Linux Kernel Memory Model

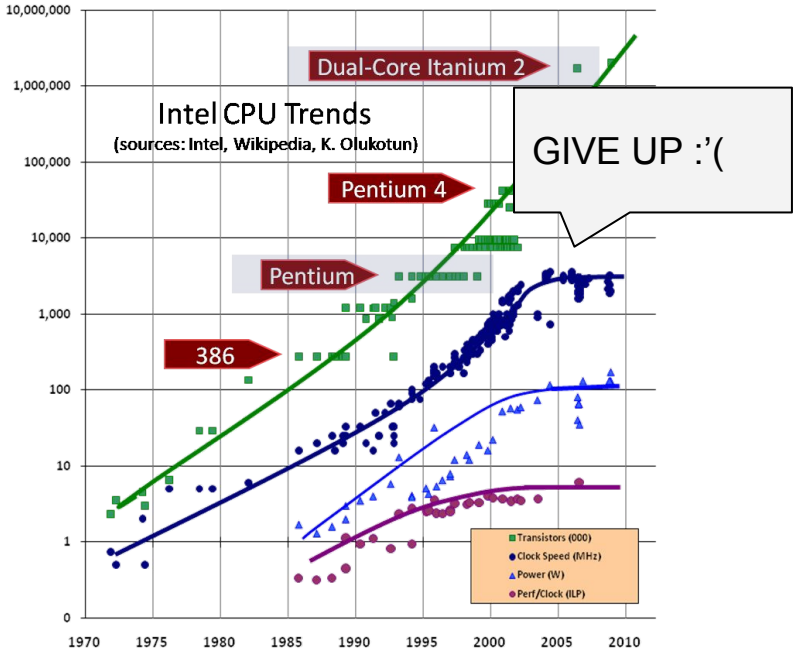
SeongJae Park <sj38.park@gmail.com>

I, SeongJae Park

- SeongJae Park <sj38.park@gmail.com>
- PhD candidate researcher at DCSLAB, SNU
- Part time linux kernel programmer at KOSSLAB
- Interested in memory management and parallel programming for OS kernels

Programmers in Multi-core Land

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



http://www.gotw.ca/images/CPU.png

Writing Correct Parallel Program is Hard

- Nature of parallelism is counter-human-intuitive
  - Time is relative, order is ambiguous, and only observations exist
- Compilers and processors do reorderings for performance
- C language developed with Uni-Processor assumption
  - Standard for the kernel (C99) is still oblivious of multi-processor systems

CPU 0	CPU 1

`X = 1;`
`Y = 1;`
`X = 2;`
`Y = 2;`
	`assert(Y == 2 && X == 1)`

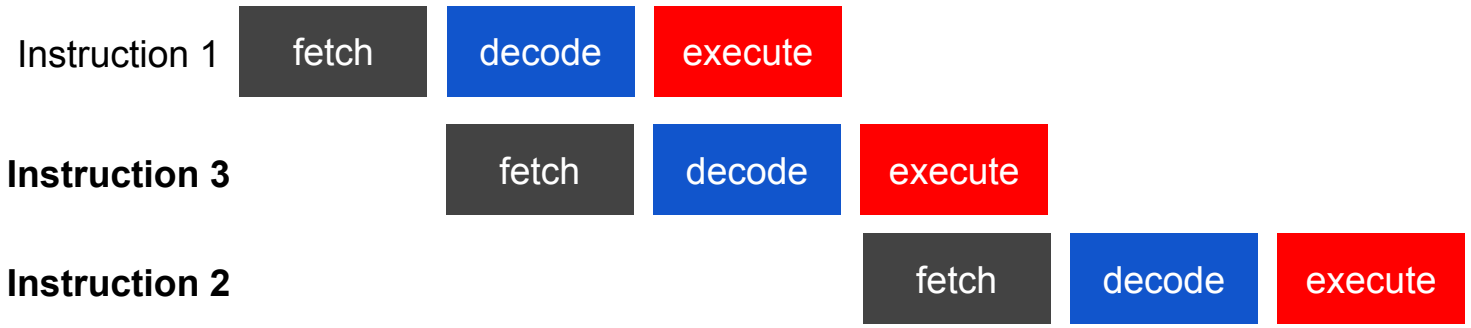
CPU 1 assertion can be true in Linux Kernel

Hardware Reorderings

Why and How?

Instruction Reordering Helps Performance

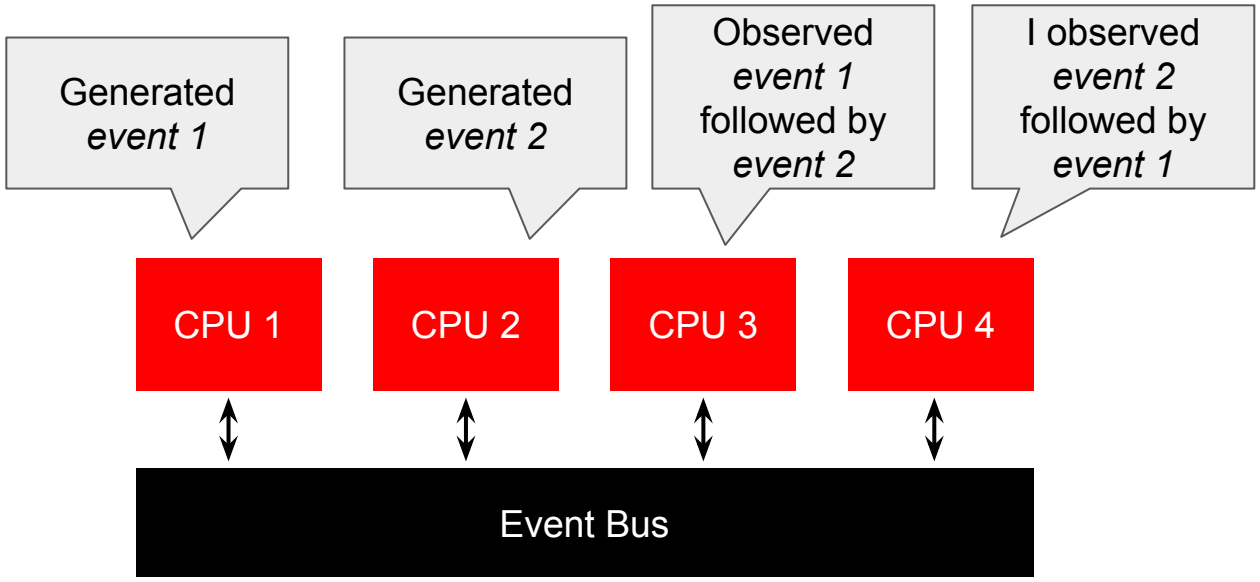
- By reordering dependent instructions to be located in far away, total execution time can be shorten
- If the reordering is guaranteed to not change the result of the instruction sequence, it would be helpful for better performance
- Such reordering is legal and recommended



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**

Time and Order is Relative

- Each CPU concurrently generates their events and observes effects of events from others
- It is impossible to define absolute order of two concurrent events; Only relative observation order is possible

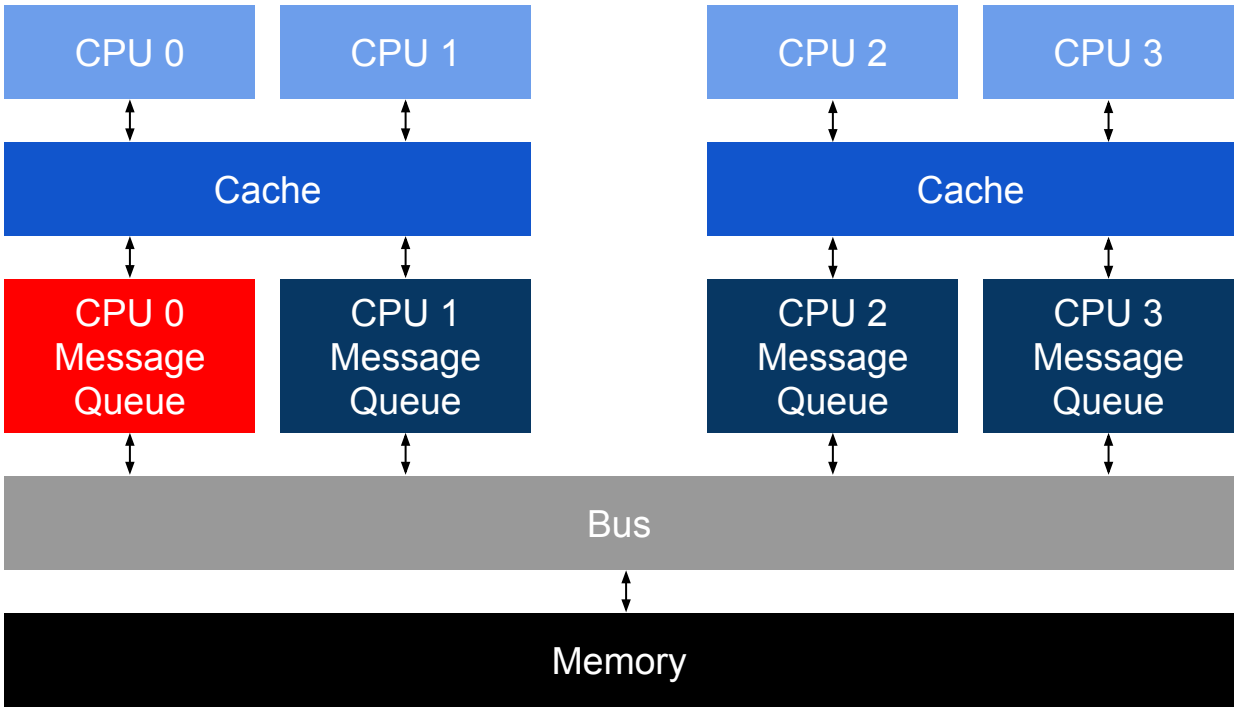


Cache Coherency is Per-CacheLine

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

An Example with A Mythological Architecture

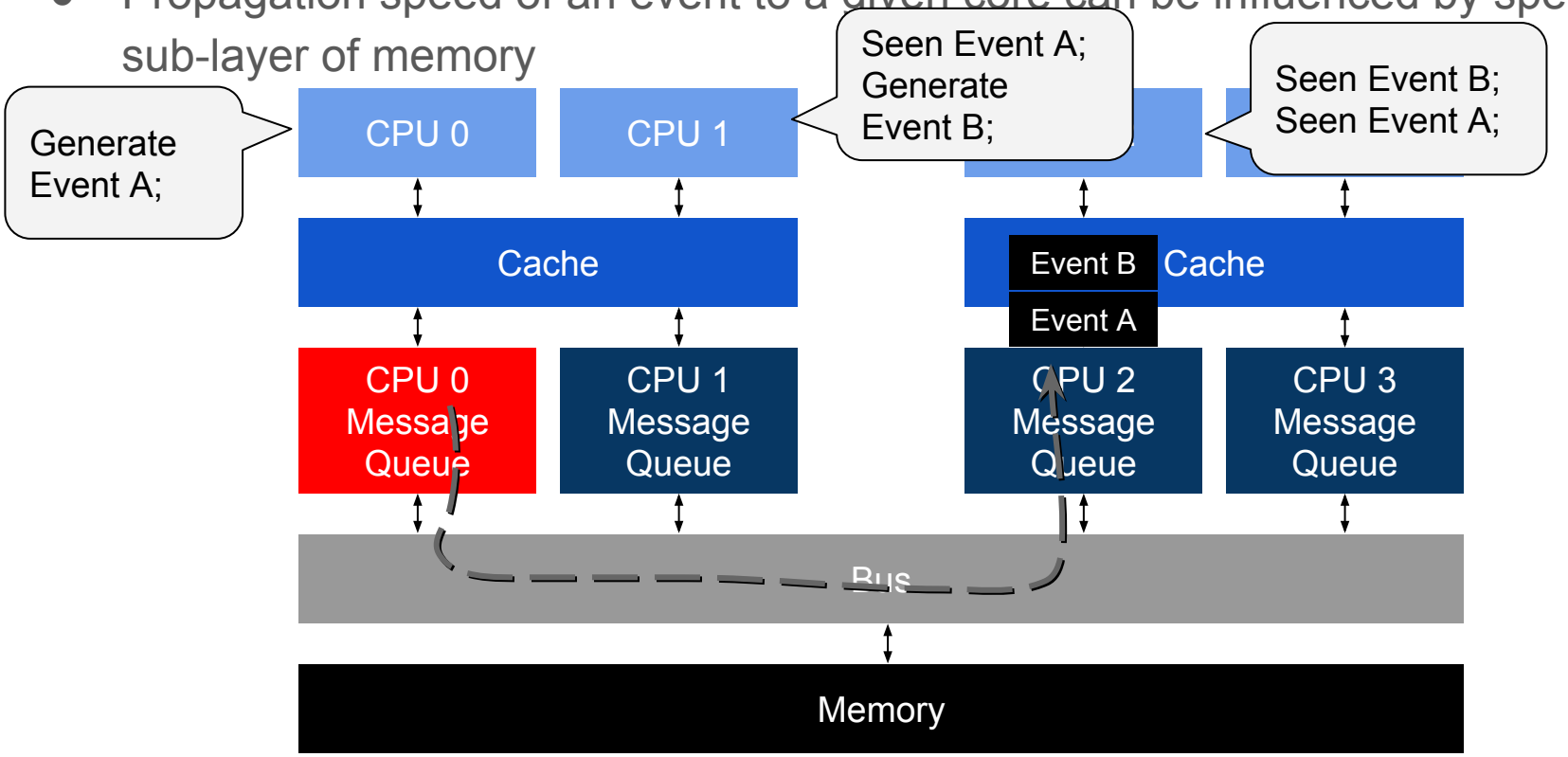
- Most system equip hierarchical memory for better performance and space
- Propagation speed of an event to a given core can be influenced by specific sub-layer of memory



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*

An Example with A Mythological Architecture

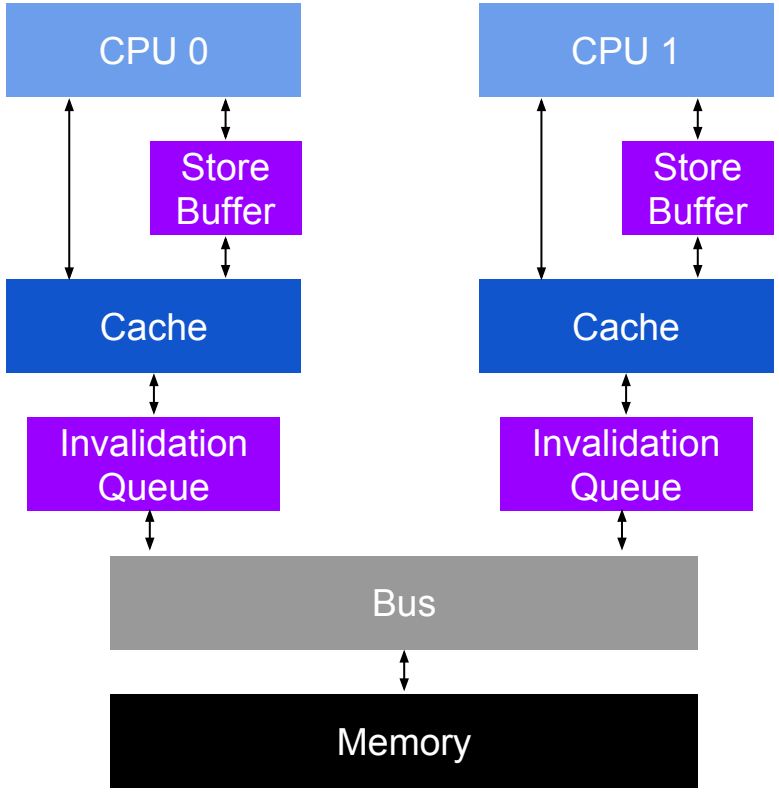
- Most system equip hierarchical memory for better performance and space
- Propagation speed of an event to a given core can be influenced by specific sub-layer of memory



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*

Yet Another Mythological Architecture

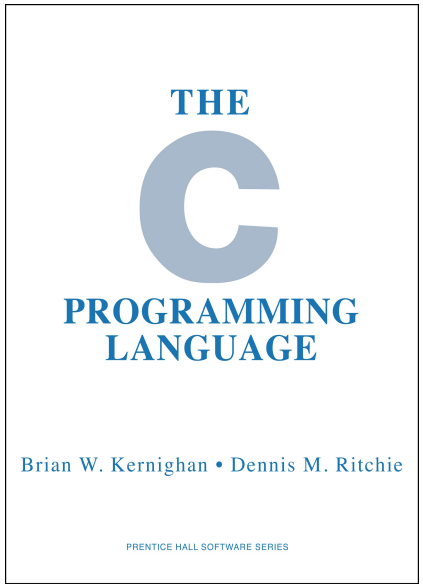
- Store Buffer and Invalidation Queue can reorder observation of stores and loads



Compiler Reorderings

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

Compiler Optimizes (Reorders) Code

- Clever compilers try hard (really hard) to optimize code for high IPC (not for human perspective goals)
  - Converts small, private functions 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
- Compilers reorder memory accesses based on the C-standard, which is oblivious of multi-processor systems; this can result in unintended programs
- C11 standard aware the multi-processor systems, though

Synchronization Primitives

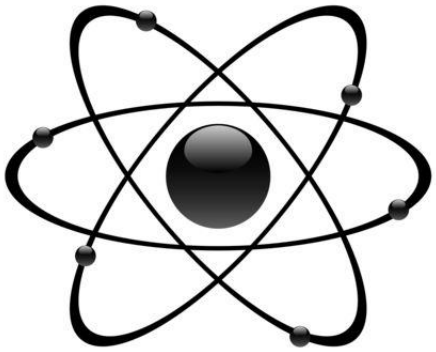
Synchronization Primitives

- Because reordering and asynchronous effect propagation is legal, synchronization primitives are necessary to write human intuitive program
- Most environments including the Linux kernel provide sufficiently enough synchronization primitives for human intuitive programs

https://s-media-cache-ak0.pinimg.com/236x/42/bc/55/42bc55a6d7e5affe2d0dbe9c872a3df9.jpg

Atomic Operations

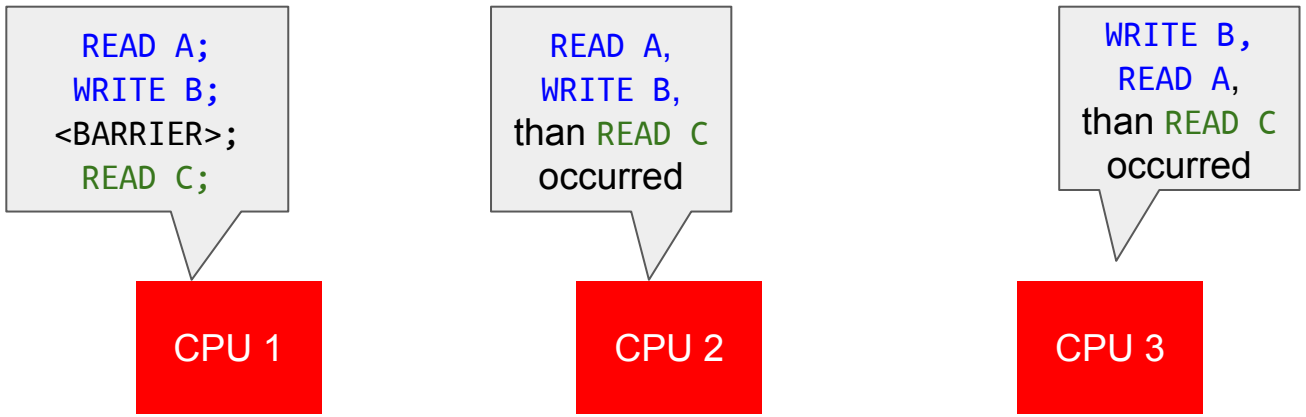
- Atomic operations are configured with multiple sub operations
  - E.g., compare[1)]-and-exchange[2)], fetch[1)]-and-add[2)], and test[1)]-and-set[2)]
- Atomic operations have mutual exclusiveness
  - Middle state of atomic operation execution cannot be seen by others
  - It can be thought of as small critical section that protected by a global lock
- Linux also provides plenty of atomic operations



http://www.scienceclarified.com/photos/atomic-mass-3024.jpg

Memory Barriers

- In general, memory barriers guarantee
  effects of memory operations issued before it
  to be propagated to other components in the system (e.g., processor)
  before memory operations issued after the barrier
- Linux provides various types of memory barriers including compiler memory barriers, hardware memory barriers, load barriers, store barriers



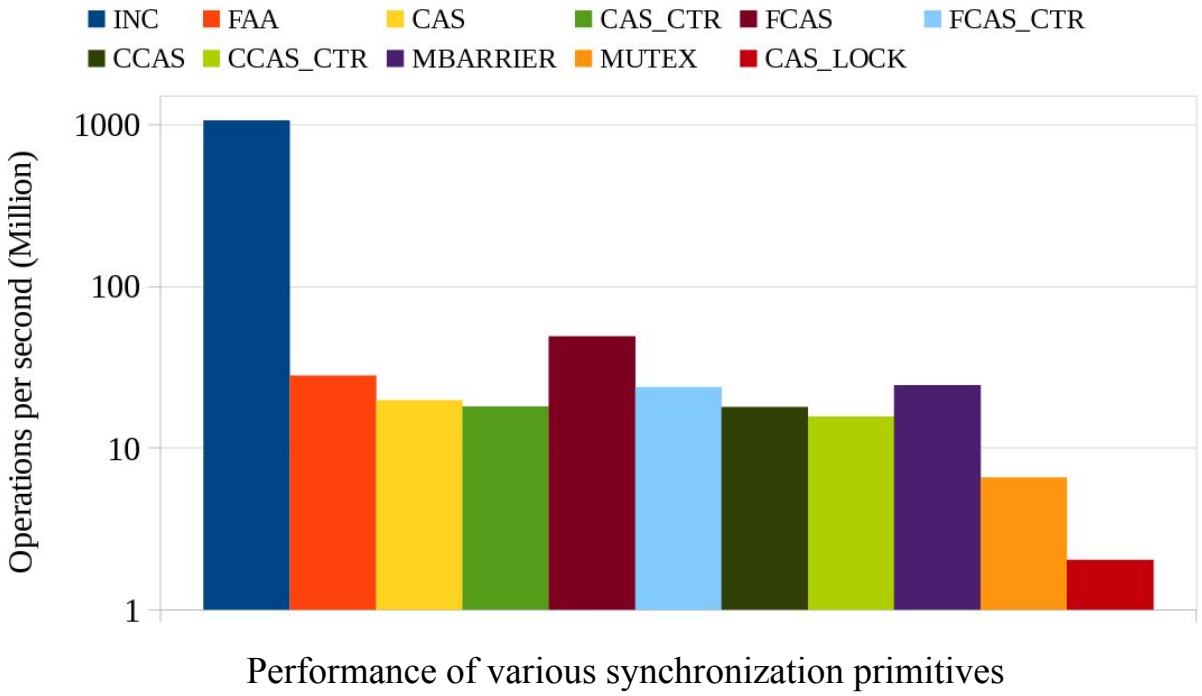
READ A and *WRITE B* can be reordered but *READ C* is guaranteed to be ordered after *{READ A, WRITE B}*

Partial Ordering Primitives

- Partial ordering primitives include read / write memory barriers, acquire / release semantics and data / address / control dependencies
- Semantics of partial ordering primitives are somewhat confusing
- Cost of partial ordering can be much cheaper than fully ordering primitives
- Linux also provides partial ordering primitives

Cost of Synchronization Primitives

- Generally, synchronization primitives are expensive, unscalable
- Each primitive costs differently; the gap is significant
- Correct selection and efficient use of synchronization primitives are important for the performance and scalability

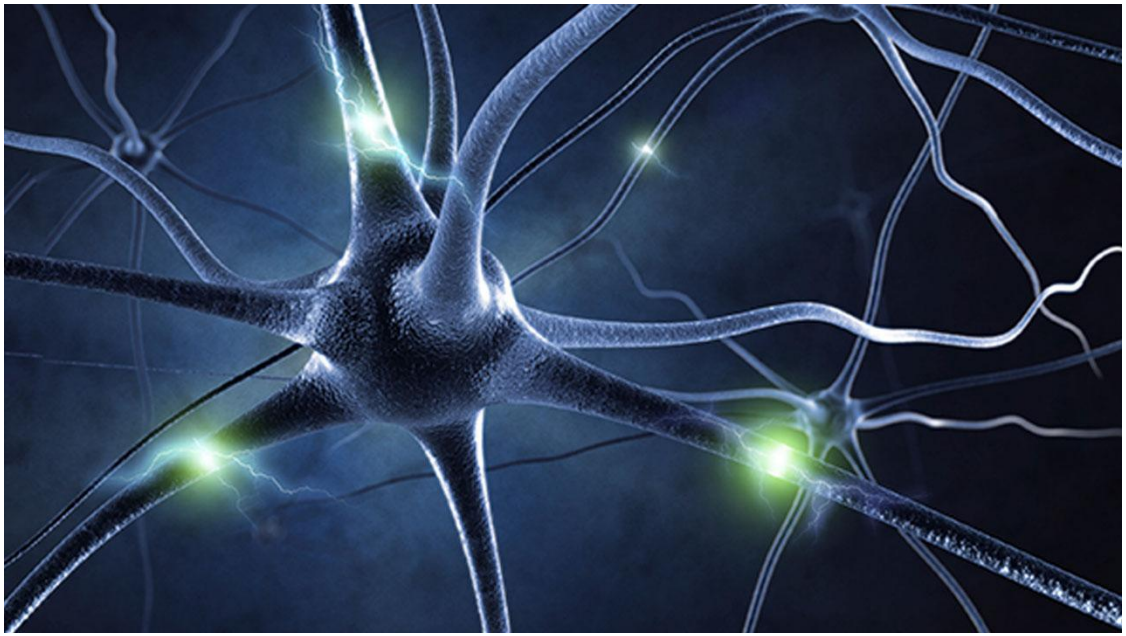


Performance of various synchronization primitives

LKMM: Linux Kernel Memory Model

Each Environment Provides Own Memory Model

- Memory model, a.k.a Memory consistency model
- Defines what values can be obtained by load instructions of given code
- Programming environments including Instruction Set Architectures, Programming languages, Operating systems, etc define their memory model
  - Modern language memory models (e.g., Golang, Rust, Java, C11, …) aware multi-processor, while C99, which is for the Linux kernel, doesn't



http://www.sciencemag.org/sites/default/files/styles/article_main_large/public/Memory.jpg?itok=4FmHo7M5

Each ISA Provides Specific Memory Model

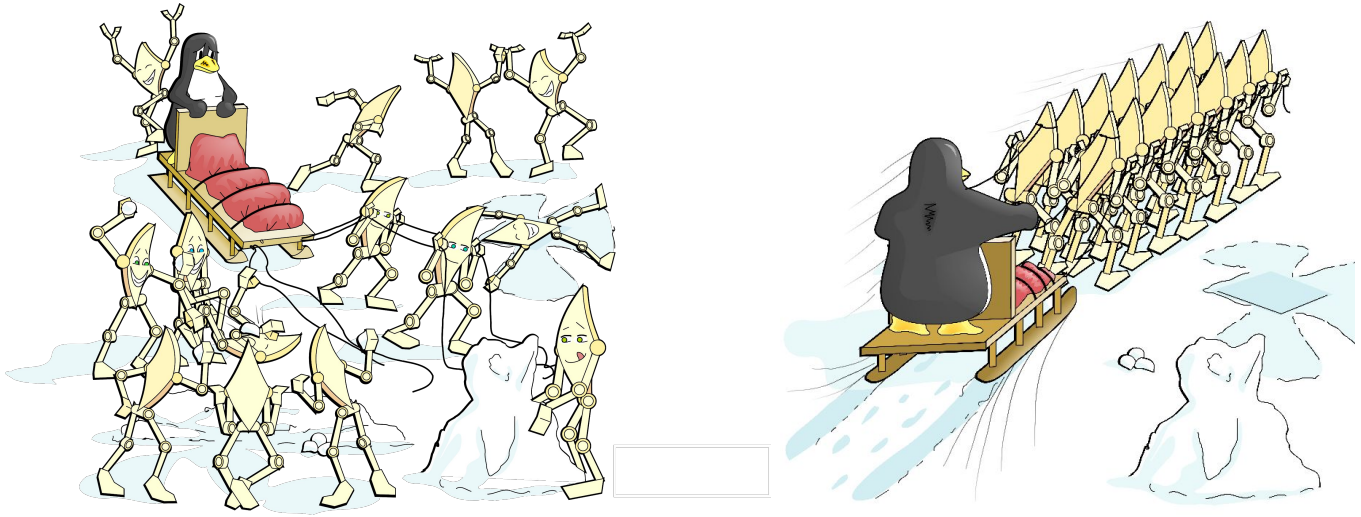
- Some architectures have stricter ordering enforcement rule than others
- PA-RISC CPUs are strictest, Alpha is weakest
- Because Linux kernel supports multiple architectures, it defines its memory model based on weakest one, **the Alpha**

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

https://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2015.01.31a.pdf

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



https://github.com/sjp38/perfbook

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
  - `atomix_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.
```

https://www.kernel.org/doc/Documentation/memory-barriers.txt

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 programming is counter-intuitive
  - Cannot define order of events without interaction
  - Ordering rule is different for different environment
  - Memory model defines their ordering rule
- 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 and executable LKMM is in the kernel tree; now you can prove your parallel code!

Thanks

Demonstration

`herdtools` Install

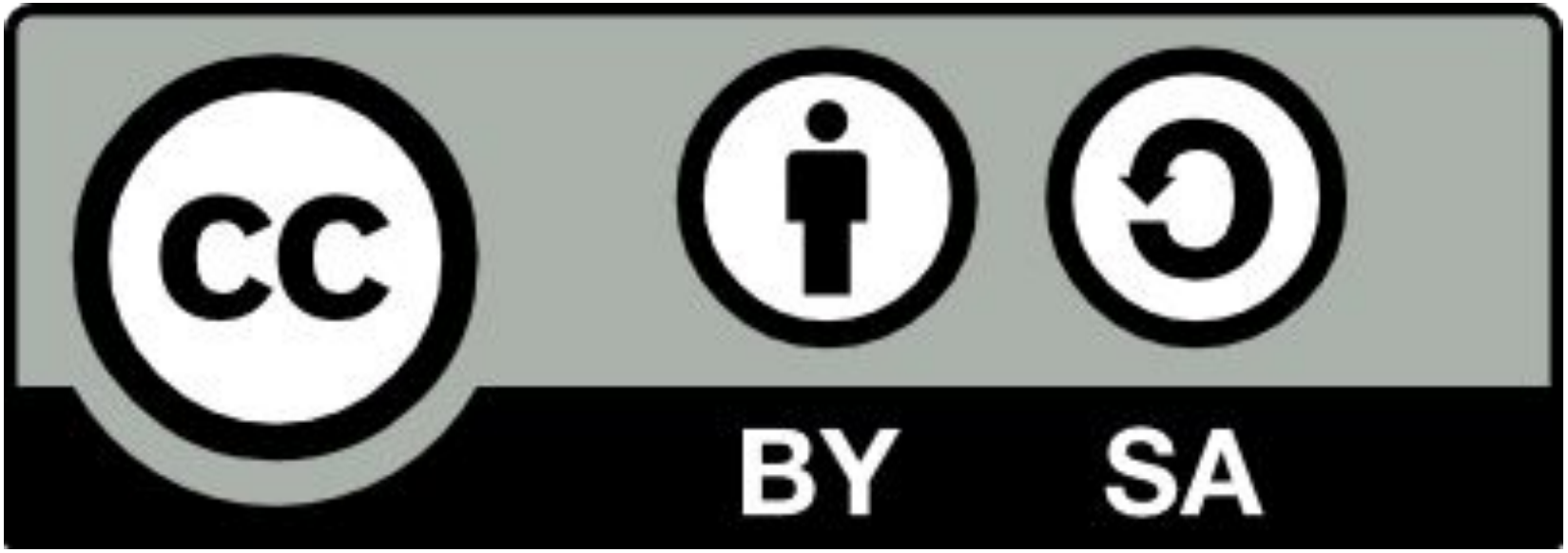
```
$ sudo apt install opam
$ opam init && sudo opam update && sudo opam upgrade
$ git clone https://github.com/herd/herdtools7 && cd herdtools7
$ git checkout 7.49
$ make all && make install
$ herd7 -version
7.49, Rev: 93dcbdd89086d5f3e981b280d437309fdeb8b427
```

Herd7 Based Userspace Litmus Tests Execution

```
$ herd7 -conf linux-kernel.cfg \
    litmus-tests/SB+fencembonceonces.litmus
Test SB+fencembonceonces Allowed
States 3
0:r0=0; 1:r0=1;
0:r0=1; 1:r0=0;
0:r0=1; 1:r0=1;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (0:r0=0 /\ 1:r0=0)
Observation SB+fencembonceonces Never 0 3
Time SB+fencembonceonces 0.01
Hash=d66d99523e2cac6b06e66f4c995ebb48
```

Klitmus7 Based Litmus Tests Execution

```
$ mkdir my; klitmus7 -o my/ \
    litmus-tests/SB+fencembonceonces.litmus
$ cd klitmus_test; make; sudo sh ./run.sh
...
Test SB+fencembonceonces Allowed
Histogram (3 states)
16580117:>0:r0=1; 1:r0=0;
16402936:>0:r0=0; 1:r0=1;
3016947 :>0:r0=1; 1:r0=1;
...
Positive: 0, Negative: 36000000
Condition exists (0:r0=0 /\ 1:r0=0) is NOT validated
Hash=d66d99523e2cac6b06e66f4c995ebb48
Observation SB+fencembonceonces Never 0 36000000
...
```



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/.