# IPC In Distributed Operating Systems, Con't

CS 239, Spring 1996

May 15, 1996



## Outline

- Distributed shared memory
- Event ordering and clocks in distributed systems
- Causally ordered multicast



## Distributed Shared Memory

Cooperating processes share a coherent memory space

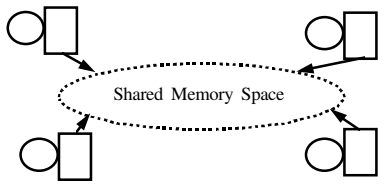
Can this be provided between processes that cannot access the same physical memory?

Yes - and sometimes at a reasonable cost

Distributed shared memory must be highly transparent to be successful



## Distributed Shared Memory Diagram






## Approaches to DSM

- Central server
- Migration
- Read replication
- Full replication



## Central Server DSM

A central server maintains all data pages

Returns data on reads

Makes updates on writes

Care needed to avoid duplicating writes

Server duties can be split between several sites

## Pros/Cons of Central Server DSM

+ Relatively simple to implement
+ Synchronization methods pretty clear
– Most data references are remote
– Prone to bottlenecks



## Migration DSM

Data lives where most recently used

When off-site memory is requested, it migrates to the requesting site

Typically at block or page levels

With locality of reference, works well

But possibility of thrashing

Servers keep track of data locations



## Pros/Cons of Migration DSM

+ If locality good, most accesses can be local
+ Synchronization behavior fairly easy
– Thrashing dangers
– Possible bottlenecks at location servers
– May put heavy burden on communications



## Read-Replication DSM

Multiple copies of readable blocks stored

If block is written, invalidate other copies

Each block has owner that permanently stores it

If writes frequent, invalidation can be complicated and expensive



## Pros/Cons of Read Replication DSM

+ Reads are usually local
+ Synchronization not too difficult
– Requires invalidations
– Possibly heavy burden on owners of popular pages
– Writes are never local (unless site owns the page)



## Full-Replication DSM

Full-service replicas of data blocks/pages stored at several machines

Requires consistency mechanism

For same reasons as replicated files

But different mechanisms may be appropriate

## Pros/Cons of Full Replication DSM

+ Access usually local
+ Bottlenecks generally less likely
– Synchronization tricky
– Requires full replication mechanism
– Likely to be overkill unless data sharing very common



## Memory Coherence in Replicated DSM Schemes

If more than one copy of a data item exists, problems arise when data is written

*Consistency model* describes what level of consistency a DSM system provides



## Consistency Models for DSM

- Strict consistency(as if only one copy)
- Sequential consistency (serializability)
- General consistency (eventual consistency)
- Processor consistency (per machine)
- Weak consistency (up to programmer)



## Protocols to Provide DSM Coherence

- Write-invalidate protocol
  Invalidate other copies on write
  Expensive for those other copies
- Write-update protocol
  Update other copies on write
  Higher network traffic results
  Requires some kind of locking



## Controlling Thrashing in DSM

How to prevent 2 sites sharing a page for write from thrashing the net?

One way is to lock a written page down at a site for some period $\Delta$

If another site wants to write it, that site must wait for $\Delta$ to expire

Used in Mirage system



## Granularity in DSM

How big should the unit of shared memory be?

Should be related to hardware's page size

Size of page has many important implications

## Pros/Cons of Large Pages for DSM

+ Takes advantage of locality of reference
+ Minimizes cost of running paging code
+ Can minimize communications cost
  If lots of data being moved
– Leads to greater contention



## Pros/Cons of Small Page Size for DSM

+ Less contention
  Minimizes *false sharing*
+ Lower cost to move small data objects
– Greater overheads



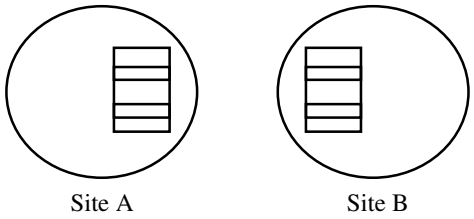
## False Sharing in DSM

False sharing occurs when an apparent, but not real, conflict occurs between two copies of a page

Caused by viewing replication unit as a page, rather than a memory location

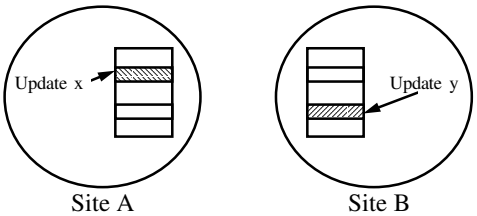


## False Sharing Example






## False Sharing Example, Con't




There appears to be a conflict, but there really isn't



## Effects of False Sharing

Depends on approach used

In migration systems, causes more page migration

In write-invalidate replication systems, causes more invalidations

In update-propagation replication systems, causes more locking

## Page Replacement in DSM

When a local node needs to bump a shared memory page, what does it do?

LRU alone not a sufficient answer

Must consider private vs. shared pages

And replication issues

And read-only pages

LRU within classes often used



## Some DSM Systems

- Ivy (Yale)
- Mirage (UCLA)
- Clouds (Georgia Tech)
- Munin (Rice U.)
- Memnet (U. of Delaware)



## Another Way to Provide Distributed Consistency

Assume a set of cooperating processes start out consistent

With a distributed state

And assume the only interactions are through messages

If all processes see the messages in the same order, all see a consistent state



## Event Ordering and Clocks

Derived from work by Lamport in late 70s

How can you determine which of two events occurred first in a distributed system?

Sometimes impossible to say

When can we determine it?



## What About Clock Time?

Stamp each event with the time of the local clock

Only works if you have clocks available

What if the local clock doesn't have the same time as other clocks?

Need some better definition



## Building a Partial Order of Events

Assume system is composed of collection of processes

Each process is sequence of events

Events within a single process have a natural total order

How about inter-process relationships?

## The "Happened Before" Relation

$a \rightarrow b$

Means that event a happened before event b

Equivalently, event a could have had a causal influence on event b

Complete set of "happened before" relationships for a computation defines a partial order of its events



## How Do We Define the Partial Order?

1). If a and b are events in the same process, and a comes before b,

$a \rightarrow b$

The natural ordering of events in a sequential process



## How Do We Define the Partial Order, Con't?

2). If a is the sending of message m and b is the receipt of message m,

$a \rightarrow b$

Means you can't receive a message until after it is sent



## How Do We Define the Partial Order, Con't?

3). If $a \rightarrow b$ and $b \rightarrow c$, then

$b \rightarrow c$

Transitivity applies

Also, $a \not\rightarrow a$

An event can't happen before itself



## Concurrent Events

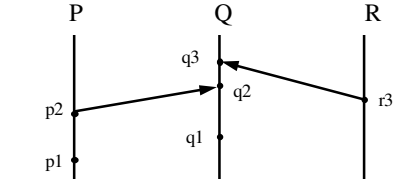
Two events a and b are said to be concurrent if

$a \not\rightarrow b$ and $b \not\rightarrow a$

Concurrent events cannot have a causal influence on each other



## Examples of Partial Ordering

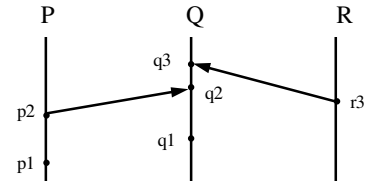



$p1 \rightarrow p2$ $q1 \rightarrow q2 \rightarrow q3$

$p2 \rightarrow q2$ $r3 \rightarrow q3$

$p1 \rightarrow q3$

## Concurrent Event Examples




**p2** and **r3** are concurrent
**q1** and **p1** and **r3** are concurrent



## Causal Influence

Event a can have a causal influence on event b only if $\mathbf{a} \rightarrow \mathbf{b}$

Which means, among other things, that events which are concurrent can be safely run in parallel



## Logical Clocks

We can use these concepts of event partial ordering to construct logical clocks that make the ordering explicit

Each event is assigned a timestamp

The timestamps reflect the "happened before" relation



## Defining the System

System consists of set of processes $\boldsymbol{P}$

Process $\boldsymbol{P_i}$ is the $i^{th}$ of these processes

Each process has a clock $\boldsymbol{C_i}$

Each event $\boldsymbol{a}$ of process Pi is assigned clock value $\boldsymbol{C_i[a]}$

$\boldsymbol{C[a]}$ is global clock value of $\boldsymbol{a}$

$\boldsymbol{C[a] = C_i[a]}$ if $\boldsymbol{a}$ happens at $\boldsymbol{P_i}$



## Correctness Condition for Logical Clocks

Clocks can be implemented as logical counters, rather than physical clocks

For any events a and b,

if $\mathbf{a} \rightarrow \mathbf{b}$ then $\boldsymbol{C[a] < C[b]}$

Converse isn't true, since that would imply concurrent events occur at same time



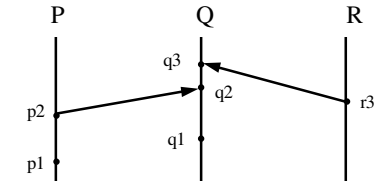
## How To Satisfy the Clock Condition?

C1. If $\boldsymbol{a}$ and $\boldsymbol{b}$ are events in $\boldsymbol{P_i}$, and $\boldsymbol{a}$ comes before $\boldsymbol{b}$, $\boldsymbol{C_i[a] < C_i[b]}$

C2. If $\boldsymbol{a}$ is the sending of a message by $\boldsymbol{P_i}$ and $\boldsymbol{b}$ is the receipt of that message by $\boldsymbol{P_j}$, then $\boldsymbol{C_i[a] < C_j[b]}$

These match the first two conditions in the definition of "happened before"

## Example of Logical Clock Values




$C_P[p1] < C_P[p2]$ $C_Q[q1] < C_Q[q2] < C_Q[q3]$

$C_P[p2] < C_Q[q2]$ $C_R[r3] < C_Q[q3]$



## How Can We Make Logical Clocks Actually Work?

Increment the local clock between any two successive events

Add timestamp $T$ to message $a$

$T = C_i[a]$

On receipt of a message at $P_j$, set local clock to larger of $T + 1$ and $C_j + 1$



## Obtaining a Total Ordering

Puts all events in system into some total order

Basic method is to provide ordering of processes

To break ties on local clock values



## Using Causal Ordering in a Real System

Messages carry updates

All messages must be handled in the appropriate order to guarantee consistency

- Absolute ordering
- Causal ordering

Easy to provide synchronously

Hard to provide asynchronously



## Isis - A Causal Message Ordering Service

Isis is a library for guaranteeing message ordering

By multicast operations

Can provide either ABCAST or CBCAST

Using low-overhead mechanisms



## The Isis Model

- Set of processes $P=\{p_1,p_2, \ldots, p_n\}$
- Process failure is detectable
- Processes structured in process groups $G=\{g_1,g_2, \ldots\}$
- Process groups are named
- Processes can belong to multiple groups
- Multicasts only go to members in sender's group

## Views in Isis

- The membership of a given group at a given moment is a *view*
- A *view sequence* of a process group is a list of its views
  - Each view differs from last by one process
- Processes learn of failures only by requesting view of a group



## Isis Message Transport

Isis is built on top of a transport mechanism

With the following characteristics

- Lossless
- Uncorrupted
- Sequenced
- Transport layer intercepts and discards messages from failed processes



## Isis Message Primitives

- $send_p(m)$
  Denotes sending of $m$ by $p$; atomic
- *dests(m)*
  Set of processes receiving message $m$
- $rcv_p(m)$
  Event of process $p$ receiving message $m$

  "Receive" is not the same as "deliver"



## Events in Isis

- The execution of a process is a partially ordered sequence of events
- Events consist of things like sending a message, receiving a message
- Acyclic event order reflects the dependence of events at p






## Virtual Synchrony

Users program Isis as if it scheduled one distributed event at a time

How to achieve this?

1). Schedule one distributed event at a time

2). Enforce degree of synchrony actually needed for each case to give this appearance



## Aspects of Virtual Synchrony

- Address expansion
  Ensure a given group multicast goes to all members of a given view
- Delivery atomicity and order
  Fault tolerant delivery
  Multiple destinations receiving same messages get them in consistent order

## Vector Clocks

Similar to version vectors, per process

For P processes, version clocks are of length P

$VT(p_i)$ is the version clock of $p_i$

$VT(p_i)[i]$ is $i^{th}$ component

Assign a unique, increasing timestamp to each send event

Properly handled, version clocks encode causality



## The CBCAST Protocol

Before multicasting message m, process $p_i$ increments $VT(p_i)[i]$

When $p_j$ receives message m from $p_i$, delay delivery until

For all k, $VT(m)[k] = VT(p_j)[k]+1$ if k=i

$VT(m)[k] <= VT(p_j)[k]$ otherwise

On delivery of m, update j's vector time



## What's Going on Here?

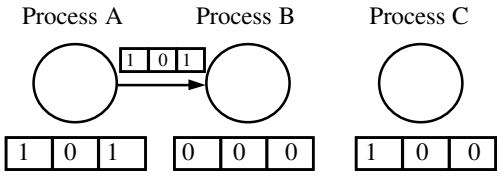
Don't deliver a message to a process until:

1). The process has received all earlier messages from the sending process

2). The process has received all earlier messages from third party processes received by the sending process



## A CBCAST Example



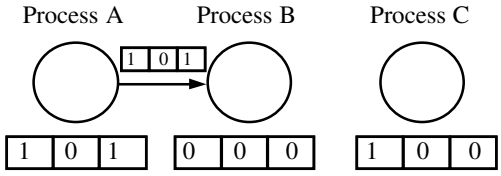

Process A has gotten a message from C

Process B hasn't gotten C's message yet

Process A sends a message to B (and C, not shown)



## CBCAST Example, Con't

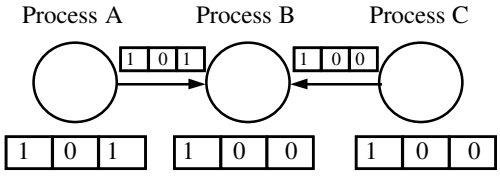



Process B cannot receive this message yet

Until he gets C's message



## CBCAST Example, Concluded




Now B can receive A's message

## Important Characteristics of CBCAST

Sites can always determine purely locally whether they can receive a message

Sites only wait for messages that should be coming to them

Sites need not send null messages to keep things going



## Can CBCAST Deadlock?

Deadlocks are based on waiting

- So when does CBCAST wait?

  When a message has arrived at a process, but cannot yet be delivered

- What is it waiting for?

  Other messages to arrive



## Is CBCAST's Message Waiting Safe?

The messages CBCAST waits for are messages that were delivered to (or originated with) the sender before the message *m* that's waiting

So they are not waiting for message *m*

Reliable delivery guarantees that the messages being waited for will arrive



## Other Issues in Isis

- Implementing ABCAST
- Delivery atomicity
- Handling group membership changes
- Minimizing communications for overhead
- Interactions of multiple overlapping groups
- Performance, performance, performance