

WIKIPEDIA

The Free Encyclopedia

Main page

Contents

Featured content

Current events

Random article

Donate to Wikipedia

Wikipedia store

Interaction

Help

About Wikipedia

Community portal

Recent changes

Contact page

Tools

What links here

Related changes

Upload file

Special pages

Permanent link

Page information

Wikidata item

Cite this page

Print/export

Create a book

Download as PDF

Printable version

Languages


한국어

日本語

Polski

Русский

中文

 Edit links

Article

Talk

Read


Edit

View history



Memory barrier

From Wikipedia, the free encyclopedia



This article **needs additional citations for verification**. Please help [improve this article](#) by [adding citations to reliable sources](#). Unsourced material may be challenged and removed. *(January 2016)* ([Learn how and when to remove this template message](#))

A **memory barrier**, also known as a **membar**, **memory fence** or **fence instruction**, is a type of [barrier instruction](#) that causes a [central processing unit](#) (CPU) or [compiler](#) to enforce an [ordering](#) constraint on [memory](#) operations issued before and after the barrier instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.

Memory barriers are necessary because most modern CPUs employ performance optimizations that can result in [out-of-order execution](#). This reordering of memory operations (loads and stores) normally goes unnoticed within a single [thread of execution](#), but can cause unpredictable behaviour in [concurrent programs](#) and [device drivers](#) unless carefully controlled. The exact nature of an ordering constraint is hardware dependent and defined by the architecture's [memory ordering model](#). Some architectures provide multiple barriers for enforcing different ordering constraints.

Memory barriers are typically used when implementing low-level [machine code](#) that operates on memory shared by multiple devices. Such code includes [synchronization](#) primitives and [lock-free](#) data structures on [multiprocessor](#) systems, and device drivers that communicate with [computer hardware](#).

Contents [\[hide\]](#)

1

[Example](#)

2

[Low-level architecture-specific primitives](#)

3

[Multithreaded programming and memory visibility](#)

4

[Out-of-order execution versus compiler reordering optimizations](#)

5

[See also](#)

6

[References](#)

7

[External links](#)

Example [\[edit \]](#)

When a program runs on a single-CPU machine, the hardware performs the necessary

bookkeeping to ensure that the program executes as if all memory operations were performed in the order specified by the programmer (program order), so memory barriers are not necessary. However, when the memory is shared with multiple devices, such as other CPUs in a multiprocessor system, or [memory-mapped peripherals](#), out-of-order access may affect program behavior. For example, a second CPU may see memory changes made by the first CPU in a sequence which differs from program order.

The following two-processor program gives an example of how such out-of-order execution can affect program behavior:

Initially, memory locations `x` and `f` both hold the value `0`. The program running on processor #1 loops while the value of `f` is zero, then it prints the value of `x`. The program running on processor #2 stores the value `42` into `x` and then stores the value `1` into `f`. Pseudo-code for the two program fragments is shown below. The steps of the program correspond to individual processor instructions.

Processor #1:

```
while (f == 0);
// Memory fence required here
print x;
```

Processor #2:

```
x = 42;
// Memory fence required here
f = 1;
```

One might expect the print statement to always print the number "42"; however, if processor #2's store operations are executed out-of-order, it is possible for `f` to be updated *before* `x`, and the print statement might therefore print "0". Similarly, processor #1's load operations may be executed out-of-order and it is possible for `x` to be read *before* `f` is checked, and again the print statement might therefore print an unexpected value. For most programs neither of these situations is acceptable. A memory barrier can be inserted before processor #2's assignment to `f` to ensure that the new value of `x` is visible to other processors at or prior to the change in the value of `f`. Another can be inserted before processor #1's access to `x` to ensure the value of `x` is not read prior to seeing the change in the value of `f`.

For another illustrative example (a non-trivial one that arises in actual practice), see [double-checked locking](#).

Low-level architecture-specific primitives [\[edit \]](#)

Memory barriers are low-level primitives and part of an architecture's [memory model](#), which, like [instruction sets](#), vary considerably between architectures, so it is not appropriate to generalize about memory barrier behavior. The conventional wisdom is that using memory barriers correctly requires careful study of the architecture manuals for the hardware being programmed.

Some architectures, including the ubiquitous [x86/x64](#), provide several memory barrier instructions including an instruction sometimes called "full fence". A full fence ensures that all load and store operations prior to the fence will have been committed prior to any loads and stores issued following the fence. Other architectures, such as the [Itanium](#), provide separate "acquire" and "release" memory barriers which address the visibility of read-after-write operations from the point of view of a reader (sink) or writer (source) respectively. Some architectures provide separate memory barriers to control ordering between different combinations of system memory and [I/O](#) memory. When more than one memory barrier instruction is available it is important to consider that the cost of different instructions may vary considerably.

Multithreaded programming and memory visibility [[edit](#)]

See also: *[Memory model \(programming\)](#)*

Multithreaded programs usually use synchronization [primitives](#) provided by a high-level programming environment, such as [Java](#) and [.NET Framework](#), or an [application programming interface](#) (API) such as [POSIX Threads](#) or [Windows API](#). Synchronization Primitives such as [mutexes](#) and [semaphores](#) are provided to synchronize access to resources from parallel threads of execution. These primitives are usually implemented with the memory barriers required to provide the expected memory visibility [semantics](#). In such environments explicit use of memory barriers is not generally necessary.

Each API or programming environment in principle has its own high-level memory model that defines its memory visibility semantics. Although programmers do not usually need to use memory barriers in such high level environments, it is important to understand their memory visibility semantics, to the extent possible. Such understanding is not necessarily easy to achieve because memory visibility semantics are not always consistently specified or documented.

Just as programming language semantics are defined at a different [level of abstraction](#) than [machine language opcodes](#), a programming environment's memory model is defined at a different level of abstraction than that of a hardware memory model. It is important to understand this distinction and realize that there is not always a simple mapping between low-level hardware memory barrier semantics and the high-level memory visibility semantics of a particular programming environment. As a result, a particular platform's implementation of [POSIX Threads](#) may employ stronger barriers than required by the specification. Programs which take advantage of memory visibility as implemented rather than as specified may not be portable.

Out-of-order execution versus compiler reordering optimizations [[edit](#)]

Memory barrier instructions address reordering effects only at the hardware level. Compilers may also reorder instructions as part of the program optimization process. Although the effects on parallel program behavior can be similar in both cases, in general it is necessary to take separate measures to inhibit compiler reordering optimizations for data that may be shared by

multiple threads of execution. Note that such measures are usually necessary only for data which is not protected by synchronization primitives such as those discussed in the prior section.

In **C** and **C++**, the `volatile` keyword was intended to allow C and C++ programs to directly access **memory-mapped I/O**. Memory-mapped I/O generally requires that the reads and writes specified in source code happen in the exact order specified with no omissions. Omissions or reorderings of reads and writes by the compiler would break the communication between the program and the device accessed by memory-mapped I/O. A C or C++ compiler may not omit reads from and writes to volatile memory locations, nor may it reorder read/writes relative to other such actions for the same volatile location (variable). The keyword `volatile` *does not guarantee a memory barrier* to enforce cache-consistency. Therefore, the use of "volatile" alone is not sufficient to use a variable for inter-thread communication on all systems and processors.^[1]

The C and C++ standards prior to C11 and C++11 do not address multiple threads (or multiple processors),^[2] and as such, the usefulness of `volatile` depends on the compiler and hardware. Although `volatile` guarantees that the volatile reads and volatile writes will happen in the exact order specified in the source code, the compiler may generate code (or the CPU may re-order execution) such that a volatile read or write is reordered with regard to non-volatile reads or writes, thus limiting its usefulness as an inter-thread flag or mutex. Preventing such is compiler specific, but some compilers, like **gcc**, will not reorder operations around in-line assembly code with `volatile` and "memory" tags, like in: `asm volatile ("" : : : "memory") ;` (See more examples in **compiler memory barrier**). Moreover, it is not guaranteed that volatile reads and writes will be seen in the same order by other processors or cores due to caching, **cache coherence** protocol and relaxed memory ordering, meaning volatile variables alone may not even work as inter-thread flags or mutexes.

See also [edit]


- [Lock-free and wait-free algorithms](#)
- [Meltdown \(security vulnerability\)](#)

 [Computer programming portal](#)

References [edit]

- [^] [Volatile Considered Harmful - Linux Kernel Documentation](#)
- [^] Boehm, Hans (June 2005). *[Threads cannot be implemented as a library](#)*. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. Association for Computing Machinery. doi:10.1145/1065010.1065042.

External links [edit]

- [Memory Barriers: a Hardware View for Software Hackers](#) 
- [Microsoft Driver Development: Memory Barriers on Multiprocessor Architectures](#)
- [HP technical report HPL-2004-209: Threads Cannot be Implemented as a Library](#)
- [Linux kernel memory barrier issues on multiple types of CPUs](#)
- [Documentation on memory barriers in the Linux kernel](#)
- [Handling Memory Ordering in Multithreaded Applications with Oracle Solaris Studio 12](#)



Update 2: Part 1, Compiler Barriers

- Handling Memory Ordering in Multithreaded Applications with Oracle Solaris Studio 12
- Update 2: Part 2, Memory Barriers and Memory Fences

- User-space RCU: Memory-barrier menagerie

V • T • E		Linux kernel				[hide]
Organization	Kernel	Linux Foundation • Linux Mark Institute • Linus's Law • Tanenbaum–Torvalds debate • Tux • SCO issues • Linaro • GNU GPL v2 • menuconfig • Supported computer architectures • Kernel names • Criticism				
	Support	Developers (<i>The Linux Programming Interface</i> • kernel.org • LKML • Linux conferences) • Users (Linux User Group (LUG))				
Technical	Debugging	CRIU • ftrace • kdump • Linux kernel oops • SystemTap • BPF				
	Startup	vmlinux • System.map • dracut • initrd • initramfs				
	ABIs	Linux Standard Base • x32 ABI				
	APIs	Of userspace	FS, daemons	devfs • devpts • debugfs • procfs • sysfs • systemd (udev) • Kmscon		
			Wrapper libraries	C standard library (glibc • uClibc • Bionic (libhybris) • dietlibc • EGLIBC • klibc • musl • Newlib) • libcgroup • libdrm • libalsa • libevdev • libusb		
		Of kernel	System Call Interface	POSIX (ioctl • select • open • read • close • sync • ...) • Linux-only (futex • epoll • splice • dnotify • inotify • readahead • ...)		
			In-kernel	ALSA • Crypto API • DRM • kernfs • Memory barrier • New API • RCU • Video4Linux		
	Components	Kernel modules • BlueZ • cgroups • Console • bcache • Device mapper • dm-cache • dm-crypt • DRM • EDAC • evdev • Kernel same-page merging (KSM) • LIO • Framebuffer • LVM • KMS driver • Netfilter • Netlink • nftables • Network scheduler • perf • SLUB • zram • zswap Process and I/O schedulers : O(n) scheduler • O(1) scheduler • Completely Fair Scheduler (CFQ) • Brain Fuck Scheduler • Noop scheduler SCHED_DEADLINE Security Modules : AppArmor • Exec Shield • grsecurity (PaX) • seccomp • SELinux • Smack • Tomoyo Linux • Linux PAM Device drivers (802.11 • graphics) • Raw device initramfs • KernelCare • kexec • kGraft • kpatch • Ksplice				
	Variants	Mainline (Linux kernel • Linux-libre) • High-performance computing (INK • Compute Node Linux • SLURM) • Real-time computing (RTLinux • RTAI • Xenomai • Carrier Grade Linux) • MMU-less (µClinux • PSXLinux)				
		Virtualization	Hypervisor (KVM • Xen) • OS-level virtualization (Linux-VServer • Lguest • LXC • OpenVZ) • Other (L4Linux • ELinOS • User-mode Linux • MkLinux • coLinux)			
	Range of use	Desktop • Embedded • Gaming • Thin client: (LTSP • Thinstation) • Server: (LAMP • LYME-LYCE) • Devices				

Adoption	
Adopters	List of Linux adopters · GENIVI Alliance · Proprietary software for Linux
Category · Commons · Book · Wikiversity · Portal	

Categories: [Computer memory](#) | [Consistency models](#) | [Instruction processing](#)

This page was last edited on 2 February 2018, at 04:17.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) · [About Wikipedia](#) · [Disclaimers](#) · [Contact Wikipedia](#) · [Developers](#) · [Cookie statement](#) · [Mobile view](#)

[Enable previews](#)

