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)
- \( R_{\text{tail}} = R_{\text{tail}} + 1 \)
- Store \( x \), (\( R_{\text{tail}} \))
- Store \( R_{\text{tail}} \), (tail)

Consumer:
- Load \( R_{\text{head}} \), (head)
- \( R_{\text{head}} = R_{\text{head}} + 1 \)
- Load \( R \), (\( R_{\text{head}} \))
- 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 $x$, ($R_{\text{tail}}$)
3. $R_{\text{tail}} = R_{\text{tail}} + 1$
4. Store $R_{\text{tail}}$, (tail)

**Consumer**:

1. Load $R_{\text{head}}$, (head)
2. Spin:
3. Load $R_{\text{tail}}$, (tail)
   - if $R_{\text{head}} == R_{\text{tail}}$ goto spin
4. Load $R$, ($R_{\text{head}}$)
   - $R_{\text{head}} = R_{\text{head}} + 1$
   - Store $R_{\text{head}}$, (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 1, (X) (X = 1)
  Store 11, (Y) (Y = 11)

T2:
  Load R₁, (Y)
  Store R₁, (Y’) (Y’ = Y)
  Load R₂, (X)
  Store R₂, (X’) (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 1, (X) \(X = 1\)
- Store 11, (Y) \(Y = 11\)

T2:
- Load \(R_1\), (Y)
- Store \(R_1\), (Y') \(Y' = Y\)
- Load \(R_2\), (X)
- Store \(R_2\), (X') \(X' = X\)

➡️ 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 x, ($R_{tail}$)
- $R_{tail} = R_{tail} + 1$
- Store $R_{tail}$, (tail)

Critical section:
Needs to be executed atomically by one consumer $\Rightarrow$ 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 $R_{head}$, (head)
- process(R)

What is wrong with this code?

Critical section:
Needs to be executed atomically by one consumer $\Rightarrow$ locks

Critical section:
Needs to be executed atomically by one consumer $\Rightarrow$ 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
• *interusions* or
• *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 ($m$), $R$:  
\[
\begin{align*}
R &\leftarrow M[m]; \\
\text{if } R &= 0 \\
M[m] &\leftarrow 1;
\end{align*}
\]

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

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

\[
\text{spin:} \quad \\
\text{Load } R_{\text{head}}, (\text{head}) \\
\text{Load } R_{\text{tail}}, (\text{tail}) \\
\text{if } R_{\text{head}} == R_{\text{tail}} \text{ goto spin} \\
\text{Load } R, (R_{\text{head}}) \\
R_{\text{head}} = R_{\text{head}} + 1 \\
\text{Store } R_{\text{head}}, (\text{head})
\]

V:
- Store 0, (mutex)
- 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

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

If memory location still equals old value
then swap in the new value

\[
\begin{align*}
\text{try:} & \quad \text{Load } R_{\text{head}}, (\text{head}) \\
\text{spin:} & \quad \text{Load } R_{\text{tail}}, (\text{tail}) \\
& \quad \text{if } R_{\text{head}}==R_{\text{tail}} \text{ goto spin} \\
& \quad \text{Load } R, (R_{\text{head}}) \\
& \quad R_{\text{newhead}} = R_{\text{head}} + 1 \\
& \quad \text{Compare}\&\text{Swap}(\text{head}), R_{\text{head}}, R_{\text{newhead}} \\
& \quad \text{if } (\text{status}==\text{fail}) \text{ goto try} \\
& \text{process}(R)
\end{align*}
\]

status is an \textit{implicit} argument

A-B-A
Problem!
CAS Double
Load-reserve & Store-conditional

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

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

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

**try:**
**spin:**
- Load-reserve $R_{\text{head}}$, (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-conditional $R_{\text{head}}$, (head)
- if (status==fail) goto try
- process(R)
Performance of Locks

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

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

\textit{later \ldots}
Issues in Implementing Sequential Consistency

Implementation of SC is complicated by

- *Out-of-order execution in processor and between memory banks capability*
  
  Load(a); Load(b) \( \rightarrow \) yes
  
  Load(a); Store(b) \( \rightarrow \) yes if \( a \neq b \)
  
  Store(a); Load(b) \( \rightarrow \) yes if \( a \neq b \)
  
  Store(a); Store(b) \( \rightarrow \) yes if \( a \neq b \)

Essentially need to use in-order execution of load/stores

- Caches

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

**SC complications motivate architects to consider weak or relaxed memory models**
Memory Fences

*Instructions to sequentialize memory accesses*

Processors with *relaxed or weak memory models* permit Loads and Stores to different addresses to be reordered, remove some/all extra dependencies imposed by SC

LL, LS, SL, SS

need to provide *memory fence* instructions to force the serialization of memory accesses

*Examples of relaxed memory models:*
- Total Store Order: LL, LS, SS, enforce SL with fence
- Partial Store Order: LL, LS, enforce SL, SS with fences
- Weak Ordering: enforce LL, LS, SL, SS with fences

Memory fences are expensive operations – *mem instructions wait for all relevant instructions in-flight to complete* (including stores to retire – need store acks) however, cost of serialization only when it is required
Using Memory Fences

Producer posting Item $x$:
- Load $R_{tail}$, $(tail)$
- Store $x$, $(R_{tail})$
- $\text{Membar}_{SS}$
- $R_{tail} = R_{tail} + 1$
- Store $R_{tail}$, $(tail)$

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
  - $\text{Membar}_{LL}$
  - Load $R$, $(R_{head})$
  - $R_{head} = R_{head} + 1$
  - Store $R_{head}$, $(head)$
  - process($R$)

ensures that $R$ is not loaded before $x$ has been stored
Memory Coherence in SMPs

Suppose CPU-1 updates A to 200.

*write-back*: memory and cache-2 have stale values
*write-through*: cache-2 has a stale value

Do these stale values matter?
What is the view of shared memory for programming?
Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

T1:
- Store 1, (X) (X = 1)
- Store 11, (Y) (Y = 11)

T2:
- Load R₁, (Y)
- Store R₁, (Y') (Y' = Y)
- Load R₂, (X)
- Store R₂, (X') (X' = X)

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

(X', Y') ∈ {(1,11), (0,10), (1,10), (0,11)} ?
Write-back Caches & SC

- **T1 is executed**
  - ST X, 1
  - ST Y, 11

- **cache-1 writes back Y**
  - cache-1
    - X = 1
    - Y = 11
  - memory
    - X = 0
    - Y = 10
    - X' = 0
    - Y' = 10
  - cache-2
    - Y = 11
    - Y' = 11
  - prog T2
    - LD Y, R1
    - ST Y', R1
    - LD X, R2
    - ST X', R2

- **T2 executed**
  - cache-1
    - X = 1
    - Y = 11
  - memory
    - X = 0
    - Y = 11
    - X' = 0
    - Y' = 11
  - cache-2
    - Y = 11
    - Y' = 11
    - X' = 0
  - prog T2
    - LD Y, R1
    - ST Y', R1
    - LD X, R2
    - ST X', R2

- **cache-1 writes back X**
  - cache-1
    - X = 1
    - Y = 11
  - memory
    - X = 0
    - Y = 11
    - X' = 0
    - Y' = 11
  - cache-2
    - Y = 11
    - Y' = 11
    - X' = 0
  - prog T2
    - LD Y, R1
    - ST Y', R1
    - LD X, R2
    - ST X', R2

- **cache-2 writes back X' & Y'**
  - cache-1
    - X = 1
    - Y = 11
  - memory
    - X = 1
    - Y = 11
    - X' = 0
    - Y' = 11
  - cache-2
    - Y = 11
    - Y' = 11
    - X' = 0
  - prog T2
    - LD Y, R1
    - ST Y', R1
    - LD X, R2
    - ST X', R2

*inconsistent*
Write-through Caches & SC

- T1 executed
  - prog T1
    - ST X, 1
    - ST Y, 11
  - cache-1
    - X = 0
    - Y = 10
  - memory
    - X = 0
    - Y = 10
    - X' = Y'
  - cache-2
    - Y = 11
    - X = 0
    - X' = Y'

- T2 executed
  - prog T2
    - LD Y, R1
    - ST Y', R1
    - LD X, R2
    - ST X', R2
  - cache-1
    - X = 1
    - Y = 11
  - memory
    - X = 1
    - Y = 11
    - X' = 0
    - Y' = 11
  - cache-2
    - Y = 11
    - X = 0
    - X' = 0

Write-through caches don’t preserve sequential consistency either.
Cache Coherence vs. Memory Consistency

- A cache coherence protocol ensures that all writes by one processor are eventually visible to other processors
  - i.e., updates are not lost
- A memory consistency model gives the rules on when a write by one processor can be observed by a read on another
  - Equivalently, what values can be seen by a load
- A cache coherence protocol is not enough to ensure sequential consistency
  - But if sequentially consistent, then caches must be coherent
Maintaining Cache Coherence

Hardware support is required such that
- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

⇒ cache coherence protocols
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 & CS152