CS 152 Computer Architecture and Engineering

Lecture 17: 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
Last Time, Lecture 16: GPUs

• Data-Level Parallelism the least flexible but cheapest form of machine parallelism, and matches application demands

• Graphics processing units have developed general-purpose processing capability for use outside of traditional graphics functionality (GP-GPUs)

• SIMT model presents programmer with illusion of many independent threads, but executes them in SIMD style on a vector-like multilane engine.

• Complex control flow handled with hardware to turn branches into mask vectors and stack to remember µthreads on alternate path

• No scalar processor, so µthreads do redundant work, unit-stride loads and stores recovered via hardware memory coalescing
Uniprocessor Performance (SPECint)


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

April 3, 2012
CS152, Spring 2012
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↑)
  $\implies$ 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 (2+ CPUs/2 yrs)
  $\implies$ Procrastination penalized: 2X sequential perf. / 5 yrs

- Even handheld systems moved to multicore
  - Nintendo 3DS, iPhone4S, iPad 3 have two cores each (plus additional specialized cores)
  - Playstation Portable Vita has four cores
Symmetric Multiprocessors

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

**symmetric**
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 $R_{\text{head}}$, (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.

Problems?
A Producer-Consumer Example

continued

Producer posting Item x:

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

Consumer:

1. Load $R_{\text{head}}$, (head)
2. spin:
   3. Load $R_{\text{tail}}$, (tail)
   4. if $R_{\text{head}} == R_{\text{tail}}$ goto spin
   5. Load R, ($R_{\text{head}}$)
   6. $R_{\text{head}} = R_{\text{head}} + 1$
   7. Store (head), $R_{\text{head}}$
   8. 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 R_1, (Y)
  Store (Y'), R_1 (Y' = Y)
  Load R_2, (X)
  Store (X'), R_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*
Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (→)

What are these in our example?

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

T2:
1. Load R₁, (Y)
2. Store (Y’), R₁ (Y’ = Y)
3. Load R₂, (X)
4. 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
Issues in Implementing Sequential Consistency

Implementation of SC is complicated by two issues

- **Out-of-order execution capability**
  
  - Load(a); Load(b)  
  - Load(a); Store(b)  
  - Store(a); Load(b)  
  - Store(a); Store(b)

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

*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

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
CS152 Administrivia

• Quiz 4 (VLIW, Vectors, Multithreading)
• Next Tuesday April 10
• Covers L13-L16, PS4, Lab 4

• PS4 due in class Thursday April 8
• Lab 4 due Tuesday April 10 (same day as quiz)
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), $R_{head}$
- process(R)

Critical section:
Needs to be executed atomically by one consumer

What is wrong with this code?

April 3, 2012 CS152, Spring 2012
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 $c_1$, $c_2$ and $\text{turn}$. Initially, both $c_1$ and $c_2$ are 0 (not busy)

**Process 1**

\[
\begin{align*}
\ldots \\
c_1 &= 1; \\
\text{turn} &= 1; \\
\text{L: if } c_2 &= 1 \& \text{turn} = 1 \\
\text{then go to L} \\
\text{< critical section>} \\
c_1 &= 0;
\end{align*}
\]

**Process 2**

\[
\begin{align*}
\ldots \\
c_2 &= 1; \\
\text{turn} &= 2; \\
\text{L: if } c_1 &= 1 \& \text{turn} = 2 \\
\text{then go to L} \\
\text{< critical section>} \\
c_2 &= 0;
\end{align*}
\]

- $\text{turn} = i$ ensures that only process $i$ can wait
- Variables $c_1$ and $c_2$ ensure mutual exclusion

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

Scenario 1

Process 1

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

Process 2

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

Scenario 2

Process 1

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

Process 2

\[ \text{c2} = 1; \]
\[ \text{turn} = 2; \]
\[ L: \text{if } \text{c1} = 1 \& \text{turn} = 2 \quad \text{then go to } L \]
\[ \text{< critical section>} \]
\[ \text{c2} = 0; \]
N-process Mutual Exclusion

Lamport’s Bakery Algorithm

Process i

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;
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 $i$**

\[
P(s) \quad \text{P(s)}
\]

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

\[
V(s) \quad \text{V(s)}
\]

*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*

Test&Set (m), R:

\[ R \leftarrow M[m]; \\
\text{if } R == 0 \text{ then} \\
M[m] \leftarrow 1; \]

Fetch&Add (m), R_V, R:

\[ R \leftarrow M[m]; \\
M[m] \leftarrow R + R_V; \]

Swap (m), R:

\[ R_t \leftarrow M[m]; \\
M[m] \leftarrow R; \\
R \leftarrow R_t; \]
Multiple Consumers Example

using the Test&Set Instruction

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

Critical Section

What if the process stops or is swapped out while in the 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:
spin:
    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];

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)

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

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

Blocking atomic read-modify-write instructions
  \textit{e.g., Test\&Set, Fetch\&Add, Swap}
vs
Non-blocking atomic read-modify-write instructions
  \textit{e.g., 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

\textit{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