CSE 490/590, Spring 2011

CSE 490/590 Computer Architecture

Synchronization and Consistency II

Steve Ko
Computer Sciences and Engineering
University at Buffalo

Last time...
- Consistency problem
  - Producer-consumer
- Sequential consistency
- Semaphores
- Instructions that can implement semaphores
  - Test&set
  - Fetch&add
  - Swap

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
- interleaved accesses to s by other processors

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;

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

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

Nonblocking Synchronization

status is an implicit argument

Try:
Load Rnew (head)
if Rnew==R then goto try

Load Rnew (tail)
if Rnew==R then goto try

Load R (Rnew) + 1
Compare&Swap(head), Rnew, Rnewnew
if (status==fail) goto try
process(R)

Multiple Consumers Example
using the Test&Set Instruction

P:
Test&Set (mutex), Rtemp
if (Rtemp==0) goto P

Load Rnew, (head)
if Rnew==Rnew then goto spin

Load R, (Rnew)
Rnew := Rnew + 1
Store (head), Rnew

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?

Critical Section
Load-reserve & Store-conditional

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

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

Store-conditional \( R \), \( m \):
\[ \text{if } \langle \text{flag, addr} \rangle = <1, m> \]
\[ \text{then cancel other proc's reservation on } m; \]
\[ M[m] \rightarrow R; \]
\[ \text{status} \rightarrow \text{succeed}; \]
\[ \text{else status} \rightarrow \text{fail}; \]

Performance of Locks

Blocking atomic read-modify-write instructions
e.g., Test&Set, Fetch&Add, Swap

Non-blocking atomic read-modify-write instructions
e.g., Compare&Swap,
Load-reserve/Store-conditional

Protocols based on ordinary Loads and Stores

Performance depends on several interacting factors:
degree of contentation,
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 \( a \neq b \)
  
  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

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:

Producer: Load \( R_{\text{head}} \) (tail)

\[ \text{Load } R_{\text{head}} \rightarrow \text{tail} \]

Membar:

\[ R_{\text{tail}} \rightarrow R_{\text{head}} + 1 \]

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

Consumer: Load \( R_{\text{head}} \) (head)

\[ \text{Load } R_{\text{head}} \rightarrow \text{head} \]

Membar:

\[ R_{\text{head}} \rightarrow R_{\text{head}} + 1 \]

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

process(R)

ensures that tail ptr is not updated before x has been stored

ensures that R is not loaded before x has been stored

CSE 490/590 Administirzia

- No class on Friday, 4/15
- Keyboards available for pickup at my office
  - Today after 2pm
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

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

Process 2

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

What is wrong?

Mutual Exclusion: second attempt

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

Process 1

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

Process 2

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

• 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

Acknowledgements

• These slides heavily contain material developed and copyright by:
  – Krste Asanovic (MIT/UCB)
  – David Patterson (UCB)
• And also by:
  – Arvind (MIT)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
• MIT material derived from course 6.823
• UCB material derived from course CS252