Lecture 3: Memory Models

Teaching assistant: Salvatore Di Girolamo

Motivational video: https://www.youtube.com/watch?v=tW2hT0g4OUc
Scientific Benchmarking: Benchmark Selection (Rule 2)

Effect of GCC -O3 optimization flag

- CoMD: no optimizations, with -O3
- HPCCG: no optimizations, with -O3
- miniAMR: no optimizations, with -O3

Time [Sec]
Based on the presented data, one may conclude that using `-O3` is always a good idea.
Based on the presented data, one may conclude that using \(-O3\) is always a good idea.

The presented data set contains only a subset of the Mantevo benchmark suite.
Based on the presented data, one may conclude that using -O3 is always a good idea.

The incompleteness of data may lead to wrong conclusions. Sometimes -O3 may not be a good idea for a code: e.g., vectorization (enabled by -O3) may segfault on a loop which does unaligned memory access on some x86. But this is not demonstrated by the presented dataset.
Based on the presented data, one may conclude that using \(-O3\) is always a good idea. However, the incompleteness of data may lead to wrong conclusions. Sometimes \(-O3\) may not be a good idea for a code: e.g., vectorization (enabled by \(-O3\)) may segfault on a loop which does unaligned memory access on some x86. But this is not demonstrated by the presented dataset.

**Rule 2:** Specify the reason for only reporting subsets of standard benchmarks or applications or not using all system resources.

- This implies: Show results even if your code/approach stops scaling!
Review of last lecture

- **Architecture case studies**
  - Memory performance is often the bottleneck
  - Parallelism grows with compute performance
  - Caching is important
  - Several issues to address for parallel systems

- **Cache Coherence**
  - Hardware support to aid programmers
  - Two guarantees:
    - Write propagation (updates are eventually visible to all readers)
    - Write serialization (writes to the same location are observed in global order)
  - Two major mechanisms:
    - Snooping
    - Directory-based – continuing today
  - Protocols: MESI (MOESI, MESIF)
DPHPC Overview

**Concepts & Techniques**
- locality
- parallelism
- caches
- memory hierarchy
- vector ISA
- shared memory
- distributed memory
- cache coherency
- memory models
- locks
- lock free
- wait free
- linearizability
- distributed algorithms
- group communications

**Models**
- Amdahl's and Gustafson's law
- memory
- α - β
- PRAM
- LogP
- I/O complexity
- balance principles I
- Little's Law
- balance principles II
- scheduling
Goals of this lecture

- **Don’t forget the projects!**
  - Project ideas shared on Thursday (send email to Salvatore for group formations)
  - Project progress presentations on 10/29 (three weeks from now)!

- **Cache-coherence is not enough**
  - Many more subtle issues for parallel programs

- **Memory Models**
  - Sequential consistency
  - Why threads cannot be implemented as a library 😊
  - Relaxed consistency models

- **Linearizability**
  - More complex objects
Directory-based cache coherence

- Snooping does not scale
  - Bus transactions must be *globally* visible
  - Implies broadcast
- Typical solution: tree-based (hierarchical) snooping
  - Root becomes a bottleneck
- Directory-based schemes are more scalable
  - Directory (entry for each CL) keeps track of all owning caches
  - Point-to-point update to involved processors
    - *No broadcast*
    - *Can use specialized (high-bandwidth) network, e.g., HT, QPI …*
Basic Scheme

- System with $N$ processors $P_i$

- For each memory block (size: cache line) maintain a directory entry
  - $N$ presence bits (light blue)
    - Set if block in cache of $P_i$
  - 1 dirty bit (red)

- First proposed by Censier and Feautrier (1978)
Directory-based CC: Read miss

- $P_0$ intends to read, misses
Directory-based CC: Read miss

- \( P_0 \) intends to read, misses

```

Main Memory

Directory

X 0 0 1 0

X = 7

Cache

Read X

P_1

Cache

P_2

Cache

X = 7
```
Directory-based CC: Read miss

- $P_0$ intends to read, misses
- If dirty bit (in directory) is off
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit (in directory) is off
  - Read from main memory
Directory-based CC: Read miss

- $P_0$ intends to read, misses
- If dirty bit (in directory) is off
  - Read from main memory
  - Set presence[i]

Diagram:

- $P_0$, $P_1$, $P_2$
- Cache: $X = 7$
- Main Memory
- Directory: $X = 7$
Directory-based CC: Read miss

- $P_0$ intends to read, misses
- If dirty bit (in directory) is off
  - Read from main memory
  - Set presence[i]
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit (in directory) is off
  - Read from main memory
  - Set presence[i]
  - Supply data to reader
Directory-based CC: Read miss

- \( P_0 \) intends to read, misses
Directory-based CC: Read miss

- $P_0$ intends to read, misses

\[
\begin{array}{c}
\text{Main Memory} \\
\text{Directory} \\
\begin{array}{cccc}
X & 0 & 0 & 1 \\
\end{array} \\
\end{array}
\]
Directory-based CC: Read miss

- $P_0$ intends to read, misses
- If dirty bit is on
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit is on
  - Recall cache line from $P_j$
    (determine by presence[])
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit is on
  - Recall cache line from $P_j$
    (determine by presence[])
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit is on
  - Recall cache line from $P_j$ (determine by presence[])
  - Update memory
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit is on
  - Recall cache line from $P_j$ (determine by presence[])
  - Update memory
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit is on
  - Recall cache line from $P_j$ (determine by presence[])
  - Update memory
  - Unset dirty bit, block shared
Directory-based CC: Read miss

- **P₀** intends to read, misses

- **If dirty bit is on**
  - Recall cache line from Pᵢ (determine by presence[])
  - Update memory
  - Unset dirty bit, block shared
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit is on
  - Recall cache line from $P_j$ (determine by presence[])
  - Update memory
  - Unset dirty bit, block shared
  - Set presence[]
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- **If dirty bit is on**
  - Recall cache line from $P_j$ (determine by presence[])
  - Update memory
  - Unset dirty bit, block shared
  - Set presence[i]
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit is on
  - Recall cache line from $P_j$ (determine by presence[])
  - Update memory
  - Unset dirty bit, block shared
  - Set presence[i]
  - Supply data to reader
Directory-based CC: Read miss

- $P_0$ intends to read, misses

- If dirty bit is on
  - Recall cache line from $P_j$
    (determine by presence[])
  - Update memory
  - Unset dirty bit, block shared
  - Set presence[i]
  - Supply data to reader
Directory-based CC: Write miss

- $P_0$ intends to write, misses
Directory-based CC: Write miss

- $P_0$ intends to write, misses
Directory-based CC: Write miss

- $P_0$ intends to write, misses
- If dirty bit (in directory) is off
Directory-based CC: Write miss

- \( P_0 \) intends to write, misses
- If dirty bit (in directory) is off
  - Send invalidations to all processors \( P_j \) with presence[j] turned on
Directory-based CC: Write miss

- $P_0$ intends to write, misses
- If dirty bit (in directory) is off
  - Send invalidations to all processors $P_j$ with presence[$j$] turned on
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- If dirty bit (in directory) is off
  - Send invalidations to all processors $P_j$ with presence$[j]$ turned on

Diagram:

- Write $X = 0$
- $P_1$
- $P_2$
- Cache
- Cache
- Cache
- Main Memory

Directory:

- $X = 7$
- $X = 0$
- $X = 0$
- $X = 1$
- $X = 0$
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- If dirty bit (in directory) is off
  - Send invalidations to all processors $P_j$ with presence[$j$] turned on
  - Unset presence bit for all processors
Directory-based CC: Write miss

- P₀ intends to write, misses

- If dirty bit (in directory) is off
  - Send invalidations to all processors P₀
    with presence[j] turned on
  - Unset presence bit for all processors
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- If dirty bit (in directory) is off
  - Send invalidations to all processors $P_j$ with presence[$j$] turned on
  - Unset presence bit for all processors
  - Set dirty bit
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- If dirty bit (in directory) is off
  - Send invalidations to all processors $P_j$ with presence$[j]$ turned on
  - Unset presence bit for all processors
  - Set dirty bit
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- If dirty bit (in directory) is off
  - Send invalidations to all processors $P_j$ with presence[$j$] turned on
  - Unset presence bit for all processors
  - Set dirty bit
  - Set presence[$i$], owner $P_i$
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- If dirty bit (in directory) is off
  - Send invalidations to all processors $P_j$ with presence[$j$] turned on
  - Unset presence bit for all processors
  - Set dirty bit
  - Set presence[$i$], owner $P_i$

![Diagram showing write miss process]
Directory-based CC: Write miss

- P₀ intends to write, misses

- If dirty bit (in directory) is off
  - Send invalidations to all processors Pⱼ with presence[j] turned on
  - Unset presence bit for all processors
  - Set dirty bit
  - Set presence[i], owner Pᵢ
Directory-based CC: Write miss

- $P_0$ intends to write, misses
Directory-based CC: Write miss

- \( P_0 \) intends to write, misses
Directory-based CC: Write miss

- $P_0$ intends to write, misses
- If dirty bit is on
Directory-based CC: Write miss

- $P_0$ intends to write, misses
  - If dirty bit is on
    - Recall cache line from owner $P_j$
Directory-based CC: Write miss

- \( P_0 \) intends to write, misses

- If dirty bit is on
  - Recall cache line from owner \( P_j \)
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- If dirty bit is on
  - Recall cache line from owner $P_j$
  - Update memory
Directory-based CC: Write miss

- $P_0$ intends to write, misses
- If dirty bit is on
  - Recall cache line from owner $P_j$
  - Update memory
Directory-based CC: Write miss

- \( P_0 \) intends to write, misses

- If dirty bit is on
  - Recall cache line from owner \( P_j \)
  - Update memory
  - Unset presence[\( j \)]
Directory-based CC: Write miss

- $P_0$ intends to write, misses
- If dirty bit is on
  - Recall cache line from owner $P_j$
  - Update memory
  - Unset presence[$j$]
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- If dirty bit is on
  - Recall cache line from owner $P_j$
  - Update memory
  - Unset presence[$j$]
  - Set presence[$i$], dirty bit remains set
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- If dirty bit is on
  - Recall cache line from owner $P_j$
  - Update memory
  - Unset presence[$j$]
  - Set presence[$i$], dirty bit remains set
Directory-based CC: Write miss

- $P_0$ intends to write, misses

- **If dirty bit is on**
  - Recall cache line from owner $P_j$
  - Update memory
  - Unset presence[$j$]
  - Set presence[$i$], dirty bit remains set
  - Acknowledge to writer
Discussion

- **Scaling of memory bandwidth**
  - No centralized memory

- **Directory-based approaches scale with restrictions**
  - Require presence bit for each cache and cache line address
  - Number of bits determined at design time
  - Directory requires memory (size scales linearly)
  - Shared vs. distributed directory

- **Software-emulation**
  - Distributed shared memory (DSM)
  - Emulate cache coherence in software (e.g., TreadMarks)
  - Often on a per-page basis, utilizes memory virtualization and paging
Open Problems (for projects, theses, research)

- Tune algorithms to cache-coherence schemes
  - What is the optimal parallel algorithm for a given scheme?
  - Parameterize for an architecture

- Measure and classify hardware
  - Read Maranget et al. “A Tutorial Introduction to the ARM and POWER Relaxed Memory Models” and have fun!
  - RDMA consistency is barely understood!
  - GPU memories are not well understood!
    *Huge potential for new insights!*

- Can we program (easily) without cache coherence?
  - How to fix the problems with inconsistent values?
  - Compiler support (issues with arrays)?

- Invent new semi-coherent schemes?
Case Study: Intel Xeon Phi
Case Study: Intel Xeon Phi
Case Study: Intel Xeon Phi
Case Study: Intel Xeon Phi
Case Study: Intel Xeon Phi
Communication?

State in source cache

\((S_{s\$}, S_{d\$})\)

State in destination cache
Communication?

State in source cache

\[(S_{ss}, S_{ds})\]

State in destination cache

Invalid read $R_I = 278$ ns
Local read: $R_L = 8.6$ ns
Remote read $R_R = 235$ ns

Inspired by Molka et al.: “Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system”
Single-Line Ping Pong

- Prediction for both in E state: 479 ns
  - Measurement: 497 ns (O=18)

Multi-Line Ping Pong

- More complex due to prefetch

\[ \mathcal{T}_N = o \cdot N + q - \frac{p}{N} \]

Number of CLs

Amortization of startup

Asymptotic Fetch Latency for each cache line (optimal prefetch!)

Startup overhead

Multi-Line Ping Pong

\[ T_N = o \cdot N + q - \frac{p}{N} \]

- **E state:**
  - o=76 ns
  - q=1,521 ns
  - p=1,096 ns

- **I state:**
  - o=95 ns
  - q=2,750 ns
  - p=2,017 ns

DTD Contention 😞

- **E state:**
  - \( a = 0 \text{ns} \)
  - \( b = 320 \text{ns} \)
  - \( c = 56.2 \text{ns} \)

\[
\mathcal{T}_C(n_{th}) = c \cdot n_{th} + b - \frac{a}{n_{th}}
\]

---

Optimizations against vendor libraries

Barrier (7x faster than OpenMP)

Ramos, Hoefler: “Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL”, IPDPS’17 (video: [https://www.youtube.com/watch?v=10Mo3MnWR74](https://www.youtube.com/watch?v=10Mo3MnWR74))
Optimizations against vendor libraries

Is Coherence Everything?

- Coherence is concerned with behavior of *individual* locations
- Consider the program (initial X,Y,Z = 0)

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y=10</td>
<td>while (X==0);</td>
</tr>
<tr>
<td>X=2</td>
<td>Z=Y</td>
</tr>
</tbody>
</table>

- Class question: what value will Z on P2 have?
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations
- Consider the program (initial $X,Y,Z = 0$)

**Class question:** what value will $Z$ on $P_2$ have?

```
while (X==0);
Z=Y
```

P1

```
Y=10
X=2
```

P2

```
while (X==0);
Z=Y
```

```
X = 0
```

```
Y = 0
```

```
X = 0
```

```
Y = 0
```
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations
- Consider the program (initial X, Y, Z = 0)

  Class question: what value will Z on P2 have?

```
P1
Y=10
X=2

P2
while (X==0);
Z=Y

W1
X = 0

L2
Y = 0

L1
Y = 0

L1
X = 0

P2
Y = 10

WT L1
X = 0

WT L1
X = 0

WB

WB
```
**Is Coherence Everything?**

- Coherence is concerned with behavior of *individual* locations
- Consider the program (initial $X,Y,Z = 0$)

\[
\text{P1} \quad \text{P2}
\]

- $Y=10$
- $X=2$

```
while (X==0);
Z=Y
```

- Class question: what value will Z on P2 have?
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations
- Consider the program (initial $X,Y,Z = 0$)

Class question: what value will $Z$ on $P_2$ have?

```
$P_1$

Y=10
X=2

$P_2$

while (X==0);
Z=Y

$X = 2$

$Y = 10$
$X = 0$

$WT \; L_1$

$WB$

$X = 0$
$Y = 0$
$P_2$

$L_1$

$WB$

$L_2$

```
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations
- Consider the program (initial X,Y,Z = 0)

  P1
  - Y=10
  - X=2

  P2
  - while (X==0);
  - Z=Y

Class question: what value will Z on P2 have?
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations
- Consider the program (initial X,Y,Z = 0)

Class question: what value will Z on P2 have?

```
Y=10
X=2
```

while (X==0); Z=Y
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations
- Consider the program (initial X,Y,Z = 0)

Class question: what value will Z on P2 have?
Is Coherence Everything?

- Coherence is concerned with behavior of *individual* locations
- Consider the program (initial $X,Y,Z = 0$)

Class question: what value will $Z$ on $P_2$ have?

```
P1
Y=10
X=2

while (X==0);
Z=Y

P2
Y = 10
X = 2
```

```
L1
X = 2

P1
WT L1
X = 2

L2
X = 2

L1
Read Y
```

```
L1
X = 2

L2
Y = 0
```
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations
- Consider the program (initial X,Y,Z = 0)

Class question: what value will Z on P2 have?

P1
Y=10
X=2

P2
while (X==0);
Z=Y

X = 2
Y = 0
Is Coherence Everything?

- Coherence is concerned with behavior of \textit{individual} locations
- Consider the program (initial $X,Y,Z = 0$)

\begin{itemize}
\item Class question: what value will $Z$ on $P_2$ have?
\end{itemize}
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations.
- Consider the program (initial X,Y,Z = 0)

Class question: what value will Z on P2 have?

Y=10 does not need to have completed before X=2 is visible to P2!

- This allows P2 to exit the loop and read Y=0
- This may not be the intent of the programmer!
- This may be due to congestion (imagine X is pushed to a remote cache while Y misses to main memory) and or due to write buffering, or ...
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations
- Consider the program (initial X,Y,Z = 0)

Class question: what value will Z on P2 have?

Y=10 does not need to have completed before X=2 is visible to P2!

- This allows P2 to exit the loop and read Y=0
- This may not be the intent of the programmer!
- This may be due to congestion (imagine X is pushed to a remote cache while Y misses to main memory) and or due to write buffering, or ...

Bonus class question: what happens when Y and X are on the same cache line (assume simple MESI and no write buffer)?
Memory Models

- Need to define what it means to “read a location” and “to write a location” and the respective ordering!
  - What values should be seen by a processor
- First thought: extend the abstractions seen by a sequential processor:
  - Compiler and hardware maintain data and control dependencies at all levels:

Two operations to the same location

- Y = 10
- ....
- T = 14
- Y = 15

One operation controls execution of others

- Y = 5
- X = 5
- T = 3
- Y = 3
- if (X==Y)
- Z = 5
- ....
Memory Models

- Need to define what it means to “read a location” and “to write a location” and the respective ordering!
  - What values should be seen by a processor
- First thought: extend the abstractions seen by a sequential processor:
  - Compiler and hardware maintain data and control dependencies at all levels:

Two operations to the same location

- Y = 10
- ....
- T = 14
- Y = 15

One operation controls execution of others

- Y = 5
- X = 5
- T = 3
- Y = 3
- if (X == Y)
- Z = 5
- ....
Memory Models

- Need to define what it means to “read a location” and “to write a location” and the respective ordering!
  - What values should be seen by a processor
- First thought: extend the abstractions seen by a sequential processor:
  - Compiler and hardware maintain data and control dependencies at all levels:

Two operations to the same location

| Y = 10 |
| .... |
| T = 14 |
| Y = 15 |

One operation controls execution of others

| Y = 5 |
| X = 5 |
| T = 3 |
| Y = 3 |
| if (X == Y) |
| Z = 5 |
| .... |
Sequential Processor

- **Correctness condition:**
  - The result of the execution is the same as if the operations had been executed in the order specified by the program “program order”
  - A read returns the value last written to the same location
    - “last” is determined by program order!

- **Consider only memory operations (e.g., a trace)**

- **N Processors**
  - P1, P2, ..., PN

- **Operations**
  - Read, Write on shared variables (initial state: most often all 0)

- **Notation:**
  - P1: R(x):3  P1 reads x and observes the value 3
  - P2: W(x,5)  P2 writes 5 to variable x
Terminology

- **Program order**
  - Deals with a *single* processor
  - Per-processor order of memory accesses, determined by program’s *Control flow*
  - Often represented as trace

- **Visibility order**
  - Deals with operations on *all* processors
  - Order of memory accesses observed by one or more processors
  - E.g., “every read of a memory location returns the value that was written last”
    - *Defined by memory model*
Memory Models

- Contract at each level between programmer and processor

<table>
<thead>
<tr>
<th>Programmer</th>
<th>Optimizing transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-level language (API/PL)</td>
<td></td>
</tr>
<tr>
<td>Compiler Frontend</td>
<td>Reordering</td>
</tr>
<tr>
<td>Intermediate Language (IR)</td>
<td></td>
</tr>
<tr>
<td>Compiler Backend/JIT</td>
<td>Operation overlap</td>
</tr>
<tr>
<td>Machine code (ISA)</td>
<td>VLIW, Vector ISA</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
</tr>
</tbody>
</table>
Sequential Consistency

- Extension of sequential processor model

- The execution happens as if
  1. The operations of all processes were executed in some sequential order (atomicity requirement), and
  2. The operations of each individual processor appear in this sequence in the order specified by the program (program order requirement)

- Applies to all layers!
  - Disallows many compiler optimizations (e.g., reordering of any memory instruction)
  - Disallows many hardware optimizations (e.g., store buffers, nonblocking reads, invalidation buffers)
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processes issue in program order

The “switch” selects arbitrary next operation

Memory

Program
A = B;

Read
B

Write
A
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

The "switch" selects arbitrary next operation

Processors issue in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation

Program
A = B;

Read B
Write A
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

The “switch” selects arbitrary next operation

Processors issue in program order

Program

\[ A = B; \]
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

The “switch” selects arbitrary next operation

Processors issue in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation

Program: `A = B;`

Read B
Read B

Memory

Read B
Read B
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Program:

```
A = B;
```

Processors issue in program order

The “switch” selects arbitrary next operation
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation

Memory

Program
A = B;

==

Read B  Write A

Read B
Write A
Read B

P1
P2
P3
P4
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation

Program
A = B;

Read B  Write A

Memory

Read B  Read B  Write A  Read B  Write A
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation

Memory

Program

\[ A = B; \]

\[
\begin{array}{c}
\text{Read B} \\
\text{Read B} \\
\text{Write A} \\
\text{Read B} \\
\text{Write A} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Read B} \\
\text{Write A} \\
\text{Read B} \\
\text{Write A} \\
\end{array}
\]
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation

Program
A = B;

Memory

Read B
Read B
Write A
Read B
Write A
Read B

Read B
Write A

P1

P2

P3

P4
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation

Program: `A = B;`
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The "switch" selects arbitrary next operation
Original SC Definition

“The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program”

(Lamport, 1979)
Alternative SC Definition

- Textbook: Hennessy/Patterson Computer Architecture

- A sequentially consistent system maintains three invariants:
  1. A load \( L \) from memory location \( A \) issued by processor \( P_i \) obtains the value of the previous store to \( A \) by \( P_j \), unless another processor has to stored a value to \( A \) in between.
  2. A load \( L \) from memory location \( A \) obtains the value of a store \( S \) to \( A \) by another processor \( P_k \) if \( S \) and \( L \) are 
     “sufficiently separated in time” and if no other store occurred between \( S \) and \( L \).
  3. Stores to the same location are serialized (defined as in (2)).

- “Sufficiently separated in time” not precise
  - Works but is not formal (a formalization must include all possibilities)
Example Operation Reordering

- Recap: “normal” sequential assumption:
  - Compiler and hardware can reorder instructions as long as control and data dependencies are met

- Examples:
  - Compiler:
    - Register allocation
    - Code motion
    - Common subexpression elimination
    - Loop transformations
  - Hardware:
    - Pipelining
    - Multiple issue (OOO)
    - Write buffer bypassing
    - Nonblocking reads
Simple compiler optimization

- Initially, all values are zero

```
P1
input = 23
ready = 1

P2
while (ready == 0) {}
compute(input)
```

- Assume P1 and P2 are compiled separately!
Simple compiler optimization

- Initially, all values are zero

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>input = 23</td>
<td>while (ready == 0) {}</td>
</tr>
<tr>
<td>ready = 1</td>
<td>compute(input)</td>
</tr>
</tbody>
</table>

- Assume P1 and P2 are compiled separately!
- What optimizations can a compiler perform for P1?
  
  *Register allocation or even replace with constant, or Switch statements*
Simple compiler optimization

- Initially, all values are zero

P1

- input = 23
- ready = 1

P2

- while (ready == 0) {}
- compute(input)

- Assume P1 and P2 are compiled separately!
- What optimizations can a compiler perform for P1?
  - Register allocation or even replace with constant, or
  - Switch statements
- What happens?
  - P2 may never terminate, or
  - Compute with wrong input
Sequential Consistency Examples

- Relying on **program order**: Dekker’s algorithm
  - Initially, all zero

  ```
  P1
  a = 1
  if(b == 0)
  critical section
  a = 0
  
  P2
  b = 1
  if(a == 0)
  critical section
  b = 0
  ```

- What can happen at compiler and hardware level?
Sequential Consistency Examples

- Relying on **program order**: Dekker’s algorithm
  - Initially, all zero

  ```
  P1
  a = 1
  if(b == 0)
    critical section
  a = 0

  P2
  b = 1
  if(a == 0)
    critical section
  b = 0
  ```

  - What can happen at compiler and hardware level?
Sequential Consistency Examples

- Relying on program order: Dekker’s algorithm
  - Initially, all zero

  - What can happen at compiler and hardware level?
Sequential Consistency Examples

- Relying on program order: Dekker’s algorithm
  - Initially, all zero

P1

```
a = 1
if(b == 0)
critical section
a = 0
```

P2

```
b = 1
if(a == 0)
critical section
b = 0
```

- What can happen at compiler and hardware level?
Sequential Consistency Examples

- Relying on **program order**: Dekker’s algorithm
  - Initially, all zero

- What can happen at compiler and hardware level?

```plaintext
P1
a = 1
if(b == 0)
critical section
a = 0

P2
b = 1
if(a == 0)
critical section
b = 0
```
Sequential Consistency Examples

- Relying on program order: Dekker’s algorithm
  - Initially, all zero

  P1
  a = 1
  if(b == 0)
  critical section
  a = 0

  P2
  b = 1
  if(a == 0)
  critical section
  b = 0

- What can happen at compiler and hardware level?

Nobody enters the critical section.
Sequential Consistency Examples

- **Relying on program order**: Dekker’s algorithm
  - Initially, all zero

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1</td>
<td>b = 1</td>
</tr>
<tr>
<td>if(b == 0)</td>
<td>if(a == 0)</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>a = 0</td>
<td>b = 0</td>
</tr>
</tbody>
</table>

- What can happen at compiler and hardware level?

Without SC, both writes may have went to a write buffer, in which case both Ps would read 0 and enter the critical section together.

Nobody enters the critical section.
Sequential Consistency Examples

- Relying on single sequential order (atomicity): three sharers

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 5</td>
<td>if (a == 1) b = 1</td>
<td>if (b == 1) print(a)</td>
</tr>
<tr>
<td>a = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What each P thinks the order is:

- What can be printed if visibility is not atomic?
Sequential Consistency Examples

- Relying on single sequential order (atomicity): three sharers

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 5</td>
<td>if (a == 1) b = 1</td>
<td>if (b == 1) print(a)</td>
</tr>
<tr>
<td>a = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What can be printed if visibility is not atomic?
Sequential Consistency Examples

- Relying on single sequential order (**atomicity**): three sharers

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 5</td>
<td>if (a == 1)</td>
<td>if (b == 1)</td>
</tr>
<tr>
<td>a = 1</td>
<td>b = 1</td>
<td>print(a)</td>
</tr>
</tbody>
</table>

- What can be printed if visibility is not atomic?
Sequential Consistency Examples

- Relying on single sequential order (**atomicity**): three sharers

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 5 )</td>
<td>( \text{if } (a == 1) ) ( b = 1 )</td>
<td>( \text{if } (b == 1) ) print(a)</td>
</tr>
</tbody>
</table>

- What can be printed if visibility is not atomic?

What each P thinks the order is:

- \( P_1: W(a,5) \)
- \( P_1: W(a,1) \)
- \( P_1: W(a,5) \)
- \( P_1: W(a,1) \)
Sequential Consistency Examples

- Relying on single sequential order (**atomicity**): three sharers

P1

a = 5
a = 1

P2

if (a == 1)
    b = 1

P3

if (b == 1)
    print(a)

- What can be printed if visibility is not atomic?
Sequential Consistency Examples

- Relying on single sequential order (atomicity): three sharers

P1

\[ a = 5 \]

\[ a = 1 \]

P2

\[ \text{if} \ (a == 1) \]

\[ b = 1 \]

P3

\[ \text{if} \ (b == 1) \]

\[ \text{print}(a) \]

- What can be printed if visibility is not atomic?
Sequential Consistency Examples

- Relying on single sequential order (atomicity): three sharers

  P1
  a = 5
  a = 1

  P2
  if (a == 1)
  b = 1

  P3
  if (b == 1)
  print(a)

  What each P thinks the order is:

  P1
  P1: W(a,5)
  P1: W(a,1)

  P2
  P2: R(a): 1
  P2: W(a,1)
  P2: W(b,1)

  P3
  P3: R(b): 1

  What can be printed if visibility is not atomic?
Sequential Consistency Examples

- Relying on single sequential order (atomicity): three sharers

P1
a = 5
a = 1

P2
if (a == 1)
    b = 1

P3
if (b == 1)
    print(a)

What can be printed if visibility is not atomic?

- P3 has not seen P1: W(a,1) yet!
Sequential Consistency Examples

- Relying on single sequential order (atomicity): three sharers

P1

a = 5
a = 1

P2

if (a == 1)
  b = 1

P3

if (b == 1)
  print(a)

- What can be printed if visibility is not atomic?

What each P thinks the order is:

P1

P1: W(a,5)
P1: W(a,1)
P1: W(a,5)
P1: W(a,1)
P1: W(a,5)
P1: W(a,1)

P2

P2: R(a): 1
P2: W(b,1)
P2: W(b,1)
P2: W(b,1)
P3

P3: R(b): 1
P3: R(a): 5
PRINT(5)
P3: W(b,1)
P3

P3 has not seen P1: W(a,1) yet!
Optimizations violating program order

- Analyzing P1 and P2 in isolation!
  - Compiler can reorder

  - Hardware can reorder, assume writes of a,b go to write buffer or speculation

```
P1
a = 1
if(b == 0)
critical section
a = 0

P2
b = 1
if(a == 0)
critical section
b = 0
```

```
P1
if(b == 0)
critical section
a = 0
else
  a = 1

P2
if(a == 0)
critical section
b = 0
else
  b = 1
```
Considerations

- Define partial order on memory requests $A \rightarrow B$
  - If $P_i$ issues two requests $A$ and $B$ and $A$ is issued before $B$ in program order, then $A \rightarrow B$
  - $A$ and $B$ are issued to the same variable, and $A$ is issued first, then $A \rightarrow B$ (on all processors)

- These partial orders can be interleaved, define a total order
  - Many total orders are sequentially consistent!

- Example:
  - P1: W(a), R(b), W(c)
  - P2: R(a), W(a), R(b)
  - Are the following schedules (total orders) sequentially consistent?
    1. P1:W(a), P2:R(a), P2:W(a), P1:R(b), P2:R(b), P1:W(c)
    2. P1:W(a), P2:R(a), P1:R(b), P2:R(b), P1:W(c), P2:W(a)
    3. P2:R(a), P2:W(a), P1:R(b), P1:W(a), P1:W(c), P2:R(b)
Write buffer example

- Write buffer
  - Absorbs writes faster than the next cache → prevents stalls
  - Aggregates writes to the same cache line → reduces cache traffic
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read (W → R)

Write buffer example:

- Process P1:
  - Write to location (a, 1)
  - Read from location (b, 0)

- Process P2:
  - Write to location (b, 1)
  - Read from location (a, 0)

**Not seq. consistent**
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read ($W \rightarrow R$)

**Write buffer example**

![Diagram showing write buffer example]

- $W(a,1)$
- $P_1$
- $L1$
- $P2$
- $L1$
- $WB$
- $WB$
- $a = 0$
- $b = 0$

-not seq. consistent

- $W(a,1)$
  - $R(b):0$
- $W(b,1)$
  - $R(a):0$
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read ($W \rightarrow R$)

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(a,1)</td>
<td>W(b,1)</td>
</tr>
<tr>
<td>R(b):0</td>
<td>R(a):0</td>
</tr>
</tbody>
</table>
```

(not seq. consistent)
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read (W → R)

![Write buffer example diagram with P1 and P2 processes, and L1 and L2 levels showing write buffer (WB) and a = 1, b = 0, a = 0 values.]
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read (W → R)

![Write buffer example diagram]

- **P1**
  - Write buffer: W(a,1)
  - Read buffer: R(b)

- **P2**
  - Write buffer: W(b,1)
  - Read buffer: R(a)

*not seq. consistent*
Reads can bypass previous writes for faster completion
- If read and write access different locations
- No order between write and following read ($W \rightarrow R$)

```
Write buffer example

WB
a = 1

L1
b = 0

WB
a = 0

L2
b = 0

P1
W(a,1) R(b):0

P2
W(b,1) R(a):0
```

*not seq. consistent*
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read (W → R)

**Write buffer example**

**P1**
- W(a,1)
- R(b): 0

**P2**
- W(b,1)
- R(a): 0

*not seq. consistent*
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read ($W \to R$)

```
Write buffer example

P1
W(a,1)  R(b):0

P2
W(b,1)

not seq. consistent
```
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read (W → R)
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read ($W \rightarrow R$)

```
Write buffer example

P1

W(a,1)
R(b): 0

P2

W(b,1)

W(a,1)
R(b): 0

P1

L1

WB
a = 1

b = 0

L2

a = 0

b = 0

P2

L1

WB
b = 1

```
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read ($W \rightarrow R$)

![Write buffer example diagram]
### Write buffer example

- **Reads can bypass previous writes for faster completion**
  - If read and write access different locations
  - No order between write and following read ($W \rightarrow R$)

```
W(a,1)  
R(b) : 0
```

```
W(b,1)  
R(a) : 0
```

**not seq. consistent**
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read (W → R)
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read (W → R)

### Diagram

**P1**
- W(a,1)
- R(b): 0

**P2**
- W(b,1)
- R(a): 0

*not seq. consistent*
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read (W → R)

### Write buffer example diagram

- **P1**:
  - W(a,1)
  - R(b): 0

- **P2**: W(b,1)
  - R(a): 0

*not seq. consistent*
Nonblocking read example

- W → W: OK
- R → W, R → R: No order between read and following read/write

P1
R(y):2
W(x,1)

P2
R(x):0
W(y,2)

not seq. consistent

Diagram:

- P1 updates L1 to x = 0
- P2 updates L1 to x = 0, y = 0
- L2 remains unchanged
Nonblocking read example

- W → W: OK
- R → W, R → R: No order between read and following read/write
Nonblocking read example

- \( W \to W: \text{OK} \)
- \( R \to W, R \to R: \) No order between read and following read/write

Diagram:

- \( P_1 \) accesses \( R(y) \) and Cache miss!
- \( P_2 \) has \( W(x, 1) \) and \( W(y, 2) \)
- Not seq. consistent
Nonblocking read example

- \( W \Rightarrow W: \text{OK} \)
- \( R \Rightarrow W, R \Rightarrow R: \text{No order between read and following read/write} \)

P1
- \( R(y): 2 \)
- \( R(x): 0 \)

P2
- \( W(x, 1) \)
- \( W(y, 2) \)

\( not \ seq. \ consistent \)
Nonblocking read example

- $W \rightarrow W$: OK
- $R \rightarrow W, R \rightarrow R$: No order between read and following read/write

P1
- $R(y):2$
- $R(x):0$

P2
- $W(x,1)$
- $W(y,2)$

*not seq. consistent*
Nonblocking read example

- W $\rightarrow$ W: OK
- R $\rightarrow$ W, R $\rightarrow$ R: No order between read and following read/write

Diagram:

- P1:
  - R(y):2
  - R(x):0

- P2:
  - W(x,1)
  - W(y,2)

Note: seq. consistent
Nonblocking read example

- W $\rightarrow$ W: OK
- R $\rightarrow$ W, R $\rightarrow$ R: No order between read and following read/write

P1
- R(y): 2
- R(x): 0

P2
- W(x, 1)
- W(y, 2)

*not seq. consistent*
Nonblocking read example

- $W \rightarrow W$: OK
- $R \rightarrow W$, $R \rightarrow R$: No order between read and following read/write
Nonblocking read example

- \( W \rightarrow W: \text{OK} \)
- \( R \rightarrow W, R \rightarrow R: \text{No order between read and following read/write} \)

\[
\begin{align*}
R(y) & : y = 0 \\
R(x) & : x = 0 \\
W(x,1) & : x = 1 \quad y = 0
\end{align*}
\]
Nonblocking read example

- W → W: OK
- R → W, R → R: No order between read and following read/write

P1
R(y):2
R(x):0
W(x,1)
W(y,2)

not seq. consistent

P2

Diagram:

P1
R(y)
R(x):0

P2
W(x,1)

L1
x = 1
y = 0

L2
x = 1
y = 0
Nonblocking read example

- $W \rightarrow W$: OK
- $R \rightarrow W$, $R \rightarrow R$: No order between read and following read/write
Nonblocking read example

- \( W \to W: \text{OK} \)
- \( R \to W, R \to R: \text{No order between read and following read/write} \)
Nonblocking read example

- $W \rightarrow W$: OK
- $R \rightarrow W$, $R \rightarrow R$: No order between read and following read/write

![Nonblocking read example diagram](image)
Nonblocking read example

- \( W \rightarrow W: \text{OK} \)
- \( R \rightarrow W, R \rightarrow R: \) No order between read and following read/write

![Diagram showing the nonblocking read example](image-url)
Discussion

- **Programmer’s view:**
  - Prefer sequential consistency
  - Easiest to reason about

- **Compiler/hardware designer’s view:**
  - Sequential consistency disallows many optimizations!
  - Substantial speed difference
  - Most architectures and compilers don’t adhere to sequential consistency!

- **Solution: synchronized programming**
  - Access to shared data (aka. “racing accesses”) are ordered by synchronization operations
  - Synchronization operations guarantee memory ordering (aka. fence)
  - More later!
Cache Coherence vs. Memory Model

- Varying definitions!

- Cache coherence: a mechanism that propagates writes to other processors/caches if needed, recap:
  - Writes are eventually visible to all processors
  - Writes to the same location are observed in (one) order

- Memory models: define the bounds on when the value is propagated to other processors
  - E.g., sequential consistency requires all reads and writes to be ordered in program order

Good read: McKenney: “Memory Barriers: a Hardware View for Software Hackers”
The fun begins: Relaxed Memory Models

- Sequential consistency
  - R\(\rightarrow\)R, R\(\rightarrow\)W, W\(\rightarrow\)R, W\(\rightarrow\)W (all orders guaranteed)

- Relaxed consistency (varying terminology):
  - Processor consistency (aka. TSO)
    - *Relaxes* W\(\rightarrow\)R
  - Partial write (store) order (aka. PSO)
    - *Relaxes* W\(\rightarrow\)R, W\(\rightarrow\)W
  - Weak consistency and release consistency (aka. RMO)
    - *Relaxes* R\(\rightarrow\)R, R\(\rightarrow\)W, W\(\rightarrow\)R, W\(\rightarrow\)W
  - Other combinations/variants possible
    - *There are even more types of orders (above is a simplification)*
## Architectures

<table>
<thead>
<tr>
<th>Type</th>
<th>Alpha</th>
<th>ARMv7</th>
<th>PA-RISC</th>
<th>POWER</th>
<th>SPARC RMO</th>
<th>SPARC PSO</th>
<th>SPARC TSO</th>
<th>x86 oostore</th>
<th>AMD64</th>
<th>IA-64</th>
<th>zSeries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Loads reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Stores reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Stores reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Atomic reordered with loads</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic reordered with stores</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependent loads reordered</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incoherent Instruction cache pipeline</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

Some older x86 and AMD systems have weaker memory ordering[4]

Source: wikipedia
Case Study: Memory ordering on Intel (x86)

- **Intel® 64 and IA-32 Architectures Software Developer's Manual**
  - Volume 3A: System Programming Guide
  - Chapter 8.2 Memory Ordering

- **Google Tech Talk: IA Memory Ordering**
  - Richard L. Hudson
  - [http://www.youtube.com/watch?v=WUfvvFD5tAA](http://www.youtube.com/watch?v=WUfvvFD5tAA)
x86 Memory model: TLO + CC

- **Total lock order (TLO)**
  - Instructions with “lock” prefix enforce total order across all processors
  - Implicit locking: `xchg` (locked compare and exchange)

- **Causal consistency (CC)**
  - Write visibility is transitive

- **Eight principles**
  - After some revisions 😊
The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R → R)
The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R → R)
2. “Writes are not reordered with other writes.” (W → W)
The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R → R)
2. “Writes are not reordered with other writes.” (W → W)
3. “Writes are not reordered with older reads.” (R → W)
The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R → R)
2. “Writes are not reordered with other writes.” (W → W)
3. “Writes are not reordered with older reads.” (R → W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W → R!)
The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)
3. “Writes are not reordered with older reads.” (R→W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W→R!)
5. “In a multiprocessor system, memory ordering obeys causality.“ (memory ordering respects transitive visibility)
The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R → R)
2. “Writes are not reordered with other writes.” (W → W)
3. “Writes are not reordered with older reads.” (R → W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W → R!)
5. “In a multiprocessor system, memory ordering obeys causality.” (memory ordering respects transitive visibility)
6. “In a multiprocessor system, writes to the same location have a total order.” (implied by cache coherence)
The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)
3. “Writes are not reordered with older reads.” (R→W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W→R!)
5. “In a multiprocessor system, memory ordering obeys causality.“ (memory ordering respects transitive visibility)
6. “In a multiprocessor system, writes to the same location have a total order.” (implied by cache coherence)
7. “In a multiprocessor system, locked instructions have a total order.“ (enables synchronized programming!)
The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)
3. “Writes are not reordered with older reads.” (R→W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W→R!)
5. “In a multiprocessor system, memory ordering obeys causality.” (memory ordering respects transitive visibility)
6. “In a multiprocessor system, writes to the same location have a total order.” (implied by cache coherence)
7. “In a multiprocessor system, locked instructions have a total order.” (enables synchronized programming!)
8. “Reads and writes are not reordered with locked instructions.” (enables synchronized programming!)
Principle 1 and 2

Reads are not reordered with other reads. \((R \rightarrow R)\)

 Writes are not reordered with other writes. \((W \rightarrow W)\)

All values zero initially. \(r_1\) and \(r_2\) are registers.

Memory

Reads and writes observed in program order. Cannot be reordered!
Principle 1 and 2

Reads are not reordered with other reads. \((R \rightarrow R)\)
Writes are not reordered with other writes. \((W \rightarrow W)\)

All values zero initially. \(r1\) and \(r2\) are registers.

```
Principle 1
a = 1
b = 2

Principle 2
r1 = b
r2 = a
```

Reads and writes observed in program order.
Cannot be reordered!

Order: from left to right
Principle 1 and 2

Reads are not reordered with other reads. \((R \rightarrow R)\)

Writes are not reordered with other writes. \((W \rightarrow W)\)

Memory

- \(W(a, 1)\)
- \(W(b, 2)\)
- \(R(b)\)
- \(R(a)\)

Reads and writes observed in program order. Cannot be reordered!

Order: from left to right

If \(r1 = 2\), then \(r2\) must be 1!
Not allowed: \(r1 = 2, r2 = 0\)

All values zero initially. \(r1\) and \(r2\) are registers.

- P1
  - \(a = 1\)
  - \(b = 2\)

- P2
  - \(r1 = b\)
  - \(r2 = a\)
Principle 1 and 2

Reads are not reordered with other reads. (R→R)
Writes are not reordered with other writes. (W→W)

Memory

Wat1,1 W(b, 2) R(b) R(a)

Reads and writes observed in program order.
Cannot be reordered!

If r1 == 2, then r2 must be 1!
Not allowed: r1 == 2, r2 == 0

Question: is r1=0, r2=1 allowed?

All values zero initially. r1 and r2 are registers.

P1

a = 1
b = 2

P2

r1 = b
r2 = a

Order: from left to right
Principle 3

Writes are not reordered with older reads. (R→W)

All values zero initially

Memory

P1

r1 = a
b = 1

P2

r2 = b
a = 1

P1

R(a) → W(b,1) → P2

P2

R(b) → W(a,1) → P1

P1

P2
Principle 3

Writes are not reordered with older reads. (R→W)

All values zero initially
Principle 3

 Writes are not reordered with older reads. (R → W)

 P1
 r1 = a
 b = 1

 P2
 r2 = b
 a = 1

 All values zero initially
Principle 3

 Writes are not reordered with older reads. (R → W)

 All values zero initially

 If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!
Principle 3

 Writes are not reordered with older reads. (R \rightarrow W)

 All values zero initially

 If \( r_1 = 1 \), then P2:W(a) \rightarrow P1:R(a), thus \( r_2 \) must be 0!
 If \( r_2 = 1 \), then P1:W(b) \rightarrow P1:R(b), thus \( r_1 \) must be 0!
Principle 3

Writes are not reordered with older reads. (R→W)

All values zero initially

Question: is r1==1 and r2==1 allowed?

If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!
If r2 == 1, then P1:W(b) → P1:R(b), thus r1 must be 0!
Principle 3

Writes are not reordered with older reads. (R→W)

Question: is r1==1 and r2==1 allowed?

Question: is r1==0 and r2==0 allowed?

If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!
If r2 == 1, then P1:W(b) → P1:R(b), thus r1 must be 0!
Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

All values zero initially

P1

\[ a = 1 \]

\[ r_1 = b \]

P2

\[ b = 1 \]

\[ r_2 = a \]
Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

All values zero initially

\[
\begin{align*}
\text{P1} & : a = 1, r_1 = b \\
\text{P2} & : b = 1, r_2 = a
\end{align*}
\]
Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

Memory

R(b)  R(a)  W(a,1)  W(b,1)

All values zero initially

P1

a = 1
r1 = b

P2

b = 1
r2 = a
Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

All values zero initially

P1
a = 1
r1 = b

P2
b = 1
r2 = a

Memory

R(b) → R(a) → W(a,1) → W(b,1)

Allowed: r1=0, r2=0.
Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

All values zero initially

P1
a = 1
r1 = b

P2
b = 1
r2 = a

Allowed: r1=0, r2=0.
Sequential consistency can be enforced with mfence.
Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

All values zero initially

```
P1
a = 1
r1 = b
```

```
P2
b = 1
r2 = a
```

Memory

R(b) → R(a) → W(a,1) → W(b,1)

Allowed: r1=0, r2=0.
Sequential consistency can be enforced with mfence.
Attention: this rule may allow reads to move into critical sections!
Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (**NO W→R!**)

All values zero initially

```
P1
a = 1
r1 = b

P2
b = 1
r2 = a
```

Question: is r1=1, r2=0 allowed?

Allowed: r1=0, r2=0.
Sequential consistency can be enforced with mfence.
Attention: this rule may allow reads to move into critical sections!
In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

**Principle 5**

All values zero initially

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1</td>
<td>r1 = a</td>
<td>r2 = b</td>
</tr>
<tr>
<td>b = 1</td>
<td>b = 1</td>
<td>r3 = a</td>
</tr>
</tbody>
</table>

Memory

- $W(a,1)$
- $R(a)$
- $W(b,1)$
- $R(b)$
- $R(a)$
Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

All values zero initially

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1</td>
<td>r1 = a</td>
<td>r2 = b</td>
</tr>
<tr>
<td></td>
<td>b = 1</td>
<td>r3 = a</td>
</tr>
</tbody>
</table>

Memory

W(a,1)  R(a)  W(b,1)  R(b)  R(a)
Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

If \( r1 == 1 \) and \( r2 == 1 \), then \( r3 \) must read 1.
Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

If \( r_1 == 1 \) and \( r_2 == 1 \), then \( r_3 \) must read 1.
Not allowed: \( r_1 == 1 \), \( r_2 == 1 \), and \( r_3 == 0 \).
In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

Principle 5

If r1 == 1 and r2 == 1, then r3 must read 1. Not allowed: r1 == 1, r2 == 1, and r3 == 0. Provides some form of atomicity.
Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

Question: is $r_1 = 1$, $r_2 = 0$, $r_3 = 1$ allowed?

If $r_1 = 1$ and $r_2 = 1$, then $r_3$ must read 1. Not allowed: $r_1 = 1$, $r_2 = 1$, and $r_3 = 0$. Provides some form of atomicity.
Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).
**Principle 6**

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

All values zero initially

- **P1**: $a = 1$
- **P2**: $a = 2$
- **P3**: $r1 = a$, $r2 = a$
- **P4**: $r3 = a$, $r4 = a$

- Not allowed: $r1 = 1$, $r2 = 2$, $r3 = 2$, $r4 = 1$
Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

- Not allowed: \( r_1 = 1, r_2 = 2, r_3 = 2, r_4 = 1 \)
- If P3 observes P1’s write before P2’s write, then P4 will also see P1’s write before P2’s write
Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

- Not allowed: \( r_1 = 1, r_2 = 2, r_3 = 2, r_4 = 1 \)
- If P3 observes P1’s write before P2’s write, then P4 will also see P1’s write before P2’s write
- Provides some form of atomicity
In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

**Principle 6**

Memory

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=1</td>
<td>a=2</td>
<td>r1 = a</td>
<td>r3 = a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2 = a</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>r4 = a</td>
</tr>
</tbody>
</table>

All values zero initially

- W(a,1)
- W(a,2)
- R(a)
- R(a)
- R(a)
- R(a)

**Question:** is r1=0, r2=2, r3=0, r4=1 allowed?

- Not allowed: r1 == 1, r2 == 2, r3 == 2, r4 == 1
- If P3 observes P1’s write before P2’s write, then P4 will also see P1’s write before P2’s write
- Provides some form of atomicity
In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

**Principle 6**

- **P1**: $a=1$
- **P2**: $a=2$
- **P3**: $r1 = a$
  - $r2 = a$
- **P4**: $r3 = a$
  - $r4 = a$

All values zero initially

---

**Question**: is $r1=0$, $r2=2$, $r3=0$, $r4=1$ allowed?

- **Not allowed**: $r1 == 1$, $r2 == 2$, $r3 == 2$, $r4 == 1$
- If P3 observes P1’s write before P2’s write, then P4 will also see P1’s write before P2’s write.
- Provides some form of atomicity
In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

All values zero initially, registers r1==r2==1

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>xchg</td>
<td>xchg(a,r1)</td>
<td>xchg(b,r2)</td>
<td>r3 = a</td>
<td>r5 = b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>r4 = b</td>
<td>r6 = a</td>
</tr>
</tbody>
</table>

Memory
In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

### Principle 7

All values zero initially, registers r1=r2=1

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>xchg(a,r1)</td>
<td>P2</td>
<td>xchg(b,r2)</td>
</tr>
<tr>
<td>P3</td>
<td>r3 = a</td>
<td>r4 = b</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>r5 = b</td>
<td>r6 = a</td>
<td></td>
</tr>
</tbody>
</table>

Question: is r3=1, r4=0, r5=0, r6=1 allowed?

---

```
Memory
```

Diagram:

- **P1**: xchg(a,r1)
- **P2**: xchg(b,r2)
- **P3**: r3 = a, r4 = b
- **P4**: r5 = b, r6 = a
Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

All values zero initially, registers $r_1=r_2=1$

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
</table>
|   | $xchg(a, r_1)$ | $xchg(b, r_2)$ | $r_3=a$ | $r_5=b$
|   |         |         | $r_4=b$ | $r_6=a$

Question: is $r_3=1$, $r_4=0$, $r_5=0$, $r_6=1$ allowed?

Memory

X(a,r1)  R(a)  R(b)  R(b)  R(a)  X(b,r2)
In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

**Principle 7**

All values zero initially, registers $r1=r2=1$

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>xchg(a, r1)</td>
<td>xchg(b, r2)</td>
<td>r3 = a</td>
<td>r5 = b</td>
</tr>
<tr>
<td>r4 = b</td>
<td>r6 = a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Question: is $r3=1$, $r4=0$, $r5=0$, $r6=1$ allowed?

- Not allowed: $r3 == 1$, $r4 == 0$, $r5 == 1$, $r6 == 0$
Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

Question: is r3=1, r4=0, r5=0, r6=1 allowed?

- Not allowed: r3 == 1, r4 == 0, r5 == 1, r6 == 0
- If P3 observes ordering P1:xchg → P2:xchg, then P4 observes the same ordering.

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>xchg(a, r1)</td>
<td>xchg(b, r2)</td>
<td>r3 = a</td>
<td>r5 = b</td>
</tr>
<tr>
<td>P2</td>
<td>xchg(b, r2)</td>
<td>r4 = b</td>
<td>r6 = a</td>
<td></td>
</tr>
</tbody>
</table>

All values zero initially, registers r1==r2==1
Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

All values zero initially, registers r1==r2==1

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xchg(a, r1)</td>
<td>xchg(b, r2)</td>
<td>r3 = a</td>
<td>r5 = b</td>
</tr>
<tr>
<td></td>
<td>r4 = b</td>
<td>r4 = b</td>
<td>r6 = a</td>
<td></td>
</tr>
</tbody>
</table>

Question: is r3=1, r4=0, r5=0, r6=1 allowed?

- Not allowed: r3 == 1, r4 == 0, r5 == 1, r6 ==0
- If P3 observes ordering P1:xchg → P2:xchg, then P4 observes the same ordering
- (xchg has implicit lock)
Principle 8

Reads and writes are not reordered with locked instructions. (enables synchronized programming!)

All values zero initially but r1 = r3 = 1

P1
\[ \text{xchg}(a,r1) \]
\[ r2 = b \]

P2
\[ \text{xchg}(b,r3) \]
\[ r4 = a \]
Principle 8

Reads and writes are not reordered with locked instructions. (enables synchronized programming!)

All values zero initially but \( r_1 = r_3 = 1 \)

- **P1**
  - Xchg(a, r1)
  - \( r_2 = b \)

- **P2**
  - Xchg(b, r3)
  - \( r_4 = a \)

Memory

- X(a, r1)
- X(b, r3)
- R(b)
- R(a)
Principle 8

Reads and writes are not reordered with locked instructions. (enables synchronized programming!)

- Not allowed: $r_2 = 0$, $r_4 = 0$
Principle 8

Reads and writes are not reordered with locked instructions. (enables synchronized programming!)

• Not allowed: $r2 == 0$, $r4 == 0$
• Locked instructions have total order, so $P1$ and $P2$ agree on the same order
Principle 8

Reads and writes are not reordered with locked instructions. (enables synchronized programming!)

All values zero initially but r1 = r3 = 1

P1
xchg(a, r1)
r2 = b

P2
xchg(b, r3)
r4 = a

• Not allowed: r2 == 0, r4 == 0
• Locked instructions have total order, so P1 and P2 agree on the same order
• If volatile variables use locked instructions → practical sequential consistency (more later)
An Alternative View: x86-TSO


“[...] **real multiprocessors typically do not provide the sequentially consistent memory** that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, **the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion. [...]** We present a new x86-TSO programmer’s model that, to the best of our knowledge, suffers from none of these problems. **It is mathematically precise** (rigorously defined in HOL4) but can be presented as an **intuitive abstract machine which should be widely accessible to working programmers. [...]**”

Newer RMA systems: A. Dan, P. Lam, TH, A. Vechev: Modeling and Analysis of Remote Memory Access Programming, ACM OOPSLA’16
Notions of Correctness

- **We discussed so far:**
  - Read/write of the same location
    *Cache coherence (write serialization and atomicity)*
  - Read/write of multiple locations
    *Memory models (visibility order of updates by cores)*

- **Now: objects (variables/fields with invariants defined on them)**
  - Invariants “tie” variables together
  - Sequential objects
  - Concurrent objects
Sequential Objects

- Each object has a type
- A type is defined by a class
  - Set of fields forms the state of an object
  - Set of methods (or free functions) to manipulate the state

- Remark
  - An interface is an abstract type that defines behavior
    
    A class implementing an interface defines several types
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0

capacity = 8
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)

capacity = 8
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)

capacity = 8
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)

```
<table>
<thead>
<tr>
<th>head</th>
<th>tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
```

capacity = 8
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)

capacity = 8
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)

capacity = 8
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
  - deq() [x]

capacity = 8
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
  - deq() [x]

capacity = 8
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
  - deq() [x]

capacity = 8
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
  - deq() [x]
  - ...

capacity = 8
Sequential Queue

class Queue {
private:
    int head, tail;
    std::vector<Item> items;

public:
    Queue(int capacity) {
        head = tail = 0;
        items.resize(capacity);
    }
    // ...
};

capacity = 8
class Queue {
    // ...

public:
    void enq(Item x) {
        if((tail+1)%items.size() == head) {
            throw FullException;
        }
        items[tail] = x;
        tail = (tail+1)%items.size();
    }

    Item deq() {
        if(tail == head) {
            throw EmtpyException;
        }
        Item item = items[head];
        head = (head+1)%items.size();
        return item;
    }
};

capacity = 8
Sequential Execution

- (The) one process executes operations one at a time
  - Sequential 😊

- Semantics of operation defined by specification of the class
  - Preconditions and postconditions
Preconditions:
- Specify conditions that must hold before method executes
- Involve state and arguments passed
- Specify obligations a client must meet before calling a method

Example: enq()
- Queue must not be full!

class Queue {
    // ...
    void enq(Item x) {
        assert((tail+1)%items.size() != head);
        // ...
    }
};
Design by Contract!

- **Postconditions:**
  - Specify conditions that must hold after method executed
  - Involve old state and arguments passed

- **Example: enq()**
  - Queue must contain element!

```java
class Queue {
   // ...
   void enq(Item x) {
      // ...
      assert(
         (tail == (old_tail + 1)%items.size()) &&
         (items[old_tail] == x)) ;
   }
}
```
Sequential specification

- if(precondition)
  - Object is in a specified state
- then(postcondition)
  - The method returns a particular value or
  - Throws a particular exception and
  - Leaves the object in a specified state

- Invariants
  - Specified conditions (e.g., object state) must hold anytime a client could invoke an objects method!
Advantages of sequential specification

- State between method calls is defined
  - Enables reasoning about objects
  - Interactions between methods captured by side effects on object state

- Enables reasoning about each method in isolation
  - Contracts for each method
  - Local state changes global state

- Adding new methods
  - Only reason about state changes that the new method causes
  - If invariants are kept: no need to check old methods
  - Modularity!
Concurrent execution - State

- Concurrent threads invoke methods on possibly shared objects
  - At overlapping time intervals!

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Meaningful only between method executions</td>
<td>Overlapping method executions → object may never be “between method executions”</td>
</tr>
</tbody>
</table>

Each method execution takes some non-zero amount of time!
Concurrent execution - State

- Concurrent threads invoke methods on possibly shared objects
  - At overlapping time intervals!

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Meaningful only between method executions</td>
<td>Overlapping method executions → object may never be “between method executions”</td>
</tr>
</tbody>
</table>

Each method execution takes some non-zero amount of time!
Concurrent execution - Reasoning

- Reasoning must now include all possible interleavings
  - Of changes caused by methods themselves

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reasoning</td>
<td>Consider each method in isolation; invariants on state before/after execution.</td>
<td>Need to consider all possible interactions; all intermediate states during execution</td>
</tr>
</tbody>
</table>

That is, now we have to consider what will happen if we execute:

- \( \text{enq()} \) concurrently with \( \text{enq()} \)
- \( \text{deq()} \) concurrently with \( \text{deq()} \)
- \( \text{deq()} \) concurrently with \( \text{enq()} \)

Each method execution takes some non-zero amount of time!
Concurrent execution - Method addition

- Reasoning must now include all possible interleavings
  - Of changes caused by and methods themselves

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Method</td>
<td>Without affecting other methods; invariants on state before/after execution.</td>
<td>Everything can potentially interact with everything else</td>
</tr>
</tbody>
</table>

- Consider adding a method that returns the last item enqueued

```java
Item peek() {
    if(tail == head) throw EmptyException;
    return items[(tail-1) % items.size()];
}

void enq(Item x) {
    items[tail] = x;
    tail = (tail+1) % items.size();
}

Item deq() {
    Item item = items[head];
    head = (head+1) % items.size();
    return item;
}
```

- If `peek()` and `enq()` run concurrently: what if tail has not yet been incremented?
- If `peek()` and `deq()` run concurrently: what if last item is being dequeued?
Concurrent objects

- How do we describe one?
  - No pre-/postconditions 😊

- How do we implement one?
  - Plan for quadratic or exponential number of interactions and states

- How do we tell if an object is correct?
  - Analyze all quadratic or exponential interactions and states
Concurrent objects

- How do we describe one?
  - No pre-/postconditions 😊
- How do we implement one?
  - Plan for quadratic or exponential number of interactions and states
- How do we tell if an object is correct?
  - Analyze all quadratic or exponential interactions and states

Is it time to panic for (parallel) software engineers? Who has a solution?
Lock-based queue

class Queue {
private:
    int head, tail;
    std::vector<Item> items;
    std::mutex lock;

public:
    Queue(int capacity) {
        head = tail = 0;
        items.resize(capacity);
    }
    // ...
};

We can use the lock to protect Queue’s fields.
Lock-based queue

class Queue {
    // ...
    public:
    void enq(Item x) {
        std::lock_guard<std::mutex> l(lock);
        if((tail+1) % items.size() == head) {
            throw FullException;
        }
        items[tail] = x;
        tail = (tail+1) % items.size();
    }

    Item deq() {
        std::lock_guard<std::mutex> l(lock);
        if(tail == head) {
            throw EmptyException;
        }
        Item item = items[head];
        head = (head+1) % items.size();
        return item;
    }
};

One of C++’s ways of implementing a critical section
Lock-based queue

class Queue {
    // ...
public:
    void enq(Item x) {
        std::lock_guard<std::mutex> l(lock);
        if((tail+1)%items.size()==head) {
            throw FullException;
        }
        items[tail] = x;
        tail = (tail+1)%items.size();
    }

    Item deq() {
        std::lock_guard<std::mutex> l(lock);
        if(tail == head) {
            throw EmptyException;
        }
        Item item = items[head];
        head = (head+1)%items.size();
        return item;
    }
};

Class question: how is the lock ever unlocked?

One of C++’s ways of implementing a critical section
C++ Resource Acquisition is Initialization

- RAII – suboptimal name
- Can be used for locks (or any other resource acquisition)
  - Constructor grabs resource
  - Destructor frees resource
- Behaves as if
  - Implicit unlock at end of block!
- Main advantages
  - Always unlock/free lock at exit
  - No “lost” locks due to exceptions or strange control flow (goto 🎉)
  - Very easy to use

```cpp
template <typename mutex_impl>
class lock_guard {
    mutex_impl& _mtx; // ref to the mutex

public:
    lock_guard(mutex_impl& mtx) : _mtx(mtx) {
        _mtx.lock(); // lock mutex in constructor
    }

    ~lock_guard() {
        _mtx.unlock(); // unlock mutex in destructor
    }
};
```
Example execution
Example execution

```c
void enq(Item x) {
    enq() is called
}
```
Example execution

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
}
```
Example execution

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
```
Example execution

deq() is called by another thread

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
}

Item deq() {
```
Example execution

deq() has to wait for the lock to be released

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if ((tail+1)%items.size()==head) {
        throw FullException;
    }
}
```

```cpp
Item deq() {
    std::lock_guard<std::mutex> l(lock);
    return items[head++];
}
```
Example execution

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
}
```

`deq()` has to wait for the lock to be released

```cpp
Item deq() {
    std::lock_guard<std::mutex> l(lock);
}```
Example execution

deq() has to wait for the lock to be released

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

```cpp
Item deq() {
    std::lock_guard<std::mutex> l(lock);
}
```
Example execution

void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

deq() releases the lock; deq() acquires it and proceeds.

Item deq() {
    std::lock_guard<std::mutex> l(lock);
Example execution

enq() releases the lock; deq() acquires it and proceeds.

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

```cpp
Item deq() {
    std::lock_guard<std::mutex> l(lock);
    if(tail == head) {
```
**Example execution**

`enq()` releases the lock; `deq()` acquires it and proceeds.

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

```cpp
Item deq() {
    std::lock_guard<std::mutex> l(lock);

    if(tail == head) {
        throw EmptyException;
    }
```
Example execution

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

Item `deq()` {
    std::lock_guard<std::mutex> l(lock);

    if(tail == head) {
        throw EmptyException;
    }
}

`enq()` releases the lock; `deq()` acquires it and proceeds.
Example execution

`enq()` releases the lock; `deq()` acquires it and proceeds.

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

Item deq() {
    std::lock_guard<std::mutex> l(lock);

    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
```
Example execution

enq() releases the lock; deq() acquires it and proceeds.

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```
Example execution

enq() releases the lock; deq() acquires it and proceeds.

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

```cpp
Item deq() {
    std::lock_guard<std::mutex> l(lock);

    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
}
```
Example execution

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

```cpp
deq() releases the lock

Item deq() {
    std::lock_guard<std::mutex> l(lock);

    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
}
```
Example execution

```cpp
template void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

```cpp
Item deq() {
    std::lock_guard<std::mutex> l(lock);

    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
}
```
Example execution

```cpp
template <typename Item, typename Exception>
class Queue {
public:
    Queue() : head(0), tail(0), size(0) {};
    ~Queue() {};

    bool isEmpty() const { return size == 0; }
    bool isFull() const { return tail == size; }

    void enq(Item x) {
        std::lock_guard<std::mutex> l(lock);
        if((tail+1)%items.size()==head) {
            throw FullException;
        }
        items[tail] = x;
        tail = (tail+1)%items.size();
    }

    Item deq() {
        std::lock_guard<std::mutex> l(lock);
        if(tail == head) {
            throw EmptyException;
        }
        Item item = items[head];
        head = (head+1)%items.size();
        return item;
    }
private:
    size_t head, tail;
    size_t size;
    std::vector<Item> items;
};
```

Methods effectively execute one after another, sequentially.
Correctness

- **Is the locked queue correct?**
  - Yes, only one thread has access if locked correctly
  - Allows us again to reason about pre- and postconditions
  - Smells a bit like sequential consistency, no?
- **Class question: What is the problem with this approach?**
Correctness

- **Is the locked queue correct?**
  - Yes, only one thread has access if locked correctly
  - Allows us again to reason about pre- and postconditions
  - Smells a bit like sequential consistency, no?

- **Class question: What is the problem with this approach?**
  - Same as for SC 😊

It does not scale!
What is the solution here?