Lecture 6

Synchronization

Distributed Shared Memory Architecture
Announcements
More synchronization problems
Discussion follows
Voelker and Marzullo
(CSE 120)
Bounded Buffer

- Problem: A set of resource buffers shared by producer and consumer threads
  - **Producer** inserts resources into the buffer set
  - **Consumer** removes resources from the buffer set

- Producer and consumer run at different rates
  - Tasks are independent
  - The buffer set allows each to run without explicit handoff

- Safety:
  - Sequence of consumed values is prefix of sequence of produced values
  - If $nc$ is number consumed, $np$ number produced, and $N$ the size of the buffer, then $0 \leq np - nc \leq N$
Synchronization variables

- \(0 \leq np - nc \leq N\) and \(0 \leq (nc - np) + N \leq N\)
- 2 counting semaphores
  - \textit{empty} – count of empty buffers
    - \(\text{empty} = (nc - np) + N\)
  - \textit{full} – count of full buffers
    - \(np - nc = \text{full}\)
- \textbf{Mutex} – mutual exclusion to shared set of buffers
Producer and Consumer Code

Semaphore mutex = 1;  // mutual exclusion to shared set of buffers
Semaphore empty = N;  // count of empty buffers (all empty to start)
Semaphore full = 0;    // count of full buffers (none full to start)

producer {
    while (1) {
        Produce new resource;
        wait(empty); // wait for empty buffer
        wait(mutex); // lock buffer list
        Add resource to an empty buffer;
        signal(mutex); // unlock buffer list
        signal(full);  // note a full buffer
    }
}

c consumer {
    while (1) {
        wait(full);       // wait for a full buffer
        wait(mutex);      // lock buffer list
        Remove resource from a full buffer;
        signal(mutex);    // unlock buffer list
        signal(EMPTY);    // note an empty buffer
        Consume resource;
    }
}
Synchronization issues

• Why need the mutex at all?
• Where are the critical sections?
• What are the conditions for deadlock
  – empty = 0 and full = 0
  – \((nc - np) + N = 0\) and \(np - nc = 0\)
  – \(N = 0\)
• What happens if operations on mutex and full/empty are swapped?
  – The pattern of signal/wait on full/empty is a common construct often called an interlock
End Discussion
Distributed Shared Memory
NUMA Architectures
NUMA Architectures

- Recall that the address space is global to all processors
- A **directory** keeps track of sharers, one for each block of memory
- Point-to-point messages manage coherence
Origin 2000 Interconnect

32 processor system

64 processor system
Inside a directory

- Each processor has a 1-bit “sharer” entry in the directory
- There is also a dirty bit and a PID identifying the owner in the case of a dirt block
Operation of a directory

- Assume a 4 processor system (only P0 & P1 shown)
- A is a location with home P1
- Initial directory entry for block containing A is empty
Operation of a directory

- P0 loads A
- Set directory entry for A (on P1) to indicate that P0 is a sharer
Operation of a directory

- P2, P3 load A (not shown)
- Set directory entry for A (on P1) to indicate that P0 is a sharer
Acquiring ownership of a block

• P0 writes A
• P0 becomes the owner of A
Acquiring ownership of a block

- P0 becomes the owner of A
- P1’s directory entry for A is set to Dirty
- Outstanding sharers are invalidated
- Access to line is blocked until all invalidations are acknowledged
Forwarding

Store A, #1
(home & owner)

P1

P0

A ← dirty

Load A

Directory

1 1 D P0
Forwarding

Store A, #1

\[\downarrow\]

(home & (P1)

owner)
Forwarding

Store A, #1

\[ \downarrow \]

(P1) ← (P2)

Store A, #2
Forwarding

Store A, #1

\[\downarrow\]

mark A as dirty

(P1) \[\leftarrow\] (P2) owner

Store A, #2
Forwarding

Store A, #1

\[ \text{mark } A \text{ as dirty} \]

(P1) \[\rightarrow\] (P2) owner

Store A, #2

(P3) Load A
Forwarding

Store A, #1

\[ \downarrow \quad \text{mark A as dirty} \]

(P1) \quad \leftrightarrow \quad \text{(P2) owner}

\quad \downarrow \quad \text{Store A, #2}

(P3) Load A
Forwarding

Store A, #1
↓
mark A as dirty
(P1) ←→ (P2) owner

Store A, #2

(P3) Load A
Performance issues

• Locality, locality, locality
• False sharing
Locality
Poor Locality
Quick primer on paging

• We group the physical and virtual address spaces into units called *pages*
• Pages are backed up on disk
• Virtual to physical mapping done by the Translation Lookaside Buffer (TLB), backs up page tables set up by the OS
• When we allocate a block of memory, we don’t need to allocate physical storage to pages; we do it on demand
Remote access latency

• When we allocate a block of memory, which processor(s) is (are) the owner(s)?
• Page allocation policies
  – First touch
  – Round robin
• Page placement and Page migration
• Copying v. redistribution
• Layout
Example

• Consider the following loop

\[
\begin{align*}
\text{for } r &= 0 \text{ to } n\text{Reps} \\
&\text{for } i = 0 \text{ to } n-1 \\
&\quad a[i] = b[i] + q*c[i]
\end{align*}
\]
Page Migration

\[ a[i] = b[i] + q \cdot c[i] \]

http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/0650/bks/SGI_Developer/books/OrOn2_PfTunesgi_html/ch08.html#id5224855
Migration eventually reaches the optimal time

Round robin initial, w/ migration
- Parallel initialization
- Serial initialization

Parallel initialization, first touch, migration

Parallel initialization, first touch (optimal)

One node initial placement, w/ migration
Cumulative effect of Page Migration
Migration Level

![Graph showing the migration level over iteration number and iteration time. The graph includes multiple lines with markers indicating specific iteration counts: 90, 70, 50, 30, and 10. The red line indicates 'migration off'.]
Eliminating false sharing
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
An example of false sharing

```c
float  a[m,n], s[m]
// Outer loop is in parallel
// Consider m=4, 128 byte cache line size
// Thread i updates  element s[i]
!$omp parallel do private(i,j), shared(s,a)
  for i = 0, m-1
    s[i] = 0.0
    for  j = 0, n-1
      s[i] += a[i,j]
    end for
  end for
```
Removing false sharing

float a[m,n], s[m,32]
!$omp parallel do private(i,j), shared(s,a)
for i = 0, m-1
  s[i,1] = 0.0
  for j = 0, n-1
    s[i,1] += a[i,j]
  end for
end for
Revisiting Memory consistency

• Recall the 3 conditions for consistency
  – Program order
  – Definition of a coherent view of memory
  – Serialization of writes

• Under a non-causal consistency model, it is possible for processor P2 to observe $A==1 \land B==1$ while processor P3 sees $B==1 \land A == 0$!

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=1</td>
<td>if (A==1)</td>
<td>if (B==1)</td>
</tr>
<tr>
<td>B=1</td>
<td></td>
<td>C=A</td>
</tr>
</tbody>
</table>