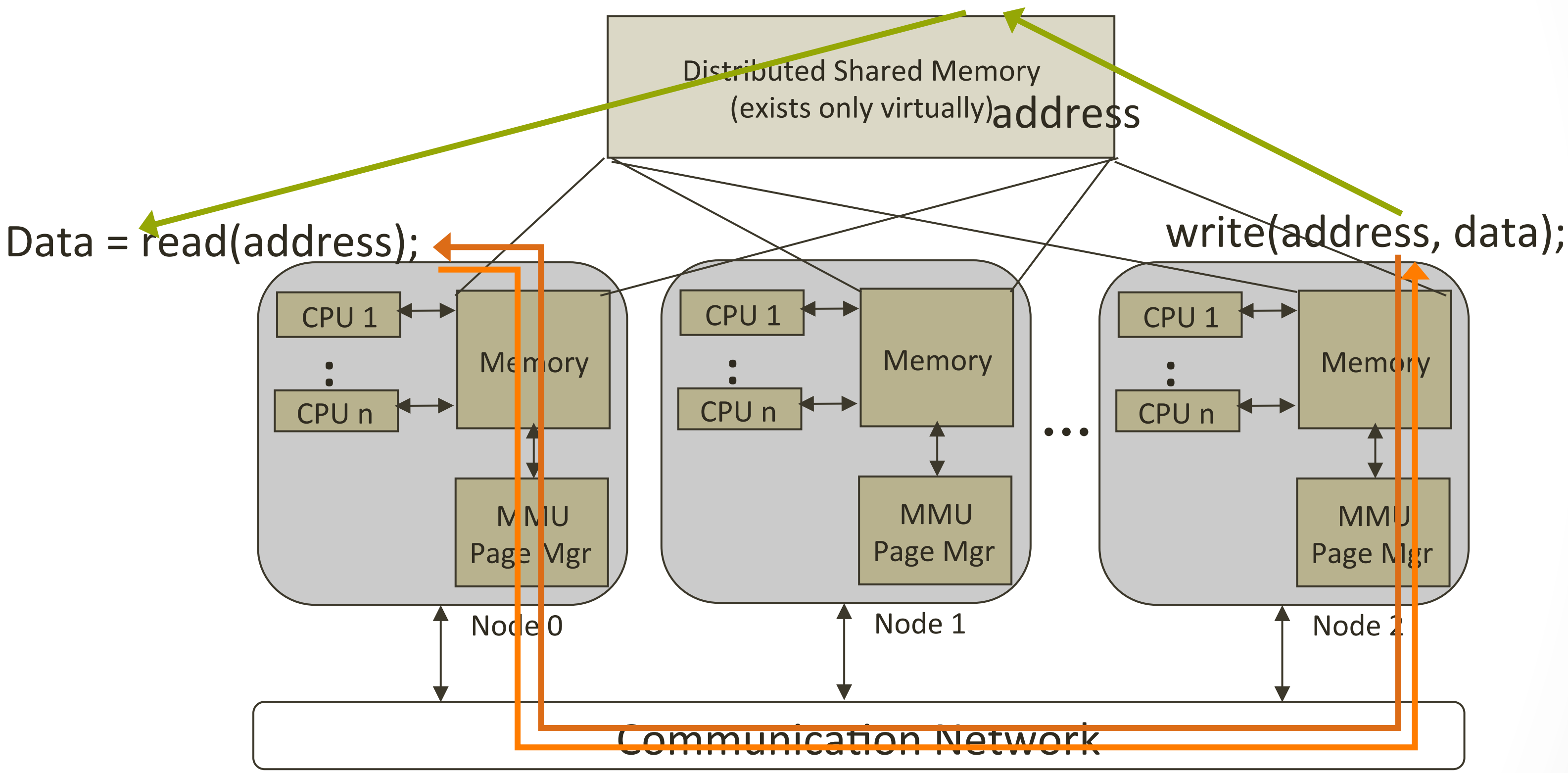
Distributed Shared Memory (DSM)

17-Nov-2015

DSM Basics



A cache line or a page is transferred to and cached in the requested computer.

Simple example

```
struct        shared { int a, b; };
```

Program *Writer*:

```
main()
{
        struct shared *p = (struct shared *)  allocShared(…)
        p->a = p->b = 0;                /* initialize fields to zero */
        while(TRUE)    {                /* continuously update structure fields */
                p –>a = p –>a + 1;
                p –>b = p –>b - 1;
        }
}
```

Program Reader:

```
main()
{
        struct shared *p = ….
        while(TRUE) {  /* read the fields once every second */
                printf("a = %d, b = %d\n", p –>a, p –>b);
                sleep(1);
        }
}
```

Why DSM

- Simpler abstraction
  - Underlying tedious communication primitives shielded by memory accesses
- Better portability of distributed programs
  - Natural transition from sequential to distributed application
- Better performance of some applications
  - Data locality, one-demand data movement, and large memory space reduce network traffic and paging/swapping activities.
- Flexible communication environment
  - Sender and receiver have no need to know each other. They even need not coexist.
- Ease of process migration
  - Migration is completed only by transferring the corresponding PCB to the destination.

Main Issues

- Granularity
  - **Fine** (less false sharing but more network traffic)
    - Cache line (e.g. Dash and Alewife), Object (e.g. Orca and Linda), Page (e.g. Ivy)
  - **Coarse**(more false sharing but less network traffic)
- Memory consistence and access synchronization
  - Strict, Sequential, Causal, Weak, and Release Consistency models
- Data location and access
  - Broadcasting, centralized data locator, fixed distributed data locator, and dynamic distributed data locator
- Replacement strategy
  - LRU or FIFO (The same issue as OS virtual memory)
- Thrashing
  - How to prevent a block from being exchanged back and forth between two nodes.
- Heterogeneity
- Implementation
  - hardware implementation, OS implementation, and User-level implementation.

Consistency Models

Two processes accessing shared variables

At the beginning a = b = 0;

Process 1

```
br := b;
ar := a;
if(ar ≥ br) then
      print ("OK");
```

Process 2

```
a := a + 1;
b := b + 1;
```

Condition satisfied

a == 1
b == 1

Condition satisfied

a == 1
b == 0

b == 1
a == 0

This may happen if new contents are transmitted through a different route.

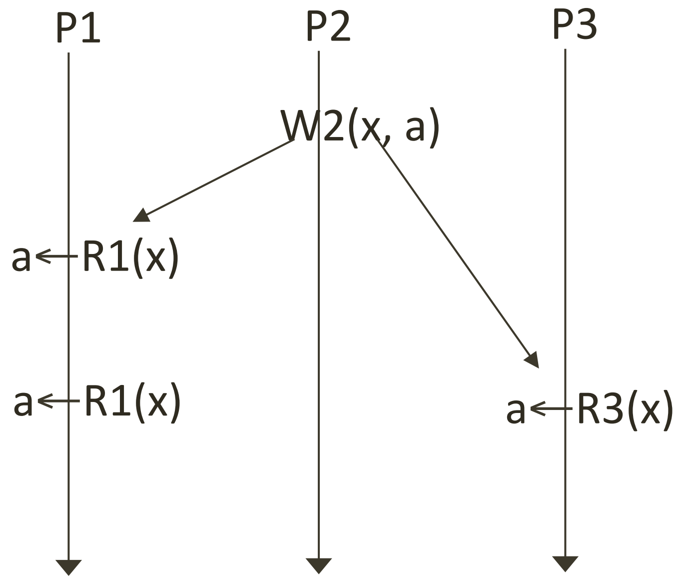
DSM needs a consistency model.

Consistency Models

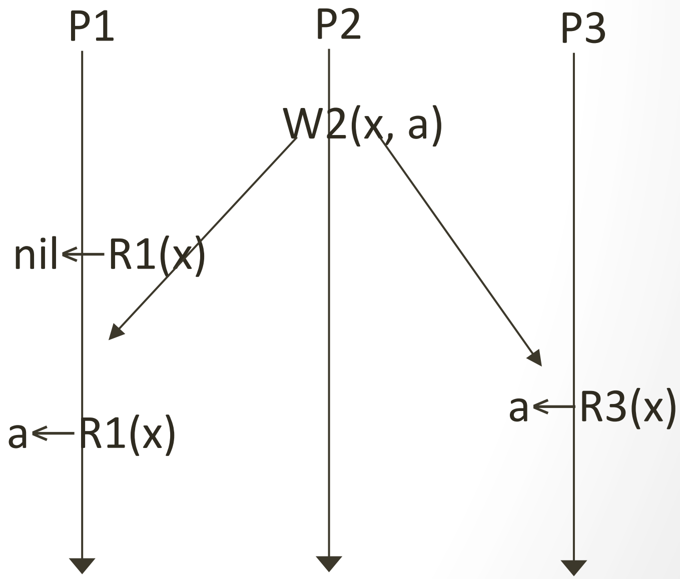
Strict Consistency

- Wi(x, a): Processor i writes a on variable x, (i.e., x = a;).
- b←Ri(x): Processor i reads b from variable x. (i.e., y = x; && y == b;).
- Any read on x must return the value of the most recent write on x.

Strict Consistency



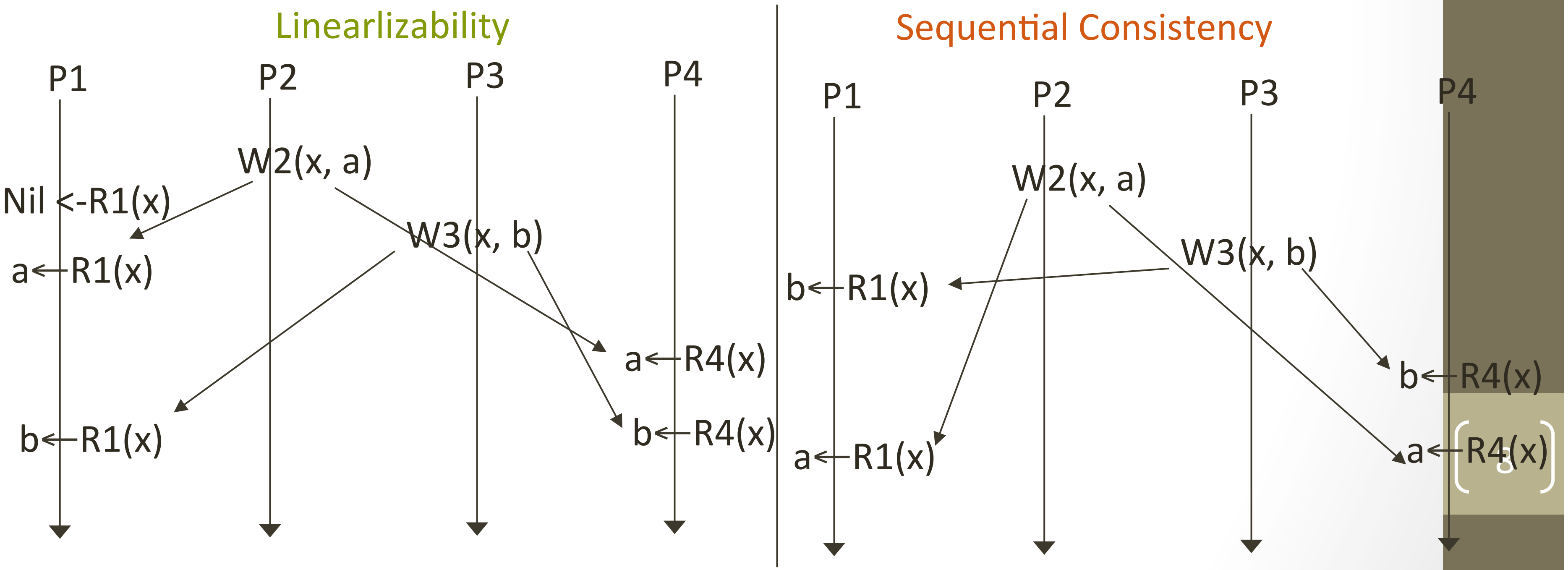
Not Strict Consistency



Consistency Models

Linearizability and Sequential Consistency

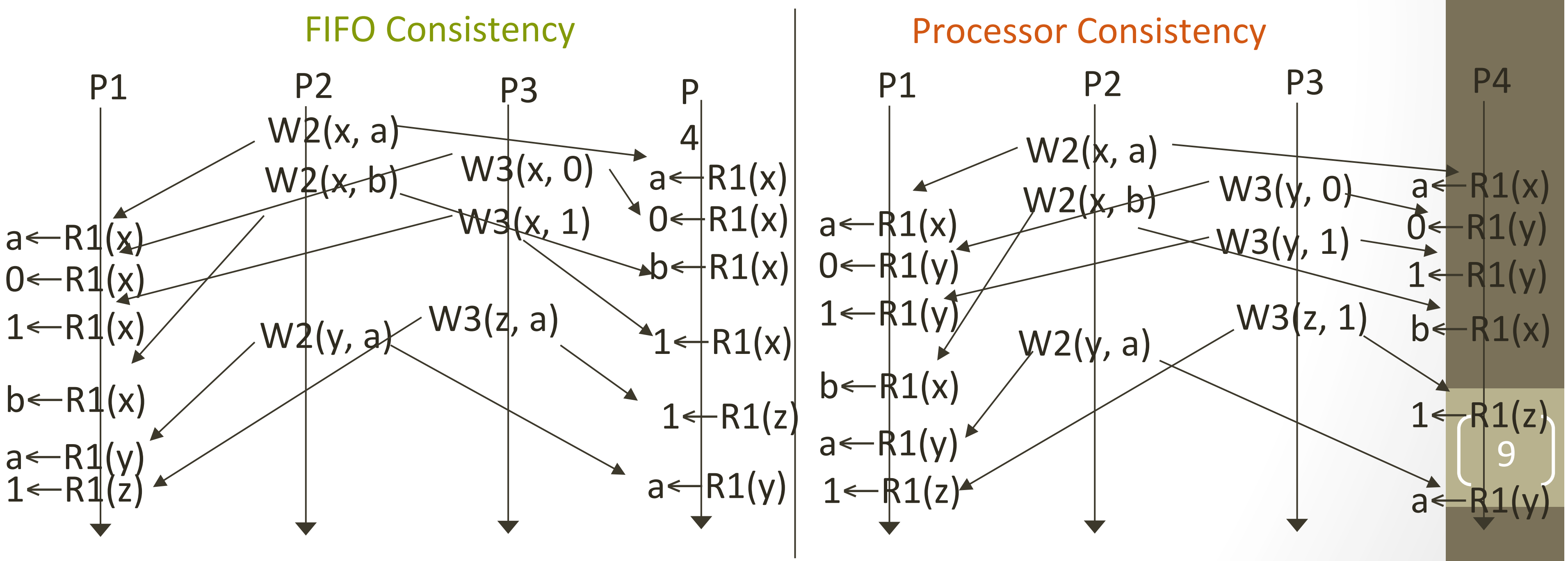
- **Linearlizability:** Operations of each individual process appear to all processes in the same order as they happen.
- **Sequential Consistency:** Operations of each individual process appear in the same order to all processes.



Consistency Models

FIFO and Processor Consistency

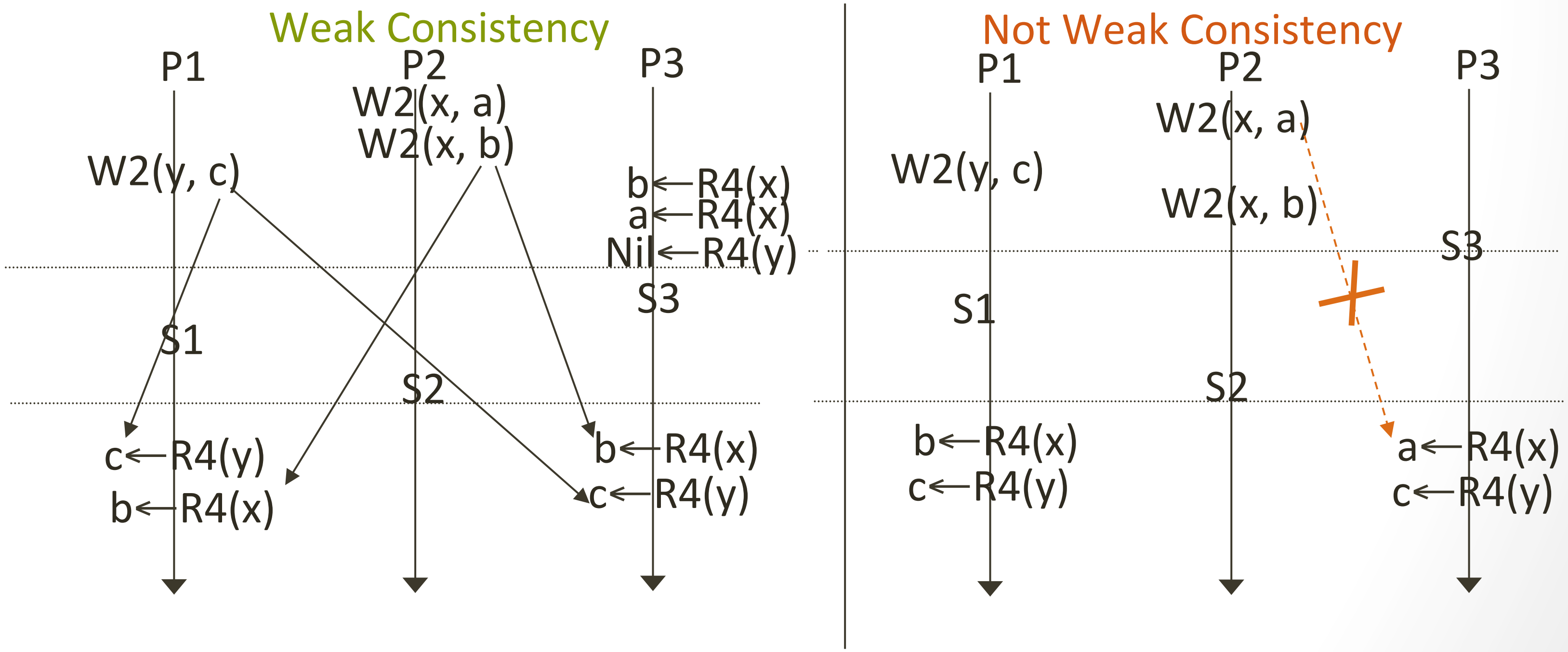
- **FIFO Consistency:** writes by a single process are visible to all other processes in the order in which they were issued.
- **Processor Consistency:** FIFO Consistency + all write to the same memory location must be visible in the same order.



Consistency Models

Weak Consistency

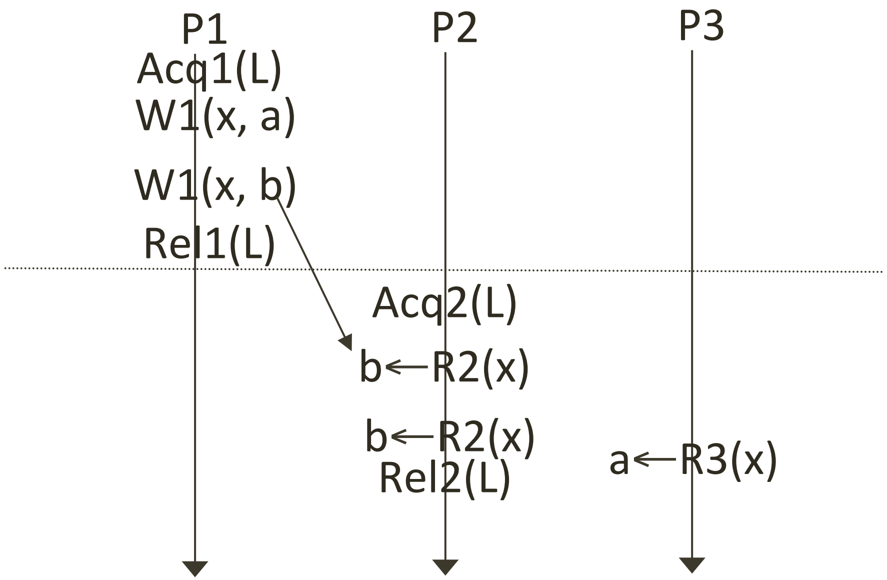
- Accesses to synchronization variables must obey sequential consistency.
- All previous writes must be completed before an access to a synchronization variable.
- All previous accesses to synchronization variables must be completed before access to non-synchronization variable.



Consistency Models

Release Consistency

- Access to acquire and release variables obey processor consistency.
- Previous acquires requested by a process must be completed before the process performs a data access.
- All previous data accesses performed by a process must be completed before the process performs a release.



Consistency Models

Release Consistency (Example)

Process 1:

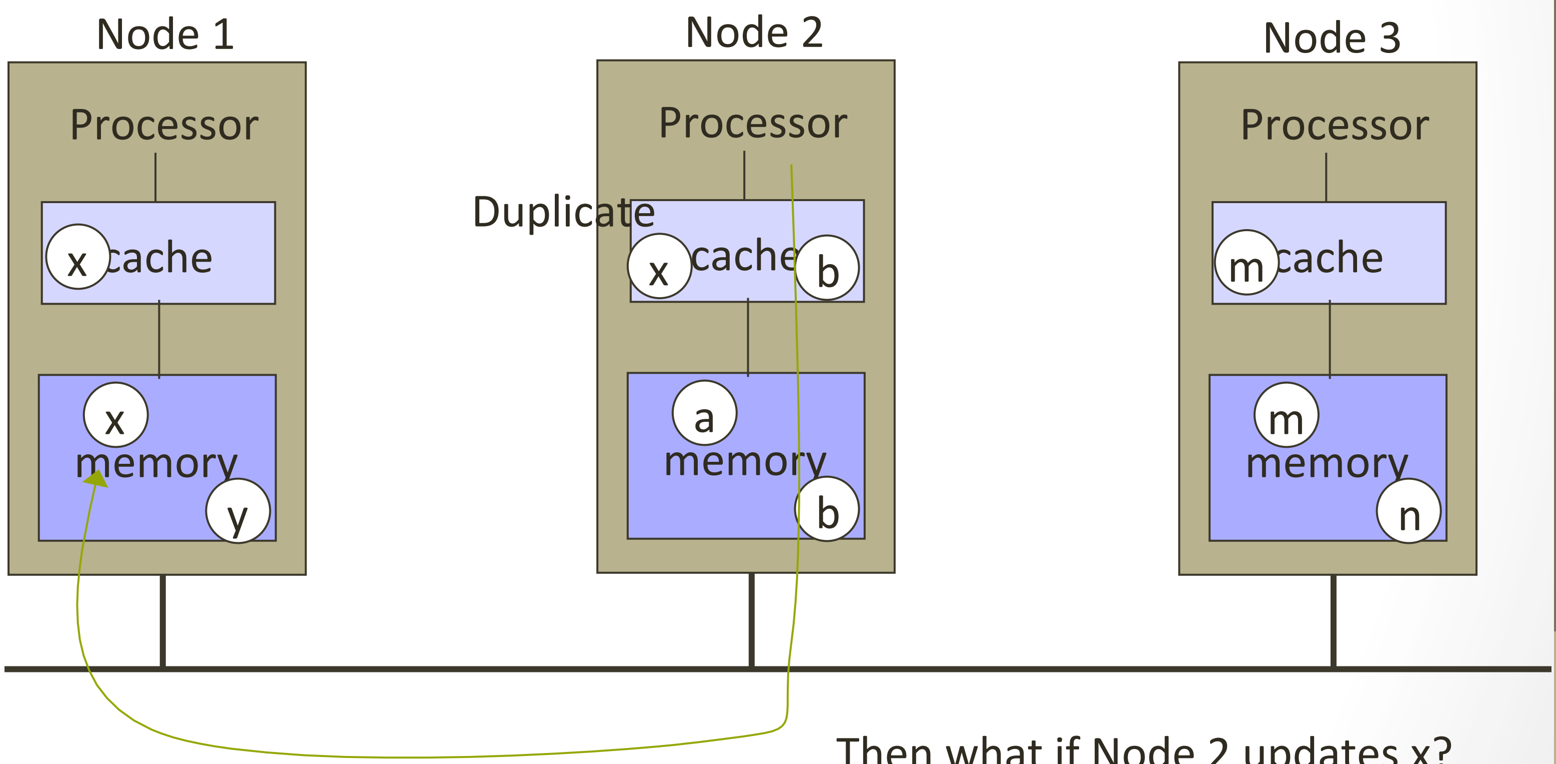
```
acquireLock();                  // enter critical section
a := a + 1;
b := b + 1;
releaseLock();                  // leave critical section
```

Process 2:

```
acquireLock();                  // enter critical section
print ("The values of a and b are: ", a, b);
releaseLock();                  // leave critical section
```

Implementing Sequential Consistency

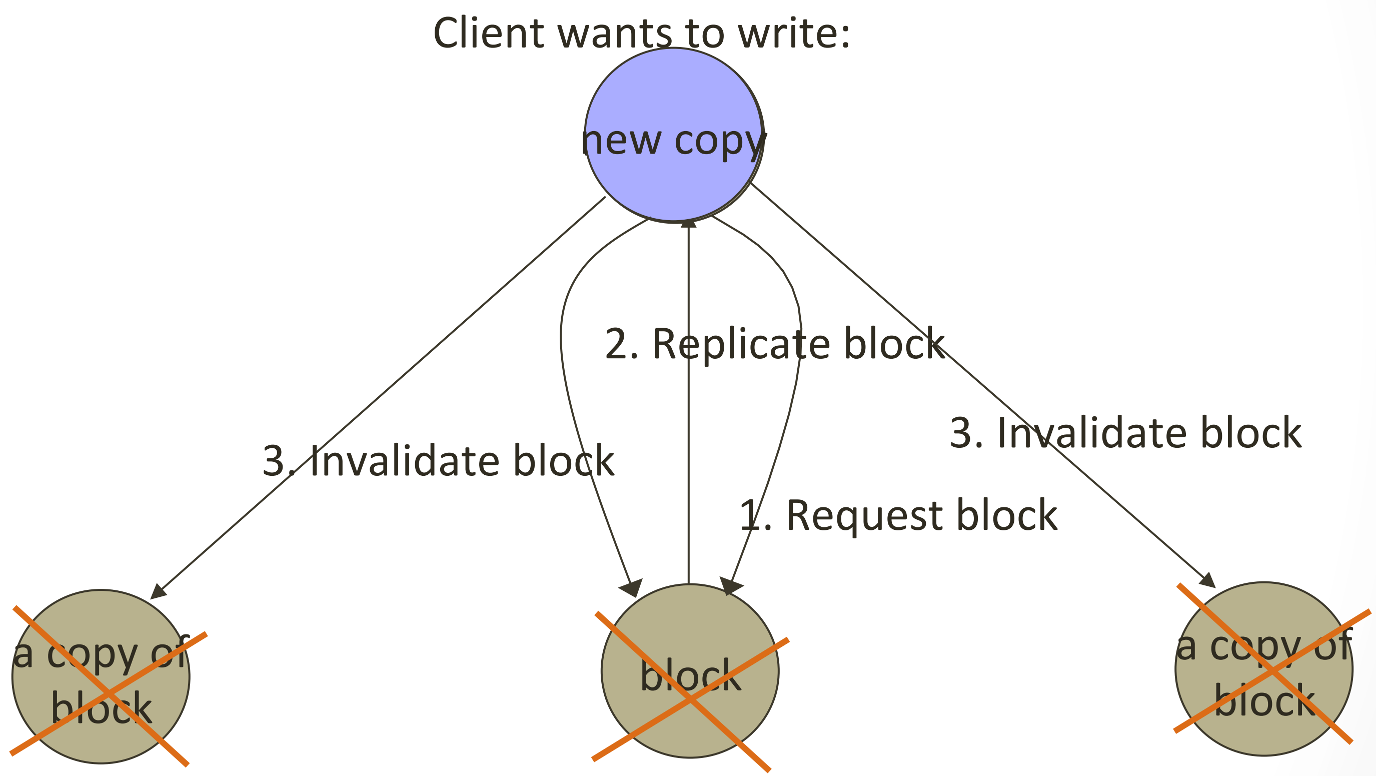
Replicated and Migrating Data Blocks



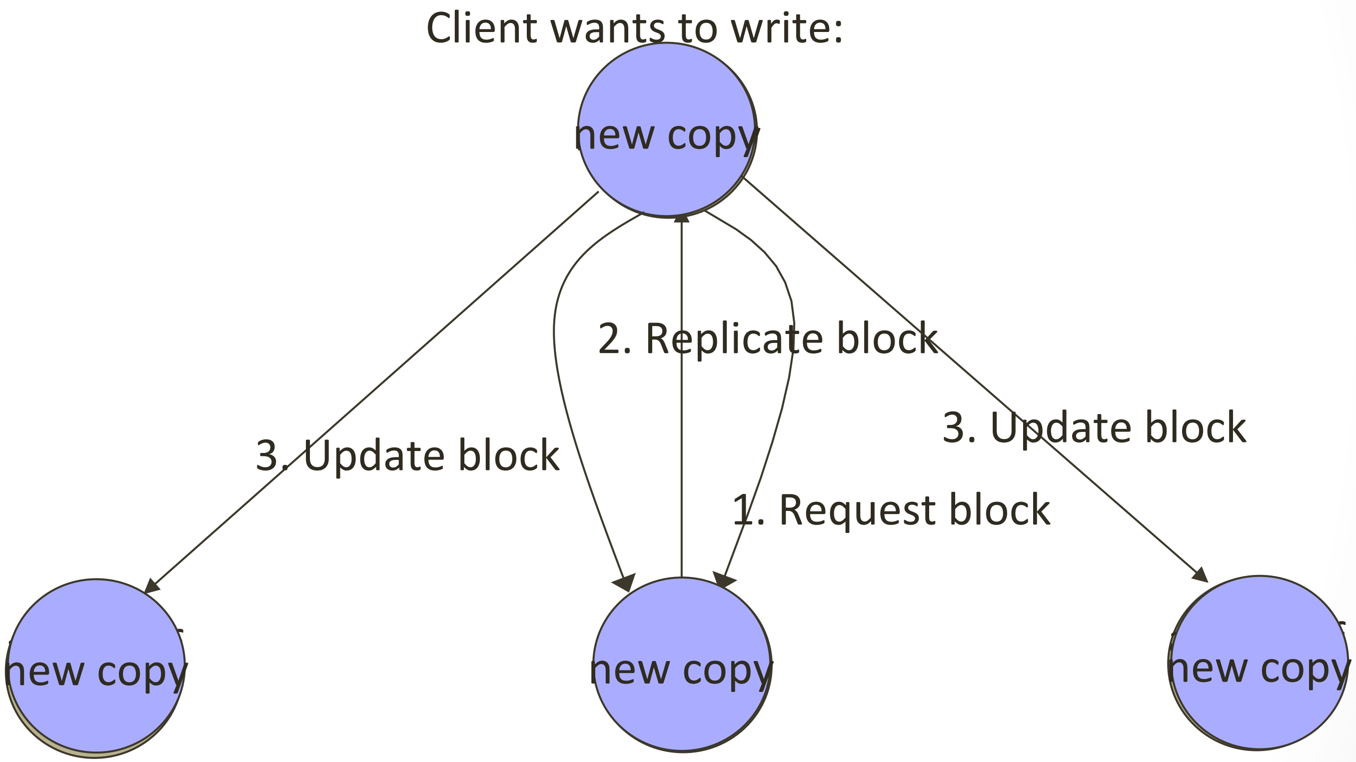
Then what if Node 2 updates x?

Implementing Sequential Consistency

Write Invalidation

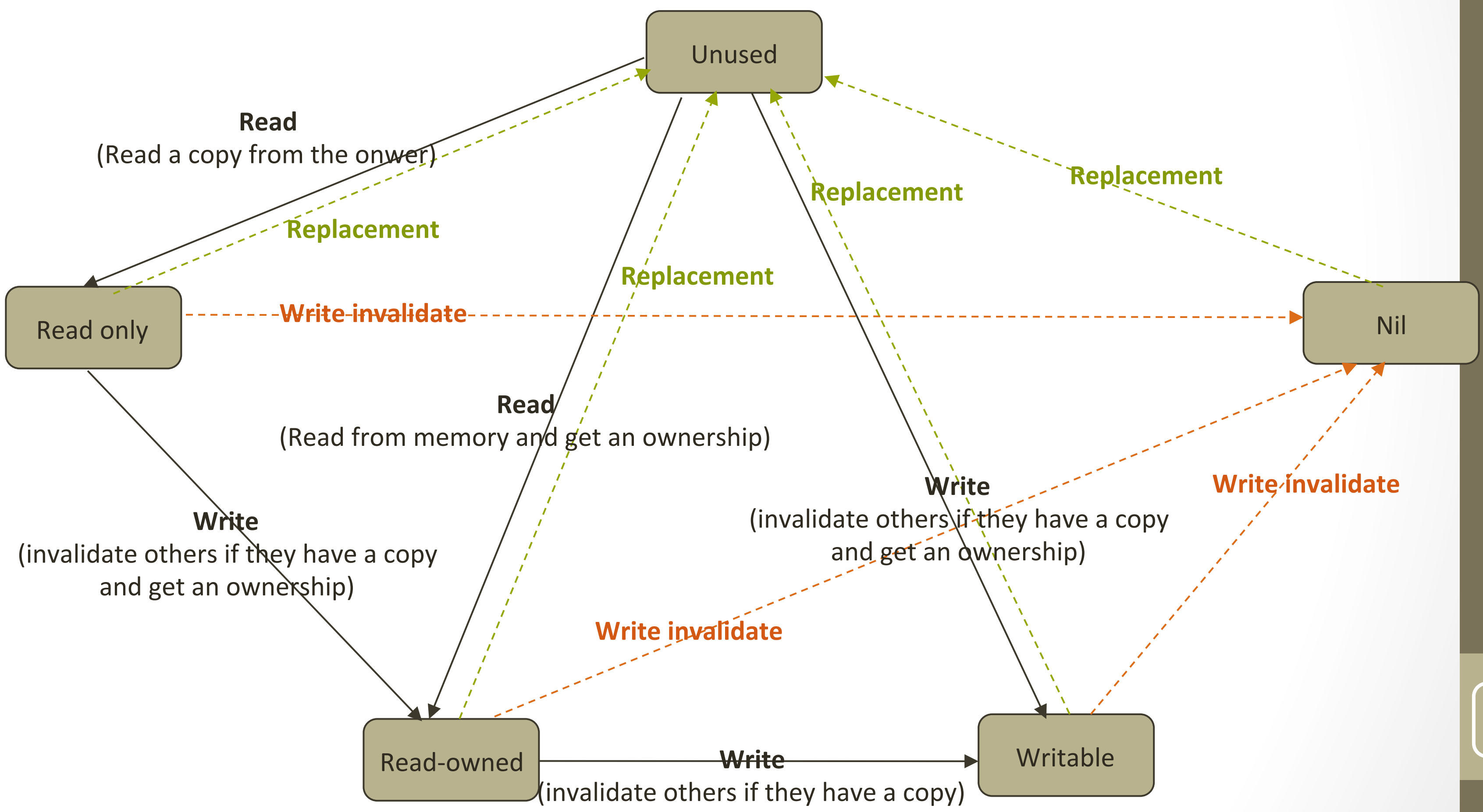


Implementing Sequential Consistency
Write Update



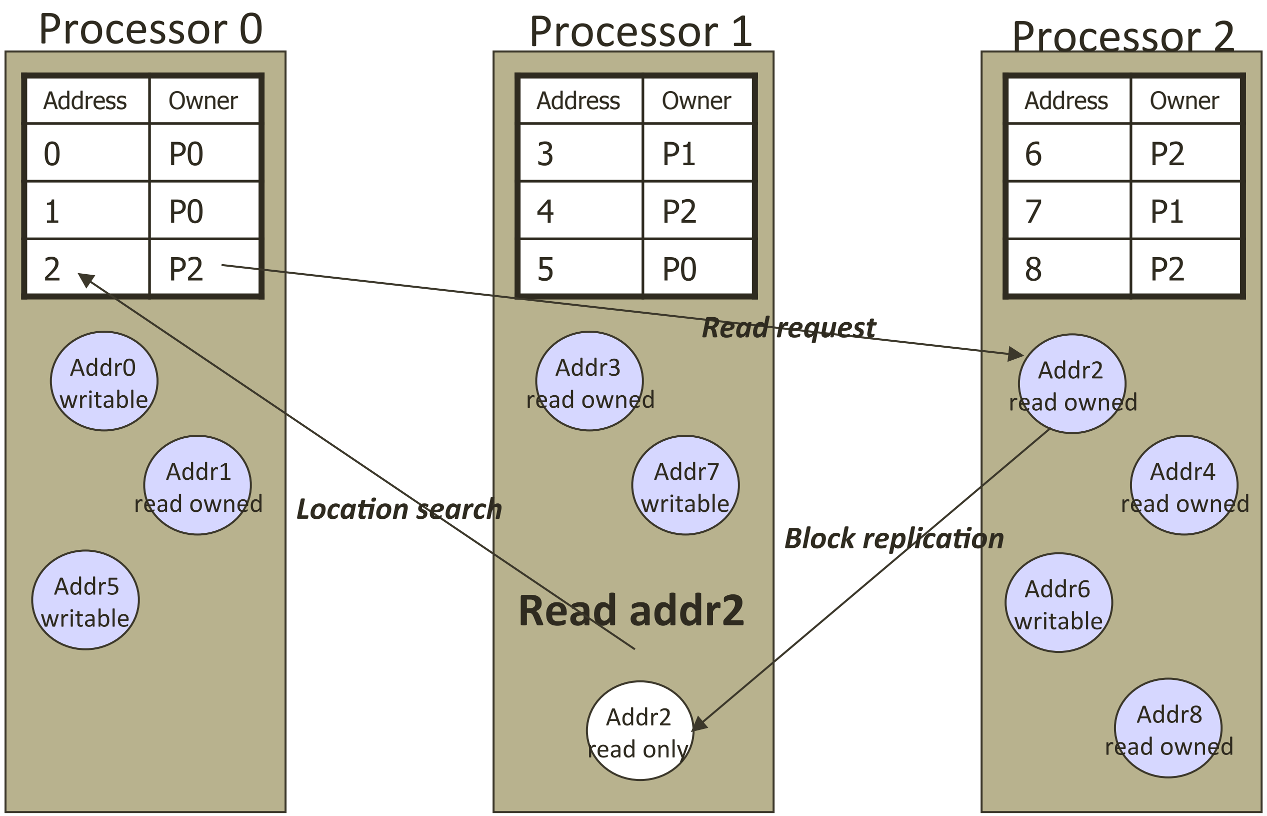
Implementing Sequential Consistency

Read/Write Request



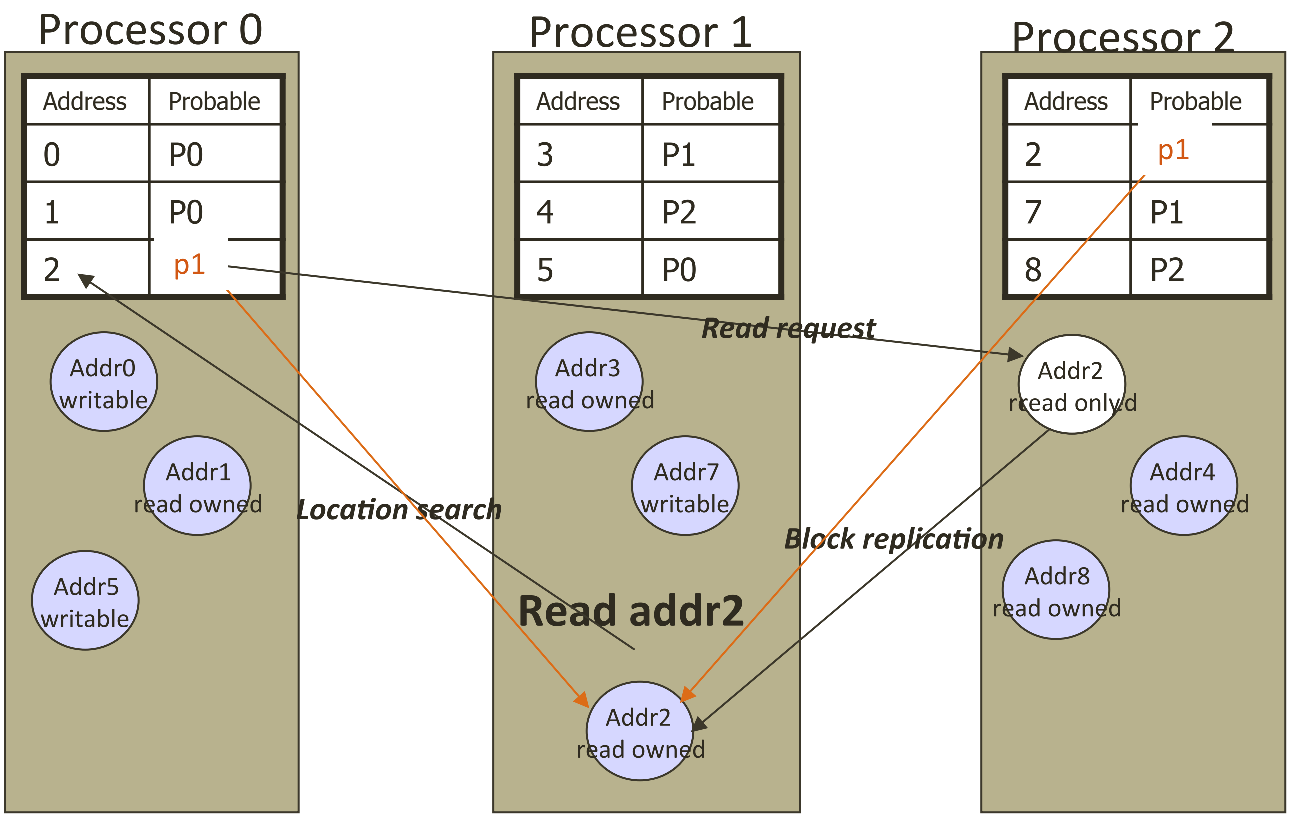
Implementing Sequential Consistency

Locating Data –Fixed Distributed-Server Algorithms



Implementing Sequential Consistency

Locating Data – Dynamic Distributed-Server Algorithms



- Breaking the chain of nodes:
  - When the node receives an invalidation
  - When the node relinquishes ownership
  - When the node forwards a fault request
- The node points to a new owner

Replacement Strategy

- Which block to replace
  - Non-usage based (e.g. FIFO)
  - Usage based (e.g. LRU)
  - Mixed of those (e.g. Ivy)
    - Unused/Nil: replaced with the highest priority
    - Read-only: the second priority
    - Read-owned: the third priority
    - Writable: the lowest priority and LRU used.
- Where to place a replaced block
  - Invalidating a block if other nodes have a copy.
  - Using secondary store
  - Using the memory space of other nodes

Thrashing

- Thrashing:
  - Two or more processes try to write the same shared block.
  - An owner keeps writing its block shared by two or more reader processes.
  - The larger a block, the more chances of false sharing that causes thrashing.
- Solutions:
  - Allow a process to prevent a block from accessed from the others, using a lock.
  - Allow a process to hold a block for a certain amount of time.
  - Apply a different coherence algorithm to each block.
- What do those solutions require users to do?
- Are there any perfect solutions?