CS 152 Computer Architecture and Engineering

Lecture 19: Synchronization and Sequential Consistency

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.cs.berkeley.edu/~cs152
Summary: Multithreaded Categories

Time (processor cycle)

Superscalar

Fine-Grained

Coarse-Grained

Multiprocessing

Simultaneous Multithreading

Thread 1

Thread 2

Thread 3

Thread 4

Thread 5

Idle slot

April 8, 2010

CS152, Spring 2010
Uniprocessor Performance (SPECint)


- **VAX**: 25%/year 1978 to 1986
- **RISC + x86**: 52%/year 1986 to 2002
- **RISC + x86**: ??%/year 2002 to present

April 8, 2010
Parallel Processing: Déjà vu all over again?

“… today’s processors … are nearing an impasse as technologies approach the speed of light..”


- Transputer had bad timing (Uniprocessor performance↑)
  ⇒ Procrastination rewarded: 2X seq. perf. / 1.5 years

- “We are dedicating all of our future product development to multicore designs. … This is a sea change in computing”

  Paul Otellini, President, Intel (2005)

- All microprocessor companies switch to MP (2X CPUs / 2 yrs)
  ⇒ Procrastination penalized: 2X sequential perf. / 5 yrs

<table>
<thead>
<tr>
<th>Manufacturer/Year</th>
<th>AMD/’09</th>
<th>Intel/’09</th>
<th>IBM/’09</th>
<th>Sun/’09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors/chip</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Threads/Processor</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Threads/chip</td>
<td>6</td>
<td>16</td>
<td>32</td>
<td>128</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).

*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 $R_{\text{head}}$, (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)

The program is written assuming instructions are executed in order.
A Producer-Consumer Example

continued

Producer posting Item x:

Load $R_{\text{tail}}$, (tail)

1 Store $(R_{\text{tail}})$, x

2 $R_{\text{tail}} = R_{\text{tail}} + 1$

Consumer:

Load $R_{\text{head}}$, (head)

spin:

Load $R_{\text{tail}}$, (tail)

if $R_{\text{head}} == R_{\text{tail}}$
goto spin

Load R, $(R_{\text{head}})$

4 $R_{\text{head}} = R_{\text{head}} + 1$

Store (head), $R_{\text{head}}$

process(R)

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

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

Problem sequences are:

2, 3, 4, 1
4, 1, 2, 3
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)} ?
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 R₁, (Y)
- Store (Y’), R₁ (Y’ = Y)
- Load R₂, (X)
- Store (X’), R₂ (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
Multiple Consumer Example

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

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

Critical section:
Needs to be executed atomically by one consumer ⇒ locks

What is wrong with this code?

Critical section:
Needs to be executed atomically by one consumer ⇒ locks
Locks or Semaphores
E. W. Dijkstra, 1965

A *semaphore* is a non-negative integer, with the following operations:

- **P(s)**: *if* $s > 0$, decrement $s$ by 1, otherwise wait
- **V(s)**: increment $s$ 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 $i$

<table>
<thead>
<tr>
<th>P(s)</th>
<th>&lt;critical section&gt;</th>
<th>V(s)</th>
</tr>
</thead>
</table>

*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*

Test&Set (m), R:
R ← M[m];
if R==0 then
M[m] ← 1;

Fetch&Add (m), Rv, R:
R ← M[m];
M[m] ← R + Rv;

Swap (m), R:
Rt ← M[m];
M[m] ← R;
R ← Rt;
CS152 Administrivia
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)

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

What if the process stops or is swapped out while in the critical section?
Nonblocking Synchronization

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

try:
spin:

Load $R_{\text{head}}$, (head)
Load $R_{\text{tail}}$, (tail)
if $R_{\text{head}} == R_{\text{tail}}$ goto spin
Load $R$, ($R_{\text{head}}$)
$R_{\text{newhead}} = R_{\text{head}} + 1$
Compare\&Swap(head), $R_{\text{head}}$, $R_{\text{newhead}}$
if (status==fail) goto try
process(R)

status is an implicit argument
Load-reserve & Store-conditional

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

Load-reserve \( R, (m) \):
\[
<\text{flag}, \text{adr}> \leftarrow <1, m>;
\]
\[
R \leftarrow M[m];
\]

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

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

Store-conditional \( (m), R \):
\[
\text{if } <\text{flag}, \text{adr}> == <1, m> \text{ then cancel other procs’ reservation on } m;
\]
\[
M[m] \leftarrow R;
\]
\[
\text{status } \leftarrow \text{succeed};
\]
\[
\text{else status } \leftarrow \text{fail};
\]
Performance of Locks

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

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

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 ≠ b
  Store(a); Load(b)   yes if a ≠ b
  Store(a); Store(b)  yes if a ≠ b

- **Caches**
  
  Caches can prevent the effect of a store from being seen by other processors
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

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 $R_{tail}$, (tail)
- Store $(R_{tail})$, $x$
- Membar$_{SS}$
  - $R_{tail} = R_{tail} + 1$
- Store (tail), $R_{tail}$

Ensures that tail ptr is not updated before $x$ has been stored

Consumer:
- Load $R_{head}$, (head)
- Spin:
  - Load $R_{tail}$, (tail)
  - if $R_{head} == R_{tail}$ goto spin
  - Membar$_{LL}$
  - Load $R$, $(R_{head})$
  - $R_{head} = R_{head} + 1$
  - Store (head), $R_{head}$

Process($R$)

Ensures that $R$ is not loaded before $x$ has been stored
Mutual Exclusion Using Load/Store

A protocol based on two shared variables \( c_1 \) and \( c_2 \). Initially, both \( c_1 \) and \( c_2 \) are 0 (not busy)

**Process 1**

\[
\ldots
\]
\[
c_1=1;
\]
\[
L: \ \text{if } c_2=1 \ \text{then go to } L
\]
\[
< \text{critical section}>
\]
\[
c_1=0;
\]

**Process 2**

\[
\ldots
\]
\[
c_2=1;
\]
\[
L: \ \text{if } c_1=1 \ \text{then go to } L
\]
\[
< \text{critical section}>
\]
\[
c_2=0;
\]

What is wrong?
Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets $c_1$ 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)

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

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

- 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;
N-process Mutual Exclusion
Lamport’s Bakery Algorithm

Process i

Entry Code

choosing[i] = 1;
num[i] = max(num[0], ..., num[N-1]) + 1;
choosing[i] = 0;

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

}  

Exit Code

num[i] = 0;

Initially num[j] = 0, for all j

Initially num[j] = 0, for all j

Entry Code

choosing[i] = 1;

num[i] = max(num[0], ..., num[N-1]) + 1;

choosing[i] = 0;

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

}  

Exit Code

num[i] = 0;
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