Memory Models: A Gentle Introduction

Alex Khosrowshahi

As operating systems developers, we like to think of ourselves as the brave Haggardian heroes of computer science, striking out into the wilds and delivering the world clean abstractions. Yet, even in the uncharted territory of the lower registers, we make quite a few assumptions about how our code *ought* to run.

Indeed, things are not so simple. While we struck out into the vast ocean of multiprocessing, syscalls, and memory management, those dastardly hardware so-called 'engineers' were plotting. Now, we have to 'reason about concurrency' and 'think about the underlying hardware's instruction reordering mechanisms'—bah![1]

This document will discuss the concept of *memory consistency models,* often called **memory models**, which allow us to understand how reads and writes from code propagate to memory and caches. Despite our operating system being single-cored, you'll find this a topic of considerable depth.

What We Mean By Consistency: Memory Models

A *memory model* is a contract that defines guarantees an architecture (as in, the architecture of your CPU) provides to the developer writing code targeting their architecture regarding reads and writes to memory. A memory model can be provided both by *software* and *hardware;* [2] for example, a language supporting `async` may provide a developer with a different contract around when writes happen, but for now, we will assume we are writing in some assembly-or-equivalent language without such features.

So, why do we need a memory model? Well, code, as we write it, is generally assumed to execute in the order it is written in—at least, as programmers, we'd like to be able to think of it in this way. There are, of course, complexities here: compilers can reorder operations, processors can execute out-of-order in their cores, and multi-core threading is its own deal. That third case is actually the principal reason memory models exist, as the introduction of multi-core systems majorly messed with the mental models developers had been operating with for years! That said, you'll find that memory models can still possess some quirks on single-core processors.

So, we have an ideal memory model; we'll call it *sequential consistency*, where the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [3]. How do we know if a given hardware memory model satisfies this ideal?

Litmus Tests

A *litmus test* is a small program paired with a question about its possible outcomes. We specify the initial state, the operations performed by each participant, and an outcome whose possibility we want to investigate. The central question of a litmus test is, *"Can this program produce this outcome?"*

Here's an example of one such program—we'll assume we're running a single core, though how that core's instructions interact with memory may differ depending on architecture.

Test 1: Can the Processor Surprise Itself?

Initially, `x = 0`

[1] For more humorous resentment of hardware architects, see James Mickens' "The Slow Winter." [1]

```
CPU:
    STORE x, 1
    r = LOAD x
```

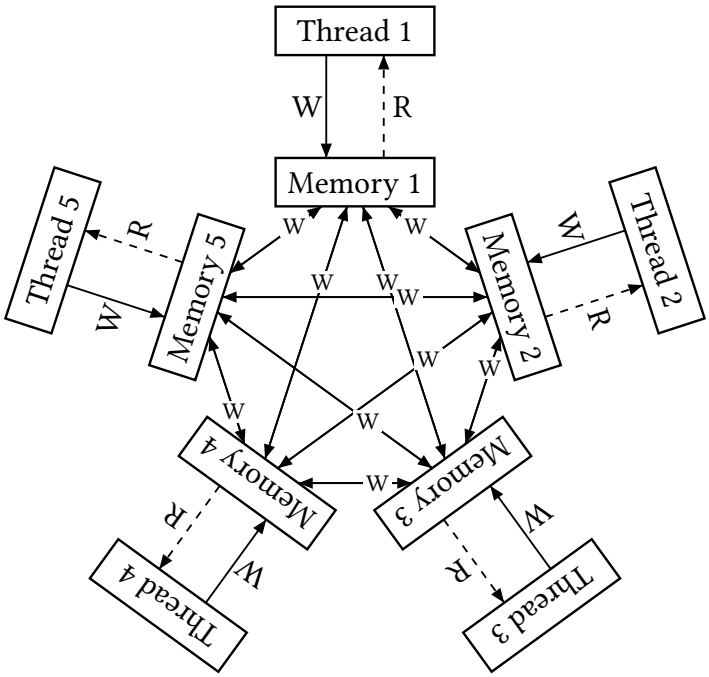
Question: Can `r == 0`?

Answer: No. ARM's memory model says that even if a write has not yet propagated to memory, its CPU-local behavior guarantees the same location read has the new value *before* memory is updated.

And just like that, you've learned something about ARM! Let's talk about it.

Arm's Memory Model

ARM employs a so-called 'relaxed' memory model. In the context of single-core devices, this doesn't mean much, but we'll discuss it regardless: on ARM, each core writes to its own copy of memory. Any writes to memory by a core to a location are then consistent with itself, but propagating those operations to different cores happens in a relaxed fashion.



But we are single-cored developers (for now) and Russ Cox already wrote an explanation miles better than this one[2], so when *we* discuss the specifics of ARM's relaxed memory model, we are talking not about interactions between *cores*, but between *a core* and *the rest of the computer*.

Single Core Woes

Consider a driver that prepares a buffer in RAM and then writes a peripheral register to tell a device to read it:

```
STORE buffer, new_data
STORE DEVICE_START, 1
```

From a language level, this seems quite simple: We write to the register and tell a device to read it. From the device's perspective, though, we make a promise when we write this register: the data must be present and available for it to read.

[2]Seriously, check out Russ' writing if you're considering doing multicore projects. [4]

To understand why this is a problem, consider three factors: CPU instructions *take time*, devices may require *explicit ordering*, and processors often employ *caching*. All of these can get us in various pickles without multicore (though multicore can get us in some particularly entangled pickles).

Thus, we need to make a promise to our devices in some way about the way we handle memory: to do this, we use *memory attributes*. The specific ones we're concerned with are **normal memory** and **device memory.**

Normal Memory
Normal memory is intended for locations that behave like ordinary storage: RAM, for example. Normal memory access has no side-effects; reading a location retrieves a value without consuming an item, acknowledging an interrupt, or commanding a peripheral to do something. [5]

This gives the processor considerable freedom, since it need not obey any other contract than "reading at (x) place gives access to (y) value." Subject to the architecture's rules, it can speculate reads, combine accesses, and reorder independent accesses while preserving the required behavior of our program externally.[3] We will treat our Normal memory as **write-back cacheable**, which means writes will immediately appear in a core's view but may not be immediately persisted to memory such that other cores or observers will see the change.

There are some cases in which ordering needs to be enforced between accesses to Normal memory, or accesses to Normal and Device memory. We'll discuss how to deal with these later.

Device Memory[4]
Device memory, in contrast to Normal memory, can have side-effects and thus must obey certain access semantics. While we may define different semantics for different device types, the baseline is that device memory is *never cacheable*. Speculative data accesses are also not permitted to Device memory, but instruction accesses may be.

When configuring Device memory, there are four valid sub-types:

1. **nGnRnE** - No gathering, no reordering, and no early write acknowledgment.
2. **nGnRE** - No gathering, no reordering, but early write acknowledgment is allowed.
3. **nGRE** - No gathering, but reordering and early write acknowledgment
4. **GRE** - gathering, reordering, and acknowledgment.

But what do all of these mean?

- **Gathering (G, nG)** - the processor may decide to merge multiple accesses to the same location into one access, or several smaller accesses into a larger access.
- **Re-ordering (R, nR)** - the processor may decide to re-order multiple accesses to a peripheral, similar to normal memory.
- **Early Write Acknowledgment (E, nE)** - Determines when a write is considered 'complete.' If allowed, a notification can be sent anywhere a write is on the bus, while if not allowed it must strictly have reached its destination.

You may ask: why not allow each of these options configure independently? Consider gathering and reordering—gathering is a strict relaxation on reordering. While reordering allows us to permute events and move their orders around, gathering changes the number and size of accesses, and in some sense necessitates the ability to *move* them as a result. Each of these permission sets acts as a

[3] We won't get into the mechanics of speculation and re-ordering too much, but if you'd like to learn more, take computer architecture (CSCI1952Y at Brown)!

[4] Much of this section is all-but stolen directly from the ARM developer manuals, with edits for clarity and engagement of the reader. [5]

relaxation of the previous, and none *necessitate* the use of all features—GRE memory can act as nGnRnE memory, only the converse is impossible. Also, note that non-cacheable Normal memory still has a distinction from device memory because it does not possess this gradual relaxation property set.

Test 2: Normal Memory or a Device?
Imagine a peripheral with a FIFO containing the values `A` and `B`, in that order. Each read pops from the FIFO, so it removes an item.

```
CPU:
    first  = LOAD FIFO
    second = LOAD FIFO
```

Question: What guarantees that these instructions produce two separate peripheral reads?

Answer: An appropriate Device mapping, such as `Device-nGnRnE`, provides the access semantics needed here. With the two reads correctly issued, `first = A` and `second = B`.

Configuring Attributes: MAIR
Memory types are encoded in a register called **MAIR**: the **M**emory **A**ttribute **I**ndirection **R**egister. MAIR is used to both configure the types of device memory and the cacheability settings of regions of normal memory, though in our OS we'll assume all normal memory is write-back cacheable in both domains.

MAIR is a 64-bit register with eight eight-bit slots. We won't go into their configuration details here, [6] but this essentially means that for each exception level, you have eight types of memory available to you.

Test 3: Where Is the New Value?
Initially, physical memory contains `buffer = 0`. Let's imagine we have a direct memory access (DMA) engine that does not snoop the cores of the CPU. Assume normal memory is, as we've discussed, write-back cacheable.

```
CPU:                              DMA engine:
    STORE buffer, 42
    start DMA ------------------> r = READ buffer
```

Question: Can the DMA engine read `0`?

Answer: If `buffer` is normal write-back cacheable memory, yes. The new value may remain dirty in a CPU cache while external memory still contains the old value [7].

Ordering, Completion, and Instruction Synchronization

ARM's Normal memory semantics, as we've discussed, are rather relaxed. That said, things get complicated when dealing with multiple cores and interactions between Normal memory and Device memory in the same process. To tighten the semantics of Normal memory *within a program,* ARM provides us with **barriers.** Barriers are instructions that force some form of ordering, synchronization, or restriction to memory accesses.

We'll consider three types of barriers: **Data Memory Barriers**, **Data Synchronization Barriers**, and **Instruction Synchronization Barriers**. [8]

DMB: Data Memory Barrier
The **Data Memory Barrier** (DMB) instruction prevents reordering of memory accesses across the barrier instruction. What this means is that, while a processor may decide to reorder two or more

sequential memory accesses, a DMB between those will ensure those instructions *after* the DMB instruction happen *strictly after* those before it to any observers.

In this example, we'll use a DMB to make sure a descriptor is published before we notify a device doorbell.

```
; Publish a descriptor before notifying a device.
; x0 = &descriptor; descriptor = 0; Normal non-cacheable.
; x1 = 42; x2 = &doorbell; w3 = 1.
; Doorbell is Device memory; device reads descriptor on notification.

str x1, [x0] ; Write descriptor.
dmb sy ; Order accesses system-wide.
str w3, [x2] ; Notify device.
```

If the DMB were not present, `str w3, [x2]` could be rearranged by hardware such that the device would be notified before the descriptor write.

DSB: Data Synchronization Barrier

The **Data Synchronization Barrier** (DSB) instruction ensures that all memory accesses before the barrier instruction have completed before those after. DSB is a strict strengthening of DMB, providing the same ordered execution with a guarantee of completion before subsequent execution[5].

In this example, we ensure a cacheable buffer is flushed to memory before a non-coherent device is notified.

```
; Publish a cacheable buffer to a non-coherent device.
; x0 = &buffer; buffer = 0; Normal write-back cacheable.
; Assume the buffer is eight bytes, aligned, and resides in one cache line.
; x1 = 42; x2 = &doorbell; w3 = 1.
; doorbell is Device memory.

str x1, [x0] ; Write buffer.
dc cvac, x0  ; Clean cache line
dsb sy ; Wait for clean completion.
str w3, [x2] ; Notify device.
```

This is a common pattern you will encounter when you turn caches on for your OS, especially when dealing with the frame buffer. Without proper data synchronization, artifacting will occur.

ISB: Instruction Synchronization Barrier

The **Instruction Synchronization Barrier** (ISB) instruction is a *context synchronization barrier.* What this means is that, while DMB and DSB can assure certain consistencies with memory access, ISB synchronizes a range of context-implicit events that are not necessarily memory reliant.

Context synchronization includes a *lot* of aspects, so we'll just borrow the list from the ARM manual:

- All unmasked interrupts that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- No instructions that appear in program order after an instruction that causes a Context synchronization event can perform any part of their functionality until the Context synchronization event has occurred.

[5]Decoding and fetching are free to continue, execution will just be delayed.

- All writes to System registers that are made before the Context synchronization event affect any instruction that appears in program order after the instruction that causes the Context synchronization event.
- All completed changes to the translation tables for entries that, before the change, were not permitted to be cached in a TLB, affect all instruction fetches that appear in program order after the instruction causing the Context synchronization event.
- All invalidations of TLBs, instruction caches, and, in AArch32 state, branch predictors, that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event. [8]

The following example uses ISB to enable floating-point access at EL1. Without the ISB, the fadd instruction after could operate in an old context and cause an error.

```
; Enable floating-point access at EL1.
; Initially CPACR_EL1.FPEN = 0b00.
; Higher exception levels permit access.
; d1 = 1.0; d2 = 2.0, initialized beforehand.

mrs x0, cpacr_el1 ; Read access controls.
orr x0, x0, #0x300000 ; Set FPEN to 0b11.
msr cpacr_el1, x0 ; Write access controls.
isb ; Synchronize instruction execution.
fadd d0, d1, d2 ; Add: d0 = 3.0.
```

Rules of Thumb

At this point you may be wondering what all this memory model information means for you. If you're in 1670, you'll never turn on caches, so you'll have much less housekeeping to do. If you are one of the poor souls who decided to take 1690 or have the misfortune of being a graduate student, you'll have a good bit more.

When do I need a data barrier?

Ask yourself: *who needs to observe these operations, and would observing them in a different order make my code incorrect?*

Data synchronization barriers are generally not needed in accesses to private Normal memory, since each core has its own view of memory. Where things get more complex is with device communication:

You may use data barriers for:

- **Preparing data before notifying a device:** establish the required ordering between the data writes and notification.
- **Ordering accesses to different peripherals:** do not assume that Device mappings alone establish the order your protocol requires.
- **Waiting for memory or maintenance operations to complete:** use the appropriate DSB when subsequent execution depends on that completion. A DMB supplies ordering without providing the same completion guarantee.

For the purposes of our class, we'll use `SY` to give all barriers full system scope, though narrower scopes can be useful.

Cleaning and Invalidating Data Caches

Before caching is enabled, you probably won't notice weirdness with talking to devices. That said, many of the devices you interact with are **cache non-coherent.** What this means is that they are not synchronized with the operations of your CPU cache. Your Raspberry Pi 3's framebuffer is one such non-coherent device. [9]

1. **Cleaning a cache line** writes back dirty data without necessarily destroying the underlying cached data.
2. **Invalidating a cache line** discards the contents of that cache line, necessitating a re-fetch.

It's useful to keep these in mind when interacting with non-coherent devices depending on their access semantics. For a non-coherent device:

- **If the device will read data written to RAM:** clean the buffer's cache lines to the Point of Coherency, then execute `DSB SY` before notifying the device to read. This ensures the data at the location in RAM is as current as possible.
- **If the device writes data into RAM:** wait for device completion, then invalidate the buffer's stale cache lines and complete the maintenance before reading it. This ensures you are not reading a stale value from the CPU's cached memory view. [7]

You have four non-coherent devices your OS will interact with:

1. The Raspberry Pi frame buffer.
2. The VideoCore GPU mailbox.
3. The UART and GPIO.
4. ARM local peripherals.

All of these will require maintenance according to the above rules when you turn caches on.

Cache Consistency

Unfortunately, we don't only have to worry about data caches—instruction caches and the TLB can also contain stale information. When you save something new to memory, ask yourself: *is something going to immediately operate on this, or is already mid-operation?* If so, you'll need to make sure *all* your caches agree. This is the context synchronization we discussed earlier.

Writing executable code: after loading or modifying instructions in cacheable memory:

1. Clean the affected data-cache lines to the Point of Unification.
2. Execute `DSB SY`.
3. Invalidate the affected instruction-cache lines.
4. Execute `DSB SY`, then `ISB`.
5. Execute the new code. [10]

Changing address translations: The TLB caches mappings for efficiency during translation. Whenever you change a mapping, then, you must make sure the TLB is properly invalidated AND the context is fully synchronized BEFORE new translations. [11]

This is part of your final quest for the VM assignment, so we won't give you the answer, but there is an instruction for TLB invalidation (aptly called `tlbi`) and each write to your translation should use data synchronization buffers.

References

[1] J. Mickens, "The Slow Winter," *;login:*, pp. 14–17, Sep. 2013, [Online]. Available: https://www.usenix.org/system/files/1309_14-17_mickens.pdf

[2] R. Cox, "Programming Language Memory Models." [Online]. Available: https://research.swtch.com/plmm

[3] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sep. 1979, [Online]. Available: https://www.microsoft.com/en-us/research/publication/make-multiprocessor-computer-correctly-executes-multiprocess-programs/

[4] R. Cox, "Hardware Memory Models." [Online]. Available: https://research.swtch.com/hwmm

[5] Arm Limited, "Armv8-A Memory Model." Apr. 01, 2019. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/Armv8-A%20memory%20model%20guide.pdf?revision=58b1dd0a-3800-4218-b21a-f95a0332034c

[6] Arm Limited, "MAIR_EL2: Memory Attribute Indirection Register (EL2)." Accessed: Nov. 18, 2026. [Online]. Available: https://support.arm.com/documentation/ddi0601/2024-06/AArch64-Registers/MAIR-EL2%E2%80%93Memory-Attribute-Indirection-Register%E2%80%93EL2-

[7] D. S. Miller, R. Henderson, and J. Jelinek, "Dynamic DMA Mapping Guide." Accessed: Nov. 18, 2026. [Online]. Available: https://docs.kernel.org/core-api/dma-api-howto.html

[8] Arm Limited, "Learn the Architecture: Memory Systems, Ordering, and Barriers." Jun. 10, 2022. [Online]. Available: https://documentation-service.arm.com/static/62a304f231ea212bb662321d

[9] dom, "Re: DMA Trouble." Accessed: Nov. 18, 2026. [Online]. Available: https://forums.raspberrypi.com/viewtopic.php?t=167244

[10] J. Bramley, "Caches and Self-Modifying Code: Implementing __clear_cache." [Online]. Available: https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/caches-self-modifying-code-implementing-clear-cache

[11] Arm Limited, "Learn the Architecture: AArch64 Memory Management Guide." 2026. [Online]. Available: https://documentation-service.arm.com/static/6ac508d3f59e042617c76666

[12] B. Simner, A. Armstrong, J. Pichon-Pharabod, C. Pulte, R. Grisenthwaite, and P. Sewell, "Relaxed Virtual Memory in Armv8-A," in *Programming Languages and Systems*, I. Sergey, Ed., in Lecture Notes in Computer Science, vol. 13240. Springer, 2022, pp. 143–173. doi: 10.1007/978-3-030-99336-8_6.