CS 152 Computer Architecture and Engineering

Lecture 19: Synchronization and Sequential Consistency

John Wawrzynek
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~johnw
http://inst.cs.berkeley.edu/~cs152
# CS152 Administrivia

## Endgame

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Chapter</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tue Nov 15</td>
<td>L19: Synchronization and Sequential Consistency</td>
<td>Ch. 5.4</td>
<td>Lab 4 due</td>
</tr>
<tr>
<td>Thu Nov 17</td>
<td>L20: Datacenters</td>
<td>Ch. 6</td>
<td>PS 5 due</td>
</tr>
<tr>
<td>Fri Nov 18</td>
<td>Section 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tue Nov 22</td>
<td>Quiz 5: Parallel Architectures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thu Nov 24</td>
<td>Holiday</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fri Nov 25</td>
<td>Section 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tue Nov 29</td>
<td>L21: Power and Energy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thu Dec 01</td>
<td>L22: The Emerging Trends and Wrapup</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Symmetric Multiprocessors

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)
Synchronization

The need for synchronization arises whenever there are concurrent processes in a system (even in a uniprocessor system)

Two classes of synchronization:

Producer-Consumer: A consumer process must wait until the producer process has produced data

Mutual Exclusion: Ensure that only one process uses a resource at a given time
A Producer-Consumer Example

Producer posting Item x:
- Load \( R_{\text{tail}} \), (tail)
- Store \((R_{\text{tail}}), x\)
- \(R_{\text{tail}} = R_{\text{tail}} + 1\)
- Store (tail), \(R_{\text{tail}}\)

Consumer:
- Load \(R_{\text{head}}, \text{(head)}\)
- spin:
  - Load \(R_{\text{tail}}, \text{(tail)}\)
  - if \(R_{\text{head}} == R_{\text{tail}}\) goto spin
  - Load \(R, \text{(R}_{\text{head}})\)
  - \(R_{\text{head}} = R_{\text{head}} + 1\)
  - Store (head), \(R_{\text{head}}\)
  - process(R)

The program is written assuming instructions are executed in order.

Problems?
Producer posting Item x:
Load $R_{\text{tail}}$, (tail)
1 Store ($R_{\text{tail}}$), x
$R_{\text{tail}} = R_{\text{tail}} + 1$
2 Store (tail), $R_{\text{tail}}$

Can the tail pointer get updated before the item x is stored?

Consumer:
Load $R_{\text{head}}$, (head)
3 spin:
Load $R_{\text{tail}}$, (tail)
if $R_{\text{head}} == R_{\text{tail}}$ goto spin
Load R, ($R_{\text{head}}$)
4 $R_{\text{head}} = R_{\text{head}} + 1$
Store (head), $R_{\text{head}}$
process(R)

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequence is:
2, 3, 4, 1
Sequential Consistency

A Memory Model

"A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport

Sequential Consistency =

arbitrary order-preserving interleaving

of memory references of sequential programs
Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

T1:
Store (X), 1 (X = 1)
Store (Y), 11 (Y = 11)

T2:
Load R₁, (Y)
Store (Y'), R₁ (Y' = Y)
Load R₂, (X)
Store (X'), R₂ (X' = X)

what are the legitimate answers for X' and Y'?

(X', Y') ∈ {(1, 11), (0, 10), (1, 10), (0, 11)} ?

If y is 11 then x cannot be 0
Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies.

What are these in our example?

T1:
- Store (X), 1 \( (X = 1) \)
- Store (Y), 11 \( (Y = 11) \)

T2:
- Load \( R_1 \), (Y)
- Store (Y'), \( R_1 \) \( (Y' = Y) \)
- Load \( R_2 \), (X)
- Store (X'), \( R_2 \) \( (X' = X) \)

additional SC requirements

Does (can) a system with caches or out-of-order execution capability provide a sequentially consistent view of the memory?

more on this later
Issues in Implementing Sequential Consistency

Implementation of SC is complicated by two issues

- **Out-of-order execution capability**
  
  Load(a); Load(b)  yes
  Load(a); Store(b)  yes if a \neq b
  Store(a); Load(b)  yes if a \neq b
  Store(a); Store(b)  yes if a \neq b

- **Caches**
  
  Caches can prevent the effect of a store from being seen by other processors – cache consistency protocols fix this!

  *No common commercial architecture has a sequentially consistent memory model!*
Memory Fences

Instructions to sequentialize memory accesses

Processors with relaxed or weak memory models (i.e., permit Loads and Stores to different addresses to be reordered) need to provide memory fence instructions to force the serialization of memory accesses.

Examples of processors with relaxed memory models:
- Sparc V8 (TSO,PSO): Membar
- Sparc V9 (RMO):
  - Membar #LoadLoad, Membar #LoadStore
  - Membar #StoreLoad, Membar #StoreStore
- PowerPC (WO): Sync, EIEIO
- ARM: DMB (Data Memory Barrier)
- X86/64: mfence (Global Memory Barrier)

Memory instructions operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.

Memory fences are expensive operations, however, one pays the cost of serialization only when it is required.
Using Memory Fences

Producer posting Item x:
- Load $R_{\text{tail}}$, (tail)
- Store ($R_{\text{tail}}$), x
- \text{Membar}_{SS}
- $R_{\text{tail}} = R_{\text{tail}} + 1$
- Store (tail), $R_{\text{tail}}$

Consumer:
- Load $R_{\text{head}}$, (head)
- \text{spin:}
  - Load $R_{\text{tail}}$, (tail)
  - if $R_{\text{head}} == R_{\text{tail}}$ goto spin
  - Load R, ($R_{\text{head}}$)
  - $R_{\text{head}} = R_{\text{head}} + 1$
  - Store (head), $R_{\text{head}}$
  - process(R)

\text{ensures that tail ptr is not updated before x has been stored}
Multiple Consumer Example

Producer posting Item x:
Load \( R_{\text{tail}} \), (tail)
Store (\( R_{\text{tail}} \)), x
\( R_{\text{tail}} = R_{\text{tail}} + 1 \)
Store (tail), \( R_{\text{tail}} \)

Consumer:
Load \( R_{\text{head}} \), (head)
spin:
Load \( R_{\text{tail}} \), (tail)
if \( R_{\text{head}} == R_{\text{tail}} \) goto spin
Load R, (\( R_{\text{head}} \))
\( R_{\text{head}} = R_{\text{head}} + 1 \)
Store (head), \( R_{\text{head}} \)

process(R)

Critical section:
Needs to be executed atomically by one consumer

What is wrong with this code?
Mutual Exclusion Using Load/Store

A protocol based on two shared variables c1 and c2. Initially, both c1 and c2 are 0 (not busy)

Process 1

...  
c1=1;  
L: if c2=1 then go to L  
< critical section>  
c1=0;

Process 2

...  
c2=1;  
L: if c1=1 then go to L  
< critical section>  
c2=0;

What is wrong? Deadlock!
Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

- Deadlock is not possible but with a low probability a *livelock* may occur.

- An unlucky process may never get to enter the critical section ⇒ *starvation*
A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables c1, c2 and turn. Initially, both c1 and c2 are 0 (not busy)

Process 1

\[
\text{...}
\text{c1=1; turn = 1;}
\text{L: if c2=1 & turn=1 then go to L}
\text{< critical section> c1=0;}
\]

Process 2

\[
\text{...}
\text{c2=1; turn = 2;}
\text{L: if c1=1 & turn=2 then go to L}
\text{< critical section> c2=0;}
\]

- \( \text{turn} = i \) ensures that only process \( i \) can wait
- variables c1 and c2 ensure mutual exclusion

Solution for \( n \) processes was given by Dijkstra and is quite tricky!
Analysis of Dekker’s Algorithm

Scenario 1

... Process 1
  c1=1;
  turn = 1;
  L: if c2=1 & turn=1
     then go to L
     < critical section>
  c1=0;

Scenario 2

... Process 1
  c1=1;
  turn = 1;
  L: if c2=1 & turn=1
     then go to L
     < critical section>
  c1=0;

... Process 2
  c2=1;
  turn = 2;
  L: if c1=1 & turn=2
     then go to L
     < critical section>
  c2=0;

Scenario 2

... Process 1
  c1=1;
  turn = 1;
  L: if c2=1 & turn=1
     then go to L
     < critical section>
  c1=0;

... Process 2
  c2=1;
  turn = 2;
  L: if c1=1 & turn=2
     then go to L
     < critical section>
  c2=0;
N-process Mutual Exclusion

*Lamport’s Bakery Algorithm*

**Process $i$**

**Entry Code**

Initially $\text{num}[j] = 0$, for all $j$

$\text{choosing}[i] = 1$;
\[
\text{num}[i] = \max(\text{num}[0], \ldots, \text{num}[N-1]) + 1;
\]
$\text{choosing}[i] = 0$;

for ($j = 0$; $j < N$; $j++$) {
    while ($\text{choosing}[j]$);
    while ($\text{num}[j] &&$
        ($\text{num}[j] < \text{num}[i]$ ||
            ($\text{num}[j] == \text{num}[i]$ && $j < i$)));

} 

**Exit Code**

$\text{num}[i] = 0$;
A *semaphore* is a non-negative integer, with the following operations:

\[ P(s) : \text{if } s > 0, \text{ decrement } s \text{ by 1, otherwise wait} \]

\[ V(s) : \text{increment } s \text{ by 1 and wake up one of the waiting processes} \]

P’s and V’s must be executed **atomically**, i.e., without
- **interruptions** or
- **interleaved accesses to** \( s \) **by** other processors

**Process i**

\[ P(s) \]

\[ <\text{critical section}> \]

\[ V(s) \]

**initial value of** \( s \) **determines the maximum no. of processes in the critical section**
## Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design...

Simpler solution:

*atomic read-modify-write instructions*

Examples: *m is a memory location, R is a register*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test&amp;Set (m), R:</td>
<td>[R ← M[m]; if R==0 then M[m] ← 1;]</td>
</tr>
<tr>
<td>Fetch&amp;Add (m), R, Rv, R:</td>
<td>[R ← M[m]; M[m] ← R + Rv;]</td>
</tr>
<tr>
<td>Swap (m), R:</td>
<td>[Rt ← M[m]; M[m] ← R; R ← Rt;]</td>
</tr>
</tbody>
</table>
Multiple Consumers Example

*using the Test&Set Instruction*

**P:**
Test&Set (mutex), $R_{\text{temp}}$
if ($R_{\text{temp}}! = 0$) goto P

**spin:**
Load $R_{\text{head}}$, (head)
Load $R_{\text{tail}}$, (tail)
if $R_{\text{head}} == R_{\text{tail}}$ goto spin
Load $R$, ($R_{\text{head}}$)
$R_{\text{head}} = R_{\text{head}} + 1$
Store (head), $R_{\text{head}}$

**V:**
Store (mutex), 0
process(R)

Critical Section

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P’s and V’s
Nonblocking Synchronization

Compare&Swap(m), R_t, R_s:
  if (R_t == M[m])
    then M[m] = R_s;
    R_s = R_t;
    status ← success;
  else status ← fail;

status is an implicit argument

try:
  Load R_head, (head)
  Load R_tail, (tail)
  if R_head == R_tail goto spin
  Load R, (R_head)
  R_newhead = R_head + 1
  Compare&Swap(head), R_head, R_newhead
  if (status == fail) goto try
  process(R)
Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-reserve R, (m):
<flag, adr> ← <1, m>;
R ← M[m];

Store-conditional (m), R:
if <flag, adr> == <1, m>
then cancel other procs’ reservation on m;
    M[m] ← R;
    status ← succeed;
else status ← fail;

try:
    Load-reserve R_{\text{head}}, (head)
    Load R_{\text{tail}}, (tail)
    if R_{\text{head}} == R_{\text{tail}} goto spin
    Load R, (R_{\text{head}})
    R_{\text{head}} = R_{\text{head}} + 1
    Store-conditional (head), R_{\text{head}}
if (status==fail) goto try
process(R)
Performance of Locks

Blocking atomic read-modify-write instructions
  e.g., Test&Set, Fetch&Add, Swap
vs
Non-blocking atomic read-modify-write instructions
  e.g., Compare&Swap,
  Load-reserve/Store-conditional
vs
Protocols based on ordinary Loads and Stores

Performance depends on several interacting factors:
  degree of contention,
caches,
out-of-order execution of Loads and Stores

later ...
Acknowledgements

These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252