So Far ...

- Message passing network
• Processes could *share* memory pages instead?
• Makes it convenient to write programs
• Reuse programs

<table>
<thead>
<tr>
<th>Process</th>
<th>Process</th>
<th>Process</th>
<th>Process</th>
</tr>
</thead>
</table>

write to page 5

read page 5

Page 0 | Page 1 | Page 2 | … | Page N-1
Distributed Shared Memory

- Distributed Shared Memory = processes virtually share pages
- How do you implement DSM over a message-passing network?

write to page 5

read page 5
1. Message-passing can be implemented over DSM!
   - Use a common page as buffer to read/write messages

2. DSM can be implemented over a message-passing network!
DSM over Message-Passing Network

- **Cache** maintained at each process
  - Cache stores pages accessed recently by that process
- Read/write first goes to cache

```
+----------------+   +----------------+   +----------------+   +----------------+
| Process        |   | Process        |   | Process        |   | Process        |
| Cache          |   | Cache          |   | Cache          |   | Cache          |
```

Centralized cache simulation using message passing.
DSM over Message-Passing Network (2)

• Pages can be mapped in local memory
• When page is present in memory, page hit
• Otherwise, *page fault* (kernel trap) occurs
  – Kernel trap handler: invokes the DSM software
  – May contact other processes in DSM group, via multicast
DSM: Invalidate Protocol

• Owner = Process with latest version of page
• Each page is in either R or W state
• When page in R state, owner has an R copy, but other processes may also have R copies
  – but no W copies exist
• When page is in W state, only owner has a copy
Process 1 Attempting a Read: **Scenario 1**

- Process 1 is owner \((O)\) and has page in R state
- *Read from cache. No messages sent.*
Process 1 Attempting a Read: Scenario 2

- Process 1 is owner (O) and has page in W state
- *Read from cache. No messages sent.*
Process 1 is owner \((O)\) and has page in R state

- Other processes also have page in R state
- \textit{Read from cache. No messages sent.}

\begin{itemize}
  \item Process 1
  \item Process 2
  \item Process 3
  \item Process 4
\end{itemize}
Process 1 Attempting a Read: Scenario 4

- Process 1 has page in R state
- Other processes also have page in R state, and someone else is owner
- Read from cache. No messages sent.

```
Process 1
    page (R)

Process 2

Process 3
    page (R)

Process 4
    page (R) (O)
```
Process 1 Attempting a Read: Scenario 5

- Process 1 does not have page
- Other process(es) has/have page in (R) state
- *Ask for a copy of page. Use multicast.*
- *Mark it as R*
- **Do Read**

```
Process 1

<table>
<thead>
<tr>
<th>Process 2</th>
<th>Process 3</th>
<th>Process 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>page (R)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>page (R) (O)</td>
</tr>
</tbody>
</table>
```
End State: Read Scenario 5

- Process 1 does not have page
- Other process(es) has/have page in (R) state
- *Ask for a copy of page. Use multicast.*
- *Mark it as R*
- *Do Read*

```
Process 1
  ________
  page (R)

Process 2

Process 3
  ________
  page (R)

Process 4
  ________
  page (R) (O)
```
Process 1 Attempting a Read: Scenario 6

- Process 1 does not have page
- Another process has page in (W) state
- *Ask other process to degrade its copy to (R). Locate process via multicast*
- *Get page; mark it as R*
- *Do Read*

```
Process 1
Process 2
Process 3
process 4
```

```
page (W) (O)
```
End State: Read Scenario 6

- Process 1 does not have page
- Another process has page in (W) state
- *Ask other process to degrade its copy to (R). Locate process via multicast*
- *Get page; mark it as R*
- *Do Read*

- Process 1
  - page (R)

- Process 2

- Process 3

- Process 4
  - page (R) (O)
Process 1 Attempting a Write: Scenario 1

- Process 1 is owner (O) and has page in W state
- Write to cache. No messages sent.
Process 1 Attempting a Write: Scenario 2

- Process 1 is owner \((O)\) has page in R state
- Other processes may also have page in R state
- *Ask other processes to invalidate their copies of page. Use multicast.*
- *Mark page as \((W)\).*
- *Do write.*

```
Process 1
    page (R)(O)
```

```
Process 2
    page (R)
```

```
Process 3
    page (R)
```

```
Process 4
    page (R)
```
End State: Write Scenario 2

• Process 1 is owner (O) has page in R state
• Other processes may also have page in R state
• Ask other processes to invalidate their copies of page. Use multicast.
• Mark page as (W).
• Do write.

Process 1
---
page (W)(O)

Process 2
---

Process 3
---
page (R)

Process 4
---
page (R)
Process 1 Attempting a Write: Scenario 3

- Process 1 has page in R state
- Other processes may also have page in R state, and someone else is owner
- *Ask other processes to invalidate their copies of page. Use multicast.*
- *Mark page as (W), become owner*
- *Do write*

![Diagram](image_url)
End State: Write Scenario 3

- Process 1 has page in R state
- Other processes may also have page in R state, and someone else is owner
- *Ask other processes to invalidate their copies of page. Use multicast.*
- *Mark page as (W), become owner*
- *Do write*

Process 1

Process 2

Process 3

Process 4

\begin{itemize}
\item page (W) (O)
\item page (R)
\item page (R) (O)
\end{itemize}
Process 1 Attempting a Write: Scenario 4

- Process 1 does not have page
- Other process(es) has/have page in (R) or (W) state
- *Ask other processes to invalidate their copies of the page. Use multicast.*
- *Fetch all copies; use the latest copy; mark it as (W); become owner*
- *Do Write*

```
Process 1

Process 2

Process 3
  page (R)

Process 4
  page (R) (O)
```
End State: Write Scenario 4

- Process 1 does not have page
- Other process(es) has/have page in (R) or (W) state
- *Ask other processes to invalidate their copies of the page. Use multicast.*
- *Fetch all copies; use the latest copy; mark it as (W); become owner*
- *Do Write*

```
Process 1
    ____________________________
    page (W) (O)

Process 2

Process 3
    ____________________________
    page (R)

Process 4
    ____________________________
    page (R) (O)
```
That was the invalidate approach

- If two processes write same page concurrently
  - Flip-flopping behavior where one process invalidates the other
  - Lots of network transfer
  - Can happen when unrelated variables fall on same page
  - Called false sharing

- Need to set page size to capture a process’ locality of interest
- If page size much larger, then have false sharing
- If page size much smaller, then too many page transfers => also inefficient
An Alternative Approach: Update

• Instead: could use **Update** approach
  – Multiple processes allowed to have page in W state
  – On a write to a page, multicast newly written value (or part of page) to all other holders of that page
  – Other processes can then continue reading and writing page

• **Update** preferable over **Invalidate**
  – When lots of sharing among processes
  – Writes are to small variables
  – Page sizes large

• Generally though, **Invalidate** better and preferred option
Consistency

• Whenever multiple processes share data, consistency comes into picture

• DSM systems can be implemented with:
  – Linearizability
  – Sequential Consistency
  – Causal Consistency
  – Pipelined RAM (FIFO) Consistency
  – Eventual Consistency
  – (Also other models like Release consistency)
  – These should be familiar to you from the course!

• As one goes down this order, speed increases while consistency gets weaker
DSM was very popular over a decade ago
But may be making a comeback now
  – Faster networks like Infiniband + SSDs => Remote Direct Memory Access (RDMA) becoming popular
  – Will this grow? Or stay the same as it is right now?
  – Time will tell!
Summary

- DSM = Distributed Shared Memory
  - Processes share pages, rather than sending/receiving messages
  - Useful abstraction: allows processes to use same code as if they were all running over the same OS (multiprocessor OS)
- DSM can be implemented over a message-passing interface
- Invalidate vs. Update protocols