Symmetric Multiprocessors: Synchronization and Sequential Consistency

Arvind
Computer Science and Artificial Intelligence Lab
M.I.T.

Based on the material prepared by Arvind and Krste Asanovic
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 parallel processes in a system (even in a uniprocessor system)

Forks and Joins: In parallel programming a parallel process may want to wait until several events have occurred

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

Exclusive use of a resource: Operating system has to ensure that only one process uses a resource at a given time

November 7, 2005
A Producer-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, $R_{head}$)
  - process($R$)

The program is written assuming instructions are executed in order.

Problems?

November 7, 2005
A Producer-Consumer Example

Producer posting Item x:

1. \( \text{Load}(R_{\text{tail}}, \text{tail}) \)
2. \( \text{Store}(R_{\text{tail}}, x) \)
3. \( R_{\text{tail}} = R_{\text{tail}} + 1 \)
4. \( \text{Store}(\text{tail}, R_{\text{tail}}) \)

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

Consumer:

1. \( \text{Load}(R_{\text{head}}, \text{head}) \)
2. \( \text{spin: } \text{Load}(R_{\text{tail}}, \text{tail}) \)
3. \( \text{if } R_{\text{head}} == R_{\text{tail}} \text{ goto spin} \)
4. \( \text{Load}(R, R_{\text{head}}) \)
5. \( R_{\text{head}} = R_{\text{head}} + 1 \)
6. \( \text{Store}(\text{head}, R_{\text{head}}) \)
7. \( \text{process}(R) \)

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

Problem sequences are:

- 2, 3, 4, 1
- 4, 1, 2, 3

November 7, 2005
Sequential Consistency
A Memory Model

“A system is \textit{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”

\textit{Leslie Lamport}

Sequential Consistency =

arbitrary \textit{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(R\textsubscript{1}, Y)
Store(Y', R\textsubscript{1}) (Y' = Y)
Load(R\textsubscript{2}, X)
Store(X', R\textsubscript{2}) (X' = X)

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

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

If y is 11 then x cannot be 0

November 7, 2005
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, $R_{tail}$)

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

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, $R_{head}$)
- process(R)

What is wrong with this code?

November 7, 2005
Locks or Semaphores
E. W. Dijkstra, 1965

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

\[
\begin{array}{c}
\text{Process } i \\
P(s) \\
\text{<critical section>} \\
V(s)
\end{array}
\]

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

November 7, 2005
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):

\[
\begin{align*}
R &\leftarrow M[m]; \\
\text{if } R == 0 \text{ then } M[m] &\leftarrow 1;
\end{align*}
\]

Fetch&Add(m, R, R):

\[
\begin{align*}
R &\leftarrow M[m]; \\
M[m] &\leftarrow R + R_v;
\end{align*}
\]

Swap(m, R):

\[
\begin{align*}
R_t &\leftarrow M[m]; \\
M[m] &\leftarrow R; \\
R &\leftarrow R_t;
\end{align*}
\]
Multiple Consumers Example
using the Test&Set Instruction

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

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

```plaintext
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:
  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)
```

status is an implicit argument

November 7, 2005
Load-reserve & Store-conditional

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

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

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

try:
Load-reserve(R_{head}, head)
Load(R_{tail}, tail)
if R_{head} == R_{tail} goto spin
Load(R, R_{head})
R_{head} = R_{head} + 1
Store-conditional(head, R_{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 ...*
Issues in Implementing Sequential Consistency

Implementation of SC is complicated by two issues

• **Our-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

November 7, 2005
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

November 7, 2005
Data-Race Free Programs  
a.k.a. Properly Synchronized Programs

Process 1

...  
Acquire(mutex);  
< critical section>  
Release(mutex);

Process 2

...  
Acquire(mutex);  
< critical section>  
Release(mutex);

Synchronization variables (e.g. mutex) are disjoint from data variables  
Accesses to writable shared data variables are protected in critical regions  
⇒ no data races except for locks  
(Formal definition is elusive)

In general, it cannot be proven if a program is data-race free.

November 7, 2005
Fences in Data-Race Free Programs

Process 1
...
Acquire(mutex);
membar;
< critical section>
membar;
Release(mutex);

Process 2
...
Acquire(mutex);
membar;
< critical section>
membar;
Release(mutex);

- Relaxed memory model allows reordering of instructions by the compiler or the processor as long as the reordering is not done across a fence
- The processor also should not speculate or prefetch across fences
Five-minute break to stretch your legs
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.

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

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

- An unlucky process may never get to enter the critical section \(\Rightarrow\) \textit{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**

```
...  
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;
```

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

... 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 num[j] = 0, for all j

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;
next time

Effect of caches on Sequential Consistency
Thank you!