Shared memory and Consistency Models

Algorithmique répartie avancée - ARA
Master2

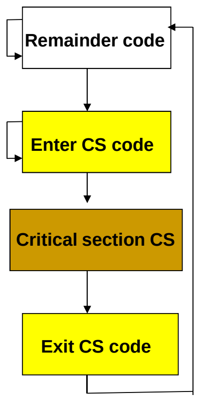
Luciana Arantes

Shared Memory

- **Shared Memory abstractions are programming abstractions that encapsulate read-writes forms of storage among processes**
  - ***Motivation***: programming with shared memory model is considered easier than with message passing.
- **In shared-memory model, processes access concurrently data objects or memory location.**
- **Shared Memory variables are usually called *read-write registers***

Shared memory mutual Exclusion

- **Mutual exclusion problem** [Taubenfeld06]
  - Two requirements should be satisfied:
    - ***Safety*** : no two processes (threads) are in the critical section (CS) at the same time.
    - ***Deadlock- free***: if a process (thread) is trying to enter its CS, then some process (thread), eventually enters its CS.
      - Guarantees global progress property.
      - Does not prevent starvation.
  - Some algorithms provide a third property which ensures lack of starvation
    - ***Starvation-freedom***: if a process is trying to enter its CS, then this process must eventually enter its CS.



Mutual exclusion: Peterson algorithm

- **Two processes**
- **The algorithm uses two shared variables: *flag[2]* and *turn*.**
  - **- A *flag* value of 1 indicates that the process wants to enter the CS.**
  - **- The variable *turn* holds the ID of the process whose turn it is.**
- **The algorithm is starvation-free**

Shared Variables:
boo flag[2] = {0,0}; int turn;

***P_0*:**

```
flag[0] = 1; turn = 1;
while (flag[1] && turn==1);
// critical section
  ...
// end of critical section
flag[0] = 0 ;
```

***P_1*:**

```
flag[1] = 1 ; turn = 0 ;
 while (flag[0] && turn == 0);
 // critical section
   ...
// end of critical section
flag[1] = 0;
```

Mutual exclusion: Lamport's Bakery Algorithm

- *N* **processes**
- **Starvation-free: satisfies mutual exclusion in first-come first-served**

Shared Variables:

```
boo choosing[n];
int timestamp[n];
```

Initialization:

```
choosing[1..n] := 0;
timestamp[1..n] := 0;
```

Entry CS Code:

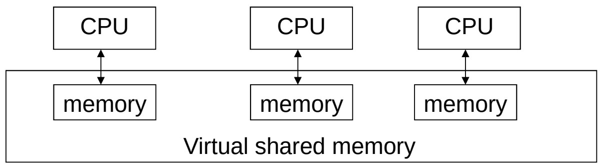
```
choosing [i] := 1;
timestamp[i] := 1 + max_1..n(timestamp[k]);
chosing[i] := 0;
for j := 1 to n do {
   await(chosing[j]=0 );
   await (timestamp[j] == 0 or (timestam[i],i < timestamp[j],j)) ;
```

Exit CS Code:

```
timestamp[i] := 0;
```


Read-write Registers

- **A register is an abstraction of shared variable**
- **Implementation**
  - Provided by multiprocessors machine at hardware level
    - Array of hardware shared registers
  - Can also be implemented over processes that communicate through message passing and do not share any shared device [Guerraoui and Rodrigues 06], [Kshemkalayani and Singhal 08]
    - Shared memory emulation (distributed shared memory)



Read-write Registers

- **Store values that are accessed by *read* and *write* operations**
  - Process/Threads exchange information by invoking these operations
    - RW registers used for process/thread communication and synchronization

RW Registers [Lamport 86]

- **Definition 1: A RW register *x* is characterized by two operations:**
  - ***write*** *(x,v)* $\longrightarrow$ *ok*: writes value *v* to register *x* and returns *ok*
  - **read**(x) $\longrightarrow$ *v*: reads the register *x* and returns its value *v*.
- **Definition2 (Precedence) : for two operation o_1 and o_2, we say that:**
  - o_1 *precedes* o_2 whenever o_1 returns before o_2 is invoked (*sequential*)
  - o_1 is *concurrent* with o_2 when neither operation precedes the other one.

RW Registers [Lamport 86]

- **If a register is used by a single process, and we assume that there is no failure, we can define the following properties:**
  - **Safety**: Every read returns the *last* value written
  - **Liveness**: every operation eventually completes
- **Concurrency:**
  - In practice execution is not sequential
  - What is the meaning of "a read returns the last write" if both operations are concurrent?
    - It depends on the ***semantics*** of concurrent accesses offered by the register.
      - Safe, regular, and atomic registers.

Memory consistency model

- **Memory coherence is the ability of the system to execute memory operations correctly.**
  - Considering all the possible interleaving of operations issued by concurrent processes/threads, ensuring memory coherence becomes identifying which of these sequences of interleaving are correct.
  - Memory consistency model defines the sets of allowable memory access ordering.

Linearizibility

- **Consistency criteria for ordering concurrent accesses**
  - All operations appear to be executed atomically and sequentially
  - A global time scale needs to be simulated.
    - All processes (threads) need to agree on a common total order.
- **Other consistency model more relaxed**
  - Sequential consistency, causal consistency, PRAM, weak, etc
    - Discussed later

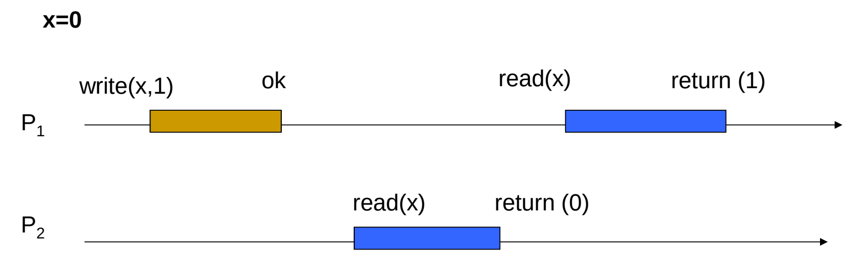
Linearizibility

- **For any concurrent execution, there is a total order of the operations such that each read to a location (variable) returns the value written by the last write to this location (variable) that precedes it in the order.**
- **This total order must be consistent with the temporal order of operations**
  - If one operation finishes before another begins, the former must precedes the latter in the total order.
    - Respect of the order of non overlapping operations.
  - For operations that overlap, all the processes/threads see the same ordering of events, which is equivalent to the global time occurrence of non overlapping events.
    - All operations appear to be executed atomically and sequentially.

Linearizibility

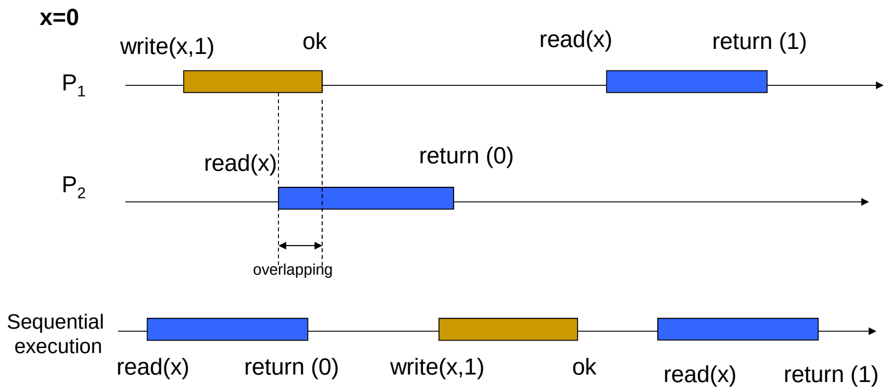
- **Each operation *op* (*Read* or *Write*) has an *invocation* and *response* events.**
  - An execution in global time is viewed as sequence *Seq* of such invocations and responses.
    - A *Seq* is linearizable is there is a permutation *Seq*' such that:
      - For every variable *v*, Seq'_v is such that each *Read* returns the most recent *Write* that immediately preceded it.
      - If the response of op_1 occurred before the invocation of op_2 in *Seq*, then op_1 occurs before op_2 en *Seq'*.

Linearizibility



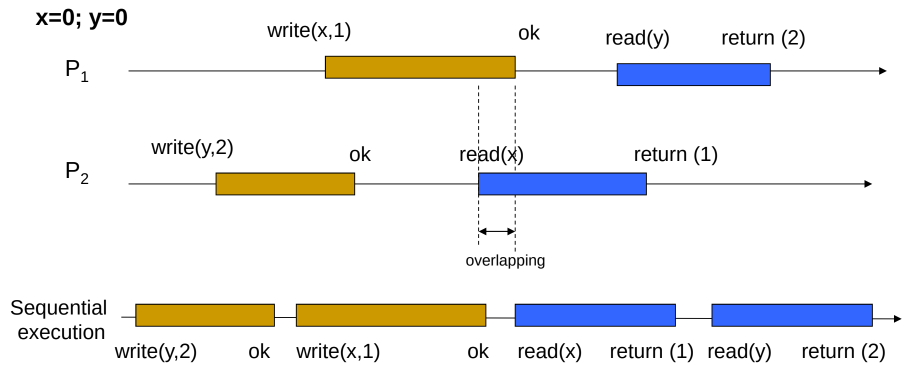
Not linearizable!!

Linearizibility



Linearizable

Linearizibility



Linearizable

Implementing Linearizibility on Message Passing

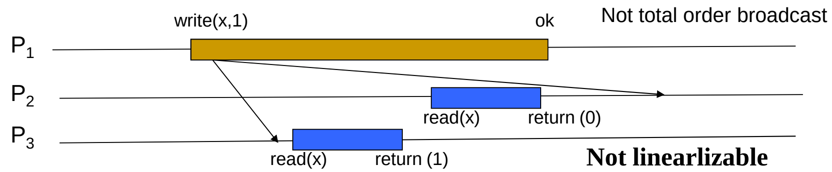
```
int x ; /*local variable */

upon operation(op,val)  from application
   /* Read or Write */
   total_order_broadcast  (op, val, id_i);

upon deliver of message <read, val, id>
   if (id = id_i)
      /* own request which was broadcast */
      return x;

upon deliver of message  <write, val, id>
   x=val;
   if (id = id_i)
      /* own request which was broadcast */
      return ack to application
```

Reads must also participate in the total order broadcast

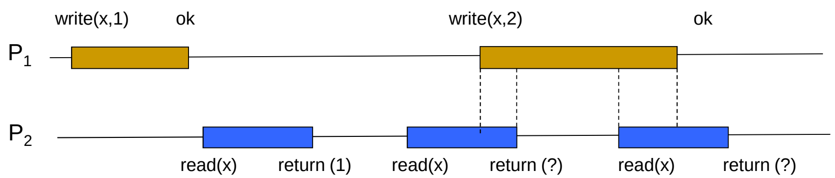


Types of Registers

- **Semantics of *Read* and *Write* operations under concurrent accesses.**
  - In the face of concurrent *read* and *write* operations, the value returned by a *read* is unpredictable.
    - The order of access depends on the properties of the register
    - Implicit assumption of a global time
  - Three types of registers [Lamport 86]:
    - Safe
    - Regular
    - Atomic

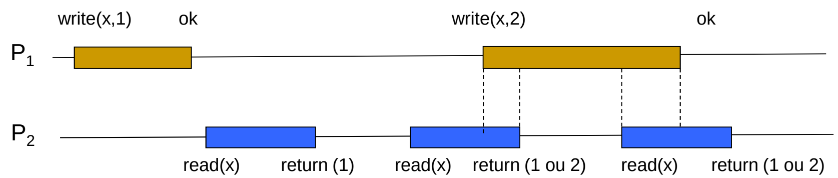
Safe register

- ***Read* does not overlap with a *write***
  - *Read* returns the most recently written value.
- ***Read* overlaps with a *write***
  - *Read* returns any value that the register could possibly have.



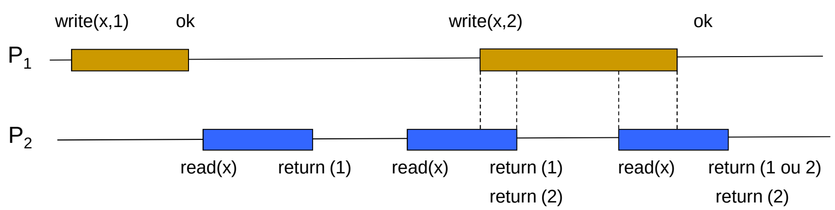
Regular Register

- **It is a *safe* register and**
  - If *read* overlaps with a write
    - *Read* returns either the most recently value or a concurrently written value.



Atomic Register

- It is a ***regular*** **and**
  - *read* and *write* that overlap are *linearizable*
    - There exists an equivalent totally ordered sequential execution of them.



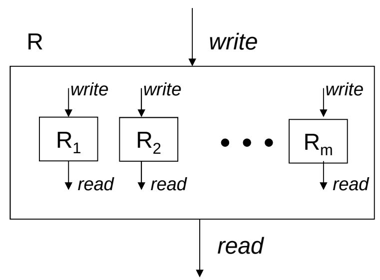
Characteristics of Registers

24 types of registers

- **Semantics**
  - Safe, regular, atomic
- **Value**
  - Binary, integer
- **Write accesses**
  - Single-writer (SW), multi-writer (MW)
- **Read accesses**
  - Single-reader (SR), multi-reader (MR)

Register Construction

- **Design a more complex register using simpler registers.**



m individual registers

Construction of MRSW safe (regular) register with SRSW safe (regular) registers

- ***n*** **SRSW safe (regular) registers:** $R_1, \dots R_n$
- **The single writer is process** P_0 **and the** ***n*** **readers are** $P_1 \dots P_n$.
  - Multiple readers are not allowed to access the same SRSW safe (regular) register
    - A reader P_i can read only SRSW register R_i (the only reader)
      - Data must be replicated
  - P_0 can write to the n registers (the only writer)
    - P_0 writes the same value to the n registers

Construction of MRSW safe (regular) register with SRSW safe (regular) registers (cont.)

When a *read* by *Pi* and a *write* by P_0 do not overlap at *Ri*, the *read* returns the correct value; otherwise:

- ***safe***: the *read* returns a legitimate value.
- ***regular***: the *read* returns either the earlier value or the value being written.

SRSW safe (regular) registers $R_1 \ldots R_n$

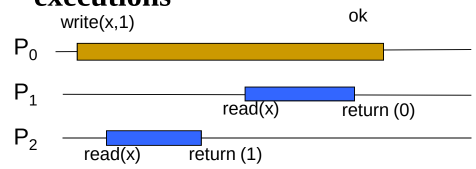
Single writer: P_0
multiple reader: $P_1,\ldots P_n$

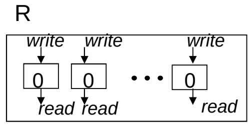
```
Write(val)
    for i=1 to n
        Ri = val.
```

```
Read (val)
    val = Ri
    return val
```

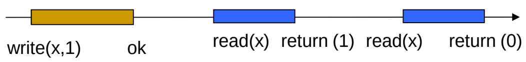

Construction of MRSW atomic register with SRSW atomic registers

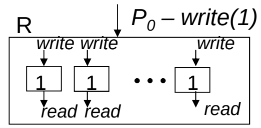
- ***n* SRSW atomic registers: $R_1,\ldots R_n$**
- **The single writer is process P_0 and the *n* readers are $P_1\ldots P_n$.**
- **The previous solution does not always ensure linearizable executions**





Non linearizable execution :
- P_1 reads R_1 before the write
- P_2 reads R_2 after the write



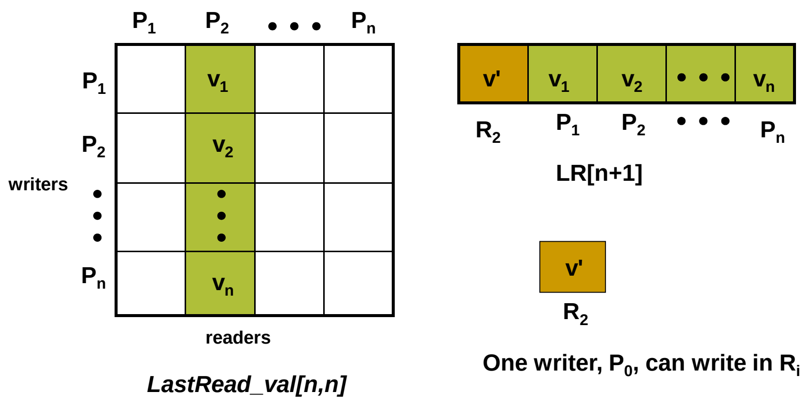


Construction of MRSW atomic register with SRSW atomic registers (cont.)

- **Solution [Israeli and Li 93]**
  - ***Read***: P_i must chose a value among R_i and the values that the other processes have last read;
    - *LastRead_val [n,n] of <data,seq>*: nxn SRSW atomic register that provides such information.
      - *LastRead_val [i,j]:* value of P_i's last returned read which was informed to P_j.
    - Before returning the value, the reader informs the other processes of the returned value.

Construction of MRSW atomic register with SRSW atomic registers

π= {P_0,P_1, P_2 … P_n}



Construction of MRSW atomic register with SRSW atomic registers (cont.)

SRSW atomic registers of type <data,seq>: R_1 … R_n

SRSW atomic registers of type <data,seq>: LastRead_val[n,n]

Local Variables:
int seq, j, latest;
<data,seq> LR[n+1]; /* last returned read value of other processes*/

Write(val)
seq++;
for j=1 **to** n
R_j = <val,seq>.

Read (val)
LR[0]=R_i
for j = 1 **to** n
/* get latest value stored for P_i by P_j */
LR[j]=LastRead_val [j,i]

find max **such that for all** latest<> k
LR[latest].seq >= LR[k].seq;

for j = 1 **to** n
LastRead_val [i,j] = LR[latest];

val = LR[latest]
return (val)

Construction of MRSW regular register on message passing system with crashes

- **Emulation of a MRSW regular register**
  - One specific process P_0 can invoke a *write* operation and any other can invoke a *read* operation on the register.
- **Use of a perfect failure detector**
- **Each process stores a copy of the current register value in a variable that is local to a process**
- **Read-one Write-all algorithm:**
  - The writer updates the value of all processes which it does not detect as faulty.
    - All processes acknowledge the receipt of the new value.
    - *Write* completes when all acknowledges from correct process is received.
  - The reader just return the value stored locally.

Construction of MRSW regular on message passing system with crashes

Local Variables:
writeSet= Ø;
reg = 0;
correct = Π

Read ( )
return reg;

Write(val)
Bebbroadcast <val>

Upon event Bebdelivery <val,j>
reg = val;
send <j,ack>

Upon event crash <j>
correct = correct / {j}
if (correct *C*= writeSet)
writeSet= Ø;
return <ok>;

Upon event reception <ack,j>
writeSet = writeSet U {j}
if (correct *C*= writeSet)
writeSet= Ø;
return <ok>;

Atomic Operations

- **Read, write**
- **Examples of other atomic operations:**
  - ***test-and-set (r:register; val:value): value***
    - The value *val* is assigned to *r*, and the old value of *r* is returned.
  - ***read-modify-write (r:register; f:function):value***
    - The value of *f(r)* is assigned to *r*, and the old value of *r* is returned.
  - ***compare-and-swap (r:register; key, new :value):value***
    - If the current value of *r* is equal to *key*, then the value of *r* is set to *new* ; otherwise *r* is not changed. The old value of *r* is returned.
  - ***fetch-and-add (r:register; val:value): value***
    - The value of *r* is incremented by *val* and the old value of *r* is returned
  - ***enqueue (Q:queue,val:value) ; dequeue (Q:queue):value***
    - Operations for a FIFO queue object.
  - ...

Mutual exclusion using test-and-set

- *N* processes
- Not starvation-free

```
function T&S (r:register, val:value): value;
   /* atomic*/
    temp = r;
    r=val;
    return (temp);
```

```
Shared Variables:
   register reg =false;

Local Variable:
   blocked;

Entry_CS ( )
  blocked=true;

  repeat
      blocked=T&S(reg,true);
  until blocked=false;

Exit _CS ( ):
  reg=false;
```


Shared Atomic Objects

- **A shared atomic object is a data structure exporting a set of operations that can be invoked concurrently by the processes (threads) of the system.**
  - Each object has a type which defines the set of operations (primitives, methods) that the object supports.
    - Object is accessed only by using such operations
    - Each object has sequential specification that specifies how the object behaves when these operations are applied atomically.
  - There are objects which have more synchronization power than atomic *Read/Write* registers.

Examples of atomic shared objects

- **Registers**
- **Test-and-Set object**
  - A shared register that supports *write* and *test-and-set* operations.
- **Read-modify-write object**
  - A shared register that supports *read-modify-write* operation.
- **Compare-and-swap object**
  - A shared register that supports *compare-and-swap* operation.
- **Queue**
  - A shared register that supports *enqueue* and *dequeue* operations.
- **….**

Registers : Failure Issues

- **If a process (thread) can fail by crashing, the operation invoked by it might not complete.**
  - If a process (thread) invokes a *write* and crashes, the *write* is considered to be concurrent with any *read* that did not precede it.
- **Any process (thread) that invokes a *read* or *write* operation and does not crash eventually returns from this invocation.**

Synchronization of operations

- **Blocking operations**
  - **A delay or crash of a process (thread) can prevent others from making progress.**
    - **e.g. Mutex, producer-consumer, etc.**
- **Non blocking operations**
  - **A delay or crash of a process (thread) can not prevent others from making progress.**
    - **Processes (threads) competing for a shared resource do not have their execution indefinitely postponed by other processes.**

Non blocking operations

- **Wait-freedom**
  - An operation on a shared object is *wait-free* if every invocation of the operation completes in a finite number of steps regardless of the number of steps taken by any other process.
    - A concurrent object is *wait-free* if all its operation is *wait-free*
    - Wait-freedom provides robustness and ensures per-process (per-thread) progress.
    - A process (thread) does not depend on other process (thread) , and its execution is does *wait-free*.
    - If *n* is the number of processes (thread) , *n-1* processes (threads) can crash
      - A wait-free algorithm in a system with *n* processes (threads) is a ***(n-1)-crash resilient*** algorithm.

Wait-free shared memory consensus

- **Consensus is impossible in an asynchronous shared memory system in the crash failure model.**
  - Extended from the impossibility in message-passing (FLP)
    - Shared memory can be emulated by message passing
  - In the face of a potential crash it is not possible to distinguish between a crashed process and a process which extremely slow in doing its *Read* or *Write* operation.

Wait-free shared memory consensus

- **There are objects for which there is a *wait-free* algorithm for reaching consensus in a *n-processes* system.**
  - Objects that provide stronger synchronization than safe, regular and atomic objects.
    - e.g. test-and-set objects, compare-and-swap objects, FIFO queue objects, etc.

Consensus Number

- **An object of type *X* has consensus number *k*, if *k* is the largest number for which the object *X* can solve wait-free *k*-process consensus in an asynchronous system subject to *k-1* crash failures, using only objects of type *X* and *read/write* objects.**

Consensus number of some objects

(∞)	Compare&Swap		
	…		
(3)			
(2)	FIFO Queue	Test&Set	Fetch&Add
(1)	Register		

Wait-free consensus : Test-and-set object

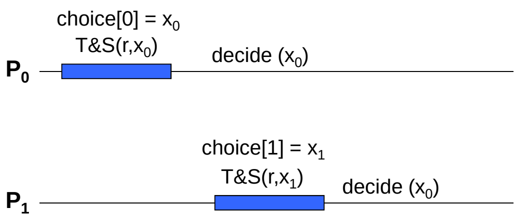
- **Two processes**

x_i: initial choice

Shared Variables :
test-and-set reg r = ⊥;
r/w reg choice[2] ={⊥, ⊥};

Local Variables:
int val;

choice[i] =x_i;
val= T&S(r,x_i)
if (val = ⊥)
decide (x_i)
else
decide (choice[(i+1)%2);



It does not work with more than 2 processes: which value to chose from *choice[ ]* ?

Wait-free consensus : FIFO Queue object

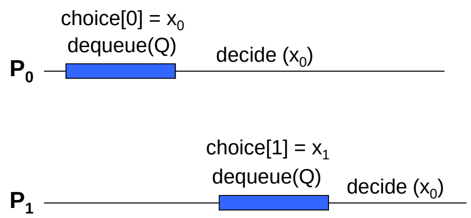
- **Two processes**

x_i: initial choice

Shared Variables :
queue reg Q= <0> /* Q initializé */
r/w reg choice[2] ={⊥, ⊥};

Local Variables:
int temp;

choice[i] =x_i;
temp= dequeue(Q) ;
if (temp == ⊥)
/* queue empty*/
decide (choice[(i+1)%2]);
else
decide (x_i)



Wait-free consensus : Compare-and-set object

• ∞ processes

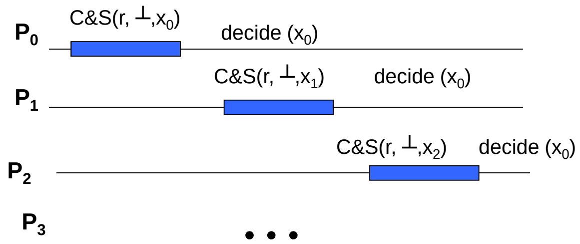
x_i: initial choice

Shared Variables :
compare-and-swap reg;

Local Variables:
int temp;

temp= C&S(reg, ⊥, x_i);
if (temp = ⊥)
decide (x_i)
else
decide (temp);

```
function C&S (r:register; key, new: value): value;
  /* atomic*/
   temp = r;
   if (key=temp) r=new;
   return (temp);
```



Memory Consistency Models

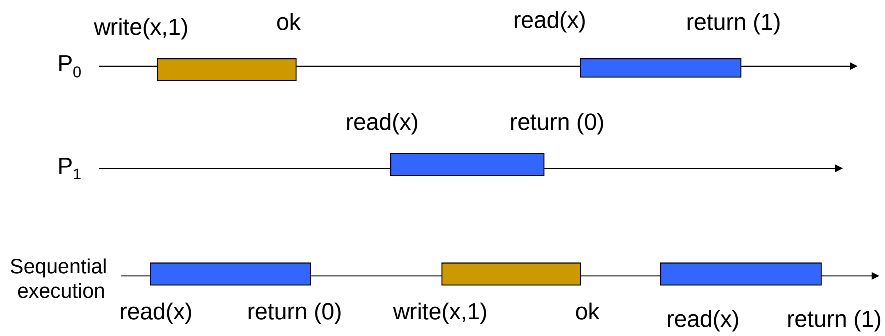
- **Linearizability = Strict or atomic consistency:**
- **Other models more relaxed :**
  - Sequential consistency
  - Causal consistency
  - PRAM consistency
  - Consistency models based on synchronization variables
    - Weak
    - Entry consistency
    - Release
    - Lazy release

Sequential Consistency

- The result of any execution is the same as if all operations of the processors were executed in some sequential order.
- The operations of each individual processor appear in this sequence in the local program order.

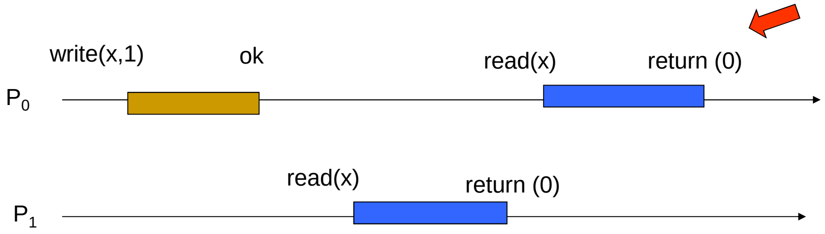
• Any interleaving of the operations from the different processes is possible. But all processors must see the same interleaving.
• Even if two operations from different processes (on the same or different variables) do not overlap in a global time scale, they may appear in reverse order in the common sequential order seen by all.

Sequential Consistency



Sequentially consistency !

Sequential Consistency

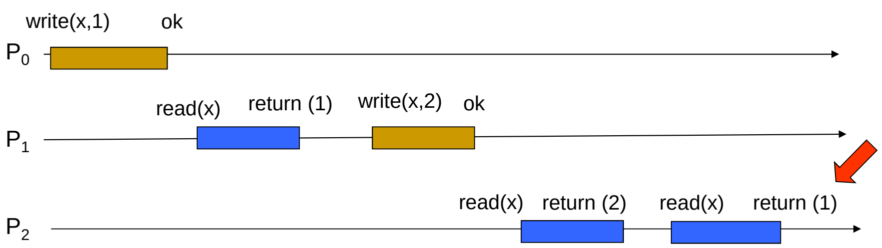


Not Sequential consistency !

Causal Consistency

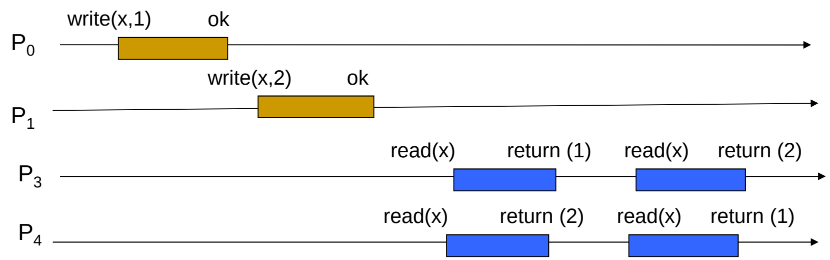
- **Only *writes* that are *causally related* must be seen by all processes in the same order. Concurrent writes may be seen in a different order.**
  - The causal relation is defined as:
    - Local order of events of a processes define local causal order.
    - A *write* operation causally precedes a *read* operation of another process if the *read* returns the value written by the *write* operation.
    - The transitive operation of the above two relations defines the global causal order.

Causal Consistency



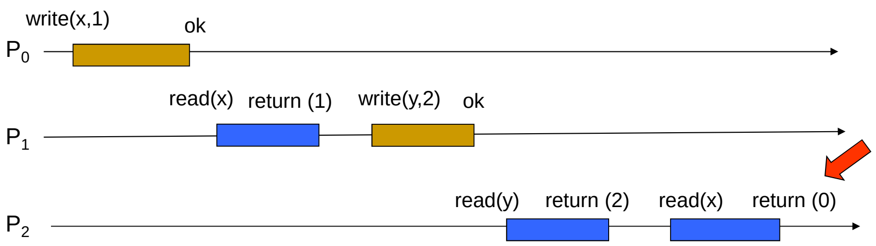
Not causal consistency !

Causal Consistency



Causal consistency !
The writes are concurrent

Causal Consistency

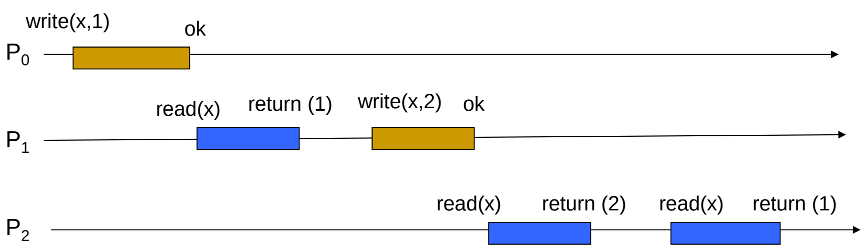


Not causal consistency !

PRAM Consistency

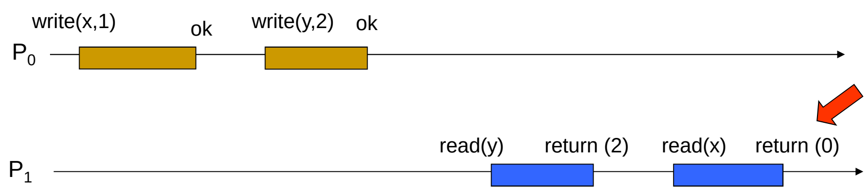
- ***Writes* done by a single processor are received by all other processes in the order in which they were issued but *writes* from different processes may be seen in a different order by different processes**
  - Only the local causality relation needs to be seen by other processes.
    - Writes from the same processes must be seen by the others in order they were issued.

PRAM Consistency



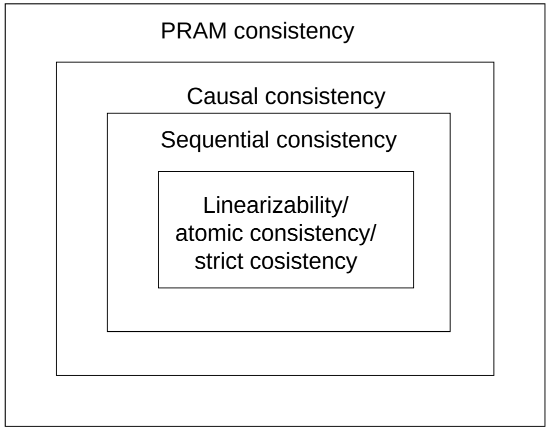
PRAM consistency !

PRAM Consistency



Not PRAM consistency !

Hierarchy of consistency models

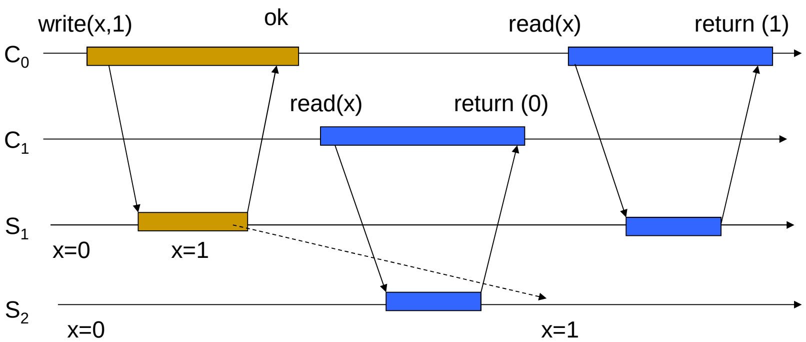


Consistency models: applications

- **Memory consistency models can be applied to other domains**
  - Example: data base systems where data are replicated for fault tolerance and performance reasons on several servers.
    - Clients: c_1, c_2,...
    - Servers: s_1, ..., s_n
    - Operations at client side: **read** and **write**
      - *read(x)* returns the value of data item *x*
      - *write(x,val)* updates the value of x with *val* and returns an acknowledgement
    - An update from a client is broadcast to all servers

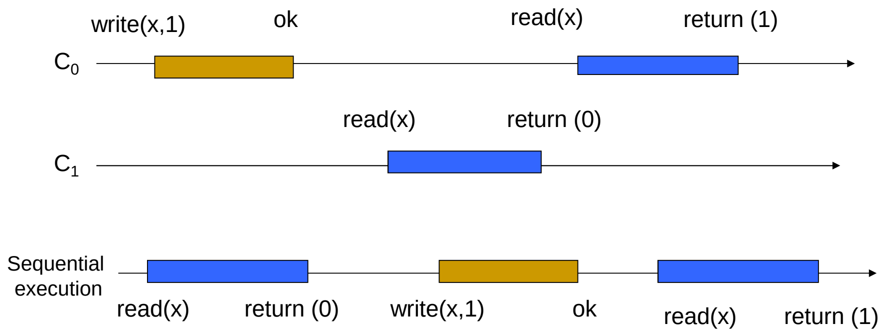
Multiple clients and multiple servers

Is it inconsistent?



Multiple clients and multiple servers

No, if sequential consistency provided



Futex

- **Fast Userspace Mutex**
  - kernel system call that programmers can use.
  - blocking construct in the context of shared-memory synchronization.
  - Shared variable :
    - No contention : the operations are done in user space, otherwise, the need for kernel services is required
  - The *futex()* system call provides two methods for :
    - a program to wait for a value at a given shared address to change,
    - wake up threads/processes waiting on a particular address

Futex ( )

```
#define futex_wait(addr, val)  syscall(SYS_futex, addr, FUTEX_WAIT, val, NULL)
#define futex_wakeup(addr, nb)   syscall(SYS_futex, addr, FUTEX_WAKE, nb)


void LOCK(int *f) {
    int old;
  while (1) {
    old = compare_and_swap(f, 0, 1);  // opération  atomique
    if (old  == 0) // The futex variable was free. Now, it is set to 1
        return;
   else
        futex_wait(f, 1); // LOCK
}
```


Futex ( )

```
void UNLOCK(int *f) {
  fetch_and_sub(f, 1);  // opération atomique
  futex_wakeup(f, 1); // UNLOCK
}

shared int futex_var;

void *thread_exmple (void *p) {

while(1) {
  …
  LOCK(&futex_var);
  ….
  UNLOCK(&futex_var);
}
```


Bibliography

- G. Taubenfeld, *Synchronization Algorithms And Concurrent Programming*, Pearson Prentice Hall, 2006.
- M. Herlihy, and N. Shavit, *The Art of Multiprocessor Programming*, Morgan Kauman Publishers, 2008.
- A. D. Kshemkalyani, and M. Singhal, *Distributed Computing: principles, algorithms, and systems*, Cambridge University Press, 2008.
- R. Guerraoui, and L. Rodrigues. *Reliable Distributed Programming*, Springer, 2006.
- M. Herlihy. Wait-free synchronization. *ACM Transaction on Programming Languages and Systems*, 13(1), 1991, pages 124-149.
- Nancy Lynch, *Distributed Algorithms*, Morgan Kaufman Publishers, 1996.

Bibliography (cont.)

- L. Lamport. On interprocess communication, *Distributed computing*, 1(2),1986, pages 77-85.
- L. Lamport, A new solution of Dijkstra's concurrent programming problem, *Communication of the ACM*, 17(8), 1974, pages 453-455.
- A. Israeli, and M. Li. Bounded timestamps. *Distributed Computing*, 6(4), 1993, pages 205-209.
- M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995, pages 37-49
- R. Lipton and J. Sandberg. *PRAM: a Scalable Shared Memory*. Technical Report CS-TR-180-88, Princeton University, 1988.
- N. Shavit, and D. Touitou. Software Transactional Memory. *PODC* 1995, pages 204-213.