ECE151 – Lecture 9

Chapter 6

Consistency and Replication

Consistency & Replication

Introduction

Data-centric consistency

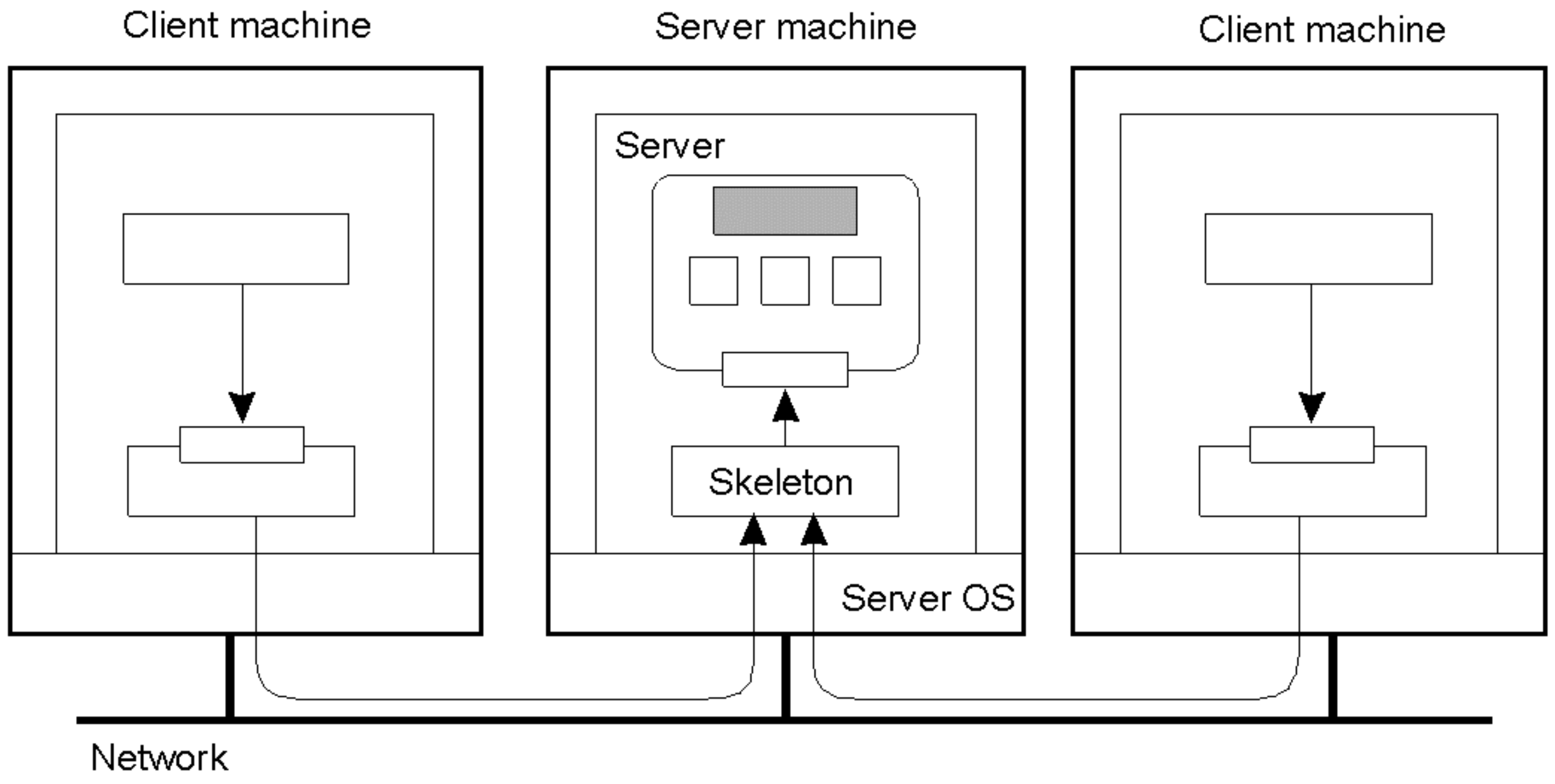
Client-centric consistency

Distribution protocols

Consistency protocols

Examples

Object Replication

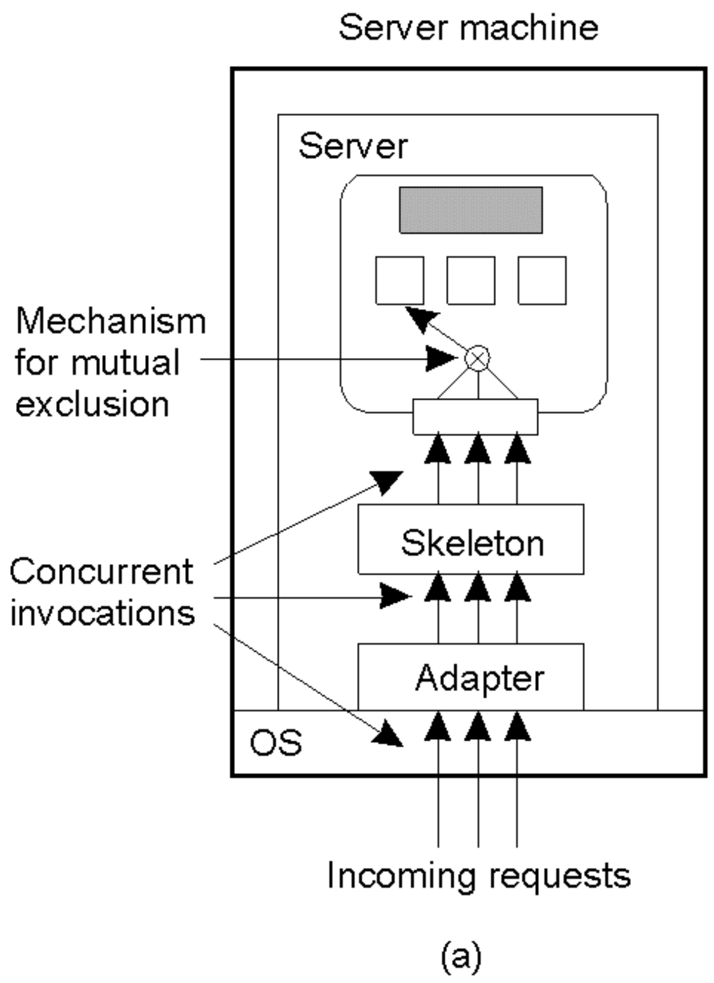


Problem: If objects (or data) are shared, we need to do something about concurrent accesses to guarantee state consistency.

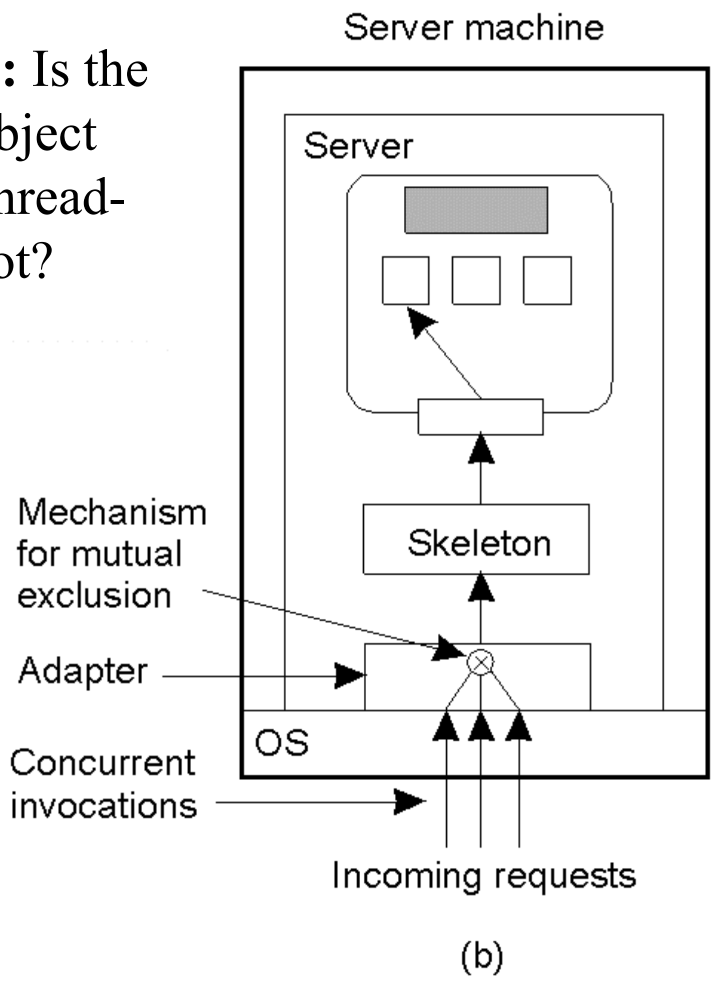
Object Replication

a) A remote object capable of handling concurrent invocations on its own.

b) A remote object for which an object adapter is required to handle concurrent invocations

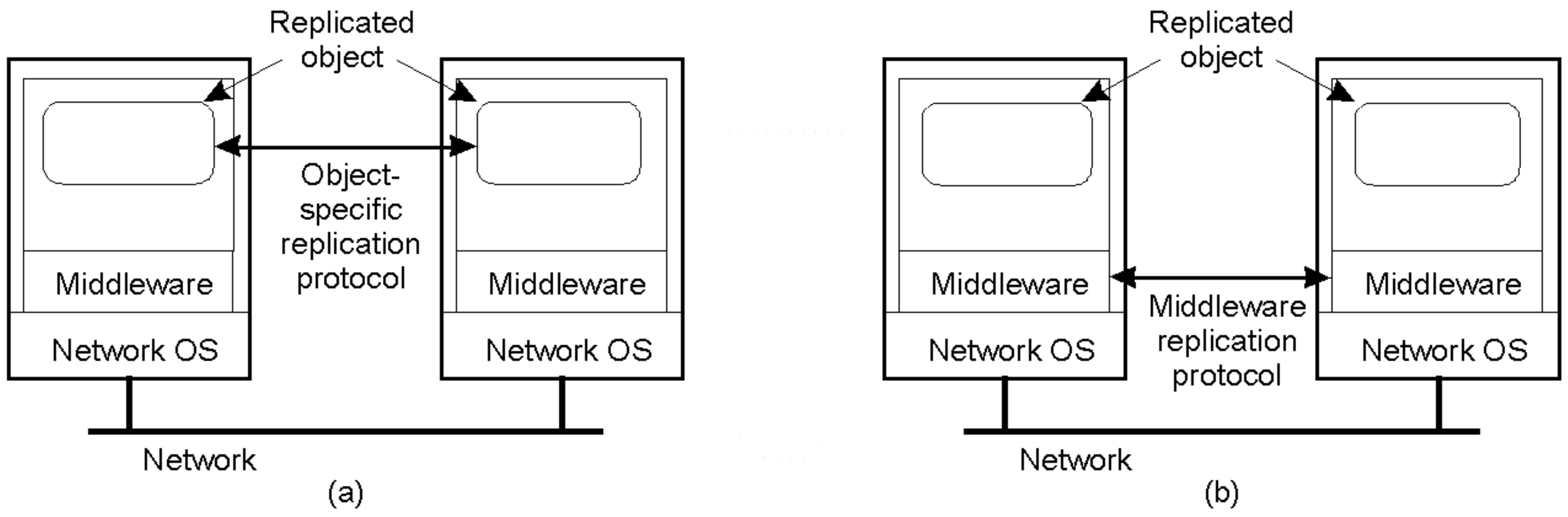


Problem: Is the remote object already thread-safe or not?



Object Replication

Problem: Should we seek for object-specific solutions, or generally applicable ones?
Question: Why would we want object-specific replication protocols?



a) A distributed system for replication-aware distributed objects.
b) A distributed system responsible for replica management

Performance and Scalability

Main issue: To keep replicas consistent, we generally need to ensure that all *conflicting* operations are done in the the same order everywhere

Conflicting operations: From the world of transactions:

Read–write conflict: a read operation and a write operation act concurrently

Write–write conflicts: two concurrent write operations

Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability

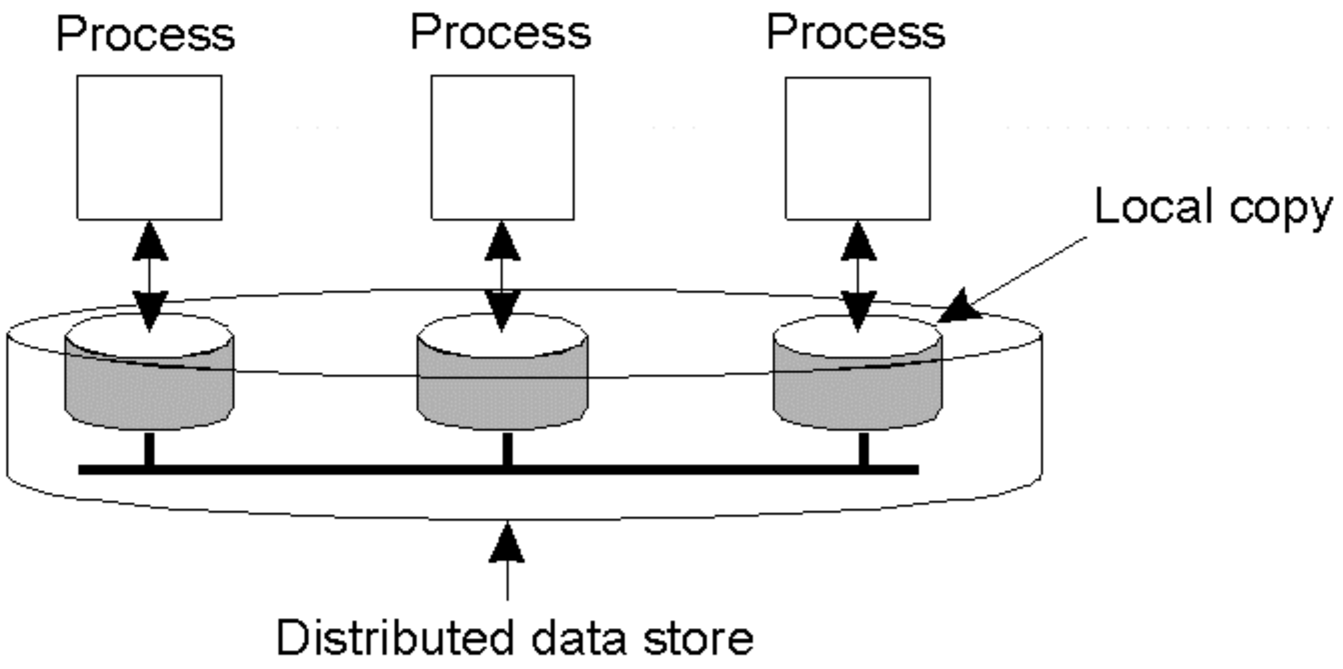
Solution: weaken consistency requirements so that hopefully global synchronization can be avoided

Data-Centric Consistency Models

The general organization of a logical data store, physically distributed and replicated across multiple processes.

Consistency model: a contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

Essence: A data store is a distributed collection of storages accessible to clients:



Data-Centric Consistency Models

Strong consistency models: Operations on shared data are synchronized:

Strict consistency (related to time)

Sequential consistency (what we are used to)

Causal consistency (maintains only causal relations)

FIFO consistency (maintains only individual ordering)

Weak consistency models: Synchronization occurs only when shared data is locked and unlocked:

General weak consistency

Release consistency

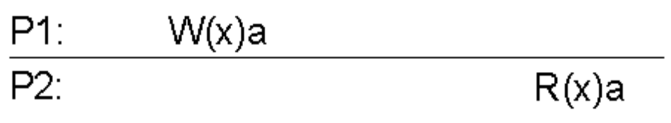
Entry consistency

Observation: The weaker the consistency model, the easier it is to build a scalable solution.

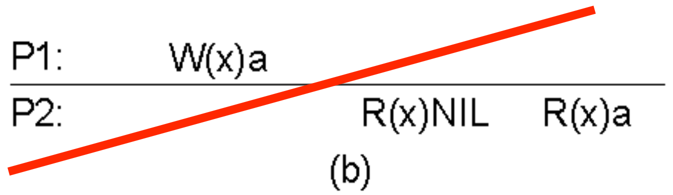
Strict Consistency

Any read to a shared data item X returns the value stored by the most recent write operation on X.

Observation: It doesn't make sense to talk about "the most recent" in a distributed environment.



(a)



(b)

Behavior of two processes, operating on the same data item.

- A strictly consistent store.
- A store that is not strictly consistent.

Assume all data items have been initialized to NIL

W(x)a: value *a* is written to *x*

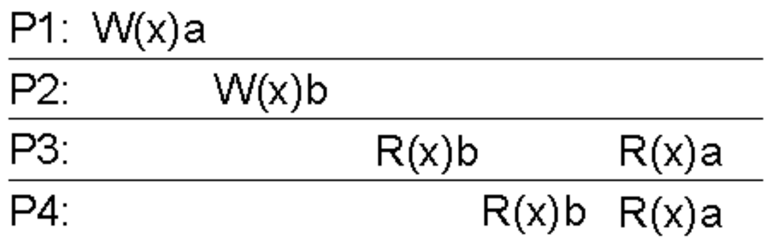
R(x)a: reading *x* returns the value *a*

Note: Strict consistency is what you get in the normal sequential case, where your program does not interfere with any other program.

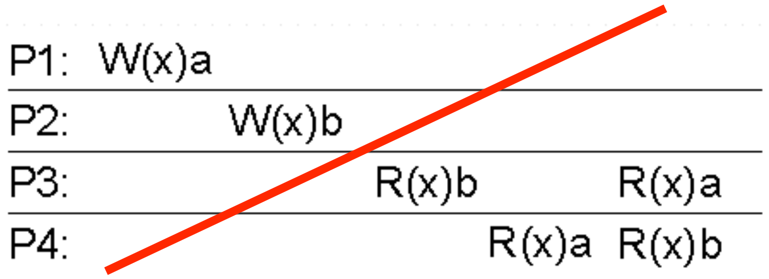
Linearizability and Sequential Consistency

The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

Note: We're talking about **interleaved** executions: there is some total ordering for all operations taken together.



(a)



(b)

a) A sequentially consistent data store.
b) A data store that is not sequentially consistent.

Linearizable: Sequential plus operations are ordered according to a global time.

Linearizability and Sequential Consistency

Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

Three concurrently executing processes.

Linearizability and Sequential Consistency

Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

```
x = 1;              x = 1;              y = 1;              y = 1;
print ((y, z);      y = 1;              z = 1;              x = 1;
y = 1;              print (x,z);        print (x, y);       z = 1;
print (x, z);       print(y, z);        print (x, z);       print (x, z);
z = 1;              z = 1;              x = 1;              print (y, z);
print (x, y);       print (x, y);       print (y, z);       print (x, y);

Prints:  001011     Prints: 101011      Prints: 010111      Prints: 111111

Signature:          Signature:          Signature:          Signature:
  001011              101011              110101              111111
    (a)                 (b)                 (c)                 (d)
```

Casual Consistency

Necessary condition:

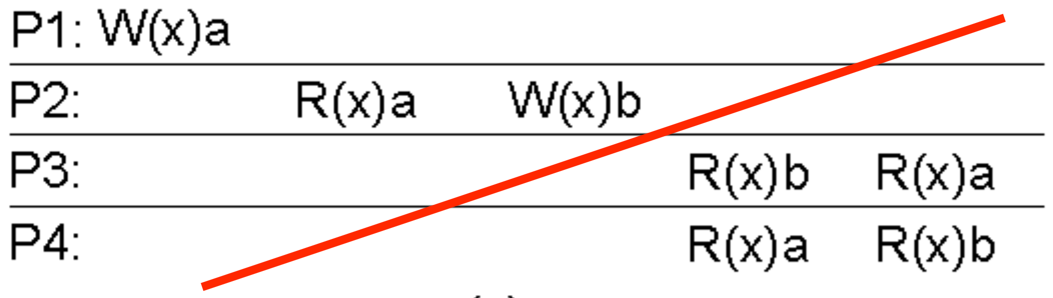
Writes that are potentially casually related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

Casual Consistency

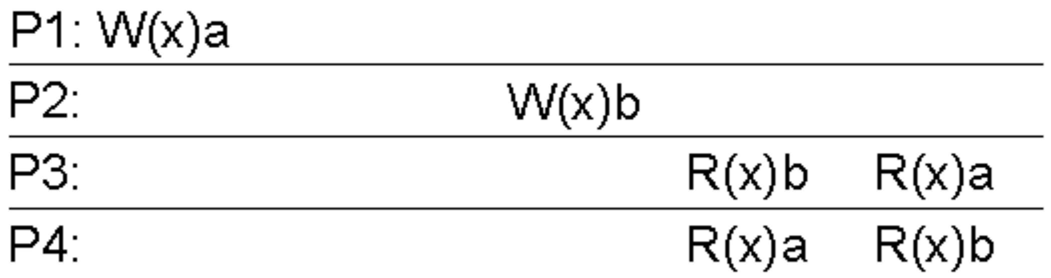
P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

This sequence is allowed with a casually-consistent store, but not with sequentially or strictly consistent store.

Casual Consistency



(a)



(b)

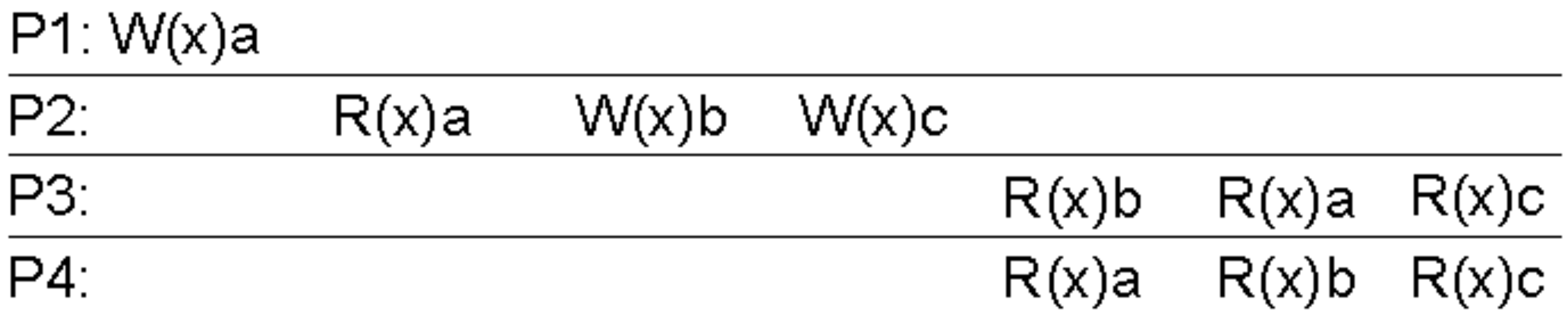
a) A violation of a casually-consistent store.
b) A correct sequence of events in a casually-consistent store.

FIFO Consistency

Necessary Condition:

Writes done by a single process are seen 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.

FIFO Consistency



A valid sequence of events of FIFO consistency

FIFO Consistency

x = 1;
print (y, z);
y = 1;
print(x, z);
z = 1;
print (x, y);

Prints: 00

(a)

x = 1;
y = 1;
print(x, z);
print (y, z);
z = 1;
print (x, y);

Prints: 10

(b)

y = 1;
print (x, z);
z = 1;
print (x, y);
x = 1;
print (y, z);

Prints: 01

(c)

Statement execution as seen by the three processes from the previous slide. The statements in bold are the ones that generate the output shown.

FIFO Consistency

Process P1	**Process P2**
x = 1;	y = 1;
if (y == 0) kill (P2);	if (x == 0) kill (P1);

Two concurrent processes.

Weak Consistency

Properties:

- Accesses to synchronization variables associated with a data store are sequentially consistent
- No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
- No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

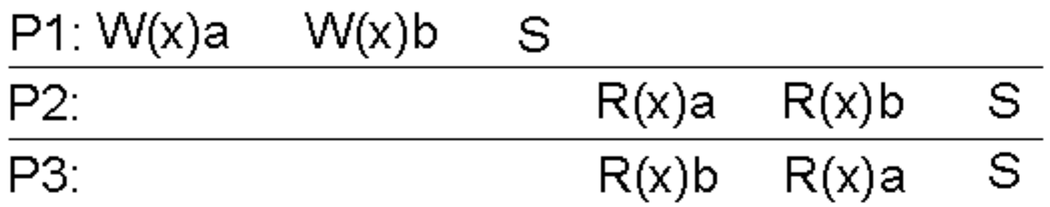
Basic idea: You don't care that reads and writes of a *series* of operations are immediately known to other processes. You just want the *effect* of the series itself to be known.

Weak Consistency

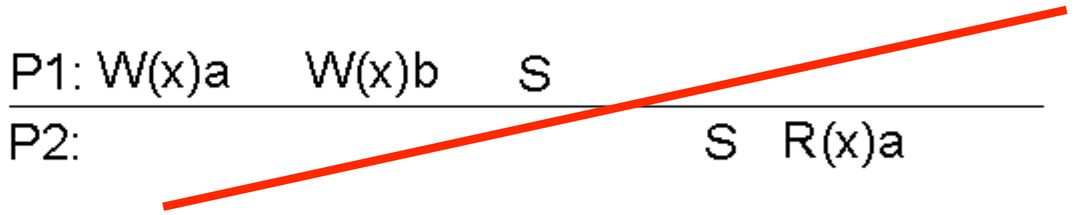
```
int a, b, c, d, e, x, y;                    /* variables */
int *p, *q;                                 /* pointers */
int f( int *p, int *q);                     /* function prototype */

a = x * x;                                  /* a stored in register */
b = y * y;                                  /* b as well */
c = a*a*a + b*b + a * b;                    /* used later */
d = a * a * c;                              /* used later */
p = &a;                                     /* p gets address of a */
q = &b                                      /* q gets address of b */
e = f(p, q)                                 /* function call */
```

A program fragment in which some variables may be kept in registers.

Weak Consistency (3)



(a)



(b)

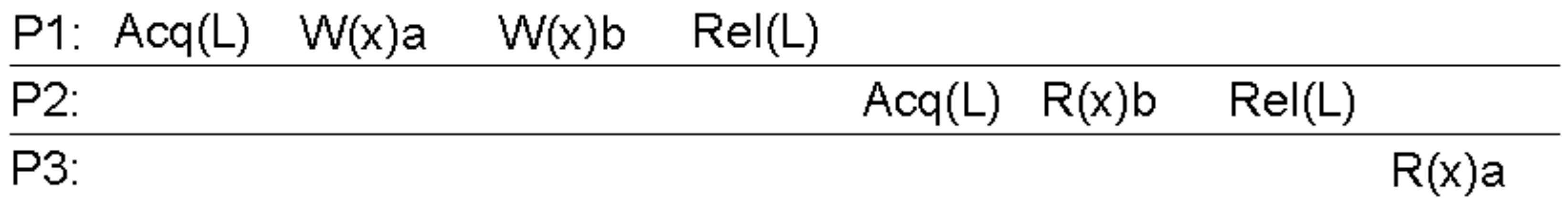
a) A valid sequence of events for weak consistency.
b) An invalid sequence for weak consistency.

Release Consistency

Idea: Divide access to a synchronization variable into two parts: an **acquire** and a **release** phase.

Acquire forces a requester to wait until the shared data can be accessed

Release sends requester's local value to other servers in data store.



A valid event sequence for release consistency.

Release Consistency

Rules:

- Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
- Before a release is allowed to be performed, all previous reads and writes by the process must have completed
- Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).

Entry Consistency

With release consistency, all local updates are propagated to other copies/servers during release of shared data.

With entry consistency, each shared data item is associated with a synchronization variable.

When acquiring the synchronization variable, the most recent values of its associated shared data item are fetched.

Note: Where release consistency affects *all* shared data, entry consistency affects only those shared data associated with a synchronization variable.

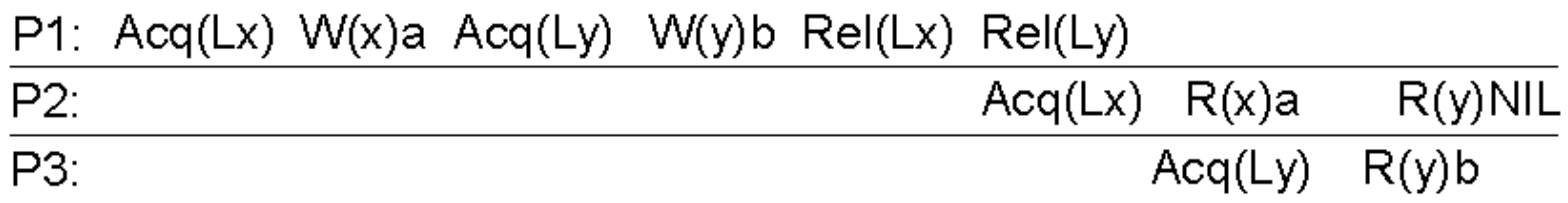
Question: What would be a convenient way of making entry consistency more or less transparent to programmers?

Entry Consistency

Conditions:

- An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
- Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
- After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Entry Consistency (1)


P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)
P2: Acq(Lx) R(x)a R(y)NIL
P3: Acq(Ly) R(y)b

A valid event sequence for entry consistency.

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

a) Consistency models not using synchronization operations.
b) Models with synchronization operations.