Understanding of Linux Kernel Memory Model

SeongJae Park <sj38.park@gmail.com>

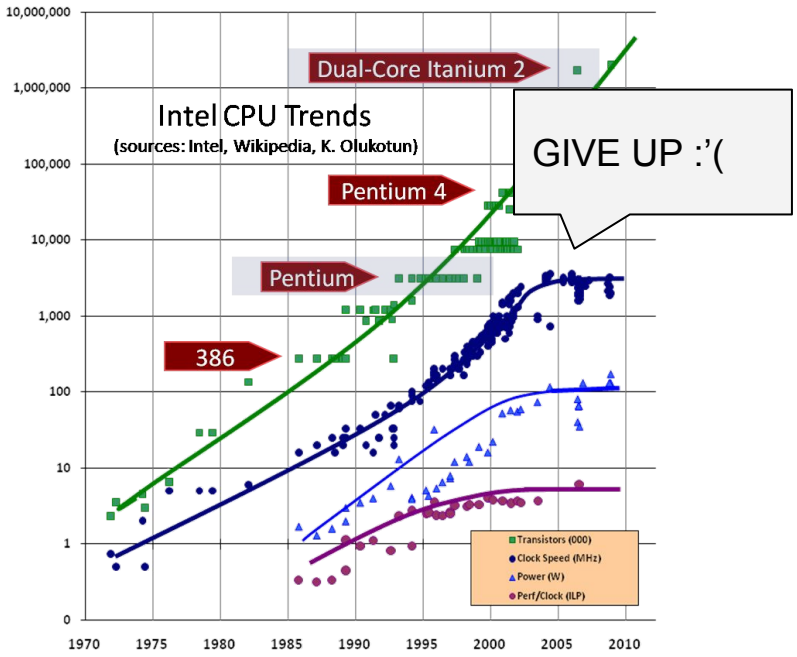
Great To Meet You

- SeongJae Park <sj38.park@gmail.com>
- Started contribution to Linux kernel just for fun since 2012
- Developing Guaranteed Contiguous Memory Allocator
  - Source code is available: https://lwn.net/Articles/634486/
- 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

https://www.linux.com/sites/lcom/files/styles/rendered_file/public/kernel-dev-2015.jpg?itok=9s3mv2Nc

Programmers in Multi-core Land

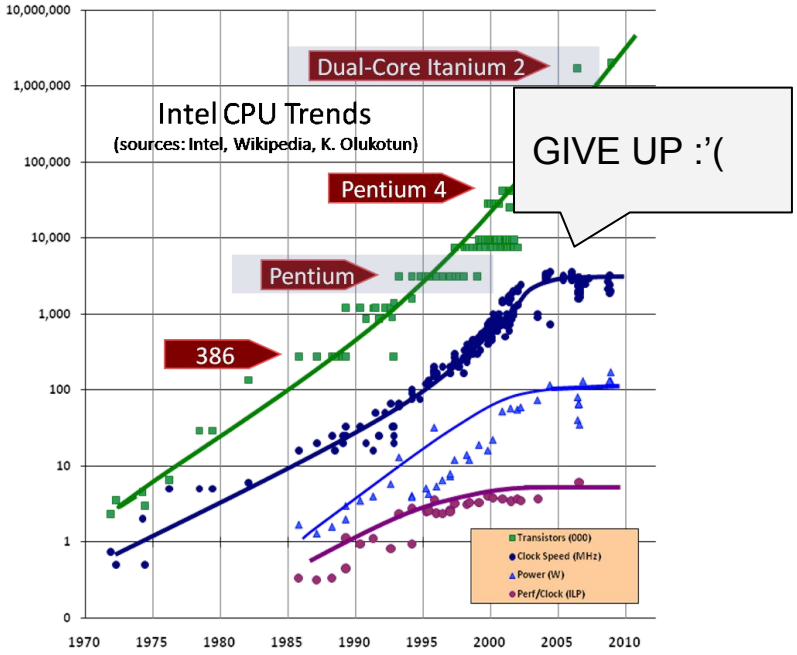
- Processor vendors changed their mind to increase number of cores instead of clock speed a decade ago
  - Now, multi-core system is prevalent
  - Even octa-core portable bomb in your pocket, maybe?



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

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



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

Writing Correct Parallel Program is Hard

- Compilers and processors care only Instructions Per Cycle, not programmer’s goal such as response time or throughput of meaningful progress

Writing Correct Parallel Program is Hard

- Compilers and processors care only Instructions Per Cycle, not programmer's goal such as response time or throughput of meaningful progress
- Nature of parallelism is counter-intuitive
  - Time is relative, before and after is ambiguous, even simultaneous available

CPU 0	CPU 1

`A = 1;`
`B = 1;`
`A = 2;`
`B = 2;`
	`assert(B == 2 && A == 1)`

CPU 1 assertion can be true on most parallel programming environments

Writing Correct Parallel Program is Hard

- Compilers and processors care only Instructions Per Cycle, not programmer's goal such as response time or throughput of meaningful 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

`A = 1;`
`B = 1;`
`A = 2;`
`B = 2;`
	`assert(B == 2 && A == 1)`

CPU 1 assertion can be true on most parallel programming environments

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

https://ih1.redbubble.net/image.218483193.6460/sticker,220x200-pad,220x200,ffffff.u2.jpg

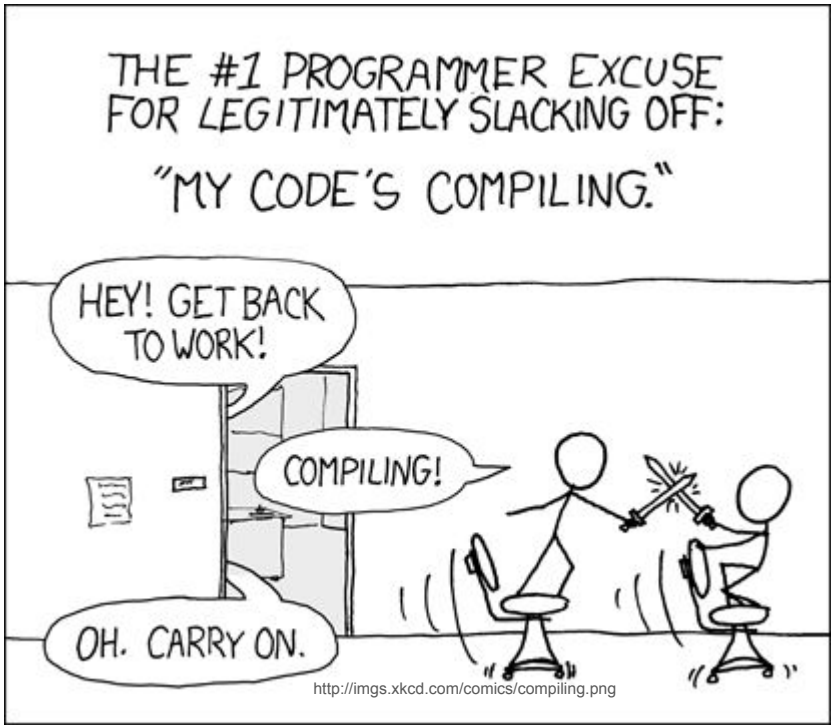
Reordering for Better IPC

Simple Program Execution Sequence

- Programmer writes program in C-like human readable language

Simple Program Execution Sequence

- Programmer writes program in C-like human readable language
- Compiler translates human readable code into assembly language



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

Simple Program Execution Sequence

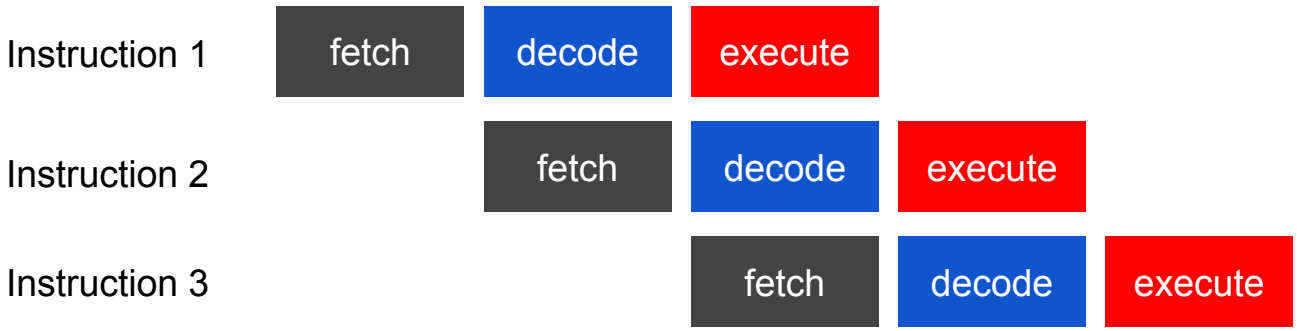
- Programmer writes program in C-like human readable language
- Compiler translates human readable code into assembly language
- Assembler generates executable binary from the assembly code
- Processor ***sequentially(?)*** executes instruction sequence in the binary

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 ***sequentially(?)*** executes instruction sequence in the binary
  - Actually, processor can execute instruction as it wants only if the result is same with sequential execution

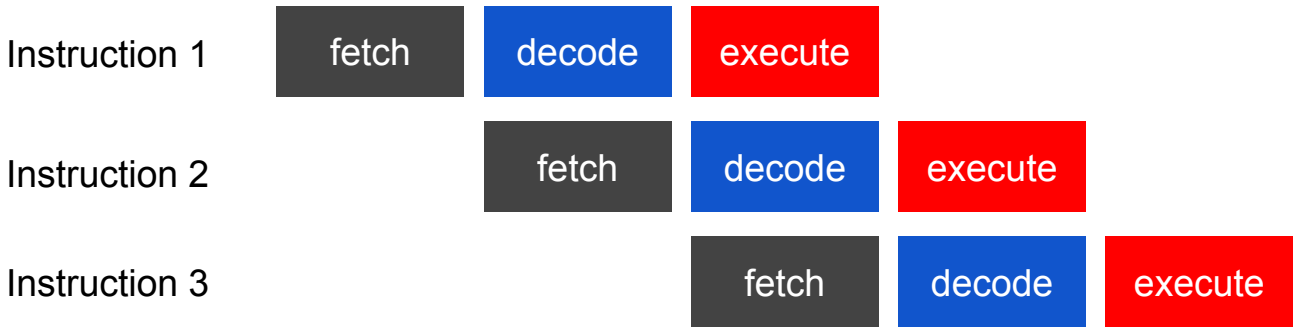
Instruction Level Parallelism (ILP)

- 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



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)

Dependent Instructions Harm ILP

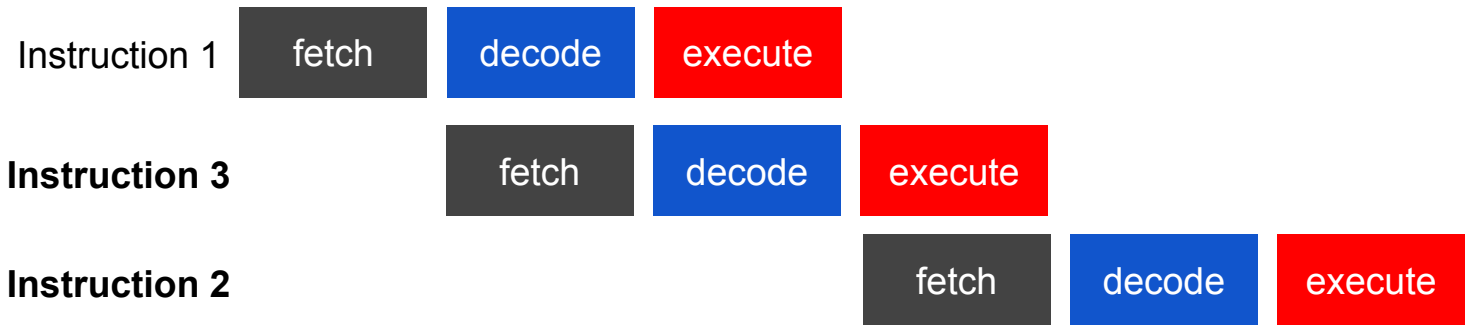
- If an instruction is dependent to result of previous instruction, it should wait until the previous one finishes execution
  - E.g., `a = b + c;`

```
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

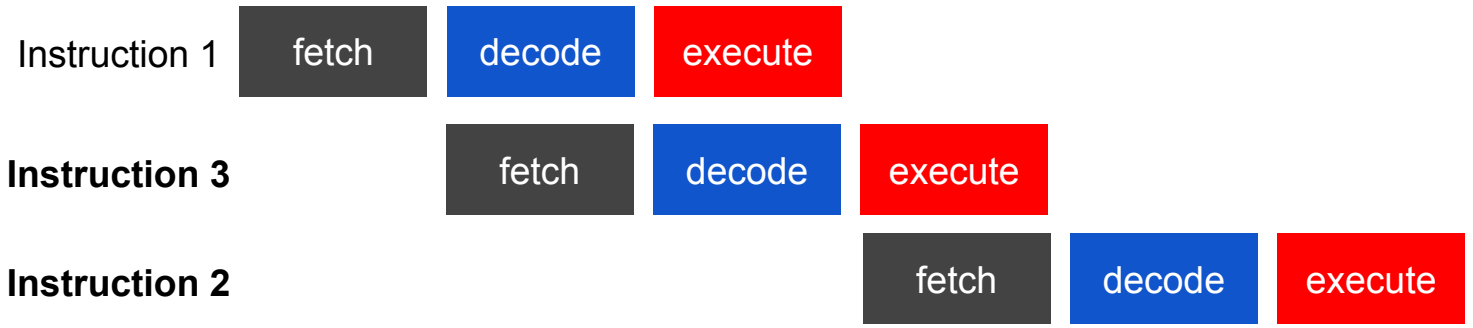
- By reordering dependent instructions to be located in far away, total execution time can be shorten
- If the reordering is guaranteed to not change the result of the instruction sequence, it would be helpful for better performance



instruction 2 depends on result of *instruction 1*
(e.g., first instruction modifies opcode of next instruction)

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**

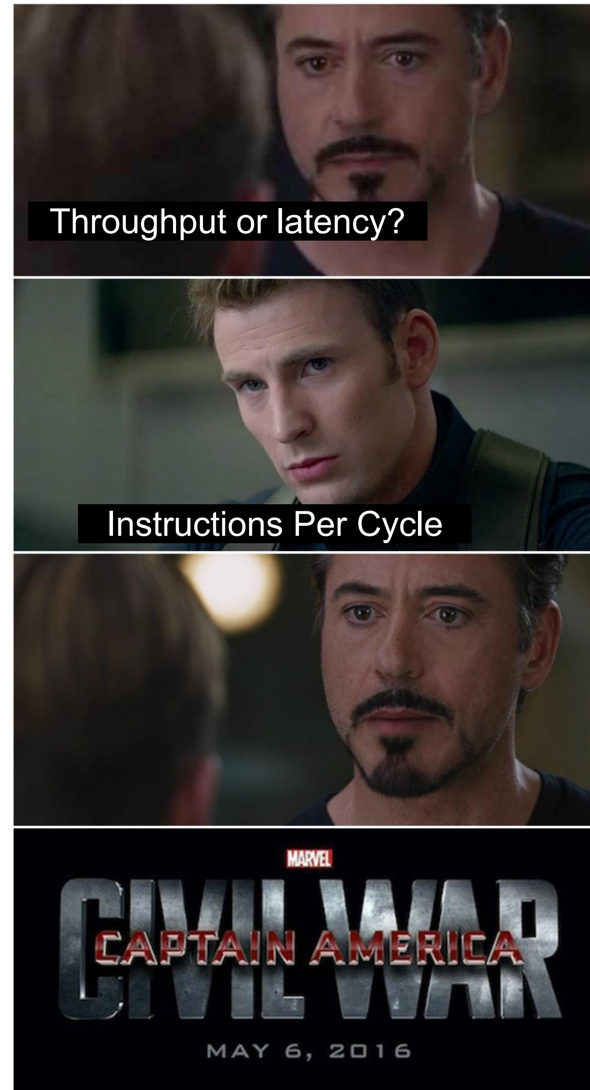
Reordering is Legal, Encouraged Behavior, But...

- If *program causality* is guaranteed, any reordering is legal
- Processors and compilers can make reordering of instructions for **better IPC**

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 unawares programmer perspective performance goal like throughput or latency
- On Multi-processor system, reordering can harm not only programmer perspective correctness, but also performance

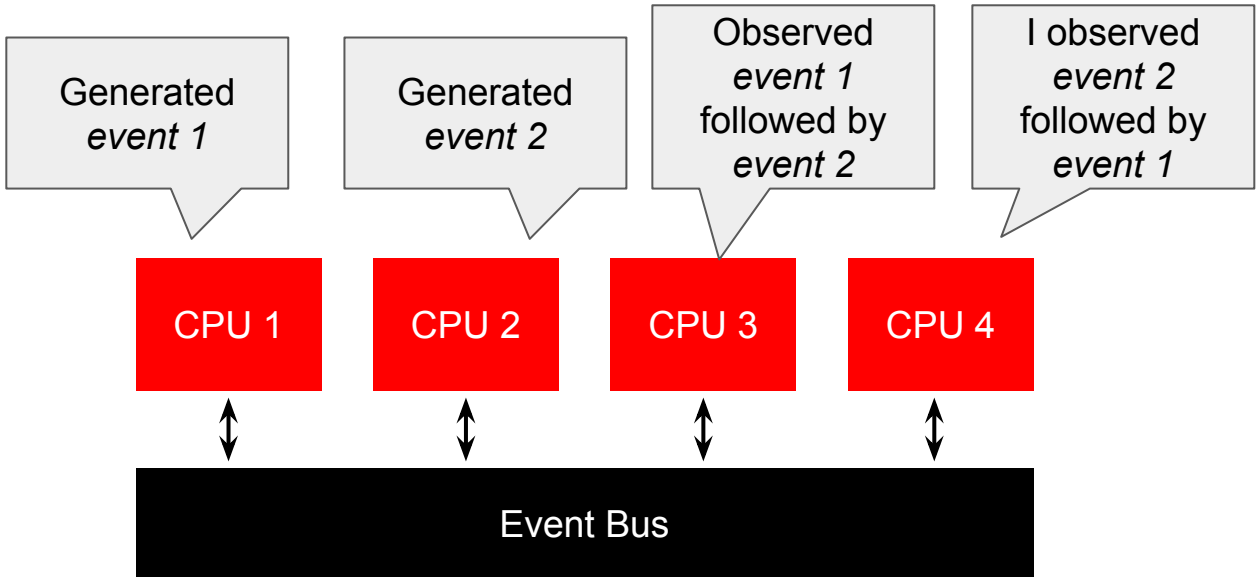


https://s-media-cache-ak0.pinimg.com/originals/c2/1a/00/c21a007e05421f7da57dacd15a86d478d.jpg

Counter-intuitive Nature of Parallelism

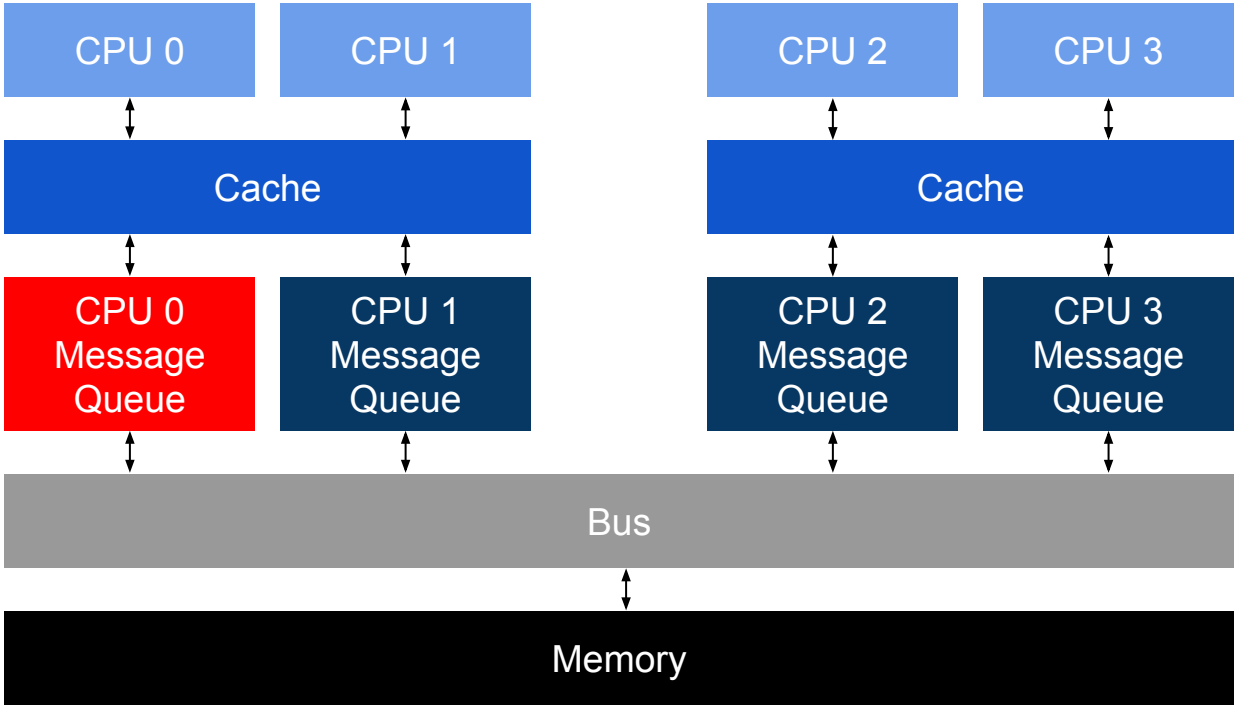
Time is Relative ($E = MC^2$)

- Each CPU generates their events in their time, observes effect 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

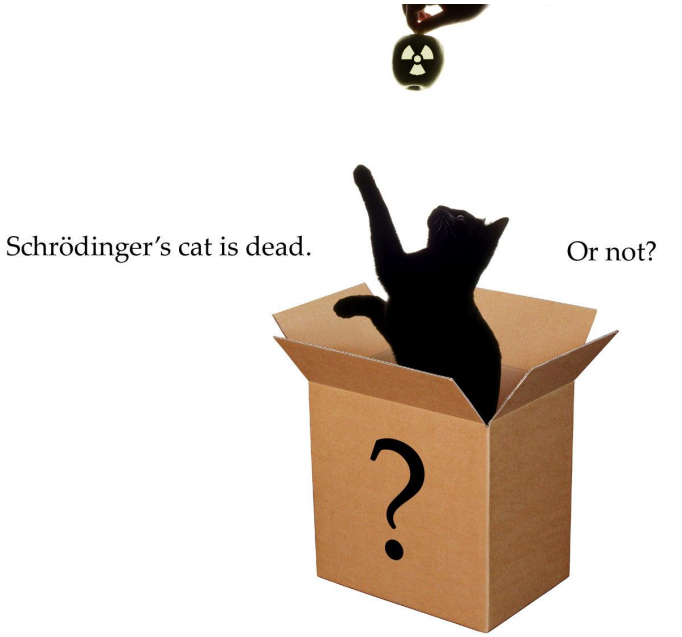
- Most systems equip hierarchical memory for better performance and space
- Propagation speed of an event to a given core can be influenced by specific sub-layer of memory



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*

Cache Coherency is Eventual

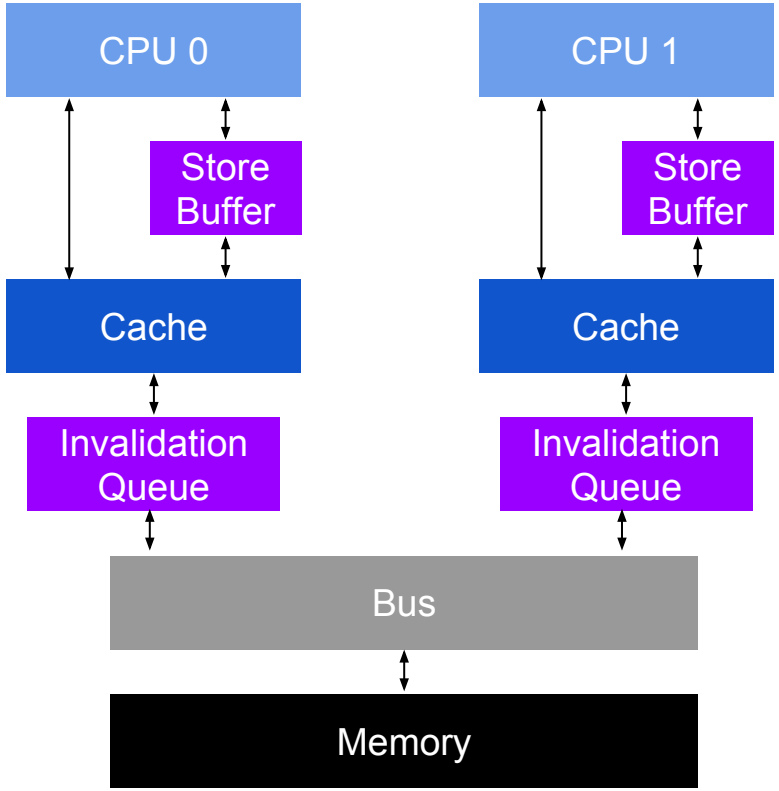
- It is well known that cache coherency protocol helps system memory consistency
- In actual, cache coherency guarantees eventual consistency only
- Every effect of each CPU will eventually become visible on all CPUs, but There's no guarantee that they will become apparent in the same order on those other CPUs



http://img06.deviantart.net/0663/i/2005/112/b/6/schrodinger__s_cat___2_by_firefoxcentral.jpg

System with Store Buffer and Invalidation Queue

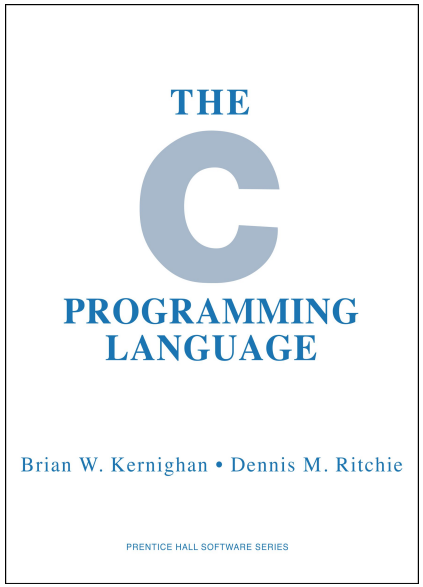
- Store Buffer and Invalidation Queue deliver effect of event but does not guarantee order of observation on each CPU



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
- Linux kernel uses compiler directive and volatile keyword to enforce memory ordering
- C11 has much more improvement, though

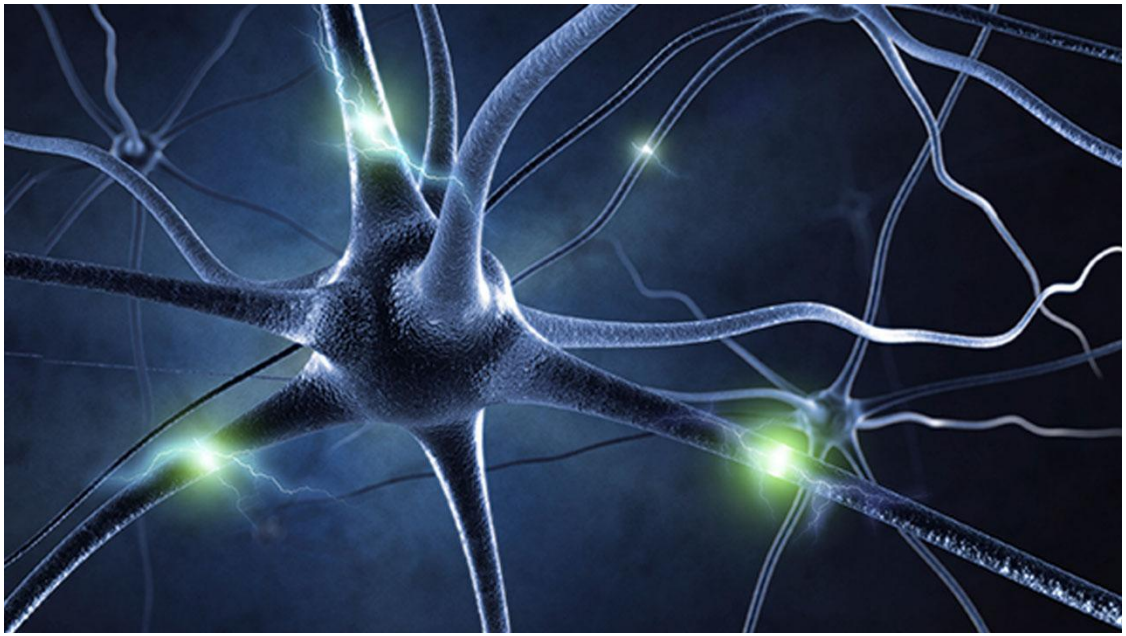


https://upload.wikimedia.org/wikipedia/commons/thumb/9/95/The_C_Programming_Language,_First_Edition_Cover_(2).svg/2000px-The_C_Programming_Language,_First_Edition_Cover_(2).svg.png

Memory Models

Each Environment Provides Own Memory Model

- Memory Model defines how memory operations are generated, what effects it makes, how their effects will be propagated
- Each programming environment like Instruction Set Architecture, Programming language, Operating system, etc defines own memory model
  - Most modern language memory models (e.g., Golang, Rust, …) aware multi-processor



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 CPU_S is strictest, Alpha is weakest
- Because Linux kernel supports multiple architectures, it defines its memory model based on weakest one, 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

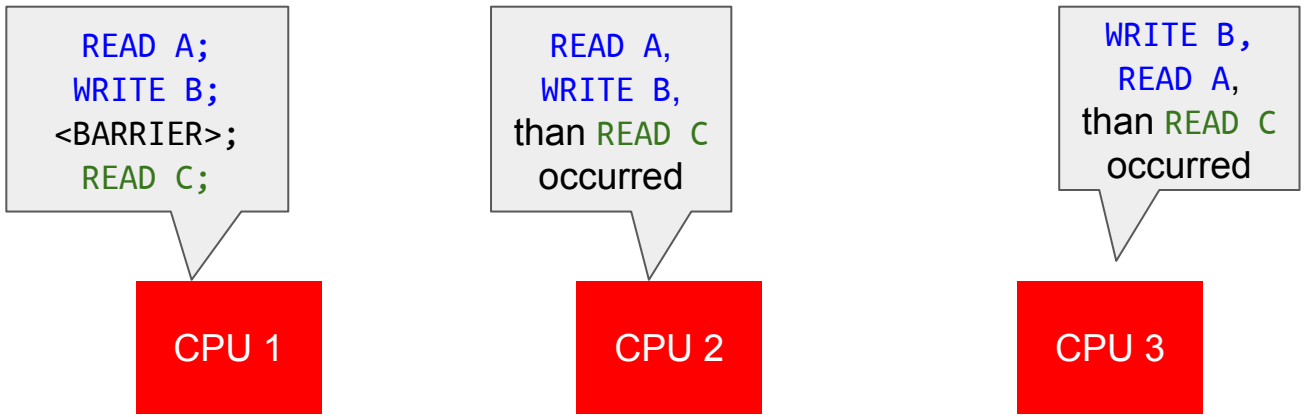
Synchronization Primitives

- Though reordering and asynchronous effect propagation is legal, synchronization primitives are necessary to write human intuitive program
- Most memory model provides synchronization primitives like atomic instructions, memory barrier, etc

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

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
- In general, memory barrier is expensive operation



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

Linux Kernel Memory Model

- Defined by weakest architecture, Alpha
  - Almost every combination of reordering is possible
- Provides rich set of atomic instructions
  - `atomix_xchg(), atomic_inc_return(), atomic_dec_return(), ...`
- Provides CPU level barriers, Compiler level barriers, semantic level barriers
  - Compiler barriers: `WRITE_ONCE(), READ_ONCE(), barrier(), ...`
  - CPU barriers: `mb(), wmb(), rmb(), smp_mb(), smp_wmb(), smp_rmb(), …`
  - Semantical barriers: `ACQUIRE` operations, `RELEASE` operations, …
  - For detail, refer to https://www.kernel.org/doc/Documentation/memory-barriers.txt
- Because different barrier has different overhead, only necessary barrier should be used in necessary case for high performance and scalability

Case Studies

Memory Operation Reordering

- Memory Operation Reordering is totally LEGAL unless it breaks causality
- Both of CPU and Compiler can do it, even in Single Processor

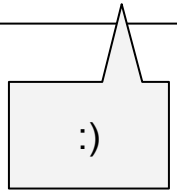
CPU 0	CPU 1	CPU 2

`A = 1;`
`B = 1;`
	`while (B == 0) {}`
	`  C = 1;`
		`Z = C;`
		`X = A;`
		`assert(z == 0 || x == 1)`

Memory Operation Reordering

- Memory Operation Reordering is totally LEGAL unless it breaks causality
- Both of CPU and Compiler can do it, even in Single Processor

CPU 0	CPU 1	CPU 2
`A = 1;` `B = 1;`		`Z = C;`
	`while (B == 0) {}` `C = 1;`	`X = A;` `assert(z == 0 || x == 1)`

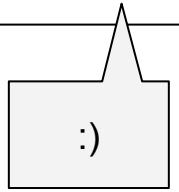


Memory Operation Reordering

- Memory Operation Reordering is totally LEGAL unless it breaks causality
- Both of CPU and Compiler can do it, even in Single Processor

CPU 0	CPU 1	CPU 2

`A = 1;`
`B = 1;`
	`while (B == 1) {}`
	`C = 1;`
		`Z = C;`
		`X = A;`
		`assert(z == 0 || x == 1)`



Memory Operation Reordering

- Memory Operation Reordering is totally LEGAL unless it breaks causality
- Both of CPU and Compiler can do it, even in Single Processor

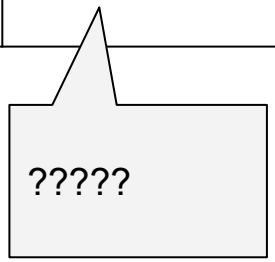
CPU 0	CPU 1	CPU 2

`B = 1;`
 	
`A = 1;`
	`while (B == 0) {}`
	`C = 1;`
		`Z = C;`
		`X = A;`
		`assert(z == 0 || x == 1)`

Memory Operation Reordering

- Memory Operation Reordering is totally LEGAL unless it breaks causality
- Both of CPU and Compiler can do it, even in Single Processor

CPU 0	CPU 1	CPU 2
`B = 1;`		`X = A;`
	`while (B == 0) {}`	`Z = C;`
	`C = 1;`	
`A = 1;`		`assert(z == 0 || x == 1)`



Memory Operation Reordering

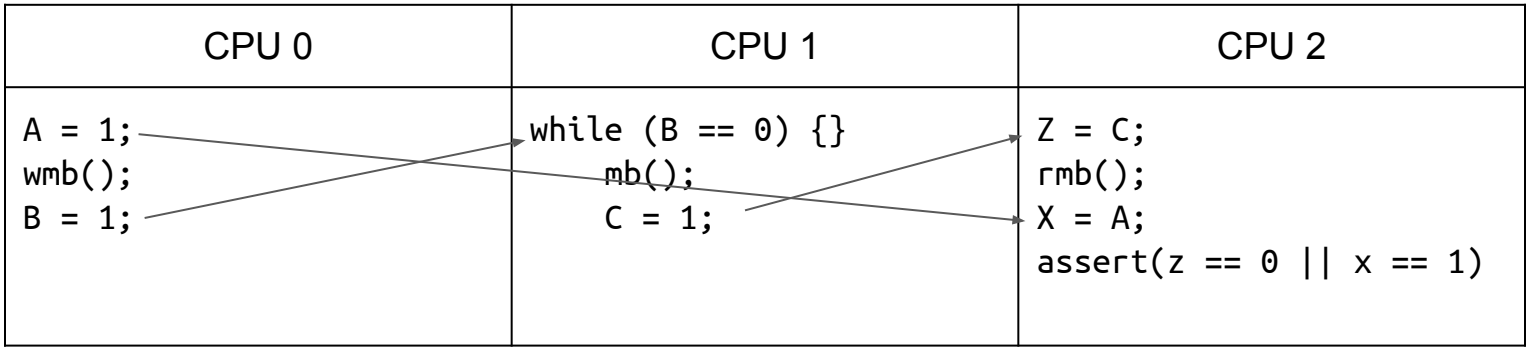
- Memory Operation Reordering is totally LEGAL unless it breaks causality
- Both of CPU and Compiler can do it, even in Single Processor
- Memory barrier enforces operations specified before it appear as happened to operations specified after it

CPU 0	CPU 1	CPU 2

`A = 1;`
`wmb();`
`B = 1;`
	`while (B == 0) {}`
	`mb();`
	`C = 1;`
		`Z = C;`
		`rmb();`
		`X = A;`
		`assert(z == 0 || x == 1)`

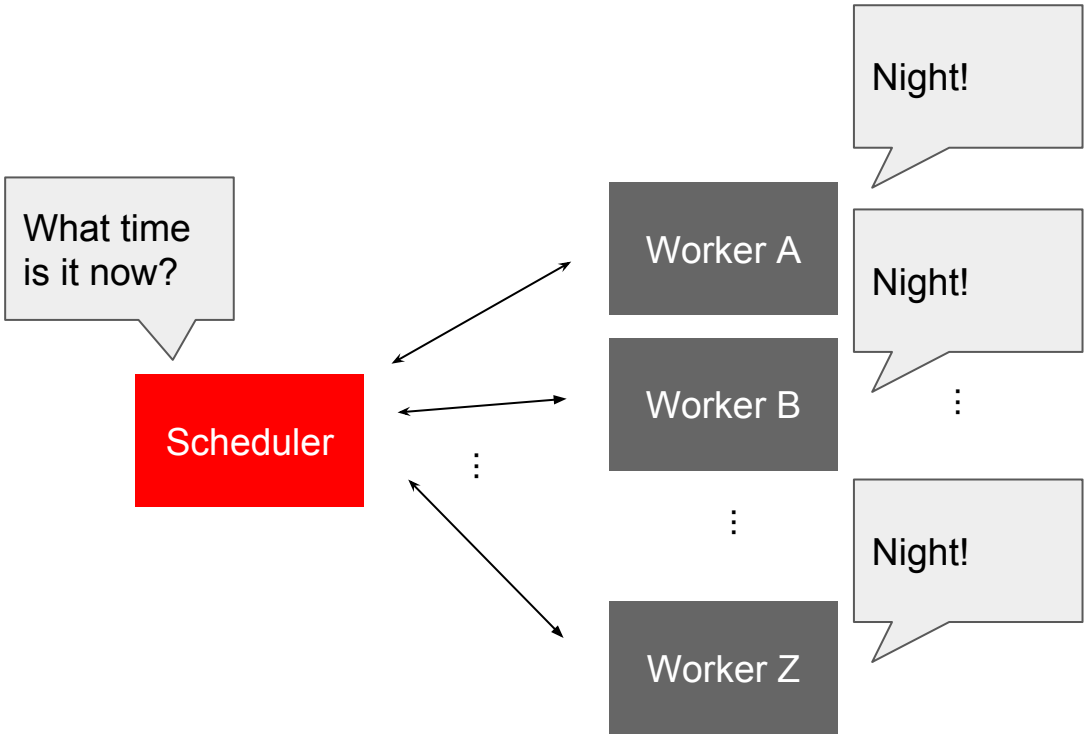
Memory Operation Reordering

- Memory Operation Reordering is totally LEGAL unless it breaks causality
- Both of CPU and Compiler can do it, even in Single Processor
- Memory barrier enforces operations specified before it appear as happened to operations specified after it
- In some architecture, even Transitivity is not guaranteed
  - Transitivity: B happened after A; C happened after B; then C happened after A



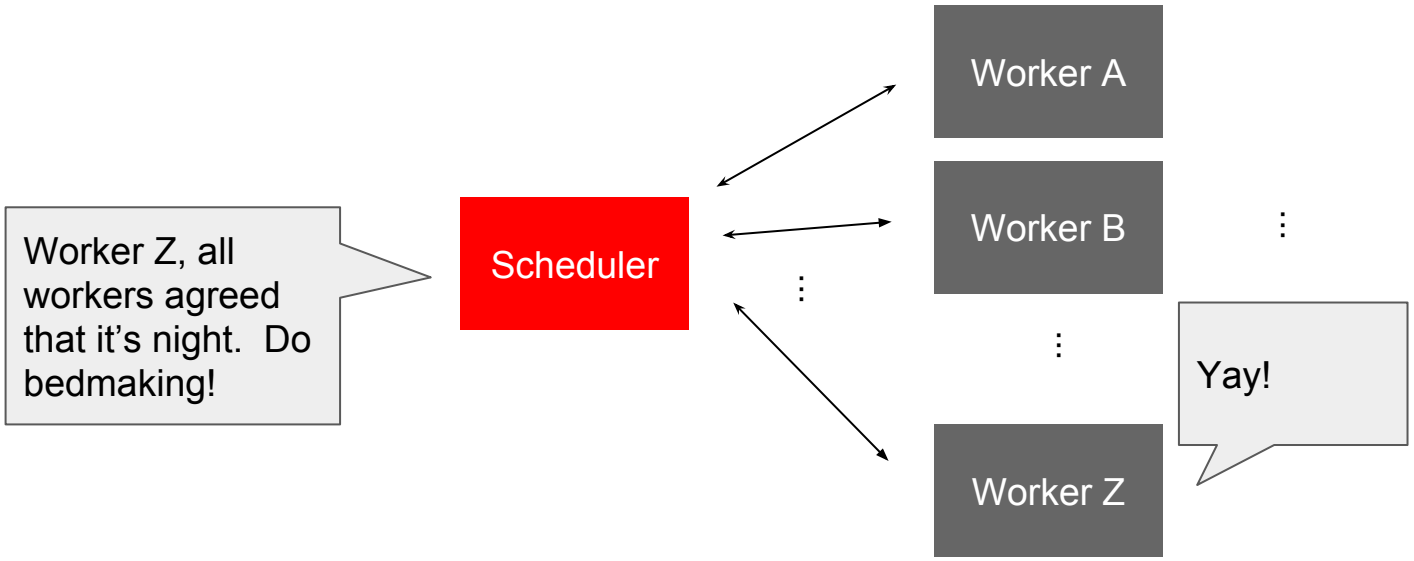
Transitivity for Scheduler and Workers

Scheduler and each workers made consensus about order



Transitivity between Scheduler and Worker

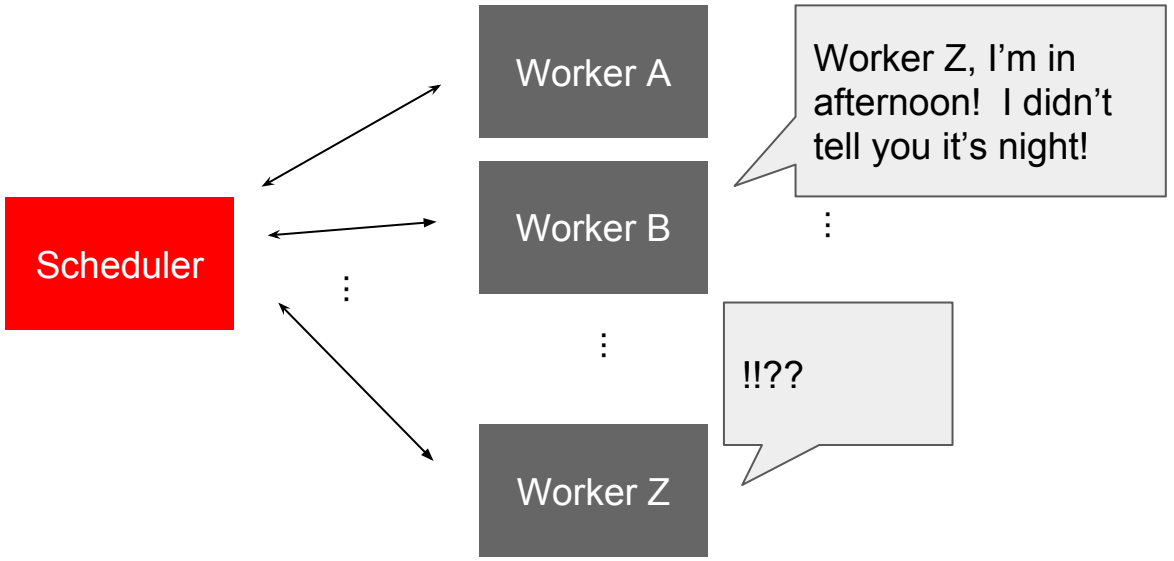
Scheduler and each workers made consensus about order



Transitivity between Scheduler and Worker

Scheduler and each workers made consensus about order

But, worker B and worker Z didn’t made consensus



Code Examples

Compiler Reordering Avoidance

Compiler can remove loop entirely

C code	Assembly language code

`static int the_var;`
`void loop(void)`
`{`
`  int i;`
`  for (i = 0; i < 1000; i++)`
`  the_var++;`
`}`
	`loop:`
	`.LFB106:`
	`  .cfi_startproc`
	`  addl  $1000, the_var(%rip)`
	`  ret`
	`  .cfi_endproc`
	`.LFE106:`

Compiler Reordering Avoidance

Store to `the_var` could not be seen by others

C code	Assembly language code

`static int the_var;`
`void loop(void)`
`{`
`  int i;`
`  for (i = 0; ACCESS_ONCE(i) < 1000; i++)`
`  the_var++;`
`}`
	`loop:`
	`  ...`
	`  movl  the_var(%rip), %ecx`
	`.L175:`
	`  ...`
	`  addl  $1, %eax`
	`  ...`
	`  cmpl  $999, %edx`
	`  jle  .L175`
	`  movl  %esi, the_var(%rip)`
	`.L170:`
	`  rep ret`

Compiler Reordering Avoidance

Still, store to `the_var` not issued for every iteration

C code	Assembly language code

`static int the_var;`
`void loop(void)`
`{`
`  int i;`
`  for (i = 0; ACCESS_ONCE(i) < 1000; i++)`
`  the_var++;`
`}`
	`loop:`
	`  ...`
	`  movl  the_var(%rip), %ecx`
	`.L175:`
	`  ...`
	`  addl  $1, %eax`
	`  ...`
	`  cmpl  $999, %edx`
	`  jle  .L175`
	`  movl  %esi, the_var(%rip)`
	`.L170:`
	`  rep ret`

Compiler Reordering Avoidance

- Volatile enforce compiler to issue memory operation (Note that it is not enforced to do access DRAM)
- However, repetitive LOAD may harm performance

C code

```
static volatile int the_var;
void loop(void)
{
        int i;
        for (i = 0; ACCESS_ONCE(i) < 1000; i++)
                the_var++;
}
```

Assembly language code

```
loop:
        ...
.L174:
        movl    the_var(%rip), %edx
        ...
        addl    $1, %edx
        movl    %edx, the_var(%rip)
        ...
        cmpl    $999, %edx
        jle     .L174
.L170:
        rep ret
        .cfi_endproc
```

Compiler Reordering Avoidance

- Complete memory barrier can help the case
- Does LOAD once and uses register for loop condition check

C code	Assembly language code

`static int the_var;`
`void loop(void)`
`{`
`  int i;`
`  for (i = 0; i < 1000; i++)`
`  the_var++;`
`  barrier();`
`}`
	`loop:`
	`.LFB106:`
	`  ...`
	`.L172:`
	`  addl  $1, the_var(%rip)`
	`  subl  $1, %eax`
	`  jne  .L172`
	`  rep ret`
	`  .cfi_endproc`

Progress perception

- Code does issue LOAD and STORE, but…
- `see_progress()` sees no progress because change made by a processor propagates to other processor eventually, not immediately

C code

```
static int prgrs;
void do_progress(void)
{
        prgrs++;
}

void see_progress(void)
{
        static int last_prgrs;
        static int seen;
        static int nr_seen;
        seen = prgrs;
        if (seen > last_prgrs)
                nr_seen++;
        last_prgrs = seen;
}
```

Assembly language code

```
do_progress:
        ...
        addl    $1, prgrs(%rip)
        ret
        ...
see_progress:
        ...
        movl    prgrs(%rip), %eax
        ...
        jle     .L193
        addl    $1, nr_seen.5542(%rip)
.L193:
        movl    %eax, last_prgrs.5540(%rip)
        ret
        .cfi_endproc
```

Progress perception

- Read barrier and write barrier helps the situation

C code

```
static int prgrs;
void do_progress(void)
{
        prgrs++;
        smp_wmb();
}

void see_progress(void)
{
        static int last_prgrs;
        static int seen;
        static int nr_seen;
        smp_rmb();
        seen = prgrs;
        if (seen > last_prgrs)
                nr_seen++;
        last_prgrs = seen;
}
```

Assembly language code

```
do_progress:
        ...
        addl    $1, prgrs(%rip)
        ...
        sfence
        ret
see_progress:
        ...
        lfence
        ...
        movl    prgrs(%rip), %eax
        ...
        jle     .L193
        addl    $1, nr_seen.5542(%rip)
.L193:
        movl    %eax, last_prgrs.5540(%rip)
```

Memory Ordering of X86

Neither Loads Nor Stores Are Reordered with Likes

CPU 0	CPU 1
`STORE 1 X` `STORE 1 Y`	`R1 = LOAD Y` `R2 = LOAD X`
`R1 == 1 && R2 == 0` impossible	

Stores Are Not Reordered With Earlier Loads

CPU 0	CPU 1
`R1 = LOAD X` `STORE 1 Y`	`R2 = LOAD Y` `STORE 1 X`
`R1 == 1 && R2 == 1` impossible	

Loads May Be Reordered with Earlier Stores to Different Locations

CPU 0	CPU 1
`STORE 1 X` `R1 = LOAD Y`	`STORE 1 Y` `R2 = LOAD X`
`R1 == 0 && R2 == 0` possible	

Intra-Processor Forwarding Is Allowed

CPU 0	CPU 1

`STORE 1 X`
`R1 = LOAD X`
`R2 = LOAD Y`
	`STORE 1 Y`
	`R3 = LOAD Y`
	`R4 = LOAD X`

`R2 == 0 && R4 == 0` possible	

Stores Are Transitively Visible

CPU 0	CPU 1	CPU 2
`STORE 1 X`	`R1 = LOAD X` `STORE 1 Y`	`R2 = LOAD Y` `R3 = LOAD X`
`R1 == 1 && R2 == 1 && R3 == 0` impossible		

Stores Are Seen in a Consistent Order by Others

CPU 0	CPU 1	CPU 2	CPU 3
`STORE 1 X`	`STORE 1 Y`	`R1 = LOAD X` `R2 = LOAD Y`	`R3 = LOAD Y` `R4 = LOAD X`
`R1 == 0 && R2 == 0 && R3 == 1 && R4 == 0` impossible			

X86 Memory Ordering Summary

- LOAD after LOAD never reordered
- STORE after STORE never reordered
- STORE after LOAD never reordered
- STOREs are transitively visible
- STOREs are seen in consistent order by others
- Intra-processor STORE forwarding is possible
- LOAD from different location after STORE may be reordered

- In short, quite reasonably strong enough
- For more detail, refer to `Intel Architecture Software Developer's Manual`

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
  - Every memory model provides synchronization primitives like atomic instruction and memory barrier, etc
  - Such interaction is expensive in common
- Linux kernel memory model is based on weakest memory model, Alpha
  - Kernel programmers should assume Alpha when writing architecture independent code
  - Because of the expensive cost of synchronization primitives, programmer should use only necessary primitives on necessary location

Thanks