have to accept that an concurrent process writes while your snapshots. Then make the update do the work.

upon snapshot()
$[x_1,...,x_N]$:= scan($R_1,...,R_N$)
$[y_1,...,y_N]$:= scan($R_1,...,R_N$)
if $[y_1,...,y_N] = [x_1,...,x_N]$ then
return $[x_1,...,x_N]$
else
let j be such that
$x_j \neq y_j$ and x_j=(u,U)
return U

upon update$_i$(v)
S := snapshot()
R_i.write(v,S)

But of two scans differ, some update succeeded ! Would this work ?

No!



We can fix that : **If a process moved twice: its last snapshot is valid!**

upon snapshot()

$[x_1,\ldots,x_N]$:= scan($R_1,\ldots,R_N$)
while true do
  $[y_1,\ldots,y_N]$:= $[x_1,\ldots,x_N]$
  $[x_1,\ldots,x_N]$:= scan($R_1,\ldots,R_N$)
  if $[y_1,\ldots,y_N] = [x_1,\ldots,x_N]$ then
    return $[x_1,\ldots,x_N]$
  else if $moved_j$ and $x_j \neq y_j$ then
    let $x_j = (u,U)$
    return U
  for each j: $moved_j$:= $moved_j \vee x_j \neq y_j$

upon update$_i$(v)

S := snapshot()
R_i.write(v,S)

If a process moved twice: its last snapshot is valid!

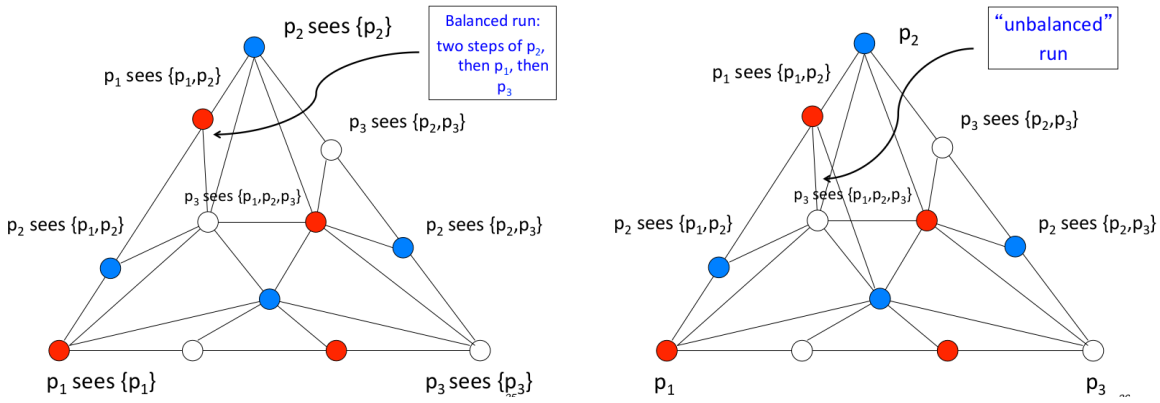
We claim that the complexity is in $O(N^2)$ steps and it is wait free. This implementation is also linearizable : for update when write, for snapshot if there is two identical scans, then between the scans. Otherwise, if returned U of p_j : at the linearization point of p_j 's snapshot.

The linearization is legal : every snapshot operation returns the most recent value for each proces. But also consistent with the real-time order: each linearization point is within the operation's interval and equivalent to the run (locally indistinguishable).

10 One-shot atomic snapshot (AS)

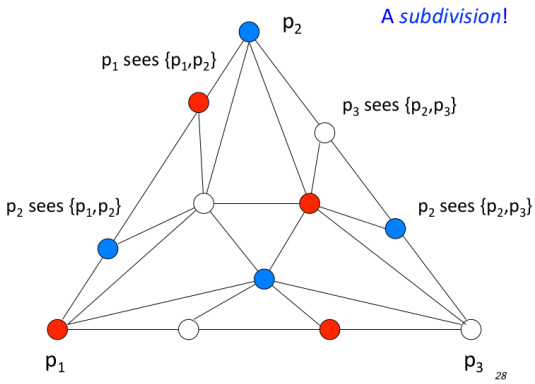
Definition 10.1. For one-shot atomic snapshot the snapshots can only be done after an update. Each process does $update_i(v_i)$, then a snapshot that returns S_i .

Theorem 2. We have that $v_i \in S_i$ (inclusion), and for all i and j, S_i is a subset of S_j or S_j is a subset of S_i (containment).



Definition 10.2. For one-shot **immediate** atomic snapshot the snapshots can only be done after an update. Each process does $update_i(v_i)$, then a snapshot that returns S_i , such that (**Immediacy**) for all i and j: if v_i is in S_j , then S_i is a subset of S_j.

Then the representation is a subdivision :



Theorem 3. IS is equivalent to AS (one-shot not immediate).

It is obvious that IS is a restriction of the AS, therefore IS is stronger than one-shot AS. A few one-shot AS objects can be used to implement IS :

shared variables:
$A_1,\ldots,A_N$ – atomic snapshot objects, initially [T,…,T]

Upon WriteRead$_i(v_i)$
r := N+1
while true do
r := r-1 // drop to the lower level
A_r.update$_i(v_i)$
S := A_r.snapshot()
if |S|=r then // |S| is the number of non-T values in S
return S

Let's prove by induction correctness and that the algorithm satisfies Self-Inclusion, Snapshot, and Immediacy. The case $N = 1$ is trivial.Suppose the algorithm is correct for $N - 1$ processes. Then N processes come to level N, at least one process returns in level N, by induction all properties hold of the ones that goes to level $N - 1$. The process that return at level N returns all N values, thus all proporeties hold for all N processes !

We define now **iterated immediate** snapshots (IIS) and shox it's representation over the number of iteration. First the iterated standard chromatic subdivision (ISDS), then one round of ISS then two round.

Shared variables:

IS_1, IS_2, IS_3,… // a series of one-shot IS

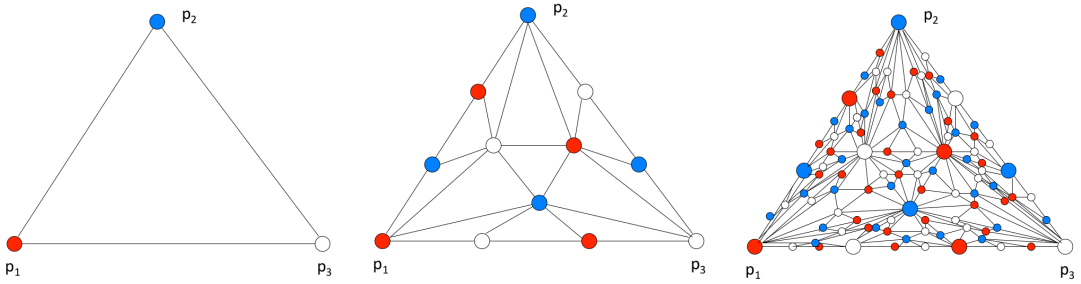
Each process p_i with input v_i:

r := 0

while true do

r := r+1

$v_i := IS_r.WriteRead_i(v_i)$



Theorem 4. IIS is equivalent to (multi-shoot) AS.

As shown before we can use several instance of AS to implement ISS. IIS can be used to implement multi-shot AS in the lock-free manner.

11 Consensus

We place ourselves in a model where there are N asynchronous (no bounds on relative speeds) processes p_0 ,...,p_{N-1} ($N \geq 2$) communicate via atomic read-write registers. Process can crash and a crashed process only takes finitely many steps. We say that a system that can resist to tprocesses crashing is a **t-resilient system**. if it is $(N-1)$-resilient system it is exactly wait-free.

Definition 11.1. Processes propose values and must agree on a common decision value so that the decided value is a proposed value of some process.
A process proposes an input value in V ($|V| \geq 2$) and tries to decide on an output value in V. In particular we define :

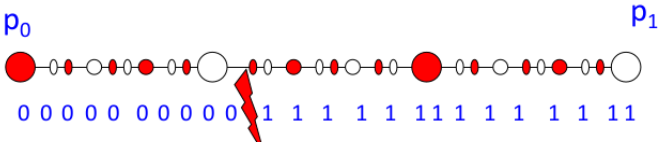
- **Agreement**: No two processes decide on different values.
- **Validity**: Every decided value is a proposed value.
- **Termination**: No process takes infinitely many steps without deciding (Every correct process decides).

If we consider a 0-resilient consensus it is easy, there are no crashes so every process can wait for 0 to choose and follow 0.

Theorem 5. No wait-free algorithm solves consensus (using only register).

Proof for $N = 2$ assuming that p_0 proposes 0 and p_1 proposes 1. This will imply the claim for all $N \geq 2$. We make the proof for IIS, it is possible because AS is equivalent to IIS. We find that solo runs remain connected, then there is no way to decide !

```
k:= 0
repeat
        k := k+1;
        R_k[i].write(v_i);
        v_i := [v_i, R_k[1-i].read()];
until not decided(v_i)
```



From there the goal is to evaluate the objects depending on their consensus power.

11.1 Consensus number

A class C of objects solves n-**consensus** if there exists a consensus protocol among n processes using any number of objects of class C and atomic registers. The **consensus number** of a class C: $(h(C))$ is the largest n for which that class solves n-consensus. This then defines a hierarchy of objects. If one can implement an object of class C from objects of class D (and registers) then $h(C) \geq h(D)$. (if $h(C) > h(D)$ there is no implementation of objects of C with objects of D).

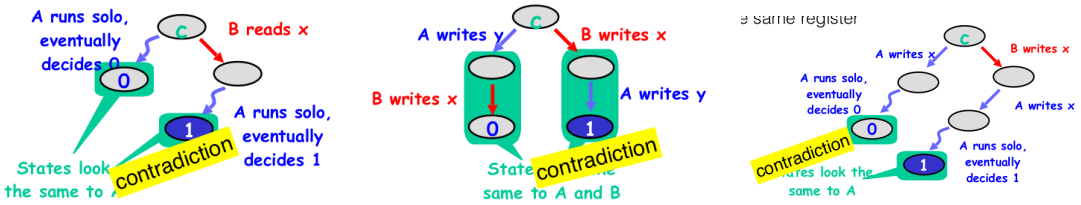
11.2 atomic registers

Theorem 6. There is no wait-free implementation of n-consensus from read-write registers for $n \geq 2$. h(Register) = 1 .

This theorem implies that asynchronous computability different from Turing computability.

Definition 11.2. A state is bivalent if the outcome is not fixed yet, and univalent if the outcome is fixed. A state is critical if it is bivalent, and every possible next step makes the state univalent.

Properties :
- Some initial state is bivalent
- If there is a consensus protocol then there is a critical state



11.3 queue FIFO list LIFO

Theorem 7. h(Queue) = 2 . FIFO (quieues) and LIFO (lists) have a consensus number of 2.

11.4 read-modify-write (RMW)

F is a set of functions of integers to integers. There is one method, method(v), which returns the previous value x, and replace x with f (v, x) for some f in F . It can encode getAnd-Set, getAndIncrement, getAndAdd, or compareAndSet. A RMW is nontrivial if it contains a function other than the identity. Any nontrivial RMW object has a consensus number of at least 2.

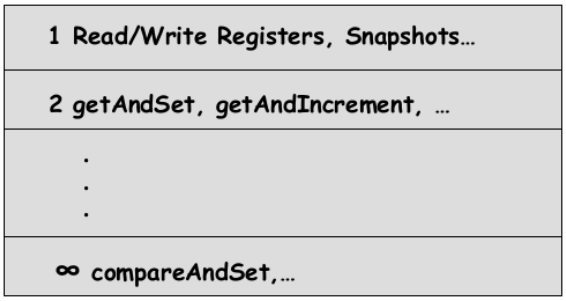
If F contains function such that for each i, j, either f_i commutes with f_j , or $f_i(f_j(v)) = f_i(v)$, then the consensus number is exactly 2.

11.5 Compare and set

CAS(expected, update) compare the value stored with expected, and set it to update if it is the same, and return true. Otherwise, return false. It has a consensus number of ∞.

11.6 Hiarchy

From the consensus number we proved we can define a hierarchy :



Theorem 8. Hirarchy
There exist non-deterministic objects a_i such that their consensus number is equal to n, while the consensus number of the union of them is $n+1$. However, this is not true for deterministic objects.

11.7 Universality

Definition 11.3. A class C of objects is universal for some set E of classes if and only if any object in E can be implemented with objects of C and registers.

Theorem 9. N-consensus is universal for N-objects (objects shared by N processes).

Proof : Construct a linked list, where processes tries to add nodes to the list. The consensus choose which node is added. This gives easily a lock-free algorithm. To have a wait-free algorithm, each thread adds the node they want to add in a table, and other processes will try to help them putting the node in the list. So, a process cannot loose at every step putting its node in the list.

11.8 k-set agreement

Definition 11.4. k-set agreement
Same as consensus, but there can be at most k distinct values decided. $(N-1)$-set agreement is called **set agreement**.
We redefine **Validity** as : Every decided value is a proposed value, and **Termination**: No process takes infinitely many steps without deciding (Every correct process decides).

Theorem 10. Impossibility of set agreement
No wait-free algorithm solves set agreement with only registers. This implies the impossibility to k-set agreement with $k < n-1$.

Theorem 11. 1-resilient consensus
No 1-resilient (assuming that one process might fail) algorithm solves consensus in read-write

12 Relaxing consensus

Definition 12.1. Safe agreement
Instead of the wait-free condition, we have that if every participant (process taking at least one step) takes enough steps, then every correct process decides.

Definition 12.2. Obstruction Freedom
We can use obstruction-freedom instead of wait-freedom: if at some point a process takes step alone then it decides.

Definition 12.3. Termination with probability 1
Each process is equipped with a random number generator. We want that there is termination with probability 1.

Definition 12.4. Leader election
Each process has a local leader, and there exist a correct process q and a time after which all sprocesses have their leader set to q.

Below are examples of safe agreements, obstruction freedom, termination with probability 1 and leader election.

Shared variables
$A[0,...,n-1]$ atomic snapshot object init $\perp$
$B[0,...,n-1]$ atomic snapshot object init $\perp$

Upon *propose*(v) **by process** p_i
A.update(v)
U:= A.snapshot()
B.update(U)
repeat
V:=B.snapshot()
until $\forall j : U[j] \neq \perp \Rightarrow V[j] \neq \perp$
Let X be the vector in V with the smallest number of non $\perp$ value
decide on the smallest non $\perp$ value in X

Shared variables
A infinite array of atomic snapshot object init $\perp$
decide init $\perp$

Upon *propose*(v) **by process** p_i
prop:=v
for r:=0 to ∞
A[2r].update(prop)
U:= A[2r].snapshot()
if all values non $\perp$ in U are equal to prop
then report:=prop
else report:=?
A[2r+1].update(report)
V:= A[2r+1].snapshot()
Let L be the non $\perp$ in V
if $|L| = 1$ then
if $L = \{1\}$ then decide:=1
if $L = \{0\}$ then decide:=0
if $L = \{?\}$ then prop:= r.n.g
else
if $L = \{1,?\}$ then prop:=1
if $L = \{0,?\}$ then prop:= 0
if decide $\neq \perp$ then return decide

Shared variables
A infinite array of atomic snapshot object init $\perp$
decide init $\perp$

Upon *propose*(v) **by process** p_i
prop:=v
for r:=0 to ∞
A[2r].update(prop)
U:= A[2r].snapshot()
if all values non $\perp$ in U are equal to prop
then report:=prop
else report:=?
A[2r+1].update(report)
V:= A[2r+1].snapshot()
Let L be the non $\perp$ in V
if $|L| = 1$ then
if $L = \{1\}$ then decide:=1
if $L = \{0\}$ then decide:=0
else
if $L = \{1,?\}$ then prop:=1
if $L = \{0,?\}$ then prop:= 0
if decide $\neq \perp$ then return decide

Shared variables
A infinite array of atomic snapshot object init $\perp$
decide init $\perp$

Upon *propose*(v) **by process** p_i
prop:=v
for r:=0 to ∞
wait until leader=p_i or decide $\neq \perp$
A[2r].update(prop)
U:= A[2r].snapshot()
if all values non $\perp$ in U are equal to prop
then report:=prop
else report:=?
A[2r+1].update(report)
V:= A[2r+1].snapshot()
Let L be the non $\perp$ in V
if $|L| = 1$ then
if $L = \{1\}$ then decide:=1
if $L = \{0\}$ then decide:=0
else
if $L = \{1,?\}$ then prop:=1
if $L = \{0,?\}$ then prop:= 0
if decide $\neq \perp$ then return decide

13 Failure detection

Definition 13.1. Failure Detector A failure detector provides (possibly incorrect) information about a failure pattern $F : T \to 2^{\Pi}$, which is the set of crashed processes. A failure detector history is the set of processes that are suspected to be crashed $H : T \times \Pi \to 2^{\Pi}$.

We call P the class of failure detectors that are perfect (each crashed process will be suspected, no correct process is suspected), and S is the class of failure detectors that are strong (at least one correct process is never suspected). P is stronger than S.

Definition 13.2. orders between failure Detector $D \geq D'$ if there exists a distributed algorithm such that processes query D and output D' . We have that $P \geq S$, but not $P \geq P$.

We also define ¡¿P as "eventually P" and the leader Ω. There exists a correct process q and a time t after which such that every correct process trusts q.The output of the failure detector is

one process. Eventually the output of all processes is the same correct process. We have a leader that we choose after a certain time and we trust him, he can decide what to output.

Theorem 12. ¡¿S and Ω are equivalent.

13.1 weakest failure detector

Definition 13.3. weakest failure detector For a problem P, a failure detector D is the weakest if :

- D can solve P
- if D' solves P then $D' \geq D$

It is generaly denoted by $wk(P)$

Intuitively a weakest failure detector encapsulates the exact amount of information on failure necessary and sufficient to solve a problem.

Theorem 13. For two problems P and P' and their respective weakest failure detector, if $wk(P) < wk(P')$ (or $wk(P) \leq wk(P')$ but not $wk(P) \geq wk(P')$) then P' is harder than P.

This theorem illustrate the importance of weakest failure detector, that's why there are many papers that chase the weakest failure detector for problems.

Theorem 14. Every problem that is solvable with a failure detector has a weakest failure detector

13.2 Failure detection and consensus

Processes construct a directed acyclic graph (DAG) that represents a "sampling" of failure detector values in the run and some temporal relationships between the values sampled. This DAG can be used to simulate runs of Consensus with D. By considering several initial configurations of Consensus, we obtain a forest of simulated runs of Consensus. It is possible to extract the identity of a process p that is correct in the run (the same correct process for all processes)

Theorem 15. The Borowsky-Gafni (BG) simulation algorithm
A set of $t+1$ asynchronous sequential processes may wait-free simulate an algorithm for n processes t-resilient ($n > t$). The reverse is also true (and trivial).

This means that if we know that it is impossible to achieve wait free consensus for two processes, then we deduce by BG that it is impossible to achieve 1-resilient consensus for n processes.

13.3 k-set agreement

If we take the algorithm such that :

- processes 1 to k write their input value in shared memory and decide their input value.
- other processes read the shared memory until they read some value and decide this value.

Then we can obviously say that :

Theorem 16. n processus can not solve k-set agreement k-resilient

13.4 BG simulation

We have $n+1$ processes that may wait free simulate and $m+1$ processes n-resilient ($m >= n$). Let S0,S1,....,Sn be the simulators.Let p0,p1, p2,..,pm the simulated processes. This processes have to execute a code Ci and used shared memory M.

Shared Memory
MEM, an n by m array of registers.
For each i,j, $MEM[i][j]$ contains a pair $(val, steps)$:
$MEM[i][j].val$, initially the initial value of j's register in $\mathcal{A}$
$MEM[i][j].steps \in N$, initially 0
For each i $MEM[i][-]$ can be written to by i
$CONSENSUS$, an m by infinite array of consensus objects
Process Local Variables Code of the simulator Pi
$input \in I_i$, initially the input of real process i
$state$, $steps$ arrays of size m, initially arbitrary
$decided$, a boolean, initially arbitrary
Thread Local Variables
$k', i', j, j', steps \in N$, initially arbitrary
val, v, w, o, variables, initially arbitrary
$mymem$, an array of n variables, initially arbitrary

upon PROPOSE($input$)
  $decided \leftarrow false$
  Parallel for all $j = 1..m$
    $w \leftarrow CONSENSUS_{(n-1)}[j][0](input)$
    $state[j] \leftarrow init\_state_j(w)$
    $steps[j] \leftarrow 1$
    repeat forever
      if $nextop_j(state[j]) = (WRITE, v)$ then
        $MEM[i][j] \leftarrow (v, steps[j])$
        $state[j] \leftarrow trans_j(state[j], DONE)$
      else if $nextop_j(state[j]) = (READ, j')$ then
        for $i' = 1..n$
          $mymem[i'] \leftarrow MEM[i'][j']$
        $(val, steps) \leftarrow VMAX(mymem)$
        $(w, k') \leftarrow CONSENSUS_{(n-1)}[j][steps[j]](val, steps)$
        $state[j] \leftarrow trans_j(state[j], w)$
      else if $nextop_j(state[j]) = (OUTPUT, o)$ then
        if $\neg decided$ **output** o; $decided \leftarrow true$
      $steps[j] \leftarrow steps[j] + 1$

Assume first that all simulators are correct

14 Message passing

In this model we send/receive messages with asynchronous communication (each message sent by a process is eventually received by a correct process). But still a process may crash (stop the execution) and we define p as correct if it doesn't crash (and t-resiliency means that at most t processes may crash).

14.1 Reliable Broadcast

It's a system with two primitives : Rbcast and Rdeliver. We redefine the notions as follows :

- **Agreement**: if p correct Rdeliver m then every correct process Rdeliver m.
- **Validity**: if p correct Rbcast m then p Rdeliver m.
- **Integrity**: if p Rdeliver m then there is a process q that has Rbcast m.

Algorithm for process p:

To execute Rbcast(m)
  send (m) to p

Rdeliver(m) occurs when
  upon receive(m) **do**
    if has not previously executed Rdeliver(m)
    then
      send (m) to all
      Rdeliver(m)

14.2 Atomic Broadcast

Still two primitives ABcast and ABdeliver. All the RBcast properties plus a **total order** : If p and q both ABdeliver m and m' then if p ABdelivers m before m' implies that q ABdelivers m before m'.

ABCast is "universal": very informally, it can be considered as a state machine replication: for any sequential state machine A, we have $t + 1$ processes simulating A, each request is made by atomic broadcast. And this gives us a t-resilient implementation of A.

Consensus in message passing is a decision algorithm such that:

- **Agreement**: if p and q decide, the decide the same value.
- **Validity**: if p decides v then v is an initial value of some processus.
- **Termination**: all correct processes decide.

From Consensus and Reliable broadcast to Atomic Broadcast

Algorithm for process p:

Initialization:
$RDelivered := \emptyset$
$ADeliveres := \emptyset$

To execute Abcast(m)
Rbcast(m)

Adeliver(_) occurs when
upon Rdeliver(m) **do**
$RDelivered := RDelivered \cup \{m\}$

do forever
$AUndelivered := RDelivered - ADelivered$
if $AUndelivered \neq \emptyset$ **then**
$k := k + 1$
$propose(k, AUndelivered)$
wait for $decide(k, msgSet)$
$batch(k) := msgSet - ADelivered$
A-deliver all messages in $batch(k)$ in some deterministic order
$ADelivered := ADelivered \cup batch(k)$

As a conclusion, Atomic broadcast and consensus are equivalent in message passing with crash failure. In other words, Consensus is "universal" in message passing with crash failure.

Message passing can implement a single-writer single-reader register, but to be corrects it needs to have a majority of correct processes. Here is the implementation and after that the proof that a majority of correct process is needed :

For the writer
to write(v)
$seq := seq + 1$
send (W, v, seq) to all
wait until receiving $\lfloor n/2 \rfloor + 1$ messages (ACK, seq)

For the reader
to read()
send(R) to all
wait until receiving $\lfloor n/2 \rfloor + 1$ messages (V, v, s) such that $s > seq$
return val
such that (V, val, S) has been received
and S is the max of the sequence number of received V messages

For all processes
when (W, v, s) is received
if $s > seq$ then
$val := v; seq := s$
send (ACK, s)

when(R) is received
send (V, val, seq) to p_r

- partition argument:
- if n≤2t then we can partition the set of processes in two set S_1 and S_2 such that $|S_1| \geq t$ and $|S_2| \geq t$.
- Run A_1: all processes in S_1 are correct and all processes in S_2 are initially dead, p_0 invokes a write(1), at some time t_0 the write terminates
- Run A_2: all processes in S_2 are correct and all processes in S_1 are initially dead, p_1 S_2 invokes a read(), at time t_0 +1 the read terminates at time t_1
- Run B: « merge » of A_1 and A_2 but no process crash. write(1) terminates before the read() and the read return 0

contradiction