

An Introduction to the Formalised Memory Model for Linux Kernel

SeongJae Park <sj38.park@gmail.com>

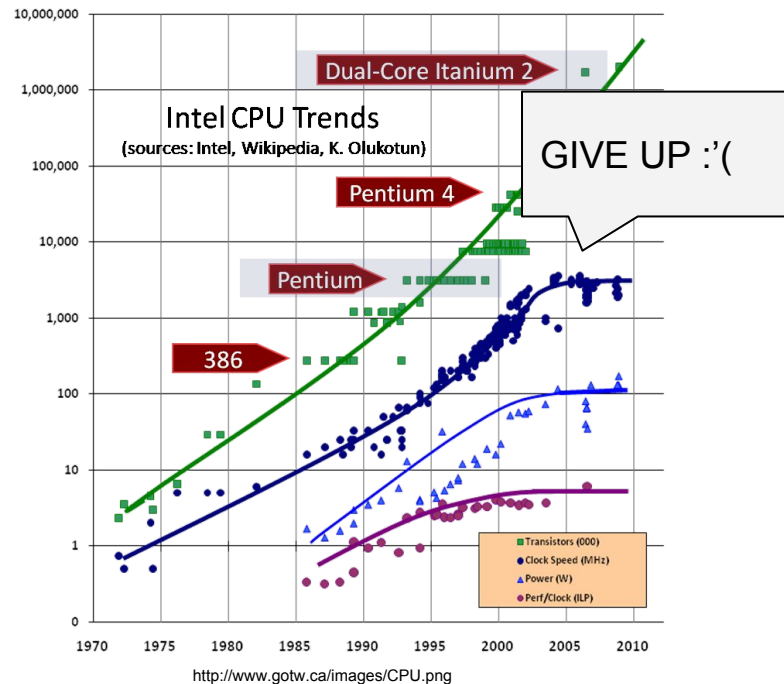
I, SeongJae Park

- SeongJae Park <sj38.park@gmail.com>
- Contributing to the Linux kernel just for fun and profit since 2012
- Developing GCMA (<https://github.com/sjp38/linux.gcma>)
- Maintaining Korean translation of Linux kernel memory barrier document
 - The translation has merged into mainline since v4.9-rc1
 - https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/ko_KR/memory-barriers.txt?h=v4.9-rc1



Programmers in Multi-core Land

- Processor vendors changed their mind to increase number of cores instead of clock speed a decade ago
 - Now, multi-core system is prevalent
 - Even octa-core portable bomb in your pocket, maybe?
- As a result, ***the free lunch is over***;
parallel programming is essential for high performance and scalability



Writing Correct Parallel Program is Hard

- Compilers and processors are heavily optimized for Instructions Per Cycle, not programmer perspective goals such as response time or throughput of meaningful (in people's context) progress
- Nature of parallelism is counter-intuitive
 - Time is relative, before and after is ambiguous, even simultaneous available
- C language developed with Uni-Processor assumption
 - *"Et tu, C?"*

CPU 0	CPU 1
<pre>X = 1; Y = 1; X = 2; Y = 2;</pre>	<pre>assert(Y == 2 && X == 1)</pre>

CPU 1 assertion can be true in Linux Kernel

TL; DR

- Memory operations can be reordered, merged, or discarded in any way unless it violates memory model defined behavior
 - In short, ``We're all mad here'' in parallel land
- Knowing memory model is important to write correct, fast, scalable parallel program



Reordering for Better IPC^[*]

^[*] IPC: Instructions Per Cycle

Simple Program Execution Sequence

- Programmer writes program in C-like human readable language
- Compiler translates human readable code into assembly language
- Assembler generates executable binary from the assembly code
- Processor executes instruction sequence in the binary
 - Execution result is guaranteed to be same with sequential execution;
In other words, the execution itself is not guaranteed to be sequential

```
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 0;
}
```



```
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
```



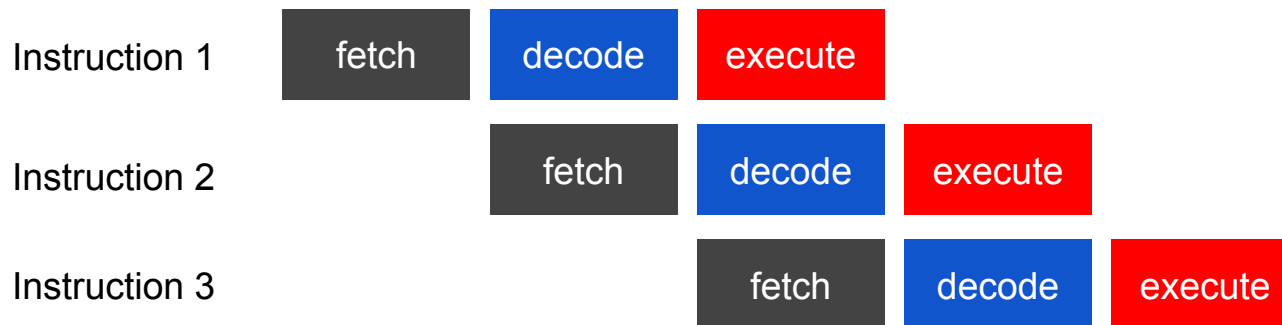
```
00000000: 7f45 4c46 0201 0100 0000
0000 0000 0000 .ELF.....
00000010: 0200 3e00 0100 0000 3004
4000 0000 0000 ..>.....0.@....
00000020: 4000 0000 0000 0000 d819
0000 0000 0000 @.....
00000030: 0000 0000 4000 3800 0900
```

Compiler

Assembler

Instruction Level Parallelism (ILP)

- Pipelining introduces instruction level parallelism
 - Each instruction is splitted up into a sequence of steps;
Each step can be executed in parallel, instructions can be processed concurrently



If not pipelined, **3 cycles per instruction**


3-depth pipeline can retire 3 instructions in 5 cycle: **1.7 cycles per instruction**

Dependent Instructions Harm ILP

- If an instruction is dependent to result of previous instruction, it should wait until the previous one finishes execution

- E.g., $a = b + c;$

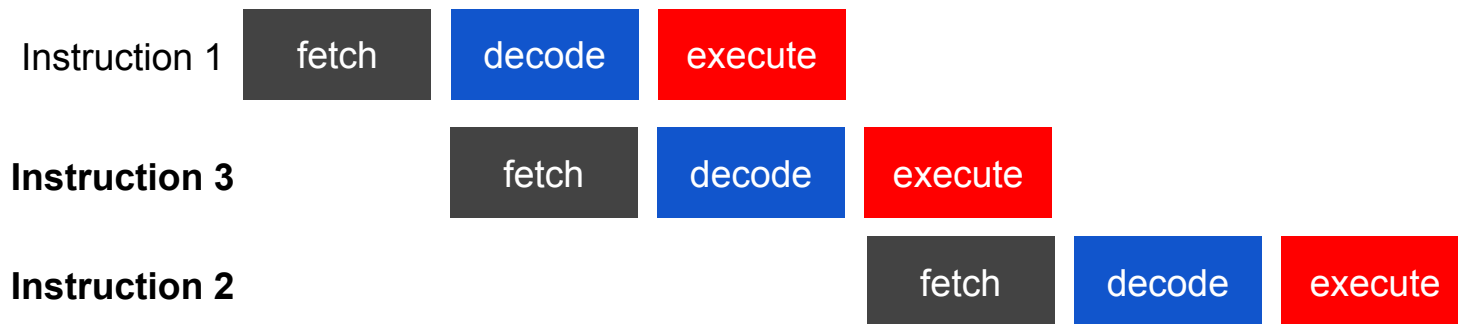
$d = a + b;$



In this case, *instruction 2* depends on result of *instruction 1*
(e.g., first instruction modifies opcode of next instruction)
7 cycles for 3 instructions: **2.3 cycles per instruction**

Instruction Reordering Helps Performance

- By reordering dependent instructions to be located in far away, total execution time can be shorten
- If the reordering is guaranteed to not change the result of the instruction sequence, it would be helpful for better performance



instruction 2 depends on result of instruction 1

(e.g., first instruction modifies opcode of next instruction)

By reordering instruction 2 and 3, total execution time can be shorten

6 cycles for 3 instructions: **2 cycles per instruction**

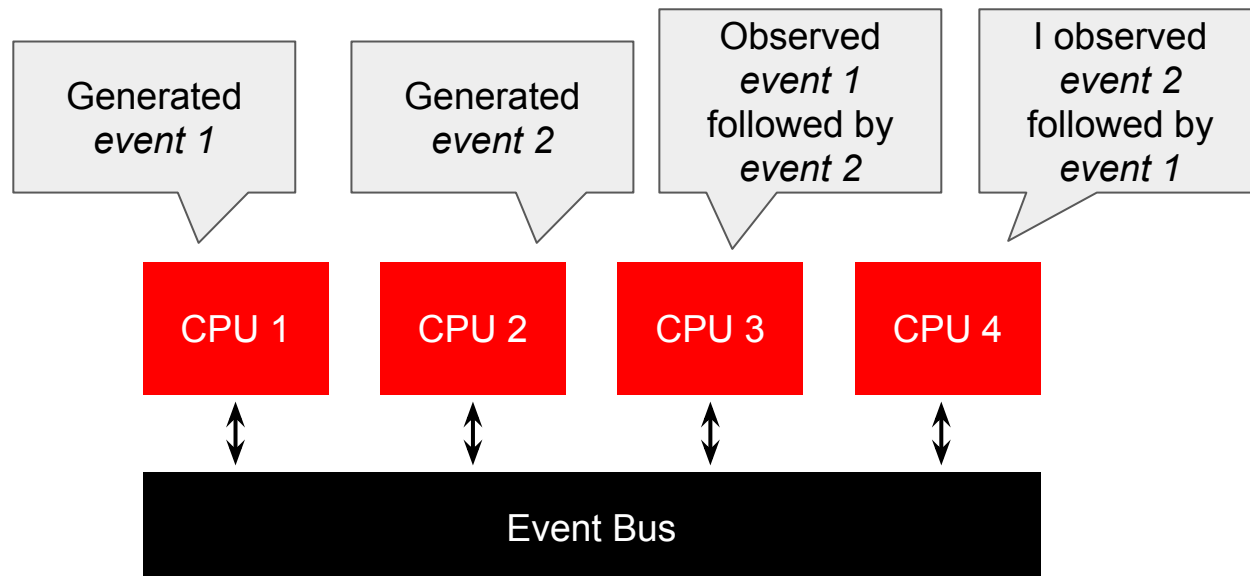
Reordering is Legal, Encouraged Behavior, But...

- If *program causality* is guaranteed, any reordering is legal
- Processors and compilers can make reordering of instructions for **better IPC**
- *program causality* defined with single processor environment
- IPC focused reordering doesn't aware programmer perspective performance goals such as throughput or latency
- On Multi-processor system, reordering could harm not only correctness, but also performance

Counter-intuitive Nature of Parallelism

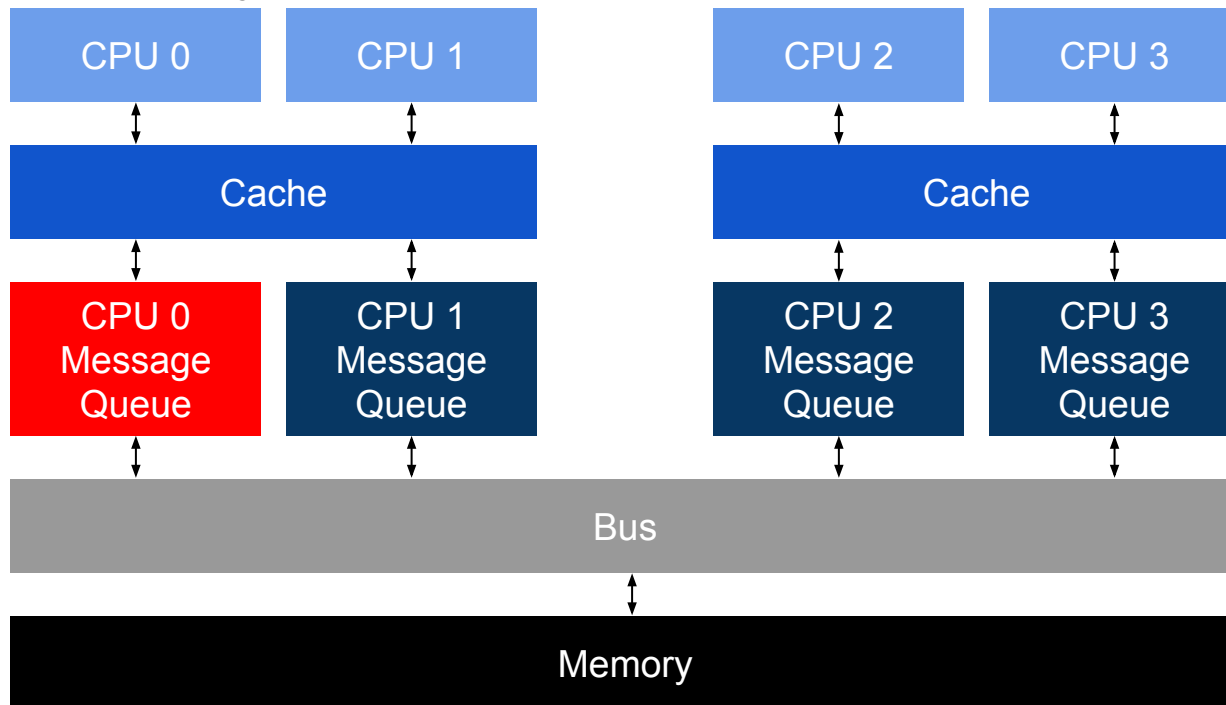
Time is Relative ($E = MC^2$)

- Each CPU generates their events in their time, observes effects of events in relative time
- It is impossible to define absolute order of two concurrent events;
Only relative observation order is possible



Relative Event Propagation of Hierarchical Memory

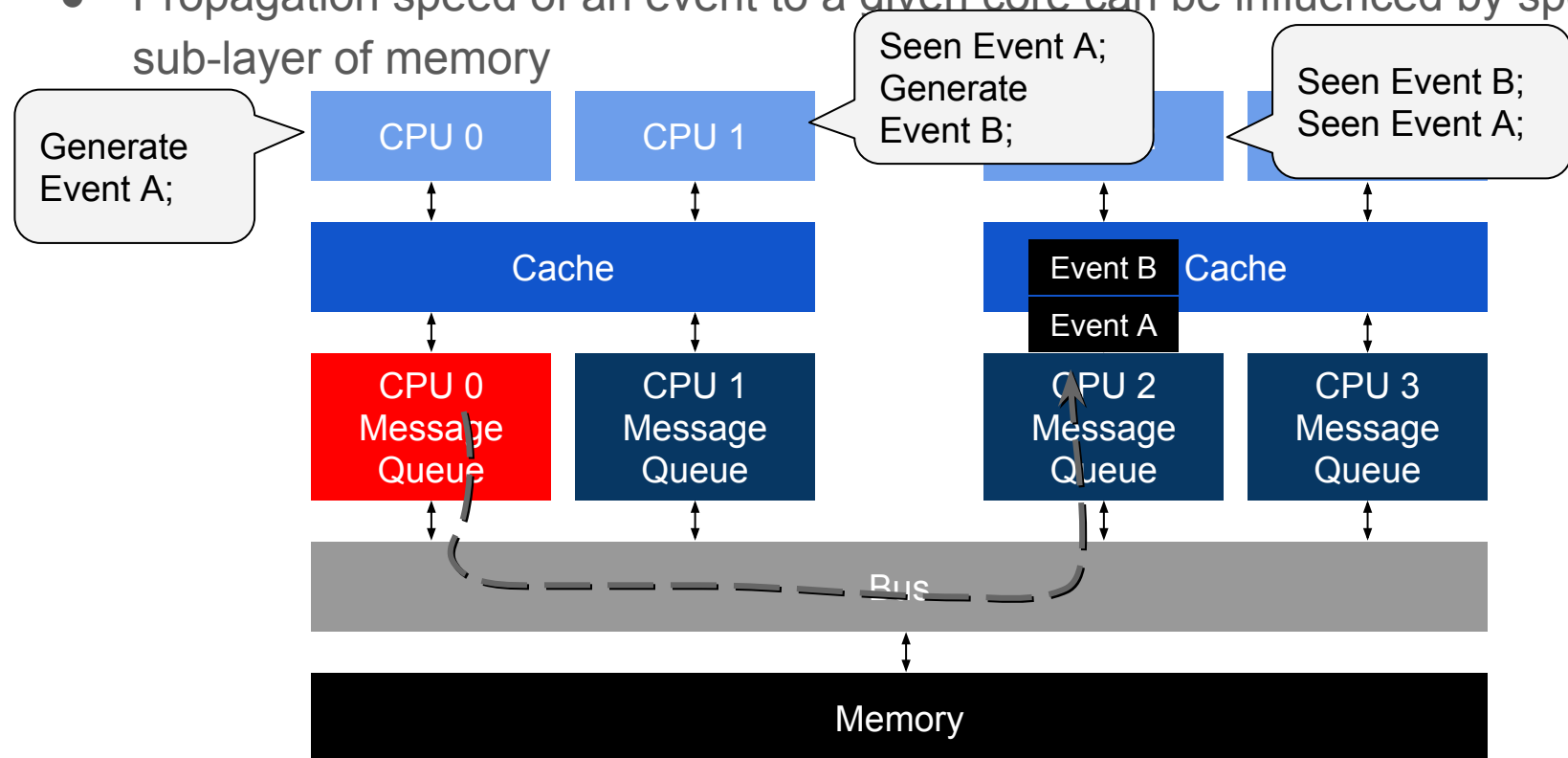
- Most system equip hierarchical memory for better performance and space
- Propagation speed of an event to a given core can be influenced by specific sub-layer of memory



If CPU 0 Message Queue is busy, CPU 2 can observe an event from CPU 1 (*event A*) followed by an event of CPU 0 (*event B*) though CPU 1 observed *event B* before generating *event A*

Relative Event Propagation of Hierarchical Memory

- Most system equip hierarchical memory for better performance and space
- Propagation speed of an event to a given core can be influenced by specific sub-layer of memory



If CPU 0 Message Queue is busy, CPU 2 can observe an event from CPU 0 (*event A*) after an event of CPU 1 (*event B*) though CPU 1 observed *event A* before generating *event B*

Cache Coherency is Per-CacheLine

- It is well known that cache coherency protocol helps system memory consistency
- In actual, it guarantees Per-Cacheline sequential consistency only
- Every CPU will eventually agree about global order of each variable, but some CPU can aware the change faster than others, order of changes between different variables is not guaranteed



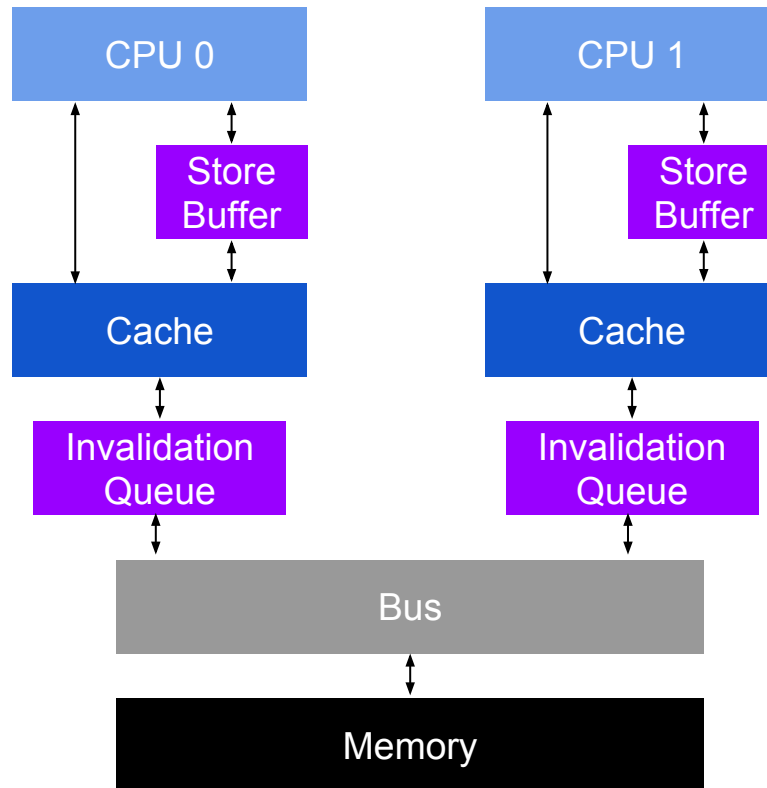
Schrödinger's cat is dead.

Or not?



System with Store Buffer and Invalidation Queue

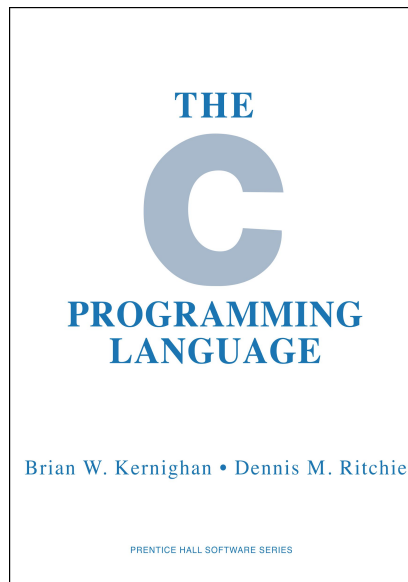
- Store Buffer and Invalidation Queue deliver effect of event but does not guarantee order of observation on each CPU



C-language and Multi-Processor

C-language Doesn't Know Multi-Processor

- By the time of initial C-language development, multi-processor was rare
- As a result, C-language has only few guarantees about memory operations on multi-processor
- ***Undefined behavior*** is allowed for undefined case



[https://upload.wikimedia.org/wikipedia/commons/thumb/9/95/The_C_Programming_Language_First_Edition_Cover_\(2\).svg/2000px-The_C_Programming_Language_First_Edition_Cover_\(2\).svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/9/95/The_C_Programming_Language_First_Edition_Cover_(2).svg/2000px-The_C_Programming_Language_First_Edition_Cover_(2).svg.png)

Compiler Optimizes Code

- Clever compilers try hard (really hard) to optimize code for high IPC (again, not for programmer perspective goals)
 - Converts small, private function to inline code
 - Reorder memory access code to minimize dependency
 - Simplify unnecessarily complex loops, ...
- Optimization uses term `Undefined behavior` as they want
 - It's legal, but sometimes do insane things in programmer's perspective
- Memory access reordering of compiler based on C-standard, which doesn't aware multi-processor system, can generate unintended program
- Linux kernel uses *compiler directives* and *volatile keyword* to enforce memory ordering
- C11 has much more improvement, though

Memory Models

Each Environment Provides Own Memory Model

- Memory model, a.k.a Memory consistency model
- Defines what values can be obtained by the code's load instructions
- Each programming environment like Instruction Set Architecture, Programming language, Operating system, etc defines own memory model
 - Modern language memory models (e.g., Golang, Rust, Java, C11, ...) aware multi-processor



Each ISA Provides Specific Memory Model

- Some architectures have stricter ordering enforcement rule than others
- PA-RISC CPUs are strictest, Alpha is weakest
- Because Linux kernel supports multiple architectures, it defines its memory model based on weakest one, the Alpha

	Alpha	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
AMD64	Y		Y	Y	Y	Y		Y	Y
ARMv7-A/R	Y	Y	Y	Y	Y	Y	Y		Y
IA64	Y	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y	Y				
PA-RISC CPUs									
POWER™	Y	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)				Y	Y		Y		Y
SPARC TSO					Y				Y
x86					Y				Y
(x86 OOSTore)	Y	Y	Y	Y	Y				Y
zSeries®				Y	Y				Y

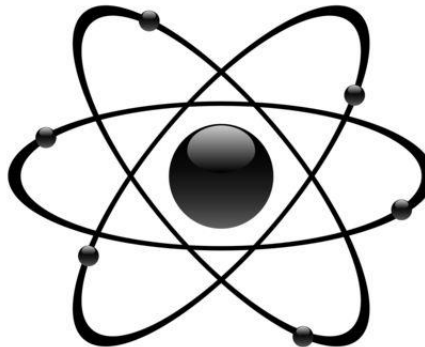
Synchronization Primitives

- Because reordering and asynchronous effect propagation is legal, synchronization primitives are necessary to write human intuitive program
- Most memory model provides synchronization primitives including atomic read-modify-write instructions and memory barriers.



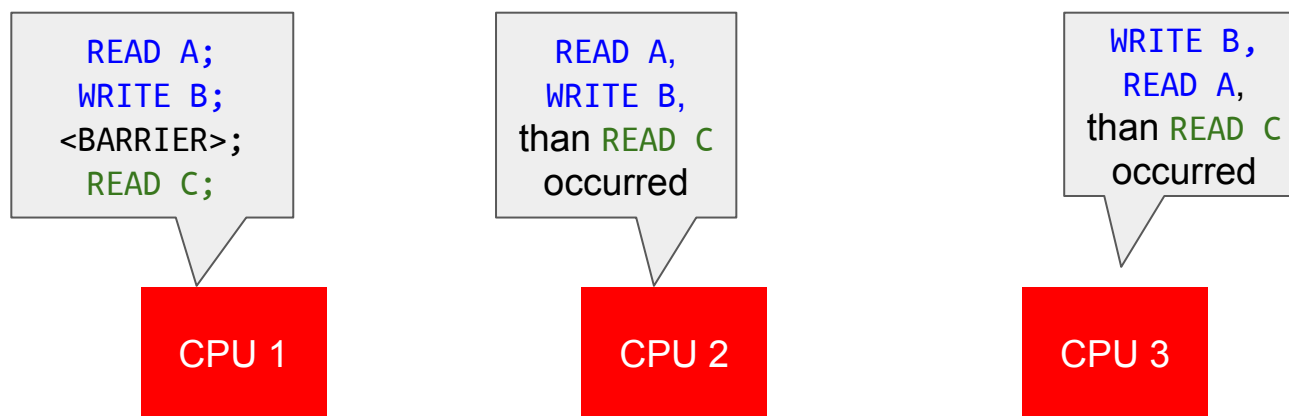
Atomic Operations

- Atomic operations are configured with multiple sub operations
 - E.g., compare-and-swap, fetch-and-add, test-and-set
- Atomic operations have mutual exclusiveness
 - Middle state of atomic operation execution cannot be seen by others
 - It can be thought of as small critical section that protected by a global lock
- Almost every hardware supports basic atomic operations



Memory Barriers

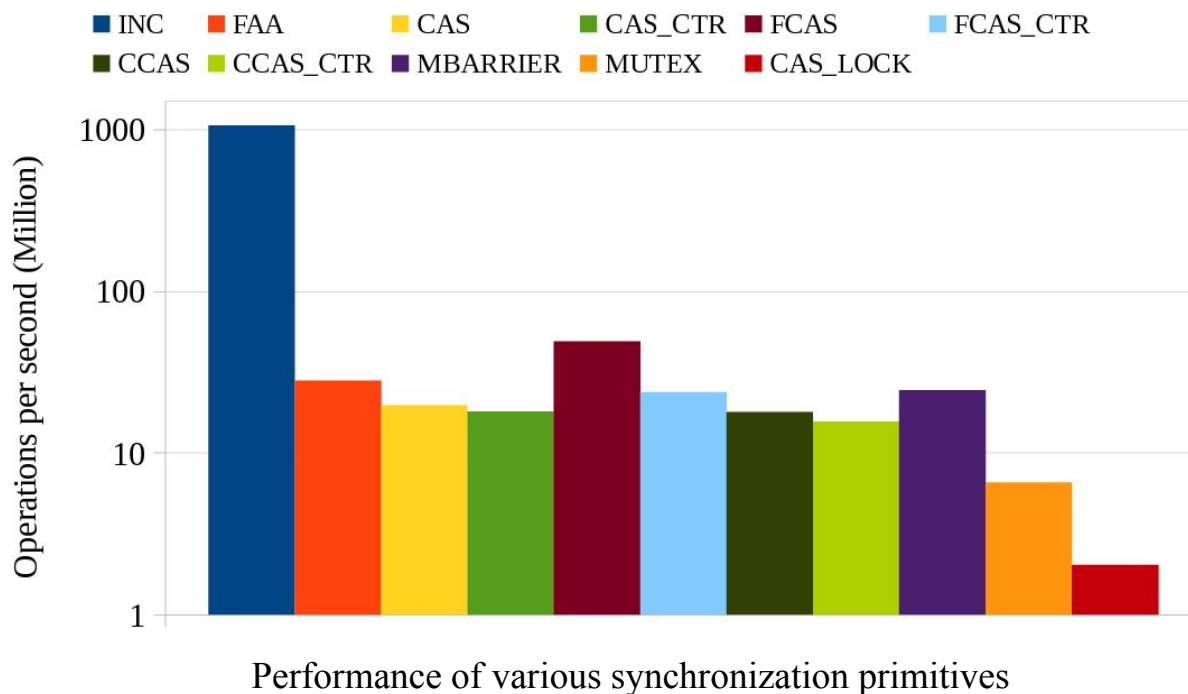
- To allow synchronization of memory operations, memory model provides enforcement primitives, namely, memory barriers
- In general, memory barriers guarantee effects of memory operations issued before it to be propagated to other components (e.g., processor) in the system before memory operations issued after the barrier



READ A and *WRITE B* can be reordered but *READ C* is guaranteed to be ordered after {*READ A*, *WRITE B*}

Cost of Synchronization Primitives

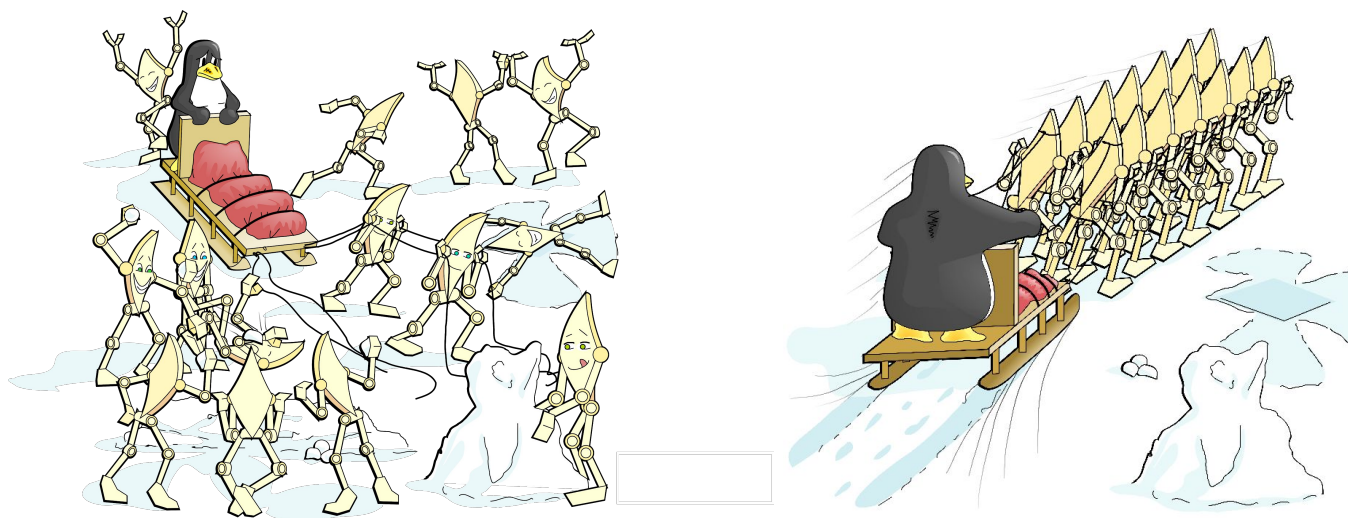
- Generally, synchronization primitives are expensive, unscalable
- Performance between synchronization primitives are different, though
- Correct selection and efficient use of synchronization primitives are important!



LKMM: Linux Kernel Memory Model

Linux Kernel Memory Model

- The memory model for Linux kernel programming environment
 - Defines what values can be obtained, given a piece of Linux kernel code, for specific load instructions in the code
- Linux kernel original memory model is necessary because
 - It uses C99 but C99 memory model is oblivious of multi-processor environment; C11 memory model aware of multi-processor, but Torvalds doesn't want it
 - It supports multiple architectures with different ISA-level memory model



LKMM: Original Memory Ordering Primitives

- Designed for weakest memory model architecture, Alpha
 - Almost every combination of reordering is possible, doesn't provide address dependency
- Atomic instructions
 - `atomic_xchg()`, `atomic_inc_return()`, `atomic_dec_return()`, ...
 - Most of those just use mapping instruction, but provide general guarantees at Kernel level
- Memory barriers
 - Compiler barriers: `WRITE_ONCE()`, `READ_ONCE()`, `barrier()`, ...
 - CPU barriers: `mb()`, `wmb()`, `rmb()`, `smp_mb()`, `smp_wmb()`, `smp_rmb()`, ...
 - Semantic barriers: ACQUIRE operations, RELEASE operations, ...
 - For detail, refer to <https://www.kernel.org/doc/Documentation/memory-barriers.txt>
- Because different memory ordering primitive has different cost, only necessary ordering primitives should be used in necessary case for high performance and scalability

Informal LKMM

- Originally, LKMM was just an informal text, 'memory-barriers.txt'
- It explains about the Linux Kernel Memory Model in English
(There is Korean translation, too: https://www.kernel.org/doc/Documentation/translations/ko_KR/memory-barriers.txt)
- To use the LKMM to prove your code, you should use Feynman Algorithm
 - Write down your code
 - Think real hard with the 'memory-barriers.txt'
 - Write down your provement
 - Hard and unreliable, of course!

```
=====
DISCLAIMER
=====
```

```
This document is not a specification; it is intentionally (for the sake of
brevity) and unintentionally (due to being human) incomplete. This document is
meant as a guide to using the various memory barriers provided by Linux, but
in case of any doubt (and there are many) please ask. Some doubts may be
resolved by referring to the formal memory consistency model and related
documentation at tools/memory-model/. Nevertheless, even this memory
model should be viewed as the collective opinion of its maintainers rather
than as an infallible oracle.
```

Formal LKMM: Help Arrives at Last

- Jade Alglave, Paul E. McKenney, and some guys made formal LKMM
- It is formal, executable memory model
 - It receives C-like simple code as input
 - The code containing parallel code snippets and a question: can this result happen?
- Based on herd7 and klitmus7
 - LKMM extends herd7 and klitmus7 to support LKMM ordering primitives in code
 - Herd7 simulates in user mode, klitmus7 runs in real kernel mode

LKMM Demonstration

- Installation

- LKMM is merged in Linux source tree at tools/memory-model;
Just get the linux source code
- Install herdtools7 (<https://github.com/herd/herdtools7>)

- Usage

- Using herd7 user mode simulation
`$ herd7 -conf linux-kernel.cfg <your-litmus-test file>`
- Using klitmus7 based real kernel mode execution
`$ mkdir mymodules`
`$ klitmus7 -o mymodules <your-litmus-test file>`
`$ cd mymodules ; make`
`$ sudo sh run.sh`

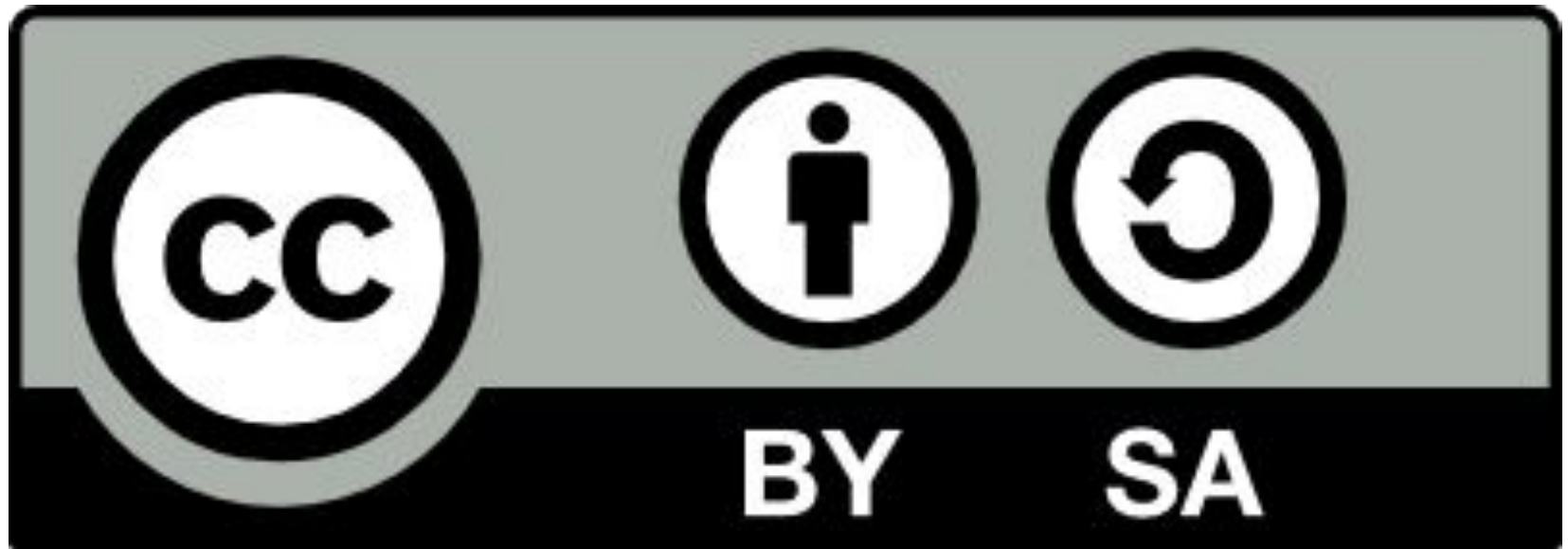
- That's it! Now you can prove your parallel code for all Linux environments!

Summary

- Nature of Parallel Land is counter-intuitive
 - Cannot define order of events without interaction
 - Ordering rule is different for different environment
 - Memory model defines their ordering rule
 - In short, **they're all mad here**
- For human-intuitive and correct program, interaction is necessary
 - Almost every environment provides memory ordering primitives including atomic instructions and memory barriers, which is expensive in common
 - Memory model defines what result can occur and cannot with given code snippet
- Formal Linux kernel memory model is available
 - Linux kernel provides its memory model based on weakest memory ordering rule architecture it supports, the Alpha, C99, and its original ordering primitives including RCU
 - Formal LKMM using herd7 is merged to mainstream; now you can prove your parallel code!

Thanks





This work by SeongJae Park is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/>.