CIS 429/529
Symmetric Multiprocessors: Synchronization and Sequential Consistency

Prof. Michel A. Kinsky
• Symmetric: All memory is equally far away from all processors
Synchronization

• The need for synchronization arises whenever there are parallel processes in a system (even in a uniprocessor system)
  
  ‣ Forks and Joins: 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
Producer-Consumer

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?
Producer-Consumer

Producer posting Item $x$:

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

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

Consumer:

Load $R_{head}$, (head)

spin:

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

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

Problem sequences are: 2, 3, 4, 1
Sequential Consistency

“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 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)}?
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)

What is wrong with this code?
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

<table>
<thead>
<tr>
<th>Process i</th>
<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:**
  \[
  \begin{align*}
  &R \leftarrow M[m]; \\
  &\text{if } R==0 \text{ then} \\
  &M[m] \leftarrow 1;
  \end{align*}
  \]

- **Fetch&Add (m), R, V:**
  \[
  \begin{align*}
  &R \leftarrow M[m]; \\
  &M[m] \leftarrow R + V;
  \end{align*}
  \]

- **Swap (m), R:**
  \[
  \begin{align*}
  &R_t \leftarrow M[m]; \\
  &M[m] \leftarrow R; \\
  &R \leftarrow R_t;
  \end{align*}
  \]
• Using the Test&Set Instruction

<table>
<thead>
<tr>
<th>P:</th>
<th>Test&amp;Set (mutex), $R_{temp}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>if ($R_{temp}$!=0) goto P</td>
</tr>
<tr>
<td>spin:</td>
<td>Load $R_{head}$, (head)</td>
</tr>
<tr>
<td></td>
<td>Load $R_{tail}$, (tail)</td>
</tr>
<tr>
<td></td>
<td>if $R_{head}$==$R_{tail}$ goto spin</td>
</tr>
<tr>
<td></td>
<td>Load R, ($R_{head}$)</td>
</tr>
<tr>
<td></td>
<td>$R_{head}=$$R_{head}$+1</td>
</tr>
<tr>
<td></td>
<td>Store head, $R_{head}$</td>
</tr>
<tr>
<td>V:</td>
<td>Store mutex, 0</td>
</tr>
<tr>
<td></td>
<td>process(R)</td>
</tr>
</tbody>
</table>

- 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**

**Compare&Swap(m), R_t, R_s:**

- if ($R_t == M[m]$)
  - then $M[m] = R_s$
  - $R_t = R_s$
  - status $\leftarrow$ success;
- else status $\leftarrow$ fail;

**try:**

- Load $R_{head}$, (head)

**spin:**

- 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
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> ← <I, m>;
  R ← M[m];

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

**try:**
**spin:**
- Load-reserve R\(_{head}\), (head)
- Load R\(_{tail}\), (tail)
- if R\(_{head}\) == R\(_{tail}\) goto spin
- Load R, (R\(_{head}\))
- R\(_{head}\) = R\(_{head}\) + I
- Store-conditional head, R\(_{head}\)
- if (status == fail) goto try
- process(R)
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*)

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$c_1 = 1; $</td>
<td>$c_2 = 1; $</td>
</tr>
<tr>
<td><strong>L:</strong> ( \text{if } c_2 = 1 \text{ then go to L} )</td>
<td><strong>L:</strong> ( \text{if } c_1 = 1 \text{ then go to L} )</td>
</tr>
<tr>
<td>&lt; critical section&gt;</td>
<td>&lt; critical section&gt;</td>
</tr>
<tr>
<td>$c_1 = 0;$</td>
<td>$c_2 = 0;$</td>
</tr>
</tbody>
</table>

**What is wrong?**

Computer Architecture and Embedded Systems Laboratory (CAES Lab)
To avoid deadlock, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

What could go wrong?
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**
- \( c_1 = 1; \)
- \( \text{turn} = 1; \)
- \( \text{L: if} \ c_2 = 1 \ \& \ \text{turn} = 1 \)
  - then go to \( L \)
  - < critical section >
- \( c_1 = 0; \)

**Process 2**
- \( c_2 = 1; \)
- \( \text{turn} = 2; \)
- \( \text{L: if} \ c_1 = 1 \ \& \ \text{turn} = 2 \)
  - then go to \( L \)
  - < critical section >
- \( c_2 = 0; \)

**Scenario 2**

**Process 1**
- \( c_1 = 1; \)
- \( \text{turn} = 1; \)
- \( \text{L: if} \ c_2 = 1 \ \& \ \text{turn} = 1 \)
  - then go to \( L \)
  - < critical section >
- \( c_1 = 0; \)

**Process 2**
- \( c_2 = 1; \)
- \( \text{turn} = 2; \)
- \( \text{L: if} \ c_1 = 1 \ \& \ \text{turn} = 2 \)
  - then go to \( L \)
  - < critical section >
- \( c_2 = 0; \)
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
Issues in Implementing Sequential Consistency

• Implementation of SC is complicated by two issues
  
  ▶ Caches
    • Caches can prevent the effect of a store from
    • being seen by other processors
Memory Fences

- 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
Memory Fences

• 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:
  ‣ 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 \( R_{\text{tail}} \) (tail)
- Store \((R_{\text{tail}}), x\)
- Membar_{ss} \( R_{\text{tail}} = R_{\text{tail}} + 1 \)
- 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
- Membar_{LL} \( R_{\text{tail}} = R_{\text{tail}} + 1 \)
- Load \( R_{\text{head}} \)
- Store head, \( R_{\text{head}} \)
- process(R)

What does this do?
Data-Race Free Programs

- 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.
Fences in Data-Race Free Programs

- 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.

Process 1

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

Process 2

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