Distributed Systems

17 DSM Examples

July-22-2009

Gerd Liefländer

System Architecture Group

Case Studies

IVY at Yale University

Mirage

Clouds

Munin

Mether

TreadMarks as separate PDF File

Example DSM

- *HW-SMPs*
  - *DASH or PLUS NUMA architectures*
- Paged virtual DSM
  - Ivy 89
  - Munin 91
  - Mirage 89
  - Clouds 91
  - Choices 90
  - COOL 93
  - Mether 89
- Middleware
  - Orca 90
  - TSpaces 98
  - Linda 89

$\}$ Our focus, i.e.
DSM ~ distributed virtual memory

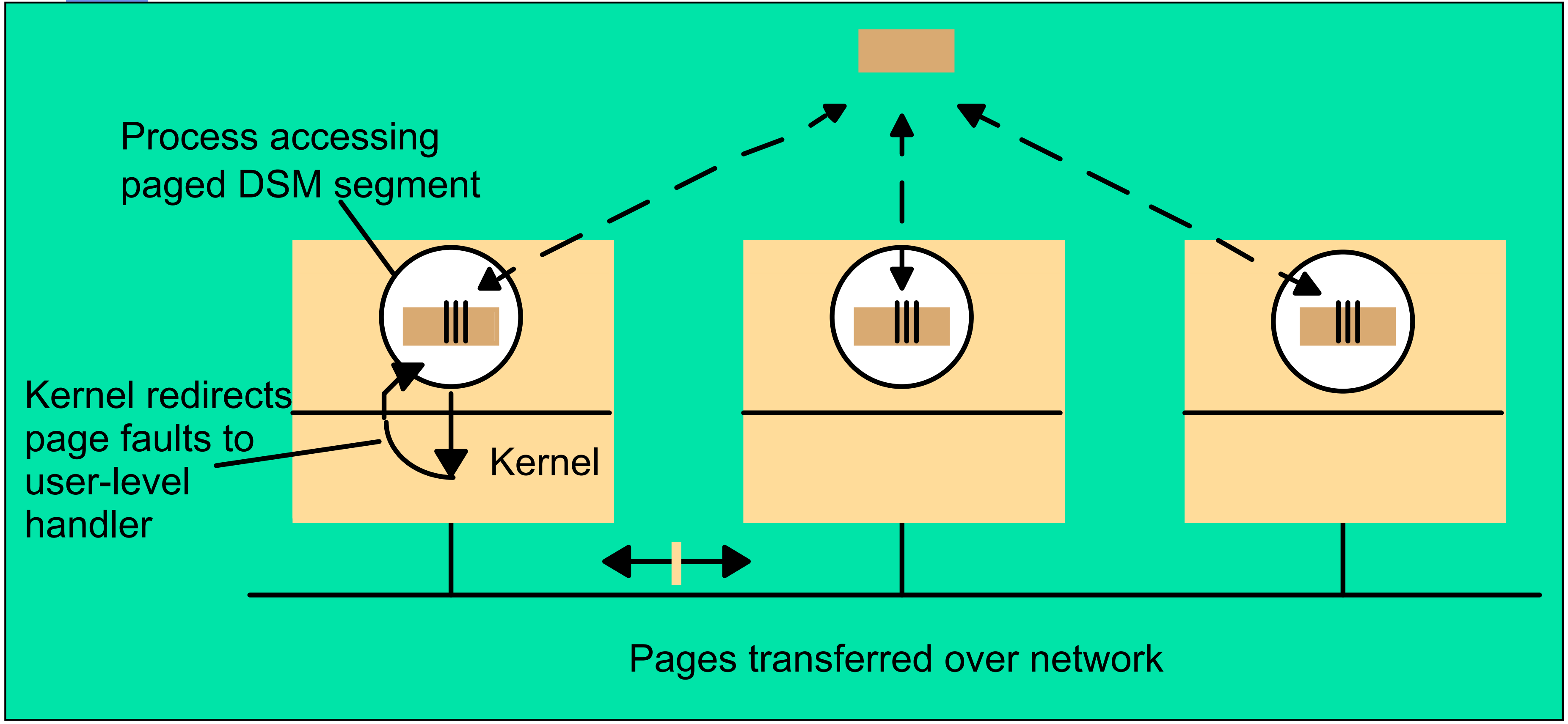
Sequential Consistency in Ivy

- This model is page-based. A single segment is shared between programs.
- The computers are equipped with a paged memory management unit.
- The DSM restricts data access permissions temporarily in order to maintain sequential consistency.
- Permissions can be none, read-only, or read-write.

Sequential Consistency and Ivy

- If a program tries to do more than it has permission for, a page fault occurs and the program is blocked until the page fault is resolved
- Since this DSM is page-based, write-update is only used if writes can be buffered
- Otherwise several consecutive updates to the same memory location or adjacent memory locations would result in several multicasts of the same page being updated

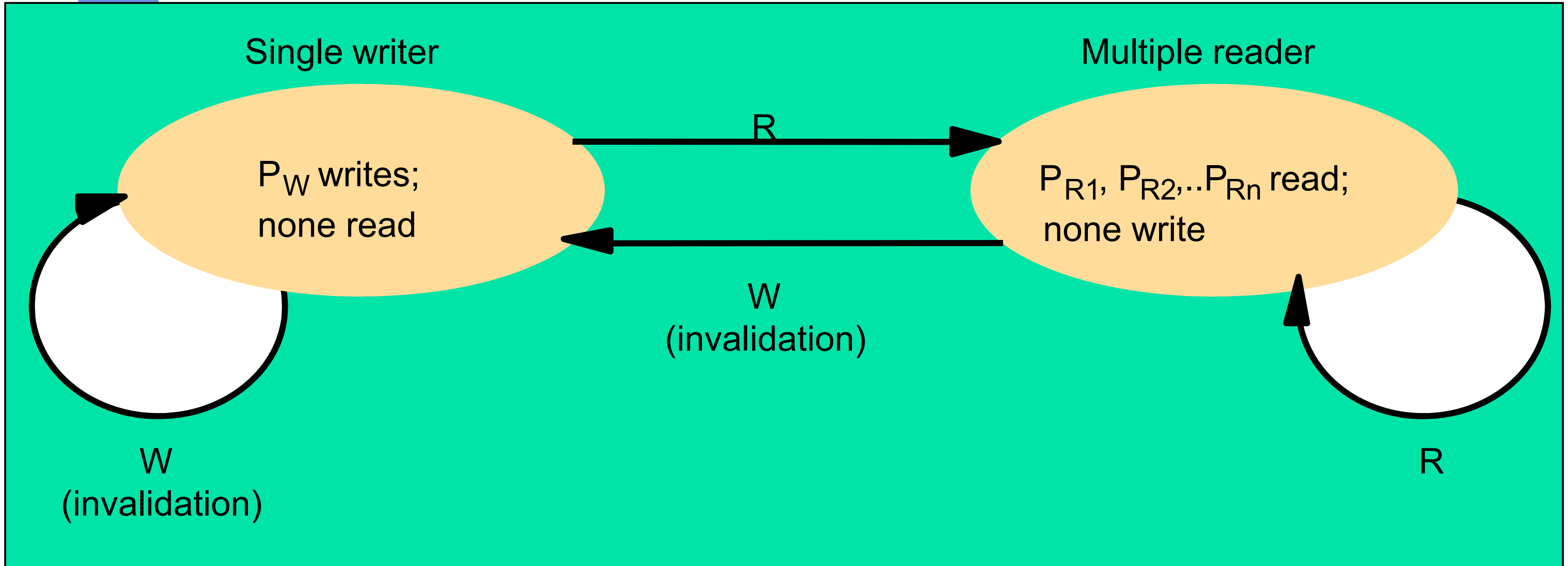
System Model for Page-based DSM


Process accessing paged DSM segment
Kernel redirects page faults to user-level handler
Kernel
Pages transferred over network

Sequential consistency in Ivy

- If writes cannot be buffered, write-invalidate is used
- The invalidation message acts as requesting a lock on the data
- When one program is updating the data it has read-write permissions and everyone else has no permissions on that page
- At all other times, all have read-only access to the page

State Transitions w. Write-Invalidation



Note: R = read fault occurs; *W* = write fault occurs.

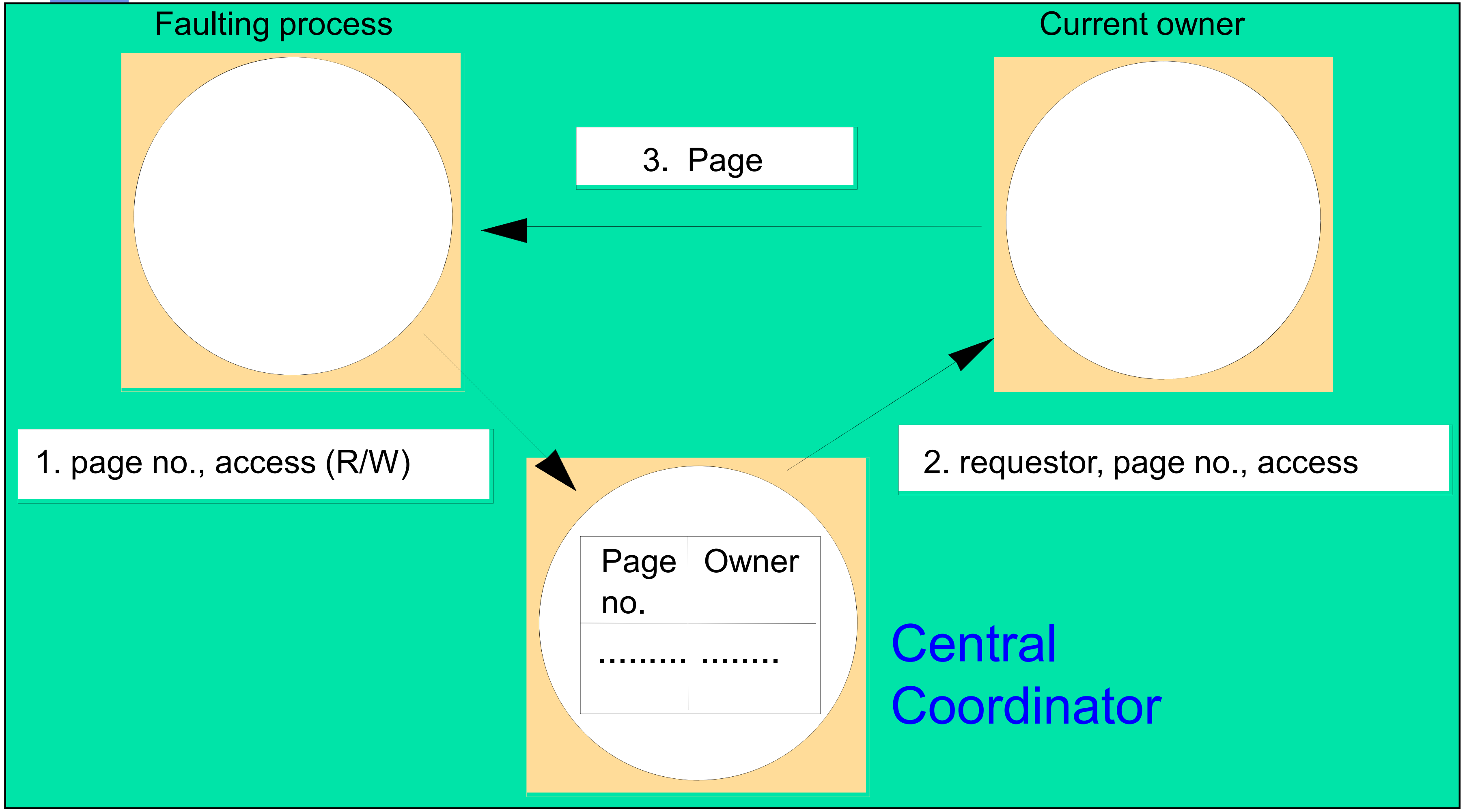
Ivy: State transitions

- When a program tries to write to a page for which it does not have read-write permission, a page fault occurs.
- An invalidate message is sent to all other programs.
- This sets the page permissions for those programs to none, and then the DSM system sets the page permissions for the writing program to read-write and unblocks it from the page fault.
- Two programs might request write access at close to the same time.

Ivy: State transitions

- If a program attempts to read a page it does not have permissions for a page fault occurs.
- The DSM system (on behalf of the reading program) will send a message (with the latest sequence number of its copy of the page) to the owner of the page.
- If the page owner determines the reader’s sequence number does not match its sequence number of the page, it sends the whole page to the reader.
- It will then grant read access to the page. If the current page owner determines it does not need to access the page soon, it may transfer ownership to another program.

Coordinator w. Associated Messages



Ivy: Invalidation Protocol

- A program must know who is the owner of the page that it needs. For this, they contact the coordinator.
- The coordinator may be just another program in the DSM system, or it may be a separate server.
- When a page fault occurs due to inappropriate permissions, the message requesting access is actually sent to the coordinator.
- The coordinator determines the page owner and forwards the message requesting access to the page owner. If the request is for a write page fault, the page ownership is transferred by the coordinator to the requester.

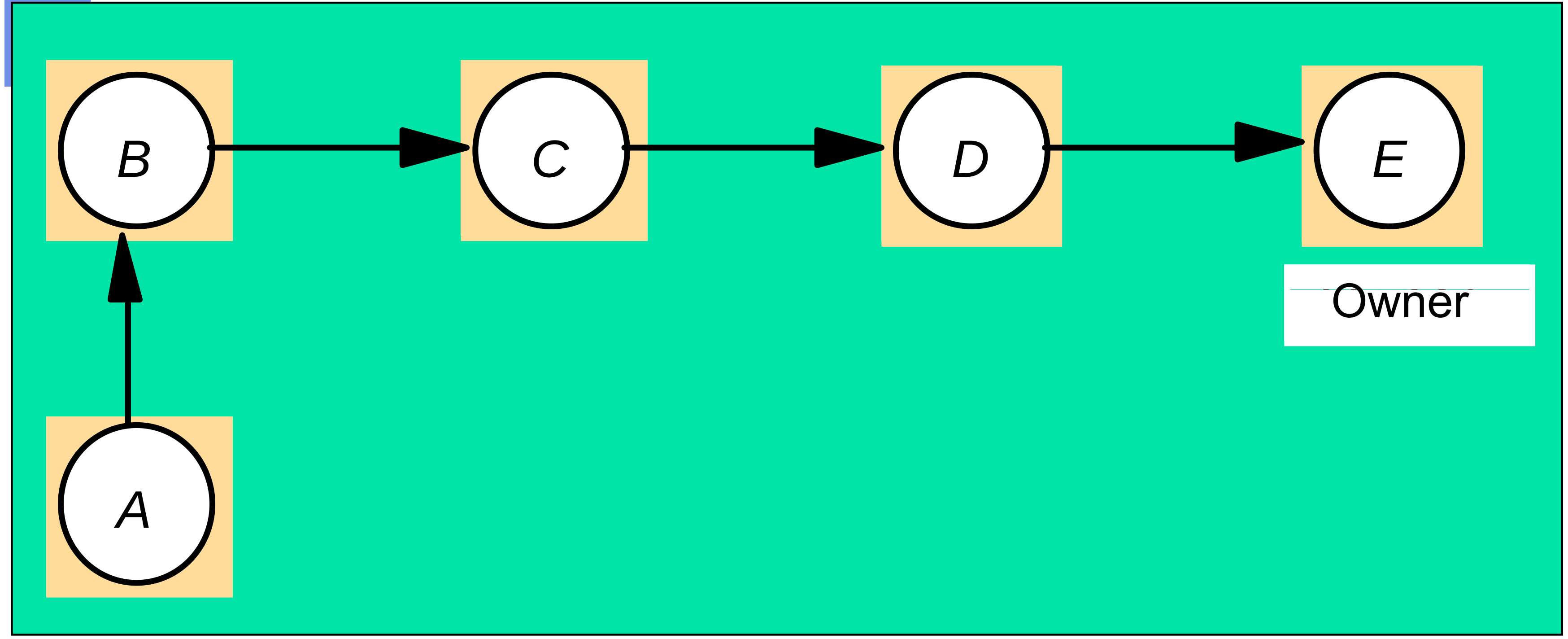
Ivy : Invalidation protocol

- For a write fault, the page’s previous owner sends the page and the page’s copy set to the new owner.
- The new owner performs the invalidation when it receives the page and copy set – it sends the invalidation message to the members of the copy set (excluding the previous owner who invalidate itself), thus revoking their read access to no access.

Ivy: Invalidation protocol

- The coodinator may become a performance bottleneck. There are a few alternatives:
- A fixed distributed page management where one program will manage a set of pages for its lifetime (even if it does not own them).
- A multicast-based management where the owner of a page manages it, read and write requests are multicast, only the owner answers.
- A dynamic distributed system where each program keeps a set of the probable owner(s) of each page.

Updating *probOwner* pointers



(a) *probOwner* pointers just before process *A* takes a page fault for a page owned by *E*

Ivy: Dynamic distributed manager

- Initially each program receives each pages owner and populates its probable ownership table.

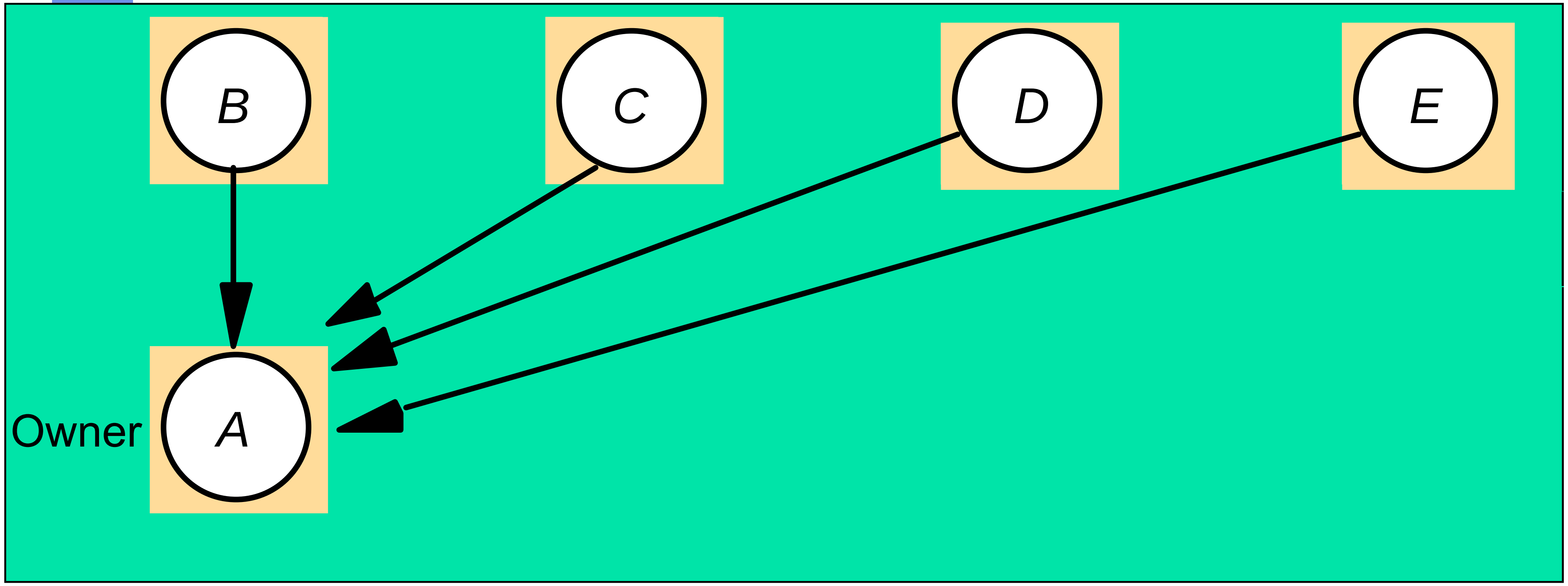
- When an owner transfers ownership, it will update its own probable ownership table with the new owner. (This guarantees at least 2 programs know the correct owner.)

- When a program receives an invalidation message for a page, it updates its table to list the sender of that message as the owner.

Ivy: Dynamic distributed manager

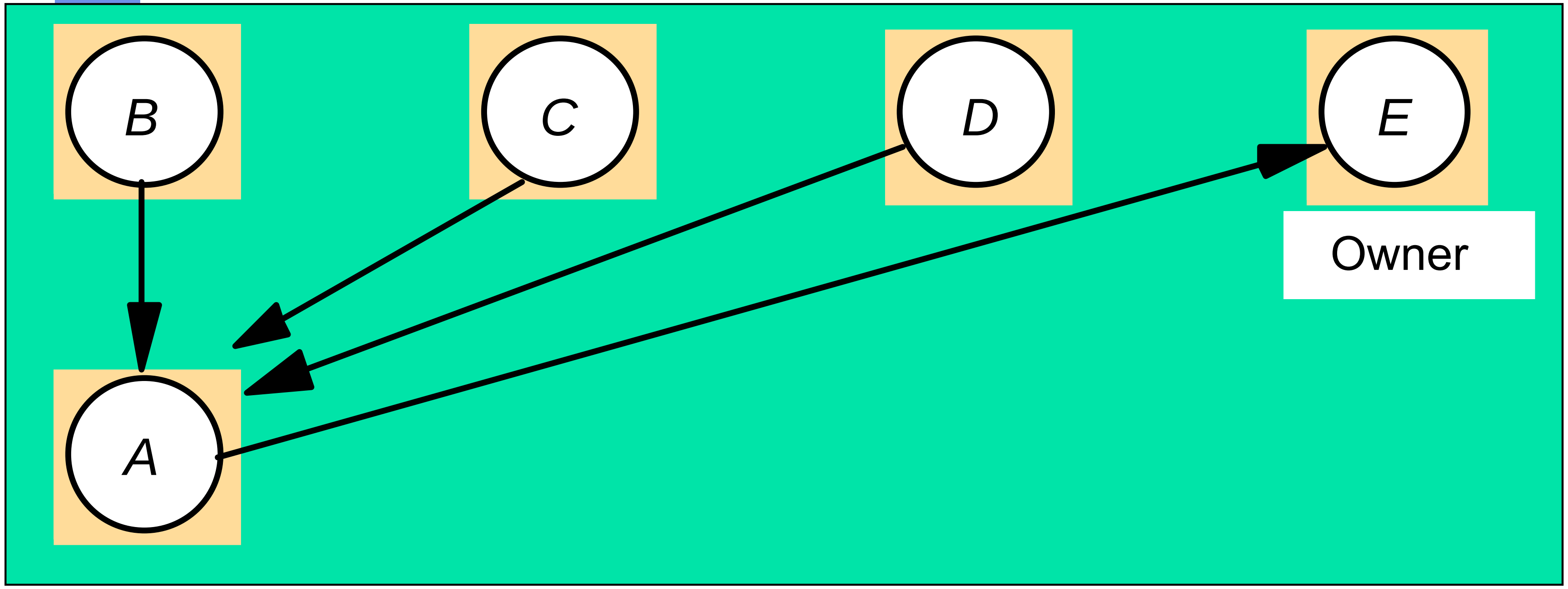
- When a program requests access to a page, it sends the request to whoever is listed in its probable owner table. When it receives the page, it will update its probable owner table with the sender of the page.
- If a program that receives a request for access does not own the page, it will forward the request to whoever is listed for the page in its probable owner table. It will then update its probable owner table to list the requester.
- Even if the requester does not become the new owner, it is about to find out who the correct owner is. By doing this the number of hops that a request can take before reaching the correct owner is limited.

Updating *probOwner* pointers



(b) Write fault: *probOwner* pointers after *A*'s write request is forwarded

Updating *probOwner* pointers



(c) Read fault: *probOwner* pointers after *A*'s read request is forwarded

Release Consistency and Munin

- Release consistency is weaker than sequential consistency, but cheaper to implement.

- Release consistency reduces overhead. It relies on the fact that programmers can use semaphores, locks, and barriers to achieve enough consistency the system may need.

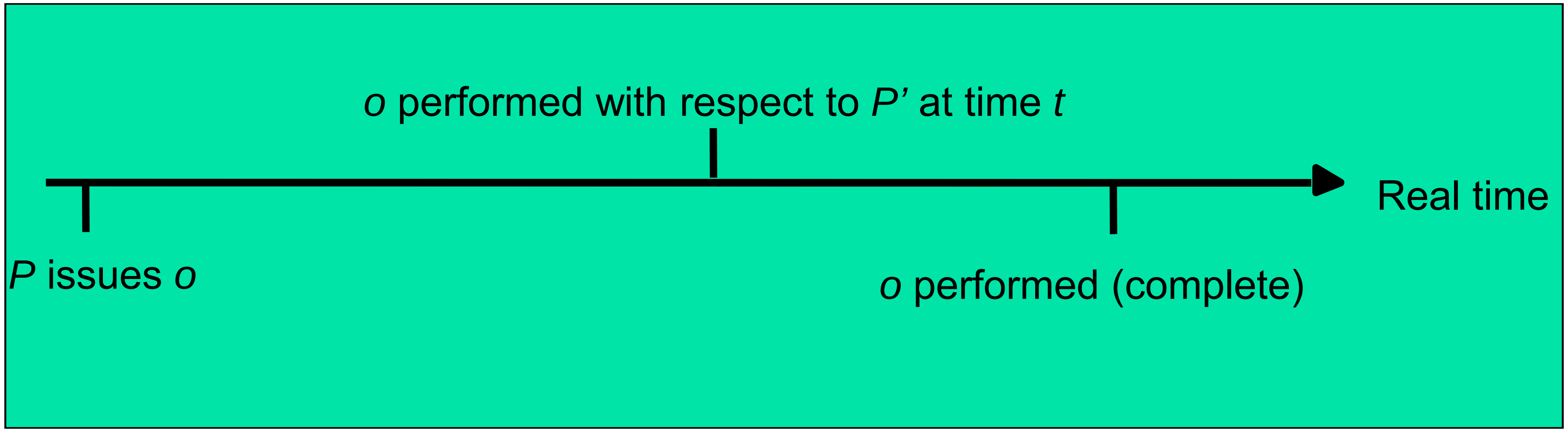
Munin: Memory accesses

- Types of memory accesses:
  - Competing accesses
    - They may occur concurrently – there is no enforced ordering between them.
    - At least one is a write
  - Non-competing or ordinary accesses
    - All read-only access, or enforced ordering

Munin: Memory accesses

- Competing memory accesses are divided into two categories:
  - Synchronization accesses are concurrent and contribute to synchronization. Examples include releasing a lock or a test-and-set operation.
  - Non-synchronization accesses are concurrent but do not contribute to synchronization.

Timeline in a DSM with *read* or *write*



Timeline for performing a DSM *read* or *write* operation

Release Consistency

Requirements

- To achieve release consistency, the system must:
  - Preserve synchronization with locks, etc.
  - Gain performance by allowing asynchronous memory operations.
  - Limit the overlap between memory operations.
  - One must acquire appropriate permissions before performing memory operations.
  - All memory operations must be performed before releasing memory.
  - Acquiring permissions and releasing memory

Munin

- Munin forces programmers to use acquireLock, releaseLock, and waitAtBarrier.
- Munin allows programmers to mark the way data is shared. Munin optimizes DSM based on this.
- These marks can also pair locks and data, which guarantees the user has the data before accessing it.
- Munin sends updates/invalidations when locks are released. An alternative has the update/invalidation sent when the lock is next acquired

Processes executing on a release-consistent DSM

```
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
```

Munin: Sharing Annotations

- The following are options with Munin on the data item level:
  - Using write-update or write-invalidate.
  - Whether several copies of data may exist.
  - Whether to send updates/invalidate immediately.
  - Whether a data has a fixed owner, and whether that data can be modified by several at once.
  - Whether the data can be modified at all.
  - Whether the data is shared by a fixed set of programs.

Munin : Standard Annotations

- Read-only : Initialized, but not allow to be updated.
- Migratory : Programs access a particular data item in turn.
- Write-shared : Programs access the same data item, but write to different parts of the data item.
- Producer-consumer : One program write to the data item. A fixed set of programs read it.
- Reduction : The data is always locked, read, updated, and unlocked
- Result : Several programs write to different parts of one data item. One program reads it.
- Conventional : Data is managed using write-invalidate.

Other Consistency Models

- Casual consistency – The happened-before relationship can be applied to read and write operations.
- Pipelining RAM – Programs apply write operations through pipelining.
- Processor consistency - Pipelining RAM plus memory coherent.

Other Consistency Models

- Entry consistency – Every shared data item is paired with a synchronization object.
- Scope consistency – Locks are applied automatically to data objects instead of relying on programmers to apply locks.
- Weak consistency – Guarantees that previous read and write operations complete before acquire or release operations.

Mether System Program -

```
#include "world.h"
struct  shared { int a,b; };
```

Program Writer:

```
main()
{
   struct shared *p;
   methersetup(); /* Initialize Mether run-time */
   p = (struct shared *)METHERBASE;
          /* overlay structure on METHER segment */
   p->a = p->b = 0; /* initialize fields to zero */
   while(TRUE){       /* update structure fields */
      p –>a = p –>a + 1;
      p –>b = p –>b - 1;
   }
}
```

Continued on next slide...

Mether System Program

```
Program Reader:
   main()
   {
      struct shared *p;
      methersetup();
      p = (struct shared *)METHERBASE;
      while(TRUE) {
    /* read the fields once every second */
         printf("a = %d, b = %d\n", p ->a, p ->b);
         sleep(1);
      }
   }
```

Literature

- B. Bershad et al: “The Midway DSM System, IEEE 1993
- N. Carreiro, D. Gelernter: “The S/Net’s Linda Kernel”, ACM Trans. On Comp. Sys., 1986
- J. Cordsen: “Virtueller gemeinsamer Speicher”, PhD TU Berlin, 1996
- M. Dubois et al.: “Synchronization, Coherence and Event Ordering in Multiprocessors”, IEEE Computer, 1988
- K. Li: “Shared Virtual Memory on Loosley Coupled Multiprocessors”, PhD Yale, 1986
- D. Mosberger: “Memory Consistency Models”, Tech. Report, Uni. Of Arizona, 1993
- B. Nitzberg: “DSM: A Survey of Issues and Algorithms”, IEEE Comp. Magazine, 1993

Appendix: Review Consistency Models

Another Notation

See Colouris et al

Processes Accessing Shared Data

Process 1

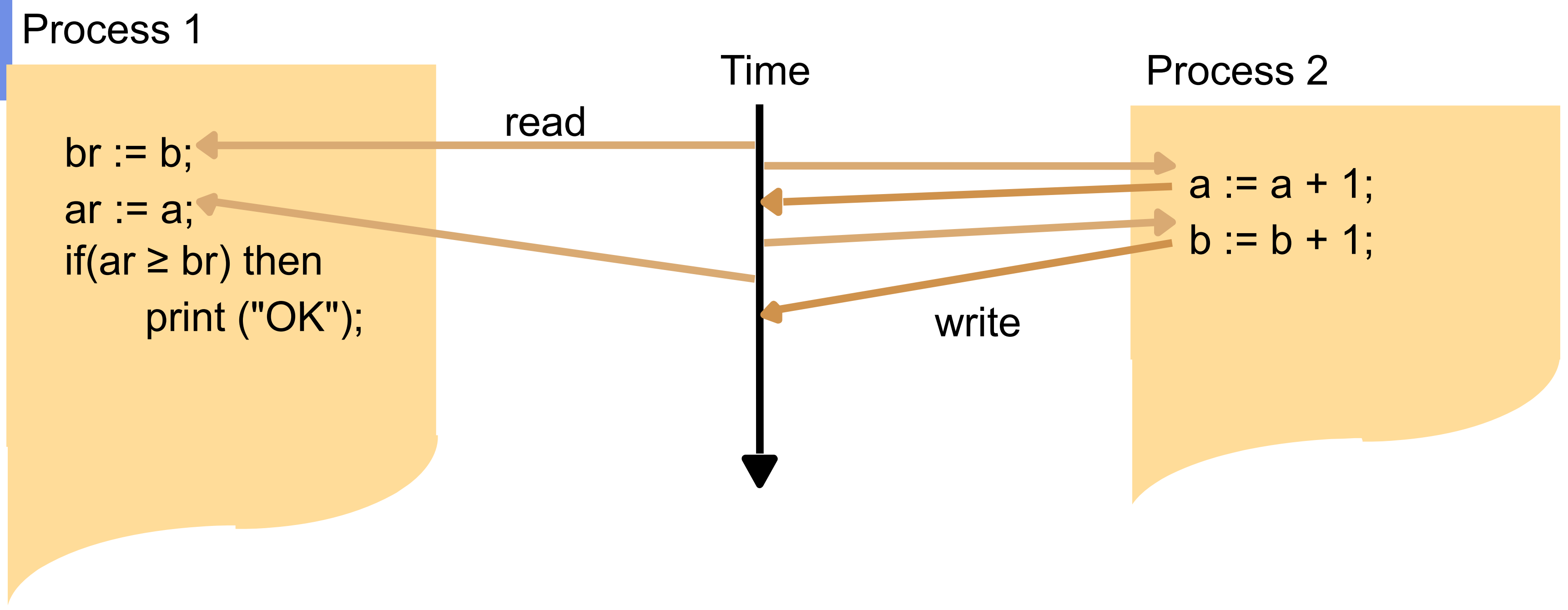
```
br := b;
ar := a;
if(ar ≥ br) then
        print ("OK");
```

Process 2

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

- a & b are initialized with 0
- Suppose, process 2 runs first, then process 1
- We expect that process1 always prints OK
- However, the update propagation of the DSM might send the updates to process1 in reverse order, i.e. ar = k, but br =k+1

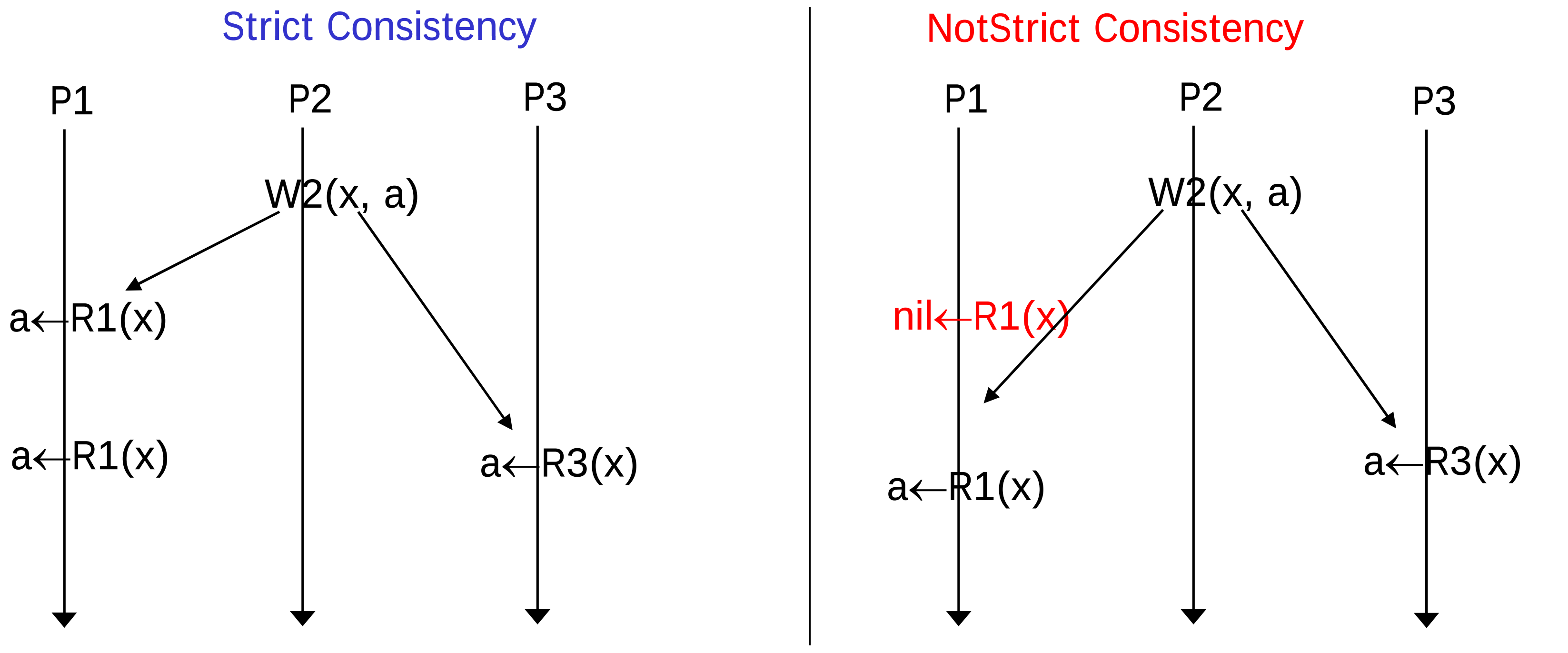
Interleaved Operations



- Allowed interleaving with sequential consistency

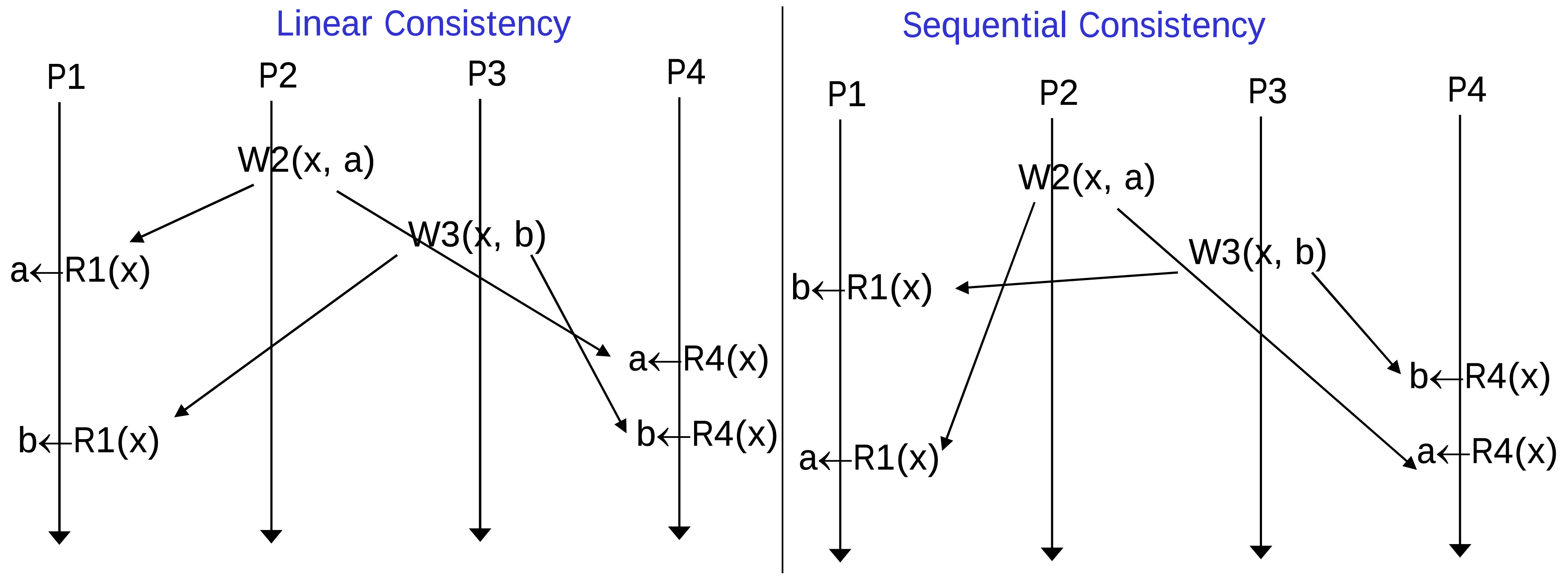
Strict Consistency

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



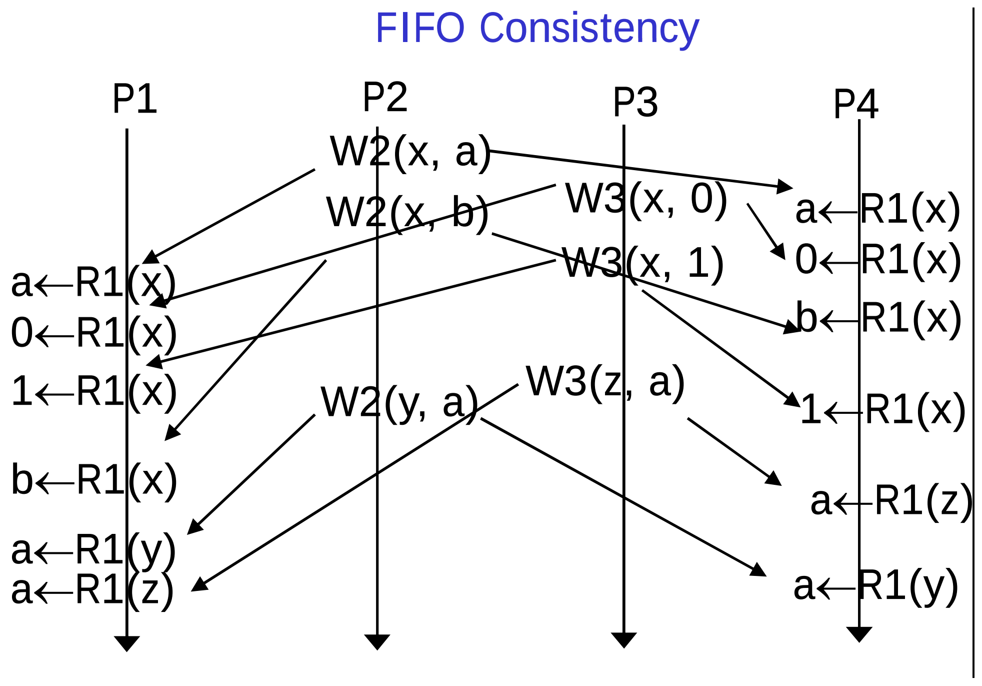
Linear & Sequential Consistency

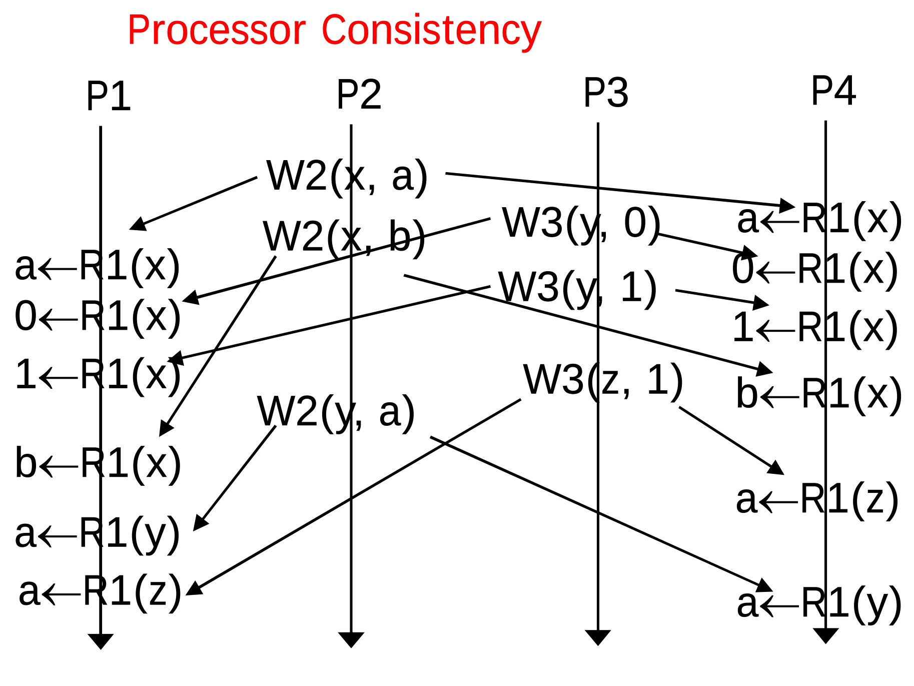
- **Linear Consistency:** Operations of each individual process appear to all processes in the same order as they happen.
- **Sequential Consistency:** Operations of each individual process appear in the same order to all processes.



FIFO and Processor Consistency

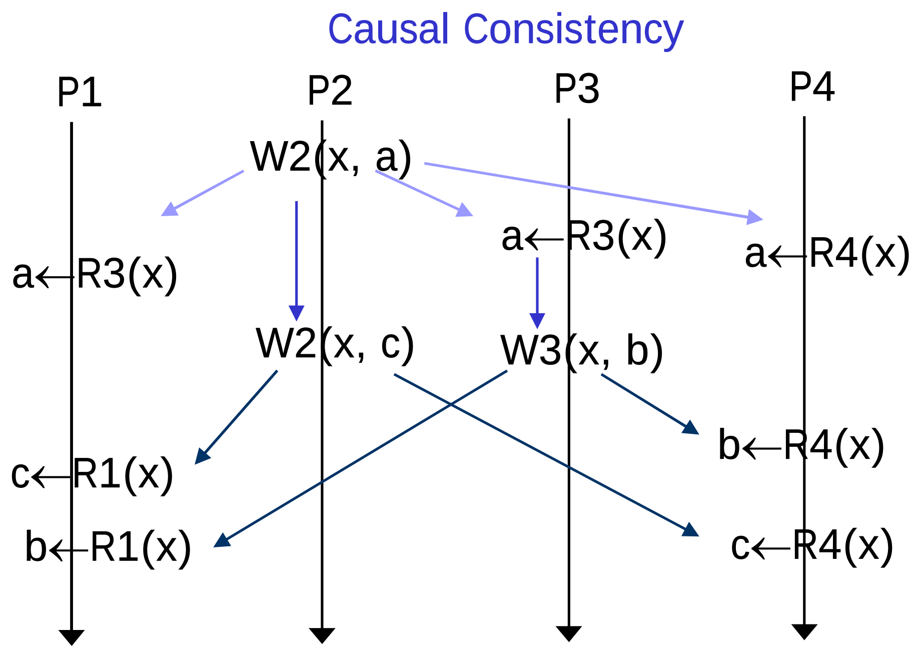
- **FIFO Consistency:** writes by a single process are visible to all other processes in the order in which they were issued.
- **Processor Consistency:** FIFO Consistency + all write to the same memory location must be visible in the same order.

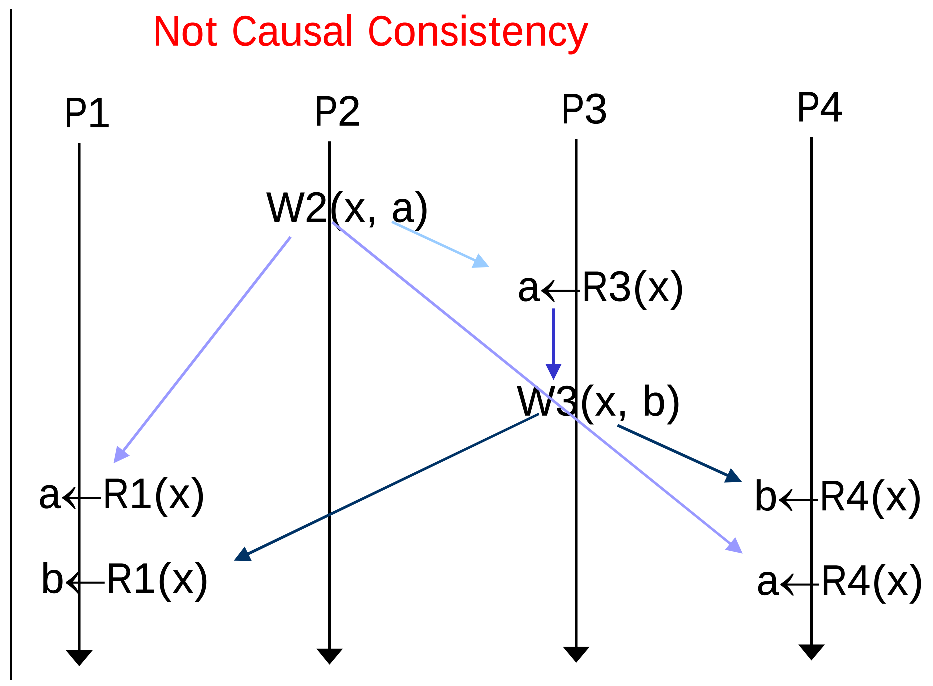




Causal Consistency

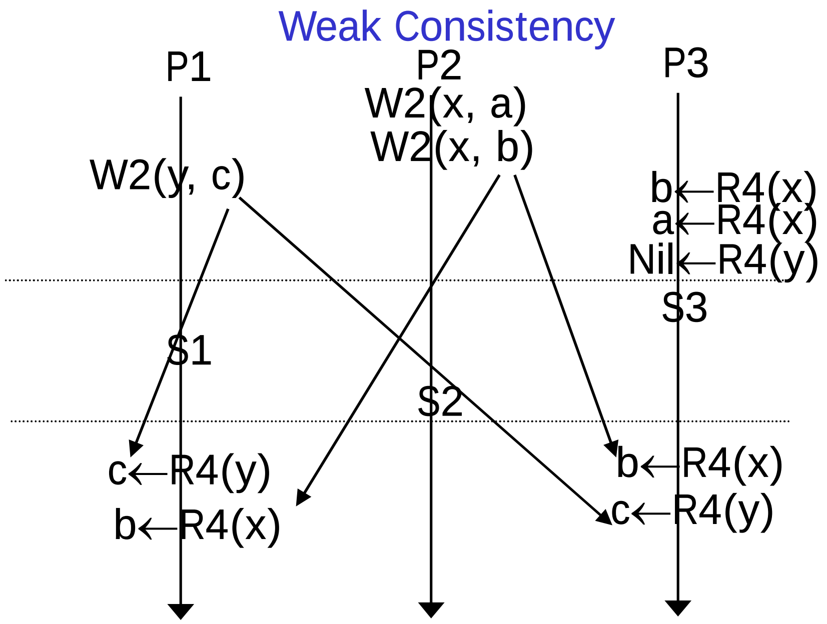
- Causally related writes must be visible to all processes in the same order. Concurrent writes may be propagated in a different order.

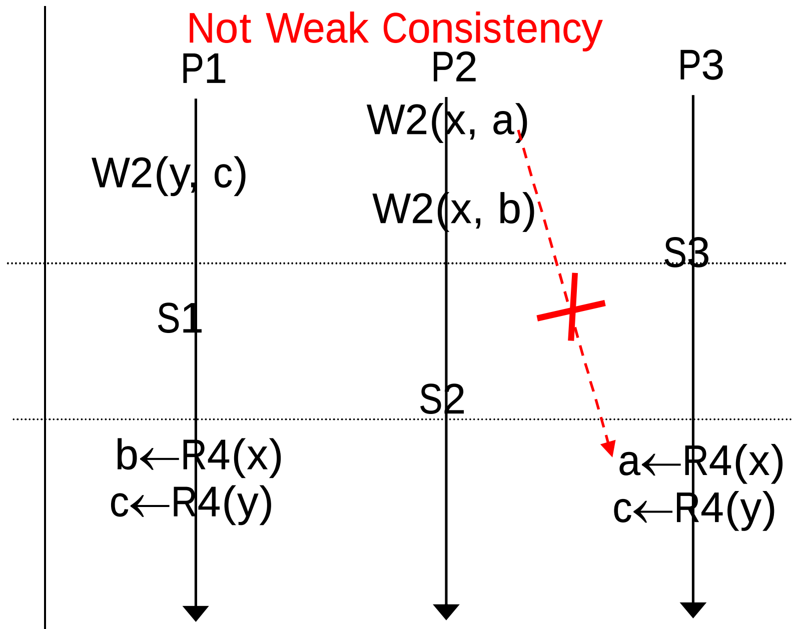




Weak Consistency

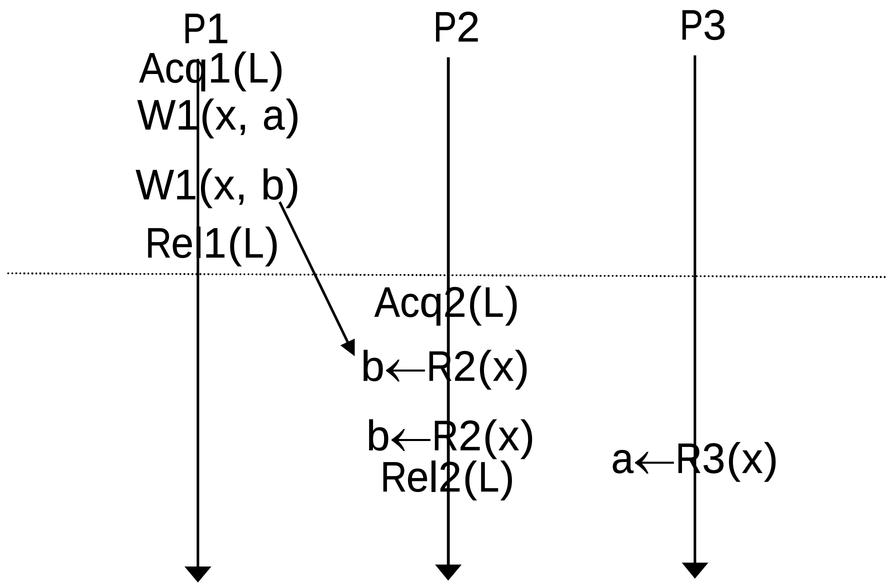
- Accesses to synchronization variables must obey sequential consistency.
- All previous writes must be completed before an access to a synchronization variable.
- All previous accesses to synchronization variables must be completed before access to non-synchronization variable.





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.


P1
P2
P3
Acq1(L)
W1(x, a)
W1(x, b)
Rel1(L)
Acq2(L)
b←R2(x)
b←R2(x)
Rel2(L)
a←R3(x)