Linux Staging Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER ONE

BRIEF TUTORIAL ON CRC COMPUTATION

A CRC is a long-division remainder. You add the CRC to the message, and the whole thing (message+CRC) is a multiple of the given CRC polynomial. To check the CRC, you can either check that the CRC matches the recomputed value, *or* you can check that the remainder computed on the message+CRC is 0. This latter approach is used by a lot of hardware implementations, and is why so many protocols put the end-of-frame flag after the CRC.

It’s actually the same long division you learned in school, except that:

- We’re working in binary, so the digits are only 0 and 1, and
- When dividing polynomials, there are no carries. Rather than add and subtract, we just xor. Thus, we tend to get a bit sloppy about the difference between adding and subtracting.

Like all division, the remainder is always smaller than the divisor. To produce a 32-bit CRC, the divisor is actually a 33-bit CRC polynomial. Since it’s 33 bits long, bit 32 is always going to be set, so usually the CRC is written in hex with the most significant bit omitted. (If you’re familiar with the IEEE 754 floating-point format, it’s the same idea.)

Note that a CRC is computed over a string of *bits*, so you have to decide on the endianness of the bits within each byte. To get the best error-detecting properties, this should correspond to the order they’re actually sent. For example, standard RS-232 serial is little-endian; the most significant bit (sometimes used for parity) is sent last. And when appending a CRC word to a message, you should do it in the right order, matching the endianness.

Just like with ordinary division, you proceed one digit (bit) at a time. Each step of the division you take one more digit (bit) of the dividend and append it to the current remainder. Then you figure out the appropriate multiple of the divisor to subtract to being the remainder back into range. In binary, this is easy - it has to be either 0 or 1, and to make the XOR cancel, it’s just a copy of bit 32 of the remainder.

When computing a CRC, we don’t care about the quotient, so we can throw the quotient bit away, but subtract the appropriate multiple of the polynomial from the remainder and we’re back to where we started, ready to process the next bit.

A big-endian CRC written this way would be coded like:

```
for (i = 0; i < input_bits; i++) {
        multiple = remainder & 0x80000000 ? CRCPOLY : 0;
```

(continues on next page)

(continued from previous page)

```
        remainder = (remainder << 1 | next_input_bit()) ^ multiple;
}
```

Notice how, to get at bit 32 of the shifted remainder, we look at bit 31 of the remainder *before* shifting it.

But also notice how the next_input_bit() bits we’ re shifting into the remainder don’ t actually affect any decision-making until 32 bits later. Thus, the first 32 cycles of this are pretty boring. Also, to add the CRC to a message, we need a 32-bit-long hole for it at the end, so we have to add 32 extra cycles shifting in zeros at the end of every message.

These details lead to a standard trick: rearrange merging in the next_input_bit() until the moment it’ s needed. Then the first 32 cycles can be precomputed, and merging in the final 32 zero bits to make room for the CRC can be skipped entirely. This changes the code to:

```
for (i = 0; i < input_bits; i++) {
        remainder ^= next_input_bit() << 31;
        multiple = (remainder & 0x80000000) ? CRCPOLY : 0;
        remainder = (remainder << 1) ^ multiple;
}
```

With this optimization, the little-endian code is particularly simple:

```
for (i = 0; i < input_bits; i++) {
        remainder ^= next_input_bit();
        multiple = (remainder & 1) ? CRCPOLY : 0;
        remainder = (remainder >> 1) ^ multiple;
}
```

The most significant coefficient of the remainder polynomial is stored in the least significant bit of the binary “remainder” variable. The other details of endianness have been hidden in CRCPOLY (which must be bit-reversed) and next_input_bit().

As long as next_input_bit is returning the bits in a sensible order, we don’ t *have* to wait until the last possible moment to merge in additional bits. We can do it 8 bits at a time rather than 1 bit at a time:

```
for (i = 0; i < input_bytes; i++) {
        remainder ^= next_input_byte() << 24;
        for (j = 0; j < 8; j++) {
                multiple = (remainder & 0x80000000) ? CRCPOLY : 0;
                remainder = (remainder << 1) ^ multiple;
        }
}
```

Or in little-endian:

```
for (i = 0; i < input_bytes; i++) {
        remainder ^= next_input_byte();
        for (j = 0; j < 8; j++) {
```

(continues on next page)

(continued from previous page)

```
                multiple = (remainder & 1) ? CRCPOLY : 0;
                remainder = (remainder >> 1) ^ multiple;
        }
}
```

If the input is a multiple of 32 bits, you can even XOR in a 32-bit word at a time and increase the inner loop count to 32.

You can also mix and match the two loop styles, for example doing the bulk of a message byte-at-a-time and adding bit-at-a-time processing for any fractional bytes at the end.

To reduce the number of conditional branches, software commonly uses the byte-at-a-time table method, popularized by Dilip V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Look-Up", Comm. ACM v.31 no.8 (August 1998) p. 1008-1013.

Here, rather than just shifting one bit of the remainder to decide in the correct multiple to subtract, we can shift a byte at a time. This produces a 40-bit (rather than a 33-bit) intermediate remainder, and the correct multiple of the polynomial to subtract is found using a 256-entry lookup table indexed by the high 8 bits.

(The table entries are simply the CRC-32 of the given one-byte messages.)

When space is more constrained, smaller tables can be used, e.g. two 4-bit shifts followed by a lookup in a 16-entry table.

It is not practical to process much more than 8 bits at a time using this technique, because tables larger than 256 entries use too much memory and, more importantly, too much of the L1 cache.

To get higher software performance, a "slicing" technique can be used. See "High Octane CRC Generation with the Intel Slicing-by-8 Algorithm", ftp://download.intel.com/technology/comms/perfnet/download/slicing-by-8.pdf

This does not change the number of table lookups, but does increase the parallelism. With the classic Sarwate algorithm, each table lookup must be completed before the index of the next can be computed.

A "slicing by 2" technique would shift the remainder 16 bits at a time, producing a 48-bit intermediate remainder. Rather than doing a single lookup in a 65536-entry table, the two high bytes are looked up in two different 256-entry tables. Each contains the remainder required to cancel out the corresponding byte. The tables are different because the polynomials to cancel are different. One has non-zero coefficients from x^32 to x^39, while the other goes from x^40 to x^47.

Since modern processors can handle many parallel memory operations, this takes barely longer than a single table look-up and thus performs almost twice as fast as the basic Sarwate algorithm.

This can be extended to "slicing by 4" using 4 256-entry tables. Each step, 32 bits of data is fetched, XORed with the CRC, and the result broken into bytes and looked up in the tables. Because the 32-bit shift leaves the low-order bits of the intermediate remainder zero, the final CRC is simply the XOR of the 4 table look-ups.

But this still enforces sequential execution: a second group of table look-ups cannot begin until the previous groups 4 table look-ups have all been completed. Thus, the processor' s load/store unit is sometimes idle.

To make maximum use of the processor, "slicing by 8" performs 8 look-ups in parallel. Each step, the 32-bit CRC is shifted 64 bits and XORed with 64 bits of input data. What is important to note is that 4 of those 8 bytes are simply copies of the input data; they do not depend on the previous CRC at all. Thus, those 4 table look-ups may commence immediately, without waiting for the previous loop iteration.

By always having 4 loads in flight, a modern superscalar processor can be kept busy and make full use of its L1 cache.

Two more details about CRC implementation in the real world:

Normally, appending zero bits to a message which is already a multiple of a polynomial produces a larger multiple of that polynomial. Thus, a basic CRC will not detect appended zero bits (or bytes). To enable a CRC to detect this condition, it' s common to invert the CRC before appending it. This makes the remainder of the message+crc come out not as zero, but some fixed non-zero value. (The CRC of the inversion pattern, 0xffffffff.)

The same problem applies to zero bits prepended to the message, and a similar solution is used. Instead of starting the CRC computation with a remainder of 0, an initial remainder of all ones is used. As long as you start the same way on decoding, it doesn' t make a difference.

CHAPTER

TWO

LZO STREAM FORMAT AS UNDERSTOOD BY LINUX' S LZO DECOMPRESSOR

2.1 Introduction

This is not a specification. No specification seems to be publicly available for the LZO stream format. This document describes what input format the LZO decompressor as implemented in the Linux kernel understands. The file subject of this analysis is lib/lzo/lzo1x_decompress_safe.c. No analysis was made on the compressor nor on any other implementations though it seems likely that the format matches the standard one. The purpose of this document is to better understand what the code does in order to propose more efficient fixes for future bug reports.

2.2 Description

The stream is composed of a series of instructions, operands, and data. The instructions consist in a few bits representing an opcode, and bits forming the operands for the instruction, whose size and position depend on the opcode and on the number of literals copied by previous instruction. The operands are used to indicate:

- a distance when copying data from the dictionary (past output buffer)
- a length (number of bytes to copy from dictionary)
- the number of literals to copy, which is retained in variable "state" as a piece of information for next instructions.

Optionally depending on the opcode and operands, extra data may follow. These extra data can be a complement for the operand (eg: a length or a distance encoded on larger values), or a literal to be copied to the output buffer.

The first byte of the block follows a different encoding from other bytes, it seems to be optimized for literal use only, since there is no dictionary yet prior to that byte.

Lengths are always encoded on a variable size starting with a small number of bits in the operand. If the number of bits isn' t enough to represent the length, up to 255 may be added in increments by consuming more bytes with a rate of at most 255 per extra byte (thus the compression ratio cannot exceed around 255:1). The variable length encoding using #bits is always the same:

```
length = byte & ((1 << #bits) - 1)
if (!length) {
        length = ((1 << #bits) - 1)
        length += 255*(number of zero bytes)
        length += first-non-zero-byte
}
length += constant (generally 2 or 3)
```

For references to the dictionary, distances are relative to the output pointer. Distances are encoded using very few bits belonging to certain ranges, resulting in multiple copy instructions using different encodings. Certain encodings involve one extra byte, others involve two extra bytes forming a little-endian 16-bit quantity (marked LE16 below).

After any instruction except the large literal copy, 0, 1, 2 or 3 literals are copied before starting the next instruction. The number of literals that were copied may change the meaning and behaviour of the next instruction. In practice, only one instruction needs to know whether 0, less than 4, or more literals were copied. This is the information stored in the <state> variable in this implementation. This number of immediate literals to be copied is generally encoded in the last two bits of the instruction but may also be taken from the last two bits of an extra operand (eg: distance).

End of stream is declared when a block copy of distance 0 is seen. Only one instruction may encode this distance (0001HLLL), it takes one LE16 operand for the distance, thus requiring 3 bytes.

Important: In the code some length checks are missing because certain instructions are called under the assumption that a certain number of bytes follow because it has already been guaranteed before parsing the instructions. They just have to "refill" this credit if they consume extra bytes. This is an implementation design choice independent on the algorithm or encoding.

Versions

0: Original version 1: LZO-RLE

Version 1 of LZO implements an extension to encode runs of zeros using run length encoding. This improves speed for data with many zeros, which is a common case for zram. This modifies the bitstream in a backwards compatible way (v1 can correctly decompress v0 compressed data, but v0 cannot read v1 data).

For maximum compatibility, both versions are available under different names (lzo and lzo-rle). Differences in the encoding are noted in this document with e.g.: version 1 only.

2.3 Byte sequences

First byte encoding:

```
0..16   : follow regular instruction encoding, see below.␣
↪It is worth
          noting that code 16 will represent a block copy␣
↪from the
          dictionary which is empty, and that it will␣
↪always be
          invalid at this place.

17      : bitstream version. If the first byte is 17, and␣
↪compressed
          stream length is at least 5 bytes (length of␣
↪shortest possible
          versioned bitstream), the next byte gives the␣
↪bitstream version
          (version 1 only).
          Otherwise, the bitstream version is 0.

18..21  : copy 0..3 literals
          state = (byte - 17) = 0..3  [ copy <state>␣
↪literals ]
          skip byte

22..255 : copy literal string
          length = (byte - 17) = 4..238
          state = 4 [ don't copy extra literals ]
          skip byte
```

Instruction encoding:

```
0 0 0 0 X X X X  (0..15)
  Depends on the number of literals copied by the last␣
↪instruction.
  If last instruction did not copy any literal (state == 0),
↪ this
  encoding will be a copy of 4 or more literal, and must be␣
↪interpreted
  like this :

     0 0 0 0 L L L L  (0..15)  : copy long literal string
     length = 3 + (L ?: 15 + (zero_bytes * 255) + non_zero_
↪byte)
     state = 4  (no extra literals are copied)

  If last instruction used to copy between 1 to 3 literals␣
↪(encoded in
  the instruction's opcode or distance), the instruction is␣
```

(continues on next page)

(continued from previous page)

```
 ↪a copy of a
  2-byte block from the dictionary within a 1kB distance.␣
 ↪It is worth
  noting that this instruction provides little savings␣
 ↪since it uses 2
  bytes to encode a copy of 2 other bytes but it encodes␣
 ↪the number of
  following literals for free. It must be interpreted like␣
 ↪this :

     0 0 0 0 D D S S  (0..15)  : copy 2 bytes from <= 1kB␣
 ↪distance
     length = 2
     state = S (copy S literals after this block)
   Always followed by exactly one byte : H H H H H H H H
     distance = (H << 2) + D + 1

  If last instruction used to copy 4 or more literals (as␣
 ↪detected by
  state == 4), the instruction becomes a copy of a 3-byte␣
 ↪block from the
  dictionary from a 2..3kB distance, and must be␣
 ↪interpreted like this :

     0 0 0 0 D D S S  (0..15)  : copy 3 bytes from 2..3 kB␣
 ↪distance
     length = 3
     state = S (copy S literals after this block)
   Always followed by exactly one byte : H H H H H H H H
     distance = (H << 2) + D + 2049

0 0 0 1 H L L L  (16..31)
     Copy of a block within 16..48kB distance (preferably␣
 ↪less than 10B)
     length = 2 + (L ?: 7 + (zero_bytes * 255) + non_zero_
 ↪byte)
  Always followed by exactly one LE16 :  D D D D D D D D :␣
 ↪D D D D D D S S
     distance = 16384 + (H << 14) + D
     state = S (copy S literals after this block)
     End of stream is reached if distance == 16384
     In version 1 only, to prevent ambiguity with the RLE␣
 ↪case when
     ((distance & 0x803f) == 0x803f) && (261 <= length <=␣
 ↪264), the
     compressor must not emit block copies where distance␣
 ↪and length
     meet these conditions.
```

(continues on next page)

(continued from previous page)

```
  In version 1 only, this instruction is also used to␣
↪encode a run of
     zeros if distance = 0xbfff, i.e. H = 1 and the D bits␣
↪are all 1.
     In this case, it is followed by a fourth byte, X.
     run length = ((X << 3) | (0 0 0 0 0 L L L)) + 4

0 0 1 L L L L L  (32..63)
     Copy of small block within 16kB distance (preferably␣
↪less than 34B)
     length = 2 + (L ?: 31 + (zero_bytes * 255) + non_zero_
↪byte)
  Always followed by exactly one LE16 :  D D D D D D D D :␣
↪D D D D D D S S
     distance = D + 1
     state = S (copy S literals after this block)

0 1 L D D D S S  (64..127)
     Copy 3-4 bytes from block within 2kB distance
     state = S (copy S literals after this block)
     length = 3 + L
   Always followed by exactly one byte : H H H H H H H H
     distance = (H << 3) + D + 1

1 L L D D D S S  (128..255)
     Copy 5-8 bytes from block within 2kB distance
     state = S (copy S literals after this block)
     length = 5 + L
   Always followed by exactly one byte : H H H H H H H H
     distance = (H << 3) + D + 1
```

2.4 Authors

This document was written by Willy Tarreau <w@1wt.eu> on 2014/07/19 during an analysis of the decompression code available in Linux 3.16-rc5, and updated by Dave Rodgman <dave.rodgman@arm.com> on 2018/10/30 to introduce run-length encoding. The code is tricky, it is possible that this document contains mistakes or that a few corner cases were overlooked. In any case, please report any doubt, fix, or proposed updates to the author(s) so that the document can be updated.

CHAPTER

THREE

REMOTE PROCESSOR FRAMEWORK

3.1 Introduction

Modern SoCs typically have heterogeneous remote processor devices in asymmetric multiprocessing (AMP) configurations, which may be running different instances of operating system, whether it’s Linux or any other flavor of real-time OS.

OMAP4, for example, has dual Cortex-A9, dual Cortex-M3 and a C64x+ DSP. In a typical configuration, the dual cortex-A9 is running Linux in a SMP configuration, and each of the other three cores (two M3 cores and a DSP) is running its own instance of RTOS in an AMP configuration.

The remoteproc framework allows different platforms/architectures to control (power on, load firmware, power off) those remote processors while abstracting the hardware differences, so the entire driver doesn’t need to be duplicated. In addition, this framework also adds rpmsg virtio devices for remote processors that supports this kind of communication. This way, platform-specific remoteproc drivers only need to provide a few low-level handlers, and then all rpmsg drivers will then just work (for more information about the virtio-based rpmsg bus and its drivers, please read *Remote Processor Messaging (rpmsg) Framework*). Registration of other types of virtio devices is now also possible. Firmwares just need to publish what kind of virtio devices do they support, and then remoteproc will add those devices. This makes it possible to reuse the existing virtio drivers with remote processor backends at a minimal development cost.

3.2 User API

```
int rproc_boot(struct rproc *rproc)
```

Boot a remote processor (i.e. load its firmware, power it on, …).

If the remote processor is already powered on, this function immediately returns (successfully).

Returns 0 on success, and an appropriate error value otherwise. Note: to use this function you should already have a valid rproc handle. There are several ways to achieve that cleanly (devres, pdata, the way remoteproc_rpmsg.c does this, or, if this becomes prevalent, we might also consider using dev_archdata for this).

```
void rproc_shutdown(struct rproc *rproc)
```

Power off a remote processor (previously booted with rproc_boot()). In case @rproc is still being used by an additional user(s), then this function will just decrement the power refcount and exit, without really powering off the device.

Every call to rproc_boot() must (eventually) be accompanied by a call to rproc_shutdown(). Calling rproc_shutdown() redundantly is a bug.

Note: we’ re not decrementing the rproc’ s refcount, only the power refcount. which means that the @rproc handle stays valid even after rproc_shutdown() returns, and users can still use it with a subsequent rproc_boot(), if needed.

```
struct rproc *rproc_get_by_phandle(phandle phandle)
```

Find an rproc handle using a device tree phandle. Returns the rproc handle on success, and NULL on failure. This function increments the remote processor’ s refcount, so always use rproc_put() to decrement it back once rproc isn’ t needed anymore.

3.3 Typical usage

```
#include <linux/remoteproc.h>

/* in case we were given a valid 'rproc' handle */
int dummy_rproc_example(struct rproc *my_rproc)
{
      int ret;

      /* let's power on and boot our remote processor */
      ret = rproc_boot(my_rproc);
      if (ret) {
              /*
               * something went wrong. handle it and leave.
               */
      }

      /*
       * our remote processor is now powered on... give it some work
       */

      /* let's shut it down now */
      rproc_shutdown(my_rproc);
}
```

3.4 API for implementors

```
struct rproc *rproc_alloc(struct device *dev, const char *name,
                              const struct rproc_ops *ops,
                              const char *firmware, int len)
```

Allocate a new remote processor handle, but don't register it yet. Required parameters are the underlying device, the name of this remote processor, platform-specific ops handlers, the name of the firmware to boot this rproc with, and the length of private data needed by the allocating rproc driver (in bytes).

This function should be used by rproc implementations during initialization of the remote processor.

After creating an rproc handle using this function, and when ready, implementations should then call rproc_add() to complete the registration of the remote processor.

On success, the new rproc is returned, and on failure, NULL.

Note: **never** directly deallocate @rproc, even if it was not registered yet. Instead, when you need to unroll rproc_alloc(), use rproc_free().

```
void rproc_free(struct rproc *rproc)
```

Free an rproc handle that was allocated by rproc_alloc.

This function essentially unrolls rproc_alloc(), by decrementing the rproc's refcount. It doesn't directly free rproc; that would happen only if there are no other references to rproc and its refcount now dropped to zero.

```
int rproc_add(struct rproc *rproc)
```

Register @rproc with the remoteproc framework, after it has been allocated with rproc_alloc().

This is called by the platform-specific rproc implementation, whenever a new remote processor device is probed.

Returns 0 on success and an appropriate error code otherwise. Note: this function initiates an asynchronous firmware loading context, which will look for virtio devices supported by the rproc's firmware.

If found, those virtio devices will be created and added, so as a result of registering this remote processor, additional virtio drivers might get probed.

```
int rproc_del(struct rproc *rproc)
```

Unroll rproc_add().

This function should be called when the platform specific rproc implementation decides to remove the rproc device. it should _only_ be called if a previous invocation of rproc_add() has completed successfully.

After rproc_del() returns, @rproc is still valid, and its last refcount should be decremented by calling rproc_free().

Returns 0 on success and -EINVAL if @rproc isn’ t valid.

```
void rproc_report_crash(struct rproc *rproc, enum rproc_crash_type␣
↪type)
```

Report a crash in a remoteproc

This function must be called every time a crash is detected by the platform specific rproc implementation. This should not be called from a non-remoteproc driver. This function can be called from atomic/interrupt context.

3.5 Implementation callbacks

These callbacks should be provided by platform-specific remoteproc drivers:

```
/**
 * struct rproc_ops - platform-specific device handlers
 * @start:    power on the device and boot it
 * @stop:     power off the device
 * @kick:     kick a virtqueue (virtqueue id given as a parameter)
 */
struct rproc_ops {
      int (*start)(struct rproc *rproc);
      int (*stop)(struct rproc *rproc);
      void (*kick)(struct rproc *rproc, int vqid);
};
```

Every remoteproc implementation should at least provide the ->start and ->stop handlers. If rpmsg/virtio functionality is also desired, then the ->kick handler should be provided as well.

The ->start() handler takes an rproc handle and should then power on the device and boot it (use rproc->priv to access platform-specific private data). The boot address, in case needed, can be found in rproc->bootaddr (remoteproc core puts there the ELF entry point). On success, 0 should be returned, and on failure, an appropriate error code.

The ->stop() handler takes an rproc handle and powers the device down. On success, 0 is returned, and on failure, an appropriate error code.

The ->kick() handler takes an rproc handle, and an index of a virtqueue where new message was placed in. Implementations should interrupt the remote processor and let it know it has pending messages. Notifying remote processors the exact virtqueue index to look in is optional: it is easy (and not too expensive) to go through the existing virtqueues and look for new buffers in the used rings.

3.6 Binary Firmware Structure

At this point remoteproc supports ELF32 and ELF64 firmware binaries. However, it is quite expected that other platforms/devices which we’d want to support with this framework will be based on different binary formats.

When those use cases show up, we will have to decouple the binary format from the framework core, so we can support several binary formats without duplicating common code.

When the firmware is parsed, its various segments are loaded to memory according to the specified device address (might be a physical address if the remote processor is accessing memory directly).

In addition to the standard ELF segments, most remote processors would also include a special section which we call “the resource table”.

The resource table contains system resources that the remote processor requires before it should be powered on, such as allocation of physically contiguous memory, or iommu mapping of certain on-chip peripherals. Remotecore will only power up the device after all the resource table’s requirement are met.

In addition to system resources, the resource table may also contain resource entries that publish the existence of supported features or configurations by the remote processor, such as trace buffers and supported virtio devices (and their configurations).

The resource table begins with this header:

```
/**
 * struct resource_table - firmware resource table header
 * @ver: version number
 * @num: number of resource entries
 * @reserved: reserved (must be zero)
 * @offset: array of offsets pointing at the various resource␣
↪entries
 *
 * The header of the resource table, as expressed by this structure,
 * contains a version number (should we need to change this format␣
↪in the
 * future), the number of available resource entries, and their␣
↪offsets
 * in the table.
 */
struct resource_table {
      u32 ver;
      u32 num;
      u32 reserved[2];
      u32 offset[0];
} __packed;
```

Immediately following this header are the resource entries themselves, each of which begins with the following resource entry header:

```
/**
 * struct fw_rsc_hdr - firmware resource entry header
 * @type: resource type
 * @data: resource data
 *
 * Every resource entry begins with a 'struct fw_rsc_hdr' header␣
↪providing
 * its @type. The content of the entry itself will immediately␣
↪follow
 * this header, and it should be parsed according to the resource␣
↪type.
 */
struct fw_rsc_hdr {
      u32 type;
      u8 data[0];
} __packed;
```

Some resources entries are mere announcements, where the host is informed of specific remoteproc configuration. Other entries require the host to do something (e.g. allocate a system resource). Sometimes a negotiation is expected, where the firmware requests a resource, and once allocated, the host should provide back its details (e.g. address of an allocated memory region).

Here are the various resource types that are currently supported:

```
/**
 * enum fw_resource_type - types of resource entries
 *
 * @RSC_CARVEOUT:   request for allocation of a physically␣
↪contiguous
 *                memory region.
 * @RSC_DEVMEM:     request to iommu_map a memory-based peripheral.
 * @RSC_TRACE:            announces the availability of a trace␣
↪buffer into which
 *                the remote processor will be writing logs.
 * @RSC_VDEV:       declare support for a virtio device, and serve␣
↪as its
 *                virtio header.
 * @RSC_LAST:       just keep this one at the end
 * @RSC_VENDOR_START: start of the vendor specific resource types␣
↪range
 * @RSC_VENDOR_END:   end of the vendor specific resource types␣
↪range
 *
 * Please note that these values are used as indices to the rproc_
↪handle_rsc
 * lookup table, so please keep them sane. Moreover, @RSC_LAST is␣
↪used to
 * check the validity of an index before the lookup table is␣
↪accessed, so
 * please update it as needed.
```

(continues on next page)

(continued from previous page)

```
 */
enum fw_resource_type {
      RSC_CARVEOUT            = 0,
      RSC_DEVMEM              = 1,
      RSC_TRACE               = 2,
      RSC_VDEV                = 3,
      RSC_LAST                = 4,
      RSC_VENDOR_START        = 128,
      RSC_VENDOR_END          = 512,
};
```

For more details regarding a specific resource type, please see its dedicated structure in include/linux/remoteproc.h.

We also expect that platform-specific resource entries will show up at some point. When that happens, we could easily add a new RSC_PLATFORM type, and hand those resources to the platform-specific rproc driver to handle.

3.7 Virtio and remoteproc

The firmware should provide remoteproc information about virtio devices that it supports, and their configurations: a RSC_VDEV resource entry should specify the virtio device id (as in virtio_ids.h), virtio features, virtio config space, vrings information, etc.

When a new remote processor is registered, the remoteproc framework will look for its resource table and will register the virtio devices it supports. A firmware may support any number of virtio devices, and of any type (a single remote processor can also easily support several rpmsg virtio devices this way, if desired).

Of course, RSC_VDEV resource entries are only good enough for static allocation of virtio devices. Dynamic allocations will also be made possible using the rpmsg bus (similar to how we already do dynamic allocations of rpmsg channels; read more about it in rpmsg.txt).

CHAPTER

FOUR

REMOTE PROCESSOR MESSAGING (RPMSG) FRAMEWORK

Note: This document describes the rpmsg bus and how to write rpmsg drivers. To learn how to add rpmsg support for new platforms, check out remoteproc.txt (also a resident of Documentation/).

4.1 Introduction

Modern SoCs typically employ heterogeneous remote processor devices in asymmetric multiprocessing (AMP) configurations, which may be running different instances of operating system, whether it's Linux or any other flavor of real-time OS.

OMAP4, for example, has dual Cortex-A9, dual Cortex-M3 and a C64x+ DSP. Typically, the dual cortex-A9 is running Linux in a SMP configuration, and each of the other three cores (two M3 cores and a DSP) is running its own instance of RTOS in an AMP configuration.

Typically AMP remote processors employ dedicated DSP codecs and multimedia hardware accelerators, and therefore are often used to offload CPU-intensive multimedia tasks from the main application processor.

These remote processors could also be used to control latency-sensitive sensors, drive random hardware blocks, or just perform background tasks while the main CPU is idling.

Users of those remote processors can either be userland apps (e.g. multimedia frameworks talking with remote OMX components) or kernel drivers (controlling hardware accessible only by the remote processor, reserving kernel-controlled resources on behalf of the remote processor, etc..).

Rpmsg is a virtio-based messaging bus that allows kernel drivers to communicate with remote processors available on the system. In turn, drivers could then expose appropriate user space interfaces, if needed.

When writing a driver that exposes rpmsg communication to userland, please keep in mind that remote processors might have direct access to the system's physical memory and other sensitive hardware resources (e.g. on OMAP4, remote cores and hardware accelerators may have direct access to the physical memory, gpio banks, dma controllers, i2c bus, gptimers, mailbox devices, hwspinlocks, etc..).

Moreover, those remote processors might be running RTOS where every task can access the entire memory/devices exposed to the processor. To minimize the risks of rogue (or buggy) userland code exploiting remote bugs, and by that taking over the system, it is often desired to limit userland to specific rpmsg channels (see definition below) it can send messages on, and if possible, minimize how much control it has over the content of the messages.

Every rpmsg device is a communication channel with a remote processor (thus rpmsg devices are called channels). Channels are identified by a textual name and have a local (“source”) rpmsg address, and remote (“destination”) rpmsg address.

When a driver starts listening on a channel, its rx callback is bound with a unique rpmsg local address (a 32-bit integer). This way when inbound messages arrive, the rpmsg core dispatches them to the appropriate driver according to their destination address (this is done by invoking the driver’ s rx handler with the payload of the inbound message).

4.2 User API

```
int rpmsg_send(struct rpmsg_channel *rpdev, void *data, int len);
```

sends a message across to the remote processor on a given channel. The caller should specify the channel, the data it wants to send, and its length (in bytes). The message will be sent on the specified channel, i.e. its source and destination address fields will be set to the channel’ s src and dst addresses.

In case there are no TX buffers available, the function will block until one becomes available (i.e. until the remote processor consumes a tx buffer and puts it back on virtio’ s used descriptor ring), or a timeout of 15 seconds elapses. When the latter happens, -ERESTARTSYS is returned.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_sendto(struct rpmsg_channel *rpdev, void *data, int len,␣
↪u32 dst);
```

sends a message across to the remote processor on a given channel, to a destination address provided by the caller.

The caller should specify the channel, the data it wants to send, its length (in bytes), and an explicit destination address.

The message will then be sent to the remote processor to which the channel belongs, using the channel’ s src address, and the user-provided dst address (thus the channel’ s dst address will be ignored).

In case there are no TX buffers available, the function will block until one becomes available (i.e. until the remote processor consumes a tx buffer and puts it back on virtio’ s used descriptor ring), or a timeout of 15 seconds elapses. When the latter happens, -ERESTARTSYS is returned.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_send_offchannel(struct rpmsg_channel *rpdev, u32 src, u32␣
↪dst,
                                                    void *data,␣
↪int len);
```

sends a message across to the remote processor, using the src and dst addresses provided by the user.

The caller should specify the channel, the data it wants to send, its length (in bytes), and explicit source and destination addresses. The message will then be sent to the remote processor to which the channel belongs, but the channel' s src and dst addresses will be ignored (and the user-provided addresses will be used instead).

In case there are no TX buffers available, the function will block until one becomes available (i.e. until the remote processor consumes a tx buffer and puts it back on virtio' s used descriptor ring), or a timeout of 15 seconds elapses. When the latter happens, -ERESTARTSYS is returned.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_trysend(struct rpmsg_channel *rpdev, void *data, int len);
```

sends a message across to the remote processor on a given channel. The caller should specify the channel, the data it wants to send, and its length (in bytes). The message will be sent on the specified channel, i.e. its source and destination address fields will be set to the channel' s src and dst addresses.

In case there are no TX buffers available, the function will immediately return -ENOMEM without waiting until one becomes available.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_trysendto(struct rpmsg_channel *rpdev, void *data, int␣
↪len, u32 dst)
```

sends a message across to the remote processor on a given channel, to a destination address provided by the user.

The user should specify the channel, the data it wants to send, its length (in bytes), and an explicit destination address.

The message will then be sent to the remote processor to which the channel belongs, using the channel' s src address, and the user-provided dst address (thus the channel' s dst address will be ignored).

In case there are no TX buffers available, the function will immediately return -ENOMEM without waiting until one becomes available.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
int rpmsg_trysend_offchannel(struct rpmsg_channel *rpdev, u32 src,␣
↪u32 dst,
                                                      void *data,␣
↪int len);
```

sends a message across to the remote processor, using source and destination addresses provided by the user.

The user should specify the channel, the data it wants to send, its length (in bytes), and explicit source and destination addresses. The message will then be sent to the remote processor to which the channel belongs, but the channel’ s src and dst addresses will be ignored (and the user-provided addresses will be used instead).

In case there are no TX buffers available, the function will immediately return -ENOMEM without waiting until one becomes available.

The function can only be called from a process context (for now). Returns 0 on success and an appropriate error value on failure.

```
struct rpmsg_endpoint *rpmsg_create_ept(struct rpmsg_device *rpdev,
                                        rpmsg_rx_cb_t cb, void␣
↪*priv,
                                        struct rpmsg_channel_info␣
↪chinfo);
```

every rpmsg address in the system is bound to an rx callback (so when inbound messages arrive, they are dispatched by the rpmsg bus using the appropriate callback handler) by means of an rpmsg_endpoint struct.

This function allows drivers to create such an endpoint, and by that, bind a callback, and possibly some private data too, to an rpmsg address (either one that is known in advance, or one that will be dynamically assigned for them).

Simple rpmsg drivers need not call rpmsg_create_ept, because an endpoint is already created for them when they are probed by the rpmsg bus (using the rx callback they provide when they registered to the rpmsg bus).

So things should just work for simple drivers: they already have an endpoint, their rx callback is bound to their rpmsg address, and when relevant inbound messages arrive (i.e. messages which their dst address equals to the src address of their rpmsg channel), the driver’ s handler is invoked to process it.

That said, more complicated drivers might do need to allocate additional rpmsg addresses, and bind them to different rx callbacks. To accomplish that, those drivers need to call this function. Drivers should provide their channel (so the new endpoint would bind to the same remote processor their channel belongs to), an rx callback function, an optional private data (which is provided back when the rx callback is invoked), and an address they want to bind with the callback. If addr is RPMSG_ADDR_ANY, then rpmsg_create_ept will dynamically assign them an available rpmsg address (drivers should have a very good reason why not to always use RPMSG_ADDR_ANY here).

Returns a pointer to the endpoint on success, or NULL on error.

```
void rpmsg_destroy_ept(struct rpmsg_endpoint *ept);
```

destroys an existing rpmsg endpoint. user should provide a pointer to an rpmsg endpoint that was previously created with rpmsg_create_ept().

```
int register_rpmsg_driver(struct rpmsg_driver *rpdrv);
```

registers an rpmsg driver with the rpmsg bus. user should provide a pointer to an rpmsg_driver struct, which contains the driver's ->probe() and ->remove() functions, an rx callback, and an id_table specifying the names of the channels this driver is interested to be probed with.

```
void unregister_rpmsg_driver(struct rpmsg_driver *rpdrv);
```

unregisters an rpmsg driver from the rpmsg bus. user should provide a pointer to a previously-registered rpmsg_driver struct. Returns 0 on success, and an appropriate error value on failure.

4.3 Typical usage

The following is a simple rpmsg driver, that sends an "hello!" message on probe(), and whenever it receives an incoming message, it dumps its content to the console.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rpmsg.h>

static void rpmsg_sample_cb(struct rpmsg_channel *rpdev, void *data,
 ↪ int len,
                                              void *priv, u32 src)
{
      print_hex_dump(KERN_INFO, "incoming message:", DUMP_PREFIX_
 ↪NONE,
                                              16, 1, data, len,␣
 ↪true);
}

static int rpmsg_sample_probe(struct rpmsg_channel *rpdev)
{
      int err;

      dev_info(&rpdev->dev, "chnl: 0x%x -> 0x%x\n", rpdev->src,␣
 ↪rpdev->dst);

      /* send a message on our channel */
      err = rpmsg_send(rpdev, "hello!", 6);
      if (err) {
              pr_err("rpmsg_send failed: %d\n", err);
              return err;
```

(continues on next page)

(continued from previous page)

```
      }

      return 0;
}

static void rpmsg_sample_remove(struct rpmsg_channel *rpdev)
{
      dev_info(&rpdev->dev, "rpmsg sample client driver is removed\n
↪");
}

static struct rpmsg_device_id rpmsg_driver_sample_id_table[] = {
      { .name = "rpmsg-client-sample" },
      { },
};
MODULE_DEVICE_TABLE(rpmsg, rpmsg_driver_sample_id_table);

static struct rpmsg_driver rpmsg_sample_client = {
      .drv.name       = KBUILD_MODNAME,
      .id_table       = rpmsg_driver_sample_id_table,
      .probe          = rpmsg_sample_probe,
      .callback       = rpmsg_sample_cb,
      .remove         = rpmsg_sample_remove,
};
module_rpmsg_driver(rpmsg_sample_client);
```

Note: a similar sample which can be built and loaded can be found in samples/rpmsg/.

4.4 Allocations of rpmsg channels

At this point we only support dynamic allocations of rpmsg channels.

This is possible only with remote processors that have the VIRTIO_RPMSG_F_NS virtio device feature set. This feature bit means that the remote processor supports dynamic name service announcement messages.

When this feature is enabled, creation of rpmsg devices (i.e. channels) is completely dynamic: the remote processor announces the existence of a remote rpmsg service by sending a name service message (which contains the name and rpmsg addr of the remote service, see struct rpmsg_ns_msg).

This message is then handled by the rpmsg bus, which in turn dynamically creates and registers an rpmsg channel (which represents the remote service). If/when a relevant rpmsg driver is registered, it will be immediately probed by the bus, and can then start sending messages to the remote service.

The plan is also to add static creation of rpmsg channels via the virtio config space, but it’s not implemented yet.

CHAPTER FIVE

SPECULATION

This document explains potential effects of speculation, and how undesirable effects can be mitigated portably using common APIs.

To improve performance and minimize average latencies, many contemporary CPUs employ speculative execution techniques such as branch prediction, performing work which may be discarded at a later stage.

Typically speculative execution cannot be observed from architectural state, such as the contents of registers. However, in some cases it is possible to observe its impact on microarchitectural state, such as the presence or absence of data in caches. Such state may form side-channels which can be observed to extract secret information.

For example, in the presence of branch prediction, it is possible for bounds checks to be ignored by code which is speculatively executed. Consider the following code:

```
int load_array(int *array, unsigned int index)
{
        if (index >= MAX_ARRAY_ELEMS)
                return 0;
        else
                return array[index];
}
```

Which, on arm64, may be compiled to an assembly sequence such as:

```
      CMP     <index>, #MAX_ARRAY_ELEMS
      B.LT    less
      MOV     <returnval>, #0
      RET
less:
      LDR     <returnval>, [<array>, <index>]
      RET
```

It is possible that a CPU mis-predicts the conditional branch, and speculatively loads array[index], even if index >= MAX_ARRAY_ELEMS. This value will subsequently be discarded, but the speculated load may affect microarchitectural state which can be subsequently measured.

More complex sequences involving multiple dependent memory accesses may result in sensitive information being leaked. Consider the following code, building on the prior example:

```
int load_dependent_arrays(int *arr1, int *arr2, int index)
{
        int val1, val2,

        val1 = load_array(arr1, index);
        val2 = load_array(arr2, val1);

        return val2;
}
```

Under speculation, the first call to load_array() may return the value of an out-of-bounds address, while the second call will influence microarchitectural state dependent on this value. This may provide an arbitrary read primitive.

CHAPTER

SIX

MITIGATING SPECULATION SIDE-CHANNELS

The kernel provides a generic API to ensure that bounds checks are respected even under speculation. Architectures which are affected by speculation-based side-channels are expected to implement these primitives.

The array_index_nospec() helper in <linux/nospec.h> can be used to prevent information from being leaked via side-channels.

A call to array_index_nospec(index, size) returns a sanitized index value that is bounded to [0, size) even under cpu speculation conditions.

This can be used to protect the earlier load_array() example:

```
int load_array(int *array, unsigned int index)
{
        if (index >= MAX_ARRAY_ELEMS)
                return 0;
        else {
                index = array_index_nospec(index, MAX_ARRAY_ELEMS);
                return array[index];
        }
}
```

CHAPTER

SEVEN

STATIC KEYS

Warning: DEPRECATED API:

The use of 'struct static_key' directly, is now DEPRECATED. In addition static_key_{true,false}() is also DEPRECATED. IE DO NOT use the following:

```
struct static_key false = STATIC_KEY_INIT_FALSE;
struct static_key true = STATIC_KEY_INIT_TRUE;
static_key_true()
static_key_false()
```

The updated API replacements are:

```
DEFINE_STATIC_KEY_TRUE(key);
DEFINE_STATIC_KEY_FALSE(key);
DEFINE_STATIC_KEY_ARRAY_TRUE(keys, count);
DEFINE_STATIC_KEY_ARRAY_FALSE(keys, count);
static_branch_likely()
static_branch_unlikely()
```

7.1 Abstract

Static keys allows the inclusion of seldom used features in performance-sensitive fast-path kernel code, via a GCC feature and a code patching technique. A quick example:

```
DEFINE_STATIC_KEY_FALSE(key);

...

if (static_branch_unlikely(&key))
        do unlikely code
else
        do likely code

...
static_branch_enable(&key);
...
```

(continues on next page)

(continued from previous page)

```
static_branch_disable(&key);
...
```

The static_branch_unlikely() branch will be generated into the code with as little impact to the likely code path as possible.

7.2 Motivation

Currently, tracepoints are implemented using a conditional branch. The conditional check requires checking a global variable for each tracepoint. Although the overhead of this check is small, it increases when the memory cache comes under pressure (memory cache lines for these global variables may be shared with other memory accesses). As we increase the number of tracepoints in the kernel this overhead may become more of an issue. In addition, tracepoints are often dormant (disabled) and provide no direct kernel functionality. Thus, it is highly desirable to reduce their impact as much as possible. Although tracepoints are the original motivation for this work, other kernel code paths should be able to make use of the static keys facility.

7.3 Solution

gcc (v4.5) adds a new ‘asm goto’ statement that allows branching to a label:

https://gcc.gnu.org/ml/gcc-patches/2009-07/msg01556.html

Using the ‘asm goto’, we can create branches that are either taken or not taken by default, without the need to check memory. Then, at run-time, we can patch the branch site to change the branch direction.

For example, if we have a simple branch that is disabled by default:

```
if (static_branch_unlikely(&key))
        printk("I am the true branch\n");
```

Thus, by default the ‘printk’ will not be emitted. And the code generated will consist of a single atomic ‘no-op’ instruction (5 bytes on x86), in the straight-line code path. When the branch is ‘flipped’, we will patch the ‘no-op’in the straight-line codepath with a ‘jump’ instruction to the out-of-line true branch. Thus, changing branch direction is expensive but branch selection is basically ‘free’. That is the basic tradeoff of this optimization.

This lowlevel patching mechanism is called ‘jump label patching’, and it gives the basis for the static keys facility.

7.4 Static key label API, usage and examples

In order to make use of this optimization you must first define a key:

```
DEFINE_STATIC_KEY_TRUE(key);
```

or:

```
DEFINE_STATIC_KEY_FALSE(key);
```

The key must be global, that is, it can’ t be allocated on the stack or dynamically allocated at run-time.

The key is then used in code as:

```
if (static_branch_unlikely(&key))
        do unlikely code
else
        do likely code
```

Or:

```
if (static_branch_likely(&key))
        do likely code
else
        do unlikely code
```

Keys defined via DEFINE_STATIC_KEY_TRUE(), or DEFINE_STATIC_KEY_FALSE, may be used in either static_branch_likely() or static_branch_unlikely() statements.

Branch(es) can be set true via:

```
static_branch_enable(&key);
```

or false via:

```
static_branch_disable(&key);
```

The branch(es) can then be switched via reference counts:

```
static_branch_inc(&key);
...
static_branch_dec(&key);
```

Thus, ‘static_branch_inc()’means ‘make the branch true’, and ‘static_branch_dec()’ means ‘make the branch false’ with appropriate reference counting. For example, if the key is initialized true, a static_branch_dec(), will switch the branch to false. And a subsequent static_branch_inc(), will change the branch back to true. Likewise, if the key is initialized false, a ‘static_branch_inc()’ , will change the branch to true. And then a ‘static_branch_dec()’ , will again make the branch false.

The state and the reference count can be retrieved with ‘static_key_enabled()’ and ‘static_key_count()’ . In general, if you use these functions, they should be protected with the same mutex used around the enable/disable or increment/decrement function.

Note that switching branches results in some locks being taken, particularly the CPU hotplug lock (in order to avoid races against CPUs being brought in the kernel while the kernel is getting patched). Calling the static key API from within a hotplug notifier is thus a sure deadlock recipe. In order to still allow use of the functionality, the following functions are provided:

> static_key_enable_cpuslocked() static_key_disable_cpuslocked()
> static_branch_enable_cpuslocked() static_branch_disable_cpuslocked()

These functions are *not* general purpose, and must only be used when you really know that you’re in the above context, and no other.

Where an array of keys is required, it can be defined as:

```
DEFINE_STATIC_KEY_ARRAY_TRUE(keys, count);
```

or:

```
DEFINE_STATIC_KEY_ARRAY_FALSE(keys, count);
```

4) Architecture level code patching interface, ‘jump labels’

There are a few functions and macros that architectures must implement in order to take advantage of this optimization. If there is no architecture support, we simply fall back to a traditional, load, test, and jump sequence. Also, the struct jump_entry table must be at least 4-byte aligned because the static_key->entry field makes use of the two least significant bits.

- **select HAVE_ARCH_JUMP_LABEL,**
  see: arch/x86/Kconfig
- **#define JUMP_LABEL_NOP_SIZE,**
  see: arch/x86/include/asm/jump_label.h
- **__always_inline bool arch_static_branch(struct static_key *key, bool branch),**
  see: arch/x86/include/asm/jump_label.h
- **__always_inline bool arch_static_branch_jump(struct static_key *key, bool branch),**
  see: arch/x86/include/asm/jump_label.h
- **void arch_jump_label_transform(struct jump_entry *entry, enum jump_label_type type),**
  see: arch/x86/kernel/jump_label.c
- **__init_or_module void arch_jump_label_transform_static(struct jump_entry *entry, enum jump_label_type type),**
  see: arch/x86/kernel/jump_label.c
- **struct jump_entry,**
  see: arch/x86/include/asm/jump_label.h

5) Static keys / jump label analysis, results (x86_64):

As an example, let’ s add the following branch to ‘getppid()’ , such that the system call now looks like:

```
SYSCALL_DEFINE0(getppid)
{
        int pid;

+       if (static_branch_unlikely(&key))
+               printk("I am the true branch\n");

        rcu_read_lock();
        pid = task_tgid_vnr(rcu_dereference(current->real_parent));
        rcu_read_unlock();

        return pid;
}
```

The resulting instructions with jump labels generated by GCC is:

```
ffffffff81044290 <sys_getppid>:
ffffffff81044290:       55                      push   %rbp
ffffffff81044291:       48 89 e5                mov    %rsp,%rbp
ffffffff81044294:       e9 00 00 00 00          jmpq   ␣
↪ffffffff81044299 <sys_getppid+0x9>
ffffffff81044299:       65 48 8b 04 25 c0 b6    mov    %gs:0xb6c0,
↪%rax
ffffffff810442a0:       00 00
ffffffff810442a2:       48 8b 80 80 02 00 00    mov    0x280(%rax),
↪%rax
ffffffff810442a9:       48 8b 80 b0 02 00 00    mov    0x2b0(%rax),
↪%rax
ffffffff810442b0:       48 8b b8 e8 02 00 00    mov    0x2e8(%rax),
↪%rdi
ffffffff810442b7:       e8 f4 d9 00 00          callq  ␣
↪ffffffff81051cb0 <pid_vnr>
ffffffff810442bc:       5d                      pop    %rbp
ffffffff810442bd:       48 98                   cltq
ffffffff810442bf:       c3                      retq
ffffffff810442c0:       48 c7 c7 e3 54 98 81    mov
↪$0xffffffff819854e3,%rdi
ffffffff810442c7:       31 c0                   xor    %eax,%eax
ffffffff810442c9:       e8 71 13 6d 00          callq  ␣
↪ffffffff8171563f <printk>
ffffffff810442ce:       eb c9                   jmp    ␣
↪ffffffff81044299 <sys_getppid+0x9>
```

Without the jump label optimization it looks like:

```
ffffffff810441f0 <sys_getppid>:
ffffffff810441f0:       8b 05 8a 52 d8 00       mov    0xd8528a(
↪%rip),%eax        # ffffffff81dc9480 <key>
```

(continues on next page)

(continued from previous page)

```
ffffffff810441f6:       55                      push   %rbp
ffffffff810441f7:       48 89 e5                mov    %rsp,%rbp
ffffffff810441fa:       85 c0                   test   %eax,%eax
ffffffff810441fc:       75 27                   jne    ␣
↪ffffffff81044225 <sys_getppid+0x35>
ffffffff810441fe:       65 48 8b 04 25 c0 b6    mov    %gs:0xb6c0,
↪%rax
ffffffff81044205:       00 00
ffffffff81044207:       48 8b 80 80 02 00 00    mov    0x280(%rax),
↪%rax
ffffffff8104420e:       48 8b 80 b0 02 00 00    mov    0x2b0(%rax),
↪%rax
ffffffff81044215:       48 8b b8 e8 02 00 00    mov    0x2e8(%rax),
↪%rdi
ffffffff8104421c:       e8 2f da 00 00          callq  ␣
↪ffffffff81051c50 <pid_vnr>
ffffffff81044221:       5d                      pop    %rbp
ffffffff81044222:       48 98                   cltq
ffffffff81044224:       c3                      retq
ffffffff81044225:       48 c7 c7 13 53 98 81    mov
↪$0xffffffff81985313,%rdi
ffffffff8104422c:       31 c0                   xor    %eax,%eax
ffffffff8104422e:       e8 60 0f 6d 00          callq  ␣
↪ffffffff81715193 <printk>
ffffffff81044233:       eb c9                   jmp    ␣
↪ffffffff810441fe <sys_getppid+0xe>
ffffffff81044235:       66 66 2e 0f 1f 84 00    data32 nopw %cs:0x0(
↪%rax,%rax,1)
ffffffff8104423c:       00 00 00 00
```

Thus, the disable jump label case adds a 'mov', 'test' and 'jne' instruction vs. the jump label case just has a 'no-op' or 'jmp 0'. (The jmp 0, is patched to a 5 byte atomic no-op instruction at boot-time.) Thus, the disabled jump label case adds:

```
6 (mov) + 2 (test) + 2 (jne) = 10 - 5 (5 byte jump 0) = 5 addition␣
↪bytes.
```

If we then include the padding bytes, the jump label code saves, 16 total bytes of instruction memory for this small function. In this case the non-jump label function is 80 bytes long. Thus, we have saved 20% of the instruction footprint. We can in fact improve this even further, since the 5-byte no-op really can be a 2-byte no-op since we can reach the branch with a 2-byte jmp. However, we have not yet implemented optimal no-op sizes (they are currently hard-coded).

Since there are a number of static key API uses in the scheduler paths, 'pipe-test' (also known as 'perf bench sched pipe') can be used to show the performance improvement. Testing done on 3.3.0-rc2:

jump label disabled:

```
Performance counter stats for 'bash -c /tmp/pipe-test' (50 runs):

       855.700314 task-clock                #    0.534 CPUs␣
 ↪utilized            ( +-  0.11% )
          200,003 context-switches          #    0.234 M/sec       ␣
 ↪            ( +-  0.00% )
                0 CPU-migrations            #    0.000 M/sec       ␣
 ↪            ( +- 39.58% )
              487 page-faults               #    0.001 M/sec       ␣
 ↪            ( +-  0.02% )
    1,474,374,262 cycles                    #    1.723 GHz         ␣
 ↪            ( +-  0.17% )
  <not supported> stalled-cycles-frontend
  <not supported> stalled-cycles-backend
    1,178,049,567 instructions              #    0.80  insns per␣
 ↪cycle          ( +-  0.06% )
      208,368,926 branches                  #  243.507 M/sec       ␣
 ↪            ( +-  0.06% )
        5,569,188 branch-misses             #    2.67% of all␣
 ↪branches          ( +-  0.54% )

      1.601607384 seconds time elapsed                             ␣
 ↪            ( +-  0.07% )
```

jump label enabled:

```
Performance counter stats for 'bash -c /tmp/pipe-test' (50 runs):

       841.043185 task-clock                #    0.533 CPUs␣
 ↪utilized            ( +-  0.12% )
          200,004 context-switches          #    0.238 M/sec       ␣
 ↪            ( +-  0.00% )
                0 CPU-migrations            #    0.000 M/sec       ␣
 ↪            ( +- 40.87% )
              487 page-faults               #    0.001 M/sec       ␣
 ↪            ( +-  0.05% )
    1,432,559,428 cycles                    #    1.703 GHz         ␣
 ↪            ( +-  0.18% )
  <not supported> stalled-cycles-frontend
  <not supported> stalled-cycles-backend
    1,175,363,994 instructions              #    0.82  insns per␣
 ↪cycle          ( +-  0.04% )
      206,859,359 branches                  #  245.956 M/sec       ␣
 ↪            ( +-  0.04% )
        4,884,119 branch-misses             #    2.36% of all␣
 ↪branches          ( +-  0.85% )

      1.579384366 seconds time elapsed
```

The percentage of saved branches is .7%, and we’ ve saved 12% on ‘branch-misses’ . This is where we would expect to get the most savings, since this optimization is about reducing the number of branches. In addition, we’ ve saved .2% on instructions, and 2.8% on cycles and 1.4% on elapsed time.

CHAPTER

EIGHT

TEE SUBSYSTEM

This document describes the TEE subsystem in Linux.

A TEE (Trusted Execution Environment) is a trusted OS running in some secure environment, for example, TrustZone on ARM CPUs, or a separate secure co-processor etc. A TEE driver handles the details needed to communicate with the TEE.

This subsystem deals with:

- Registration of TEE drivers
- Managing shared memory between Linux and the TEE
- Providing a generic API to the TEE

8.1 The TEE interface

include/uapi/linux/tee.h defines the generic interface to a TEE.

User space (the client) connects to the driver by opening /dev/tee[0-9]* or /dev/teepriv[0-9]*.

- TEE_IOC_SHM_ALLOC allocates shared memory and returns a file descriptor which user space can mmap. When user space doesn't need the file descriptor any more, it should be closed. When shared memory isn' t needed any longer it should be unmapped with munmap() to allow the reuse of memory.
- TEE_IOC_VERSION lets user space know which TEE this driver handles and its capabilities.
- TEE_IOC_OPEN_SESSION opens a new session to a Trusted Application.
- TEE_IOC_INVOKE invokes a function in a Trusted Application.
- TEE_IOC_CANCEL may cancel an ongoing TEE_IOC_OPEN_SESSION or TEE_IOC_INVOKE.
- TEE_IOC_CLOSE_SESSION closes a session to a Trusted Application.

There are two classes of clients, normal clients and supplicants. The latter is a helper process for the TEE to access resources in Linux, for example file system access. A normal client opens /dev/tee[0-9]* and a supplicant opens /dev/teepriv[0-9].

Much of the communication between clients and the TEE is opaque to the driver. The main job for the driver is to receive requests from the clients, forward them to the TEE and send back the results. In the case of supplicants the communication goes in the other direction, the TEE sends requests to the supplicant which then sends back the result.

8.2 The TEE kernel interface

Kernel provides a TEE bus infrastructure where a Trusted Application is represented as a device identified via Universally Unique Identifier (UUID) and client drivers register a table of supported device UUIDs.

TEE bus infrastructure registers following APIs:

match():
iterates over the client driver UUID table to find a corresponding match for device UUID. If a match is found, then this particular device is probed via corresponding probe API registered by the client driver. This process happens whenever a device or a client driver is registered with TEE bus.

uevent():
notifies user-space (udev) whenever a new device is registered on TEE bus for auto-loading of modularized client drivers.

TEE bus device enumeration is specific to underlying TEE implementation, so it is left open for TEE drivers to provide corresponding implementation.

Then TEE client driver can talk to a matched Trusted Application using APIs listed in include/linux/tee_drv.h.

8.2.1 TEE client driver example

Suppose a TEE client driver needs to communicate with a Trusted Application having UUID: `ac6a4085-0e82-4c33-bf98-8eb8e118b6c2`, so driver registration snippet would look like:

```
static const struct tee_client_device_id client_id_table[] = {
        {UUID_INIT(0xac6a4085, 0x0e82, 0x4c33,
                   0xbf, 0x98, 0x8e, 0xb8, 0xe1, 0x18, 0xb6, 0xc2)},
        {}
};

MODULE_DEVICE_TABLE(tee, client_id_table);

static struct tee_client_driver client_driver = {
        .id_table       = client_id_table,
        .driver         = {
                .name           = DRIVER_NAME,
                .bus            = &tee_bus_type,
                .probe          = client_probe,
                .remove         = client_remove,
```

(continues on next page)

(continued from previous page)

```
        },
};

static int __init client_init(void)
{
        return driver_register(&client_driver.driver);
}

static void __exit client_exit(void)
{
        driver_unregister(&client_driver.driver);
}

module_init(client_init);
module_exit(client_exit);
```

8.3 OP-TEE driver

The OP-TEE driver handles OP-TEE [1] based TEEs. Currently it is only the ARM TrustZone based OP-TEE solution that is supported.

Lowest level of communication with OP-TEE builds on ARM SMC Calling Convention (SMCCC) [2], which is the foundation for OP-TEE’ s SMC interface [3] used internally by the driver. Stacked on top of that is OP-TEE Message Protocol [4].

OP-TEE SMC interface provides the basic functions required by SMCCC and some additional functions specific for OP-TEE. The most interesting functions are:

- OPTEE_SMC_FUNCID_CALLS_UID (part of SMCCC) returns the version information which is then returned by TEE_IOC_VERSION
- OPTEE_SMC_CALL_GET_OS_UUID returns the particular OP-TEE implementation, used to tell, for instance, a TrustZone OP-TEE apart from an OP-TEE running on a separate secure co-processor.
- OPTEE_SMC_CALL_WITH_ARG drives the OP-TEE message protocol
- OPTEE_SMC_GET_SHM_CONFIG lets the driver and OP-TEE agree on which memory range to used for shared memory between Linux and OP-TEE.

The GlobalPlatform TEE Client API [5] is implemented on top of the generic TEE API.

Picture of the relationship between the different components in the OP-TEE architecture:

```
   User space                  Kernel                   Secure world
   ~~~~~~~~~~                  ~~~~~~                   ~~~~~~~~~~~~
+--------+                                             +------------
↪-+
| Client |                                             | Trusted    ␣
```

(continues on next page)

(continued from previous page)

```
 ↪ |
+--------+                                             |␣
 ↪Application |
   /\                                                  +------------
 ↪-+
   || +----------+                                           /\
   || |tee-      |                                           ||
   || |supplicant|                                           \/
   || +----------+                                     +------------
 ↪-+
   \/      /\                                          | TEE␣
 ↪Internal|
+-------+  ||                                          | API       ␣
 ↪ |
+ TEE   |  ||            +--------+--------+           +------------
 ↪-+
| Client|  ||            | TEE    | OP-TEE |           | OP-TEE     ␣
 ↪ |
| API   |  \/            | subsys | driver |           | Trusted OS␣
 ↪ |
+-------+----------------+----+-------+----+-----------+------------
 ↪-+
|      Generic TEE API        |       |     OP-TEE MSG             ␣
 ↪ |
|      IOCTL (TEE_IOC_*)      |       |     SMCCC (OPTEE_SMC_CALL_
 ↪*) |
+-----------------------------+       +----------------------------
 ↪-+
```

RPC (Remote Procedure Call) are requests from secure world to kernel driver or tee-supplicant. An RPC is identified by a special range of SMCCC return values from OPTEE_SMC_CALL_WITH_ARG. RPC messages which are intended for the kernel are handled by the kernel driver. Other RPC messages will be forwarded to tee-supplicant without further involvement of the driver, except switching shared memory buffer representation.

8.3.1 OP-TEE device enumeration

OP-TEE provides a pseudo Trusted Application: drivers/tee/optee/device.c in order to support device enumeration. In other words, OP-TEE driver invokes this application to retrieve a list of Trusted Applications which can be registered as devices on the TEE bus.

8.4 AMD-TEE driver

The AMD-TEE driver handles the communication with AMD’ s TEE environment. The TEE environment is provided by AMD Secure Processor.

The AMD Secure Processor (formerly called Platform Security Processor or PSP) is a dedicated processor that features ARM TrustZone technology, along with a software-based Trusted Execution Environment (TEE) designed to enable third-party Trusted Applications. This feature is currently enabled only for APUs.

The following picture shows a high level overview of AMD-TEE:

```
                                            |
   x86                                      |
                                            |
User space            (Kernel space)        |    AMD Secure␣
 ↪Processor (PSP)
~~~~~~~~~~            ~~~~~~~~~~~~~~        |    ~~~~~~~~~~~~~~~~~~~
 ↪~~~~~~~
                                            |
+--------+                                  |       +-------------+
| Client |                                  |       | Trusted     |
+--------+                                  |       | Application |
    /\                                      |       +-------------+
    ||                                      |             /\
    ||                                      |             ||
    ||                                      |             \/
    ||                                      |         +----------+
    ||                                      |         |   TEE    |
    ||                                      |         | Internal |
    \/                                      |         |   API    |
+---------+           +-----------+---------+         +----------+
| TEE     |           | TEE       | AMD-TEE |         | AMD-TEE  |
| Client  |           | subsystem | driver  |         | Trusted  |
| API     |           |           |         |         |   OS     |
+---------+-----------+----+------+---------+---------+----------+
|   Generic TEE API        |      | ASP     |      Mailbox       |
|   IOCTL (TEE_IOC_*)      |      | driver  | Register Protocol  |
+--------------------------+      +---------+--------------------+
```

At the lowest level (in x86), the AMD Secure Processor (ASP) driver uses the CPU to PSP mailbox regsister to submit commands to the PSP. The format of the command buffer is opaque to the ASP driver. It’ s role is to submit commands to the secure processor and return results to AMD-TEE driver. The interface between AMD-TEE driver and AMD Secure Processor driver can be found in [6].

The AMD-TEE driver packages the command buffer payload for processing in TEE. The command buffer format for the different TEE commands can be found in [7].

The TEE commands supported by AMD-TEE Trusted OS are:

- **TEE_CMD_ID_LOAD_TA - loads a Trusted Application (TA) binary into**
  TEE environment.

- TEE_CMD_ID_UNLOAD_TA - unloads TA binary from TEE environment.
- TEE_CMD_ID_OPEN_SESSION - opens a session with a loaded TA.
- TEE_CMD_ID_CLOSE_SESSION - closes session with loaded TA
- TEE_CMD_ID_INVOKE_CMD - invokes a command with loaded TA
- TEE_CMD_ID_MAP_SHARED_MEM - maps shared memory
- TEE_CMD_ID_UNMAP_SHARED_MEM - unmaps shared memory

AMD-TEE Trusted OS is the firmware running on AMD Secure Processor.

The AMD-TEE driver registers itself with TEE subsystem and implements the following driver function callbacks:

- get_version - returns the driver implementation id and capability.
- open - sets up the driver context data structure.
- release - frees up driver resources.
- open_session - loads the TA binary and opens session with loaded TA.
- close_session - closes session with loaded TA and unloads it.
- invoke_func - invokes a command with loaded TA.

cancel_req driver callback is not supported by AMD-TEE.

The GlobalPlatform TEE Client API [5] can be used by the user space (client) to talk to AMD's TEE. AMD's TEE provides a secure environment for loading, opening a session, invoking commands and clossing session with TA.

8.5 References

[1] https://github.com/OP-TEE/optee_os

[2] http://infocenter.arm.com/help/topic/com.arm.doc.den0028a/index.html

[3] drivers/tee/optee/optee_smc.h

[4] drivers/tee/optee/optee_msg.h

[5] http://www.globalplatform.org/specificationsdevice.asp look for
“TEE Client API Specification v1.0” and click download.

[6] include/linux/psp-tee.h

[7] drivers/tee/amdtee/amdtee_if.h

CHAPTER

NINE

XZ DATA COMPRESSION IN LINUX

9.1 Introduction

XZ is a general purpose data compression format with high compression ratio and relatively fast decompression. The primary compression algorithm (filter) is LZMA2. Additional filters can be used to improve compression ratio even further. E.g. Branch/Call/Jump (BCJ) filters improve compression ratio of executable data.

The XZ decompressor in Linux is called XZ Embedded. It supports the LZMA2 filter and optionally also BCJ filters. CRC32 is supported for integrity checking. The home page of XZ Embedded is at <https://tukaani.org/xz/embedded.html>, where you can find the latest version and also information about using the code outside the Linux kernel.

For userspace, XZ Utils provide a zlib-like compression library and a gzip-like command line tool. XZ Utils can be downloaded from <https://tukaani.org/xz/>.

9.2 XZ related components in the kernel

The xz_dec module provides XZ decompressor with single-call (buffer to buffer) and multi-call (stateful) APIs. The usage of the xz_dec module is documented in include/linux/xz.h.

The xz_dec_test module is for testing xz_dec. xz_dec_test is not useful unless you are hacking the XZ decompressor. xz_dec_test allocates a char device major dynamically to which one can write .xz files from userspace. The decompressed output is thrown away. Keep an eye on dmesg to see diagnostics printed by xz_dec_test. See the xz_dec_test source code for the details.

For decompressing the kernel image, initramfs, and initrd, there is a wrapper function in lib/decompress_unxz.c. Its API is the same as in other decompress_*.c files, which is defined in include/linux/decompress/generic.h.

scripts/xz_wrap.sh is a wrapper for the xz command line tool found from XZ Utils. The wrapper sets compression options to values suitable for compressing the kernel image.

For kernel makefiles, two commands are provided for use with $(call if_needed). The kernel image should be compressed with $(call if_needed,xzkern) which will use a BCJ filter and a big LZMA2 dictionary. It will also append a four-byte trailer containing the uncompressed size of the file, which is needed by the boot code.

Other things should be compressed with $(call if_needed,xzmisc) which will use no BCJ filter and 1 MiB LZMA2 dictionary.

9.3 Notes on compression options

Since the XZ Embedded supports only streams with no integrity check or CRC32, make sure that you don’ t use some other integrity check type when encoding files that are supposed to be decoded by the kernel. With liblzma, you need to use either LZMA_CHECK_NONE or LZMA_CHECK_CRC32 when encoding. With the xz command line tool, use –check=none or –check=crc32.

Using CRC32 is strongly recommended unless there is some other layer which will verify the integrity of the uncompressed data anyway. Double checking the integrity would probably be waste of CPU cycles. Note that the headers will always have a CRC32 which will be validated by the decoder; you can only change the integrity check type (or disable it) for the actual uncompressed data.

In userspace, LZMA2 is typically used with dictionary sizes of several megabytes. The decoder needs to have the dictionary in RAM, thus big dictionaries cannot be used for files that are intended to be decoded by the kernel. 1 MiB is probably the maximum reasonable dictionary size for in-kernel use (maybe more is OK for initramfs). The presets in XZ Utils may not be optimal when creating files for the kernel, so don’ t hesitate to use custom settings. Example:

```
xz --check=crc32 --lzma2=dict=512KiB inputfile
```

An exception to above dictionary size limitation is when the decoder is used in single-call mode. Decompressing the kernel itself is an example of this situation. In single-call mode, the memory usage doesn’t depend on the dictionary size, and it is perfectly fine to use a big dictionary: for maximum compression, the dictionary should be at least as big as the uncompressed data itself.

9.4 Future plans

Creating a limited XZ encoder may be considered if people think it is useful. LZMA2 is slower to compress than e.g. Deflate or LZO even at the fastest settings, so it isn’ t clear if LZMA2 encoder is wanted into the kernel.

Support for limited random-access reading is planned for the decompression code. I don’ t know if it could have any use in the kernel, but I know that it would be useful in some embedded projects outside the Linux kernel.

9.5 Conformance to the .xz file format specification

There are a couple of corner cases where things have been simplified at expense of detecting errors as early as possible. These should not matter in practice all, since they don't cause security issues. But it is good to know this if testing the code e.g. with the test files from XZ Utils.

9.6 Reporting bugs

Before reporting a bug, please check that it's not fixed already at upstream. See <https://tukaani.org/xz/embedded.html> to get the latest code.

Report bugs to <lasse.collin@tukaani.org> or visit #tukaani on Freenode and talk to Larhzu. I don't actively read LKML or other kernel-related mailing lists, so if there's something I should know, you should email to me personally or use IRC.

Don't bother Igor Pavlov with questions about the XZ implementation in the kernel or about XZ Utils. While these two implementations include essential code that is directly based on Igor Pavlov's code, these implementations aren't maintained nor supported by him.

CHAPTER

TEN

ATOMIC TYPES

```
On atomic types (atomic_t atomic64_t and atomic_long_t).

The atomic type provides an interface to the architecture's means of atomic
RMW operations between CPUs (atomic operations on MMIO are not supported and
can lead to fatal traps on some platforms).

API
---

The 'full' API consists of (atomic64_ and atomic_long_ prefixes omitted for
brevity):

Non-RMW ops:

  atomic_read(), atomic_set()
  atomic_read_acquire(), atomic_set_release()


RMW atomic operations:

Arithmetic:

  atomic_{add,sub,inc,dec}()
  atomic_{add,sub,inc,dec}_return{,_relaxed,_acquire,_release}()
  atomic_fetch_{add,sub,inc,dec}{,_relaxed,_acquire,_release}()


Bitwise:

  atomic_{and,or,xor,andnot}()
  atomic_fetch_{and,or,xor,andnot}{,_relaxed,_acquire,_release}()


Swap:

  atomic_xchg{,_relaxed,_acquire,_release}()
  atomic_cmpxchg{,_relaxed,_acquire,_release}()
  atomic_try_cmpxchg{,_relaxed,_acquire,_release}()


Reference count (but please see refcount_t):

  atomic_add_unless(), atomic_inc_not_zero()
  atomic_sub_and_test(), atomic_dec_and_test()


Misc:
  atomic_inc_and_test(), atomic_add_negative()
  atomic_dec_unless_positive(), atomic_inc_unless_negative()


Barriers:

  smp_mb__{before,after}_atomic()


TYPES (signed vs unsigned)
-----

While atomic_t, atomic_long_t and atomic64_t use int, long and s64
respectively (for hysterical raisins), the kernel uses -fno-strict-overflow
(which implies -fwrapv) and defines signed overflow to behave like
2s-complement.

Therefore, an explicitly unsigned variant of the atomic ops is strictly
unnecessary and we can simply cast, there is no UB.

There was a bug in UBSAN prior to GCC-8 that would generate UB warnings for
signed types.

With this we also conform to the C/C++ _Atomic behaviour and things like
P1236R1.


SEMANTICS
---------

Non-RMW ops:

The non-RMW ops are (typically) regular LOADs and STOREs and are canonically
implemented using READ_ONCE(), WRITE_ONCE(), smp_load_acquire() and
smp_store_release() respectively. Therefore, if you find yourself only using
the Non-RMW operations of atomic_t, you do not in fact need atomic_t at all
and are doing it wrong.

A note for the implementation of atomic_set{}() is that it must not break the
atomicity of the RMW ops. That is:

  C Atomic-RMW-ops-are-atomic-WRT-atomic_set

  {
    atomic_t v = ATOMIC_INIT(1);
  }

  P0(atomic_t *v)
  {
    (void)atomic_add_unless(v, 1, 0);
  }

  P1(atomic_t *v)
  {
    atomic_set(v, 0);
  }

  exists
  (v=2)

In this case we would expect the atomic_set() from CPU1 to either happen
before the atomic_add_unless(), in which case that latter one would no-op, or
_after_ in which case we'd overwrite its result. In no case is "2" a valid
```

outcome.

This is typically true on 'normal' platforms, where a regular competing STORE will invalidate a LL/SC or fail a CMPXCHG.

The obvious case where this is not so is when we need to implement atomic ops with a lock:

```
  CPU0                                          CPU1

  atomic_add_unless(v, 1, 0);
    lock();
    ret = READ_ONCE(v->counter); // == 1
                                                atomic_set(v, 0);
    if (ret != u)                                 WRITE_ONCE(v->counter, 0);
      WRITE_ONCE(v->counter, ret + 1);
    unlock();
```

the typical solution is to then implement atomic_set{}() with atomic_xchg().

RMW ops:

These come in various forms:

- plain operations without return value: atomic_{}()

- operations which return the modified value: atomic_{}_return()

  these are limited to the arithmetic operations because those are reversible. Bitops are irreversible and therefore the modified value is of dubious utility.

- operations which return the original value: atomic_fetch_{}()

- swap operations: xchg(), cmpxchg() and try_cmpxchg()

- misc; the special purpose operations that are commonly used and would, given the interface, normally be implemented using (try_)cmpxchg loops but are time critical and can, (typically) on LL/SC architectures, be more efficiently implemented.

All these operations are SMP atomic; that is, the operations (for a single atomic variable) can be fully ordered and no intermediate state is lost or visible.

ORDERING (go read memory-barriers.txt first)

The rule of thumb:

- non-RMW operations are unordered;

- RMW operations that have no return value are unordered;

- RMW operations that have a return value are fully ordered;

- RMW operations that are conditional are unordered on FAILURE, otherwise the above rules apply.

Except of course when an operation has an explicit ordering like:

```
 {}_relaxed: unordered
 {}_acquire: the R of the RMW (or atomic_read) is an ACQUIRE
 {}_release: the W of the RMW (or atomic_set)  is a  RELEASE
```

Where 'unordered' is against other memory locations. Address dependencies are not defeated.

Fully ordered primitives are ordered against everything prior and everything subsequent. Therefore a fully ordered primitive is like having an smp_mb() before and an smp_mb() after the primitive.

The barriers:

```
  smp_mb__{before,after}_atomic()
```

only apply to the RMW atomic ops and can be used to augment/upgrade the ordering inherent to the op. These barriers act almost like a full smp_mb(): smp_mb__before_atomic() orders all earlier accesses against the RMW op itself and all accesses following it, and smp_mb__after_atomic() orders all later accesses against the RMW op and all accesses preceding it. However, accesses between the smp_mb__{before,after}_atomic() and the RMW op are not ordered, so it is advisable to place the barrier right next to the RMW atomic op whenever possible.

These helper barriers exist because architectures have varying implicit ordering on their SMP atomic primitives. For example our TSO architectures provide full ordered atomics and these barriers are no-ops.

NOTE: when the atomic RmW ops are fully ordered, they should also imply a compiler barrier.

Thus:

```
  atomic_fetch_add();
```

is equivalent to:

```
  smp_mb__before_atomic();
  atomic_fetch_add_relaxed();
  smp_mb__after_atomic();
```

However the atomic_fetch_add() might be implemented more efficiently.

Further, while something like:

```
  smp_mb__before_atomic();
  atomic_dec(&X);
```

is a 'typical' RELEASE pattern, the barrier is strictly stronger than a RELEASE because it orders preceding instructions against both the read and write parts of the atomic_dec(), and against all following instructions as well. Similarly, something like:

```
  atomic_inc(&X);
  smp_mb__after_atomic();
```

is an ACQUIRE pattern (though very much not typical), but again the barrier is strictly stronger than ACQUIRE. As illustrated:

```
  C Atomic-RMW+mb__after_atomic-is-stronger-than-acquire

  {
  }

  P0(int *x, atomic_t *y)
  {
    r0 = READ_ONCE(*x);
    smp_rmb();
    r1 = atomic_read(y);
  }

  P1(int *x, atomic_t *y)
  {
    atomic_inc(y);
    smp_mb__after_atomic();
    WRITE_ONCE(*x, 1);
  }

  exists
  (0:r0=1 /\ 0:r1=0)
```

This should not happen; but a hypothetical atomic_inc_acquire() -- (void)atomic_fetch_inc_acquire() for instance -- would allow the outcome, because it would not order the W part of the RMW against the following WRITE_ONCE. Thus:

```
  P0                      P1

                          t = LL.acq *y (0)
                          t++;
                          *x = 1;
  r0 = *x (1)
  RMB
  r1 = *y (0)
                          SC *y, t;
```

is allowed.

CHAPTER

ELEVEN

ATOMIC BITOPS

```
=============
Atomic bitops
=============

While our bitmap_{}() functions are non-atomic, we have a number of operations
operating on single bits in a bitmap that are atomic.


API
---

The single bit operations are:

Non-RMW ops:

  test_bit()

RMW atomic operations without return value:

  {set,clear,change}_bit()
  clear_bit_unlock()

RMW atomic operations with return value:

  test_and_{set,clear,change}_bit()
  test_and_set_bit_lock()

Barriers:

  smp_mb__{before,after}_atomic()


All RMW atomic operations have a '__' prefixed variant which is non-atomic.


SEMANTICS
---------

Non-atomic ops:

In particular __clear_bit_unlock() suffers the same issue as atomic_set(),
which is why the generic version maps to clear_bit_unlock(), see atomic_t.txt.


RMW ops:

The test_and_{}_bit() operations return the original value of the bit.
```

(continues on next page)

(continued from previous page)

```
ORDERING
--------

Like with atomic_t, the rule of thumb is:

 - non-RMW operations are unordered;

 - RMW operations that have no return value are unordered;

 - RMW operations that have a return value are fully ordered.

 - RMW operations that are conditional are unordered on FAILURE,
   otherwise the above rules apply. In the case of test_and_set_bit_lock(),
   if the bit in memory is unchanged by the operation then it is deemed to have
   failed.

Except for a successful test_and_set_bit_lock() which has ACQUIRE semantics and
clear_bit_unlock() which has RELEASE semantics.

Since a platform only has a single means of achieving atomic operations
the same barriers as for atomic_t are used, see atomic_t.txt.
```

CHAPTER TWELVE

MEMORY BARRIERS

```
                         ============================
                         LINUX KERNEL MEMORY BARRIERS
                         ============================
```

```
By: David Howells <dhowells@redhat.com>
    Paul E. McKenney <paulmck@linux.ibm.com>
    Will Deacon <will.deacon@arm.com>
    Peter Zijlstra <peterz@infradead.org>
```

```
==========
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.

To repeat, this document is not a specification of what Linux expects from
hardware.

The purpose of this document is twofold:

 (1) to specify the minimum functionality that one can rely on for any
     particular barrier, and

 (2) to provide a guide as to how to use the barriers that are available.

Note that an architecture can provide more than the minimum requirement
for any particular barrier, but if the architecture provides less than
that, that architecture is incorrect.

Note also that it is possible that a barrier may be a no-op for an
architecture because the way that arch works renders an explicit barrier
unnecessary in that case.


========
CONTENTS
========
============================
ABSTRACT MEMORY ACCESS MODEL
============================

Consider the following abstract model of the system:
```

```
                            :                :
                            :                :
                            :                :
                +-------+   :   +--------+   :   +-------+
                |       |   :   |        |   :   |       |
                |       |   :   |        |   :   |       |
                | CPU 1 |<----->| Memory |<----->| CPU 2 |
                |       |   :   |        |   :   |       |
                |       |   :   |        |   :   |       |
                +-------+   :   +--------+   :   +-------+
                    ^       :       ^        :       ^
                    |       :       |        :       |
                    |       :       |        :       |
                    |       :       v        :       |
                    |       :   +--------+   :       |
                    |       :   |        |   :       |
                    |       :   |        |   :       |
                    +---------->| Device |<----------+
                            :   |        |   :
                            :   |        |   :
                            :   +--------+   :
                            :                :
```

Each CPU executes a program that generates memory access operations. In the abstract CPU, memory operation ordering is very relaxed, and a CPU may actually perform the memory operations in any order it likes, provided program causality appears to be maintained. Similarly, the compiler may also arrange the instructions it emits in any order it likes, provided it doesn't affect the apparent operation of the program.

So in the above diagram, the effects of the memory operations performed by a CPU are perceived by the rest of the system as the operations cross the interface between the CPU and rest of the system (the dotted lines).

For example, consider the following sequence of events:

```
        CPU 1           CPU 2
        =============== ===============
        { A == 1; B == 2 }
        A = 3;          x = B;
        B = 4;          y = A;
```

The set of accesses as seen by the memory system in the middle can be arranged in 24 different combinations:

```
        STORE A=3,      STORE B=4,      y=LOAD A->3,    x=LOAD B->4
        STORE A=3,      STORE B=4,      x=LOAD B->4,    y=LOAD A->3
        STORE A=3,      y=LOAD A->3,    STORE B=4,      x=LOAD B->4
        STORE A=3,      y=LOAD A->3,    x=LOAD B->2,    STORE B=4
        STORE A=3,      x=LOAD B->2,    STORE B=4,      y=LOAD A->3
        STORE A=3,      x=LOAD B->2,    y=LOAD A->3,    STORE B=4
        STORE B=4,      STORE A=3,      y=LOAD A->3,    x=LOAD B->4
        STORE B=4, ...
        ...
```

and can thus result in four different combinations of values:

```
        x == 2, y == 1
        x == 2, y == 3
        x == 4, y == 1
        x == 4, y == 3
```

Furthermore, the stores committed by a CPU to the memory system may not be perceived by the loads made by another CPU in the same order as the stores were committed.

As a further example, consider this sequence of events:

```
        CPU 1           CPU 2
        =============== ===============
        { A == 1, B == 2, C == 3, P == &A, Q == &C }
        B = 4;          Q = P;
        P = &B;         D = *Q;
```

There is an obvious data dependency here, as the value loaded into D depends on the address retrieved from P by CPU 2. At the end of the sequence, any of the following results are possible:

```
        (Q == &A) and (D == 1)
        (Q == &B) and (D == 2)
        (Q == &B) and (D == 4)
```

Note that CPU 2 will never try and load C into D because the CPU will load P into Q before issuing the load of *Q.

DEVICE OPERATIONS

Some devices present their control interfaces as collections of memory locations, but the order in which the control registers are accessed is very important. For instance, imagine an ethernet card with a set of internal registers that are accessed through an address port register (A) and a data port register (D). To read internal register 5, the following code might then be used:

```
        *A = 5;
        x = *D;
```

but this might show up as either of the following two sequences:

```
        STORE *A = 5, x = LOAD *D
        x = LOAD *D, STORE *A = 5
```

the second of which will almost certainly result in a malfunction, since it set the address _after_ attempting to read the register.

GUARANTEES

There are some minimal guarantees that may be expected of a CPU:

(*) On any given CPU, dependent memory accesses will be issued in order, with respect to itself. This means that for:

```
        Q = READ_ONCE(P); D = READ_ONCE(*Q);
```

the CPU will issue the following memory operations:

```
        Q = LOAD P, D = LOAD *Q
```

and always in that order. However, on DEC Alpha, READ_ONCE() also emits a memory-barrier instruction, so that a DEC Alpha CPU will instead issue the following memory operations:

```
Q = LOAD P, MEMORY_BARRIER, D = LOAD *Q, MEMORY_BARRIER
```

Whether on DEC Alpha or not, the READ_ONCE() also prevents compiler mischief.

(*) Overlapping loads and stores within a particular CPU will appear to be ordered within that CPU. This means that for:

```
a = READ_ONCE(*X); WRITE_ONCE(*X, b);
```

the CPU will only issue the following sequence of memory operations:

```
a = LOAD *X, STORE *X = b
```

And for:

```
WRITE_ONCE(*X, c); d = READ_ONCE(*X);
```

the CPU will only issue:

```
STORE *X = c, d = LOAD *X
```

(Loads and stores overlap if they are targeted at overlapping pieces of memory).

And there are a number of things that _must_ or _must_not_ be assumed:

(*) It _must_not_ be assumed that the compiler will do what you want with memory references that are not protected by READ_ONCE() and WRITE_ONCE(). Without them, the compiler is within its rights to do all sorts of "creative" transformations, which are covered in the COMPILER BARRIER section.

(*) It _must_not_ be assumed that independent loads and stores will be issued in the order given. This means that for:

```
X = *A; Y = *B; *D = Z;
```

we may get any of the following sequences:

```
X = LOAD *A,  Y = LOAD *B,  STORE *D = Z
X = LOAD *A,  STORE *D = Z, Y = LOAD *B
Y = LOAD *B,  X = LOAD *A,  STORE *D = Z
Y = LOAD *B,  STORE *D = Z, X = LOAD *A
STORE *D = Z, X = LOAD *A,  Y = LOAD *B
STORE *D = Z, Y = LOAD *B,  X = LOAD *A
```

(*) It _must_ be assumed that overlapping memory accesses may be merged or discarded. This means that for:

```
X = *A; Y = *(A + 4);
```

we may get any one of the following sequences:

```
X = LOAD *A; Y = LOAD *(A + 4);
Y = LOAD *(A + 4); X = LOAD *A;
{X, Y} = LOAD {*A, *(A + 4) };
```

And for:

```
        *A = X; *(A + 4) = Y;
```

we may get any of:

```
        STORE *A = X; STORE *(A + 4) = Y;
        STORE *(A + 4) = Y; STORE *A = X;
        STORE {*A, *(A + 4) } = {X, Y};
```

And there are anti-guarantees:

(*) These guarantees do not apply to bitfields, because compilers often generate code to modify these using non-atomic read-modify-write sequences. Do not attempt to use bitfields to synchronize parallel algorithms.

(*) Even in cases where bitfields are protected by locks, all fields in a given bitfield must be protected by one lock. If two fields in a given bitfield are protected by different locks, the compiler's non-atomic read-modify-write sequences can cause an update to one field to corrupt the value of an adjacent field.

(*) These guarantees apply only to properly aligned and sized scalar variables. "Properly sized" currently means variables that are the same size as "char", "short", "int" and "long". "Properly aligned" means the natural alignment, thus no constraints for "char", two-byte alignment for "short", four-byte alignment for "int", and either four-byte or eight-byte alignment for "long", on 32-bit and 64-bit systems, respectively. Note that these guarantees were introduced into the C11 standard, so beware when using older pre-C11 compilers (for example, gcc 4.6). The portion of the standard containing this guarantee is Section 3.14, which defines "memory location" as follows:

> memory location
>
> either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width
>
> NOTE 1: Two threads of execution can update and access separate memory locations without interfering with each other.
>
> NOTE 2: A bit-field and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields, if one is declared inside a nested structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two bit-fields in the same structure if all members declared between them are also bit-fields, no matter what the sizes of those intervening bit-fields happen to be.

=========================
WHAT ARE MEMORY BARRIERS?
=========================

As can be seen above, independent memory operations are effectively performed in random order, but this can be a problem for CPU-CPU interaction and for I/O. What is required is some way of intervening to instruct the compiler and the CPU to restrict the order.

Memory barriers are such interventions. They impose a perceived partial ordering over the memory operations on either side of the barrier.

Such enforcement is important because the CPUs and other devices in a system can use a variety of tricks to improve performance, including reordering, deferral and combination of memory operations; speculative loads; speculative branch prediction and various types of caching. Memory barriers are used to override or suppress these tricks, allowing the code to sanely control the interaction of multiple CPUs and/or devices.

VARIETIES OF MEMORY BARRIER

Memory barriers come in four basic varieties:

(1) Write (or store) memory barriers.

A write memory barrier gives a guarantee that all the STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other components of the system.

A write barrier is a partial ordering on stores only; it is not required to have any effect on loads.

A CPU can be viewed as committing a sequence of store operations to the memory system as time progresses. All stores _before_ a write barrier will occur _before_ all the stores after the write barrier.

[!] Note that write barriers should normally be paired with read or data dependency barriers; see the "SMP barrier pairing" subsection.

(2) Data dependency barriers.

A data dependency barrier is a weaker form of read barrier. In the case where two loads are performed such that the second depends on the result of the first (eg: the first load retrieves the address to which the second load will be directed), a data dependency barrier would be required to make sure that the target of the second load is updated after the address obtained by the first load is accessed.

A data dependency barrier is a partial ordering on interdependent loads only; it is not required to have any effect on stores, independent loads or overlapping loads.

As mentioned in (1), the other CPUs in the system can be viewed as committing sequences of stores to the memory system that the CPU being considered can then perceive. A data dependency barrier issued by the CPU under consideration guarantees that for any load preceding it, if that load touches one of a sequence of stores from another CPU, then by the time the barrier completes, the effects of all the stores prior to that touched by the load will be perceptible to any loads issued after the data dependency barrier.

See the "Examples of memory barrier sequences" subsection for diagrams showing the ordering constraints.

[!] Note that the first load really has to have a _data_ dependency and not a control dependency. If the address for the second load is dependent on the first load, but the dependency is through a conditional rather than actually loading the address itself, then it's a _control_ dependency and a full read barrier or better is required. See the "Control dependencies" subsection for more information.

[!] Note that data dependency barriers should normally be paired with write barriers; see the "SMP barrier pairing" subsection.

(3) Read (or load) memory barriers.

A read barrier is a data dependency barrier plus a guarantee that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system.

A read barrier is a partial ordering on loads only; it is not required to have any effect on stores.

Read memory barriers imply data dependency barriers, and so can substitute for them.

[!] Note that read barriers should normally be paired with write barriers; see the "SMP barrier pairing" subsection.

(4) General memory barriers.

A general memory barrier gives a guarantee that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other components of the system.

A general memory barrier is a partial ordering over both loads and stores.

General memory barriers imply both read and write memory barriers, and so can substitute for either.

And a couple of implicit varieties:

(5) ACQUIRE operations.

This acts as a one-way permeable barrier. It guarantees that all memory operations after the ACQUIRE operation will appear to happen after the ACQUIRE operation with respect to the other components of the system. ACQUIRE operations include LOCK operations and both smp_load_acquire() and smp_cond_load_acquire() operations.

Memory operations that occur before an ACQUIRE operation may appear to happen after it completes.

An ACQUIRE operation should almost always be paired with a RELEASE operation.

(6) RELEASE operations.

This also acts as a one-way permeable barrier. It guarantees that all memory operations before the RELEASE operation will appear to happen before the RELEASE operation with respect to the other components of the system. RELEASE operations include UNLOCK operations and smp_store_release() operations.

Memory operations that occur after a RELEASE operation may appear to happen before it completes.

The use of ACQUIRE and RELEASE operations generally precludes the need for other sorts of memory barrier. In addition, a RELEASE+ACQUIRE pair is -not- guaranteed to act as a full memory barrier. However, after an ACQUIRE on a given variable, all memory accesses preceding any prior RELEASE on that same variable are guaranteed to be visible. In other words, within a given variable's critical section, all accesses of all previous critical sections for that variable are guaranteed to have completed.

This means that ACQUIRE acts as a minimal "acquire" operation and RELEASE acts as a minimal "release" operation.

A subset of the atomic operations described in atomic_t.txt have ACQUIRE and RELEASE variants in addition to fully-ordered and relaxed (no barrier semantics) definitions. For compound atomics performing both a load and a store, ACQUIRE semantics apply only to the load and RELEASE semantics apply only to the store portion of the operation.

Memory barriers are only required where there's a possibility of interaction between two CPUs or between a CPU and a device. If it can be guaranteed that there won't be any such interaction in any particular piece of code, then memory barriers are unnecessary in that piece of code.

Note that these are the _minimum_ guarantees. Different architectures may give more substantial guarantees, but they may _not_ be relied upon outside of arch specific code.

WHAT MAY NOT BE ASSUMED ABOUT MEMORY BARRIERS?

There are certain things that the Linux kernel memory barriers do not guarantee:

(*) There is no guarantee that any of the memory accesses specified before a memory barrier will be _complete_ by the completion of a memory barrier instruction; the barrier can be considered to draw a line in that CPU's access queue that accesses of the appropriate type may not cross.

(*) There is no guarantee that issuing a memory barrier on one CPU will have any direct effect on another CPU or any other hardware in the system. The indirect effect will be the order in which the second CPU sees the effects of the first CPU's accesses occur, but see the next point:

(*) There is no guarantee that a CPU will see the correct order of effects from a second CPU's accesses, even _if_ the second CPU uses a memory barrier, unless the first CPU _also_ uses a matching memory barrier (see the subsection on "SMP Barrier Pairing").

(*) There is no guarantee that some intervening piece of off-the-CPU hardware[*] will not reorder the memory accesses. CPU cache coherency mechanisms should propagate the indirect effects of a memory barrier between CPUs, but might not do so in order.

[*] For information on bus mastering DMA and coherency please read:

```
Documentation/driver-api/pci/pci.rst
Documentation/core-api/dma-api-howto.rst
Documentation/core-api/dma-api.rst
```

DATA DEPENDENCY BARRIERS (HISTORICAL)

As of v4.15 of the Linux kernel, an smp_mb() was added to READ_ONCE() for DEC Alpha, which means that about the only people who need to pay attention to this section are those working on DEC Alpha architecture-specific code and those working on READ_ONCE() itself. For those who need it, and for those who are interested in the history, here is the story of data-dependency barriers.

The usage requirements of data dependency barriers are a little subtle, and it's not always obvious that they're needed. To illustrate, consider the following sequence of events:

```
        CPU 1                 CPU 2
        ===============       ===============
        { A == 1, B == 2, C == 3, P == &A, Q == &C }
        B = 4;
        <write barrier>
        WRITE_ONCE(P, &B);
                              Q = READ_ONCE(P);
                              D = *Q;
```

There's a clear data dependency here, and it would seem that by the end of the sequence, Q must be either &A or &B, and that:

```
        (Q == &A) implies (D == 1)
        (Q == &B) implies (D == 4)
```

But! CPU 2's perception of P may be updated _before_ its perception of B, thus leading to the following situation:

```
        (Q == &B) and (D == 2) ????
```

While this may seem like a failure of coherency or causality maintenance, it isn't, and this behaviour can be observed on certain real CPUs (such as the DEC Alpha).

To deal with this, a data dependency barrier or better must be inserted between the address load and the data load:

```
        CPU 1                 CPU 2
        ===============       ===============
        { A == 1, B == 2, C == 3, P == &A, Q == &C }
        B = 4;
        <write barrier>
        WRITE_ONCE(P, &B);
                              Q = READ_ONCE(P);
                              <data dependency barrier>
                              D = *Q;
```

This enforces the occurrence of one of the two implications, and prevents the third possibility from arising.

[!] Note that this extremely counterintuitive situation arises most easily on machines with split caches, so that, for example, one cache bank processes even-numbered cache lines and the other bank processes odd-numbered cache lines. The pointer P might be stored in an odd-numbered cache line, and the variable B might be stored in an even-numbered cache line. Then, if the even-numbered bank of the reading CPU's cache is extremely busy while the odd-numbered bank is idle, one can see the new value of the pointer P (&B), but the old value of the variable B (2).

A data-dependency barrier is not required to order dependent writes because the CPUs that the Linux kernel supports don't do writes until they are certain (1) that the write will actually happen, (2) of the location of the write, and (3) of the value to be written. But please carefully read the "CONTROL DEPENDENCIES" section and the Documentation/RCU/rcu_dereference.rst file: The compiler can and does break dependencies in a great many highly creative ways.

```
        CPU 1                 CPU 2
        ===============       ===============
        { A == 1, B == 2, C = 3, P == &A, Q == &C }
        B = 4;
        <write barrier>
        WRITE_ONCE(P, &B);
                              Q = READ_ONCE(P);
                              WRITE_ONCE(*Q, 5);
```

Therefore, no data-dependency barrier is required to order the read into Q with the store into *Q. In other words, this outcome is prohibited, even without a data-dependency barrier:

```
        (Q == &B) && (B == 4)
```

Please note that this pattern should be rare. After all, the whole point of dependency ordering is to -prevent- writes to the data structure, along with the expensive cache misses associated with those writes. This pattern can be used to record rare error conditions and the like, and the CPUs' naturally occurring ordering prevents such records from being lost.

Note well that the ordering provided by a data dependency is local to the CPU containing it. See the section on "Multicopy atomicity" for more information.

The data dependency barrier is very important to the RCU system, for example. See rcu_assign_pointer() and rcu_dereference() in include/linux/rcupdate.h. This permits the current target of an RCU'd pointer to be replaced with a new modified target, without the replacement target appearing to be incompletely initialised.

See also the subsection on "Cache Coherency" for a more thorough example.

CONTROL DEPENDENCIES

Control dependencies can be a bit tricky because current compilers do not understand them. The purpose of this section is to help you prevent the compiler's ignorance from breaking your code.

A load-load control dependency requires a full read memory barrier, not simply a data dependency barrier to make it work correctly. Consider the following bit of code:

```
        q = READ_ONCE(a);
        if (q) {
                <data dependency barrier>  /* BUG: No data dependency!!! */
                p = READ_ONCE(b);
        }
```

This will not have the desired effect because there is no actual data dependency, but rather a control dependency that the CPU may short-circuit by attempting to predict the outcome in advance, so that other CPUs see the load from b as having happened before the load from a. In such a case what's actually required is:

```
        q = READ_ONCE(a);
        if (q) {
                <read barrier>
                p = READ_ONCE(b);
        }
```

However, stores are not speculated. This means that ordering -is- provided for load-store control dependencies, as in the following example:

```
        q = READ_ONCE(a);
        if (q) {
                WRITE_ONCE(b, 1);
        }
```

Control dependencies pair normally with other types of barriers. That said, please note that neither READ_ONCE() nor WRITE_ONCE() are optional! Without the READ_ONCE(), the compiler might combine the load from 'a' with other loads from 'a'. Without the WRITE_ONCE(), the compiler might combine the store to 'b' with other stores to 'b'. Either can result in highly counterintuitive effects on ordering.

Worse yet, if the compiler is able to prove (say) that the value of variable 'a' is always non-zero, it would be well within its rights to optimize the original example by eliminating the "if" statement as follows:

```
        q = a;
        b = 1;  /* BUG: Compiler and CPU can both reorder!!! */
```

So don't leave out the READ_ONCE().

It is tempting to try to enforce ordering on identical stores on both branches of the "if" statement as follows:

```
        q = READ_ONCE(a);
        if (q) {
                barrier();
                WRITE_ONCE(b, 1);
                do_something();
        } else {
                barrier();
                WRITE_ONCE(b, 1);
                do_something_else();
        }
```

Unfortunately, current compilers will transform this as follows at high optimization levels:

```
        q = READ_ONCE(a);
        barrier();
        WRITE_ONCE(b, 1);  /* BUG: No ordering vs. load from a!!! */
        if (q) {
                /* WRITE_ONCE(b, 1); -- moved up, BUG!!! */
                do_something();
        } else {
                /* WRITE_ONCE(b, 1); -- moved up, BUG!!! */
                do_something_else();
        }
```

Now there is no conditional between the load from 'a' and the store to 'b', which means that the CPU is within its rights to reorder them: The conditional is absolutely required, and must be present in the assembly code even after all compiler optimizations have been applied. Therefore, if you need ordering in this example, you need explicit memory barriers, for example, smp_store_release():

```
        q = READ_ONCE(a);
        if (q) {
                smp_store_release(&b, 1);
                do_something();
        } else {
                smp_store_release(&b, 1);
                do_something_else();
        }
```

In contrast, without explicit memory barriers, two-legged-if control ordering is guaranteed only when the stores differ, for example:

```
        q = READ_ONCE(a);
        if (q) {
                WRITE_ONCE(b, 1);
                do_something();
        } else {
                WRITE_ONCE(b, 2);
                do_something_else();
        }
```

The initial READ_ONCE() is still required to prevent the compiler from proving the value of 'a'.

In addition, you need to be careful what you do with the local variable 'q', otherwise the compiler might be able to guess the value and again remove the needed conditional. For example:

```
        q = READ_ONCE(a);
        if (q % MAX) {
                WRITE_ONCE(b, 1);
                do_something();
        } else {
                WRITE_ONCE(b, 2);
                do_something_else();
        }
```

If MAX is defined to be 1, then the compiler knows that (q % MAX) is equal to zero, in which case the compiler is within its rights to transform the above code into the following:

```
        q = READ_ONCE(a);
        WRITE_ONCE(b, 2);
        do_something_else();
```

Given this transformation, the CPU is not required to respect the ordering between the load from variable 'a' and the store to variable 'b'. It is tempting to add a barrier(), but this does not help. The conditional is gone, and the barrier won't bring it back. Therefore, if you are relying on this ordering, you should make sure that MAX is greater than one, perhaps as follows:

```
        q = READ_ONCE(a);
        BUILD_BUG_ON(MAX <= 1); /* Order load from a with store to b. */
        if (q % MAX) {
                WRITE_ONCE(b, 1);
                do_something();
        } else {
                WRITE_ONCE(b, 2);
                do_something_else();
        }
```

Please note once again that the stores to 'b' differ. If they were identical, as noted earlier, the compiler could pull this store outside of the 'if' statement.

You must also be careful not to rely too much on boolean short-circuit evaluation. Consider this example:

```
        q = READ_ONCE(a);
        if (q || 1 > 0)
                WRITE_ONCE(b, 1);
```

Because the first condition cannot fault and the second condition is always true, the compiler can transform this example as following, defeating control dependency:

```
        q = READ_ONCE(a);
        WRITE_ONCE(b, 1);
```

This example underscores the need to ensure that the compiler cannot out-guess your code. More generally, although READ_ONCE() does force the compiler to actually emit code for a given load, it does not force the compiler to use the results.

In addition, control dependencies apply only to the then-clause and else-clause of the if-statement in question. In particular, it does not necessarily apply to code following the if-statement:

```
        q = READ_ONCE(a);
        if (q) {
                WRITE_ONCE(b, 1);
        } else {
                WRITE_ONCE(b, 2);
        }
        WRITE_ONCE(c, 1);  /* BUG: No ordering against the read from 'a'. */
```

It is tempting to argue that there in fact is ordering because the compiler cannot reorder volatile accesses and also cannot reorder the writes to 'b' with the condition. Unfortunately for this line of reasoning, the compiler might compile the two writes to 'b' as conditional-move instructions, as in this fanciful pseudo-assembly language:

```
        ld r1,a
        cmp r1,$0
        cmov,ne r4,$1
        cmov,eq r4,$2
        st r4,b
        st $1,c
```

A weakly ordered CPU would have no dependency of any sort between the load from 'a' and the store to 'c'. The control dependencies would extend only to the pair of cmov instructions and the store depending on them. In short, control dependencies apply only to the stores in the then-clause and else-clause of the if-statement in question (including functions invoked by those two clauses), not to code following that if-statement.

Note well that the ordering provided by a control dependency is local to the CPU containing it. See the section on "Multicopy atomicity" for more information.

In summary:

(*) Control dependencies can order prior loads against later stores. However, they do -not- guarantee any other sort of ordering: Not prior loads against later loads, nor prior stores against later anything. If you need these other forms of ordering, use smp_rmb(), smp_wmb(), or, in the case of prior stores and later loads, smp_mb().

(*) If both legs of the "if" statement begin with identical stores to the same variable, then those stores must be ordered, either by preceding both of them with smp_mb() or by using smp_store_release() to carry out the stores. Please note that it is -not- sufficient to use barrier() at beginning of each leg of the "if" statement because, as shown by the example above, optimizing compilers can destroy the control dependency while respecting the letter of the barrier() law.

(*) Control dependencies require at least one run-time conditional between the prior load and the subsequent store, and this conditional must involve the prior load. If the compiler is able to optimize the conditional away, it will have also optimized away the ordering. Careful use of READ_ONCE() and WRITE_ONCE() can help to preserve the needed conditional.

(*) Control dependencies require that the compiler avoid reordering the dependency into nonexistence. Careful use of READ_ONCE() or atomic{,64}_read() can help to preserve your control dependency. Please see the COMPILER BARRIER section for more information.

(*) Control dependencies apply only to the then-clause and else-clause of the if-statement containing the control dependency, including any functions that these two clauses call. Control dependencies do -not- apply to code following the if-statement containing the control dependency.

(*) Control dependencies pair normally with other types of barriers.

(*) Control dependencies do -not- provide multicopy atomicity. If you need all the CPUs to see a given store at the same time, use smp_mb().

(*) Compilers do not understand control dependencies. It is therefore your job to ensure that they do not break your code.

SMP BARRIER PAIRING

When dealing with CPU-CPU interactions, certain types of memory barrier should always be paired. A lack of appropriate pairing is almost certainly an error.

General barriers pair with each other, though they also pair with most other types of barriers, albeit without multicopy atomicity. An acquire barrier pairs with a release barrier, but both may also pair with other barriers, including of course general barriers. A write barrier pairs with a data dependency barrier, a control dependency, an acquire barrier, a release barrier, a read barrier, or a general barrier. Similarly a read barrier, control dependency, or a data dependency barrier pairs with a write barrier, an acquire barrier, a release barrier, or a general barrier:

```
        CPU 1                 CPU 2
        ===============       ===============
        WRITE_ONCE(a, 1);
        <write barrier>
        WRITE_ONCE(b, 2);     x = READ_ONCE(b);
                              <read barrier>
                              y = READ_ONCE(a);
```

Or:

```
        CPU 1                 CPU 2
        ===============       ===============================
        a = 1;
        <write barrier>
        WRITE_ONCE(b, &a);    x = READ_ONCE(b);
                              <data dependency barrier>
                              y = *x;
```

Or even:

```
        CPU 1                 CPU 2
        ===============       ===============================
        r1 = READ_ONCE(y);
        <general barrier>
        WRITE_ONCE(x, 1);     if (r2 = READ_ONCE(x)) {
                                 <implicit control dependency>
                                 WRITE_ONCE(y, 1);
                              }

        assert(r1 == 0 || r2 == 0);
```

Basically, the read barrier always has to be there, even though it can be of the "weaker" type.

[!] Note that the stores before the write barrier would normally be expected to match the loads after the read barrier or the data dependency barrier, and vice versa:

```
        CPU 1                               CPU 2
        ===================                 ===================
        WRITE_ONCE(a, 1);    }----   --->{  v = READ_ONCE(c);
        WRITE_ONCE(b, 2);    }    \ /    {  w = READ_ONCE(d);
        <write barrier>            \        <read barrier>
        WRITE_ONCE(c, 3);    }    / \    {  x = READ_ONCE(a);
        WRITE_ONCE(d, 4);    }----   --->{  y = READ_ONCE(b);
```

EXAMPLES OF MEMORY BARRIER SEQUENCES

Firstly, write barriers act as partial orderings on store operations. Consider the following sequence of events:

```
        CPU 1
        =======================
        STORE A = 1
        STORE B = 2
        STORE C = 3
        <write barrier>
        STORE D = 4
        STORE E = 5
```

This sequence of events is committed to the memory coherence system in an order that the rest of the system might perceive as the unordered set of { STORE A, STORE B, STORE C } all occurring before the unordered set of { STORE D, STORE E }:

```
        +-------+       :      :
        |       |       +------+
        |       |------>| C=3  |     }     /\
        |       |  :    +------+     }-----  \  -----> Events perceptible to
        |       |  :    | A=1  |     }        \/       the rest of the system
        |       |  :    +------+     }
        | CPU 1 |  :    | B=2  |     }
        |       |       +------+     }
        |       |   wwwwwwwwwwwwwwww }   <--- At this point the write barrier
        |       |       +------+     }        requires all stores prior to the
        |       |  :    | E=5  |     }        barrier to be committed before
        |       |  :    +------+     }        further stores may take place
        |       |------>| D=4  |     }
        |       |       +------+
        +-------+       :      :
                           |
                           | Sequence in which stores are committed to the
                           | memory system by CPU 1
                           V
```

Secondly, data dependency barriers act as partial orderings on data-dependent loads. Consider the following sequence of events:

```
        CPU 1                   CPU 2
        ======================= =======================
                { B = 7; X = 9; Y = 8; C = &Y }
        STORE A = 1
        STORE B = 2
        <write barrier>
        STORE C = &B            LOAD X
        STORE D = 4             LOAD C (gets &B)
                                LOAD *C (reads B)
```

Without intervention, CPU 2 may perceive the events on CPU 1 in some effectively random order, despite the write barrier issued by CPU 1:

```
        +-------+       :      :                :       :
        |       |       +------+                +-------+  | Sequence of update
        |       |------>| B=2  |-----       --->| Y->8  |  | of perception on
        |       |  :    +------+     \          +-------+  | CPU 2
        | CPU 1 |  :    | A=1  |      \     --->| C->&Y |  V
        |       |       +------+       |        +-------+
        |       |   wwwwwwwwwwwwwwww   |        :       :
        |       |       +------+       |        :       :
        |       |  :    | C=&B |---    |        :       :       +-------+
        |       |  :    +------+   \   |        +-------+       |       |
        |       |------>| D=4  |    ----------->| C->&B |------>|       |
        |       |       +------+       |        +-------+       |       |
        +-------+       :      :       |        :       :       |       |
                                       |        :       :       |       |
                                       |        :       :       | CPU 2 |
                                       |        +-------+       |       |
            Apparently incorrect --->  |        | B->7  |------>|       |
            perception of B (!)        |        +-------+       |       |
                                       |        :       :       |       |
                                       |        +-------+       |       |
            The load of X holds --->    \       | X->9  |------>|       |
            up the maintenance           \      +-------+       |       |
            of coherence of B             ----->| B->2  |       +-------+
                                                +-------+
                                                :       :
```

In the above example, CPU 2 perceives that B is 7, despite the load of *C (which would be B) coming after the LOAD of C.

If, however, a data dependency barrier were to be placed between the load of C and the load of *C (ie: B) on CPU 2:

```
        CPU 1                   CPU 2
        ======================= =======================
                { B = 7; X = 9; Y = 8; C = &Y }
        STORE A = 1
        STORE B = 2
        <write barrier>
        STORE C = &B            LOAD X
        STORE D = 4             LOAD C (gets &B)
                                <data dependency barrier>
                                LOAD *C (reads B)
```

then the following will occur:

```
        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| B=2  |-----       --->| Y->8  |
        |       |  :    +------+     \          +-------+
        | CPU 1 |  :    | A=1  |      \     --->| C->&Y |
        |       |       +------+       |        +-------+
        |       |   wwwwwwwwwwwwwwww   |        :       :
        |       |       +------+       |        :       :
        |       |  :    | C=&B |---    |        :       :       +-------+
        |       |  :    +------+   \   |        +-------+       |       |
        |       |------>| D=4  |    ----------->| C->&B |------>|       |
        |       |       +------+       |        +-------+       |       |
        +-------+       :      :       |        :       :       |       |
                                       |        :       :       |       |
                                       |        :       :       | CPU 2 |
                                       |        +-------+       |       |
                                       |        | X->9  |------>|       |
                                       |        +-------+       |       |
          Makes sure all effects --->   \   ddddddddddddddddd   |       |
          prior to the store of C        \      +-------+       |       |
          are perceptible to              ----->| B->2  |------>|       |
          subsequent loads                      +-------+       |       |
                                                :       :       +-------+
```

And thirdly, a read barrier acts as a partial order on loads. Consider the following sequence of events:

```
        CPU 1                   CPU 2
        ======================= =======================
                { A = 0, B = 9 }
        STORE A=1
        <write barrier>
        STORE B=2
                                LOAD B
                                LOAD A
```

Without intervention, CPU 2 may then choose to perceive the events on CPU 1 in some effectively random order, despite the write barrier issued by CPU 1:

```
        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| A=1  |------      --->| A->0  |
        |       |       +------+      \         +-------+
        | CPU 1 |   wwwwwwwwwwwwwwww   \    --->| B->9  |
        |       |       +------+        |       +-------+
        |       |------>| B=2  |---     |       :       :
        |       |       +------+   \    |       :       :       +-------+
        +-------+       :      :    \   |       +-------+       |       |
                                     ---------->| B->2  |------>|       |
                                        |       +-------+       | CPU 2 |
                                        |       | A->0  |------>|       |
                                        |       +-------+       |       |
                                        |       :       :       +-------+
                                         \      :       :
                                          \     +-------+
                                           ---->| A->1  |
                                                +-------+
                                                :       :
```

If, however, a read barrier were to be placed between the load of B and the load of A on CPU 2:

```
        CPU 1                   CPU 2
        ======================= =======================
                { A = 0, B = 9 }
        STORE A=1
        <write barrier>
        STORE B=2
                                LOAD B
                                <read barrier>
                                LOAD A
```

then the partial ordering imposed by CPU 1 will be perceived correctly by CPU 2:

```
        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| A=1  |------      --->| A->0  |
        |       |       +------+      \         +-------+
        | CPU 1 |   wwwwwwwwwwwwwwww   \    --->| B->9  |
        |       |       +------+        |       +-------+
        |       |------>| B=2  |---     |       :       :
        |       |       +------+   \    |       :       :       +-------+
        +-------+       :      :    \   |       +-------+       |       |
                                     ---------->| B->2  |------>|       |
                                        |       +-------+       | CPU 2 |
                                        |       :       :       |       |
                                        |       :       :       |       |
          At this point the read ---->   \  rrrrrrrrrrrrrrrrr   |       |
          barrier causes all effects      \     +-------+       |       |
          prior to the storage of B        ---->| A->1  |------>|       |
          to be perceptible to CPU 2            +-------+       |       |
                                                :       :       +-------+
```

To illustrate this more completely, consider what could happen if the code

```
contained a load of A either side of the read barrier:

        CPU 1                   CPU 2
        ======================= =======================
                { A = 0, B = 9 }
        STORE A=1
        <write barrier>
        STORE B=2
                                LOAD B
                                LOAD A [first load of A]
                                <read barrier>
                                LOAD A [second load of A]

Even though the two loads of A both occur after the load of B, they may both
come up with different values:

        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| A=1  |------      --->| A->0  |
        |       |       +------+      \         +-------+
        | CPU 1 |   wwwwwwwwwwwwwwww   \    --->| B->9  |
        |       |       +------+        |       +-------+
        |       |------>| B=2  |---     |       :       :
        |       |       +------+   \    |       :       :       +-------+
        +-------+       :      :    \   |       +-------+       |       |
                                     ---------->| B->2  |------>|       |
                                        |       +-------+       | CPU 2 |
                                        |       :       :       |       |
                                        |       :       :       |       |
                                        |       +-------+       |       |
                                        |       | A->0  |------>| 1st   |
                                        |       +-------+       |       |
          At this point the read ---->   \  rrrrrrrrrrrrrrrrr   |       |
          barrier causes all effects      \     +-------+       |       |
          prior to the storage of B        ---->| A->1  |------>| 2nd   |
          to be perceptible to CPU 2            +-------+       |       |
                                                :       :       +-------+


But it may be that the update to A from CPU 1 becomes perceptible to CPU 2
before the read barrier completes anyway:

        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| A=1  |------      --->| A->0  |
        |       |       +------+      \         +-------+
        | CPU 1 |   wwwwwwwwwwwwwwww   \    --->| B->9  |
        |       |       +------+        |       +-------+
        |       |------>| B=2  |---     |       :       :
        |       |       +------+   \    |       :       :       +-------+
        +-------+       :      :    \   |       +-------+       |       |
                                     ---------->| B->2  |------>|       |
                                        |       +-------+       | CPU 2 |
                                        |       :       :       |       |
                                         \      :       :       |       |
                                          \     +-------+       |       |
                                           ---->| A->1  |------>| 1st   |
                                                +-------+       |       |
                                            rrrrrrrrrrrrrrrrr   |       |
                                                +-------+       |       |
                                                | A->1  |------>| 2nd   |
                                                +-------+       |       |
                                                :       :       +-------+
```

The guarantee is that the second load will always come up with A == 1 if the load of B came up with B == 2. No such guarantee exists for the first load of A; that may come up with either A == 0 or A == 1.

READ MEMORY BARRIERS VS LOAD SPECULATION

Many CPUs speculate with loads: that is they see that they will need to load an item from memory, and they find a time where they're not using the bus for any other loads, and so do the load in advance - even though they haven't actually got to that point in the instruction execution flow yet. This permits the actual load instruction to potentially complete immediately because the CPU already has the value to hand.

It may turn out that the CPU didn't actually need the value - perhaps because a branch circumvented the load - in which case it can discard the value or just cache it for later use.

Consider:

```
        CPU 1                   CPU 2
        ======================= =======================
                                LOAD B
                                DIVIDE          } Divide instructions generally
                                DIVIDE          } take a long time to perform
                                LOAD A
```

Which might appear as this:

```
                                                :       :       +-------+
                                                +-------+       |       |
                                            --->| B->2  |------>|       |
                                                +-------+       | CPU 2 |
                                                :       :DIVIDE |       |
                                                +-------+       |       |
        The CPU being busy doing a --->     --->| A->0  |~~~~   |       |
        division speculates on the              +-------+   ~   |       |
        LOAD of A                               :       :   ~   |       |
                                                :       :DIVIDE |       |
                                                :       :   ~   |       |
        Once the divisions are complete -->     :       :   ~-->|       |
        the CPU can then perform the            :       :       |       |
        LOAD with immediate effect              :       :       +-------+
```

Placing a read barrier or a data dependency barrier just before the second load:

```
        CPU 1                   CPU 2
        ======================= =======================
                                LOAD B
                                DIVIDE
                                DIVIDE
                                <read barrier>
                                LOAD A
```

will force any value speculatively obtained to be reconsidered to an extent dependent on the type of barrier used. If there was no change made to the speculated memory location, then the speculated value will just be used:

```
                                                :       :       +-------+
                                                +-------+       |       |
                                            --->| B->2  |------>|       |
                                                +-------+       | CPU 2 |
                                                :       :DIVIDE |       |
                                                +-------+       |       |
        The CPU being busy doing a --->     --->| A->0  |~~~~   |       |
        division speculates on the              +-------+   ~   |       |
        LOAD of A                               :       :   ~   |       |
                                                :       :DIVIDE |       |
                                                :       :   ~   |       |
                                                :       :   ~   |       |
                                            rrrrrrrrrrrrrrrr~   |       |
                                                :       :   ~   |       |
                                                :       :   ~-->|       |
                                                :       :       |       |
                                                :       :       +-------+
```

but if there was an update or an invalidation from another CPU pending, then the speculation will be cancelled and the value reloaded:

```
                                                :       :       +-------+
                                                +-------+       |       |
                                            --->| B->2  |------>|       |
                                                +-------+       | CPU 2 |
                                                :       :DIVIDE |       |
                                                +-------+       |       |
        The CPU being busy doing a --->     --->| A->0  |~~~~   |       |
        division speculates on the              +-------+   ~   |       |
        LOAD of A                               :       :   ~   |       |
                                                :       :DIVIDE |       |
                                                :       :   ~   |       |
                                                :       :   ~   |       |
                                            rrrrrrrrrrrrrrrrr   |       |
                                                +-------+       |       |
        The speculation is discarded --->   --->| A->1  |------>|       |
        and an updated value is                 +-------+       |       |
        retrieved                               :       :       +-------+
```

MULTICOPY ATOMICITY

Multicopy atomicity is a deeply intuitive notion about ordering that is not always provided by real computer systems, namely that a given store becomes visible at the same time to all CPUs, or, alternatively, that all CPUs agree on the order in which all stores become visible. However, support of full multicopy atomicity would rule out valuable hardware optimizations, so a weaker form called ``other multicopy atomicity'' instead guarantees only that a given store becomes visible at the same time to all -other- CPUs. The remainder of this document discusses this weaker form, but for brevity will call it simply ``multicopy atomicity''.

The following example demonstrates multicopy atomicity:

```
        CPU 1                   CPU 2                   CPU 3
        ======================= ======================= =======================
                { X = 0, Y = 0 }
        STORE X=1               r1=LOAD X (reads 1)     LOAD Y (reads 1)
                                <general barrier>       <read barrier>
                                STORE Y=r1              LOAD X
```

Suppose that CPU 2's load from X returns 1, which it then stores to Y, and CPU 3's load from Y returns 1. This indicates that CPU 1's store to X precedes CPU 2's load from X and that CPU 2's store to Y precedes CPU 3's load from Y. In addition, the memory barriers guarantee that CPU 2 executes its load before its store, and CPU 3 loads from Y before it loads from X. The question is then "Can CPU 3's load from X return 0?"

Because CPU 3's load from X in some sense comes after CPU 2's load, it is natural to expect that CPU 3's load from X must therefore return 1. This expectation follows from multicopy atomicity: if a load executing on CPU B follows a load from the same variable executing on CPU A (and CPU A did not originally store the value which it read), then on multicopy-atomic systems, CPU B's load must return either the same value that CPU A's load did or some later value. However, the Linux kernel does not require systems to be multicopy atomic.

The use of a general memory barrier in the example above compensates for any lack of multicopy atomicity. In the example, if CPU 2's load from X returns 1 and CPU 3's load from Y returns 1, then CPU 3's load from X must indeed also return 1.

However, dependencies, read barriers, and write barriers are not always able to compensate for non-multicopy atomicity. For example, suppose that CPU 2's general barrier is removed from the above example, leaving only the data dependency shown below:

```
        CPU 1                   CPU 2                   CPU 3
        ======================= ======================= =======================
                { X = 0, Y = 0 }
        STORE X=1               r1=LOAD X (reads 1)     LOAD Y (reads 1)
                                <data dependency>       <read barrier>
                                STORE Y=r1              LOAD X (reads 0)
```

This substitution allows non-multicopy atomicity to run rampant: in this example, it is perfectly legal for CPU 2's load from X to return 1, CPU 3's load from Y to return 1, and its load from X to return 0.

The key point is that although CPU 2's data dependency orders its load and store, it does not guarantee to order CPU 1's store. Thus, if this example runs on a non-multicopy-atomic system where CPUs 1 and 2 share a store buffer or a level of cache, CPU 2 might have early access to CPU 1's writes. General barriers are therefore required to ensure that all CPUs agree on the combined order of multiple accesses.

General barriers can compensate not only for non-multicopy atomicity, but can also generate additional ordering that can ensure that -all- CPUs will perceive the same order of -all- operations. In contrast, a chain of release-acquire pairs do not provide this additional ordering, which means that only those CPUs on the chain are guaranteed to agree on the combined order of the accesses. For example, switching to C code in deference to the ghost of Herman Hollerith:

```
        int u, v, x, y, z;

        void cpu0(void)
        {
                r0 = smp_load_acquire(&x);
                WRITE_ONCE(u, 1);
                smp_store_release(&y, 1);
        }

        void cpu1(void)
        {
                r1 = smp_load_acquire(&y);
                r4 = READ_ONCE(v);
                r5 = READ_ONCE(u);
                smp_store_release(&z, 1);
        }

        void cpu2(void)
        {
                r2 = smp_load_acquire(&z);
                smp_store_release(&x, 1);
        }

        void cpu3(void)
        {
                WRITE_ONCE(v, 1);
                smp_mb();
                r3 = READ_ONCE(u);
        }
```

Because cpu0(), cpu1(), and cpu2() participate in a chain of smp_store_release()/smp_load_acquire() pairs, the following outcome is prohibited:

```
        r0 == 1 && r1 == 1 && r2 == 1
```

Furthermore, because of the release-acquire relationship between cpu0() and cpu1(), cpu1() must see cpu0()'s writes, so that the following outcome is prohibited:

```
        r1 == 1 && r5 == 0
```

However, the ordering provided by a release-acquire chain is local to the CPUs participating in that chain and does not apply to cpu3(), at least aside from stores. Therefore, the following outcome is possible:

```
        r0 == 0 && r1 == 1 && r2 == 1 && r3 == 0 && r4 == 0
```

As an aside, the following outcome is also possible:

```
        r0 == 0 && r1 == 1 && r2 == 1 && r3 == 0 && r4 == 0 && r5 == 1
```

Although cpu0(), cpu1(), and cpu2() will see their respective reads and writes in order, CPUs not involved in the release-acquire chain might well disagree on the order. This disagreement stems from the fact that the weak memory-barrier instructions used to implement smp_load_acquire() and smp_store_release() are not required to order prior stores against subsequent loads in all cases. This means that cpu3() can see cpu0()'s store to u as happening -after- cpu1()'s load from v, even though both cpu0() and cpu1() agree that these two operations occurred in the intended order.

However, please keep in mind that smp_load_acquire() is not magic. In particular, it simply reads from its argument with ordering. It does -not- ensure that any particular value will be read. Therefore, the following outcome is possible:

```
        r0 == 0 && r1 == 0 && r2 == 0 && r5 == 0
```

Note that this outcome can happen even on a mythical sequentially consistent system where nothing is ever reordered.

To reiterate, if your code requires full ordering of all operations, use general barriers throughout.

EXPLICIT KERNEL BARRIERS

The Linux kernel has a variety of different barriers that act at different levels:

(*) Compiler barrier.

(*) CPU memory barriers.

COMPILER BARRIER

The Linux kernel has an explicit compiler barrier function that prevents the compiler from moving the memory accesses either side of it to the other side:

```
        barrier();
```

This is a general barrier -- there are no read-read or write-write variants of barrier(). However, READ_ONCE() and WRITE_ONCE() can be thought of as weak forms of barrier() that affect only the specific accesses flagged by the READ_ONCE() or WRITE_ONCE().

The barrier() function has the following effects:

(*) Prevents the compiler from reordering accesses following the barrier() to precede any accesses preceding the barrier(). One example use for this property is to ease communication between interrupt-handler code and the code that was interrupted.

(*) Within a loop, forces the compiler to load the variables used in that loop's conditional on each pass through that loop.

The READ_ONCE() and WRITE_ONCE() functions can prevent any number of optimizations that, while perfectly safe in single-threaded code, can be fatal in concurrent code. Here are some examples of these sorts of optimizations:

(*) The compiler is within its rights to reorder loads and stores to the same variable, and in some cases, the CPU is within its rights to reorder loads to the same variable. This means that the following code:

```
   a[0] = x;
   a[1] = x;
```

Might result in an older value of x stored in a[1] than in a[0]. Prevent both the compiler and the CPU from doing this as follows:

```
   a[0] = READ_ONCE(x);
   a[1] = READ_ONCE(x);
```

In short, READ_ONCE() and WRITE_ONCE() provide cache coherence for accesses from multiple CPUs to a single variable.

(*) The compiler is within its rights to merge successive loads from the same variable. Such merging can cause the compiler to "optimize" the following code:

```
while (tmp = a)
        do_something_with(tmp);
```

into the following code, which, although in some sense legitimate for single-threaded code, is almost certainly not what the developer intended:

```
if (tmp = a)
        for (;;)
                do_something_with(tmp);
```

Use READ_ONCE() to prevent the compiler from doing this to you:

```
while (tmp = READ_ONCE(a))
        do_something_with(tmp);
```

(*) The compiler is within its rights to reload a variable, for example, in cases where high register pressure prevents the compiler from keeping all data of interest in registers. The compiler might therefore optimize the variable 'tmp' out of our previous example:

```
while (tmp = a)
        do_something_with(tmp);
```

This could result in the following code, which is perfectly safe in single-threaded code, but can be fatal in concurrent code:

```
while (a)
        do_something_with(a);
```

For example, the optimized version of this code could result in passing a zero to do_something_with() in the case where the variable a was modified by some other CPU between the "while" statement and the call to do_something_with().

Again, use READ_ONCE() to prevent the compiler from doing this:

```
while (tmp = READ_ONCE(a))
        do_something_with(tmp);
```

Note that if the compiler runs short of registers, it might save tmp onto the stack. The overhead of this saving and later restoring is why compilers reload variables. Doing so is perfectly safe for single-threaded code, so you need to tell the compiler about cases where it is not safe.

(*) The compiler is within its rights to omit a load entirely if it knows what the value will be. For example, if the compiler can prove that the value of variable 'a' is always zero, it can optimize this code:

```
while (tmp = a)
        do_something_with(tmp);
```

Into this:

```
do { } while (0);
```

This transformation is a win for single-threaded code because it gets rid of a load and a branch. The problem is that the compiler will carry out its proof assuming that the current CPU is the only one updating variable 'a'. If variable 'a' is shared, then the compiler's proof will be erroneous. Use READ_ONCE() to tell the compiler that it doesn't know as much as it thinks it does:

```
while (tmp = READ_ONCE(a))
        do_something_with(tmp);
```

But please note that the compiler is also closely watching what you do with the value after the READ_ONCE(). For example, suppose you do the following and MAX is a preprocessor macro with the value 1:

```
while ((tmp = READ_ONCE(a)) % MAX)
        do_something_with(tmp);
```

Then the compiler knows that the result of the "%" operator applied to MAX will always be zero, again allowing the compiler to optimize the code into near-nonexistence. (It will still load from the variable 'a'.)

(*) Similarly, the compiler is within its rights to omit a store entirely if it knows that the variable already has the value being stored. Again, the compiler assumes that the current CPU is the only one storing into the variable, which can cause the compiler to do the wrong thing for shared variables. For example, suppose you have the following:

```
a = 0;
... Code that does not store to variable a ...
a = 0;
```

The compiler sees that the value of variable 'a' is already zero, so it might well omit the second store. This would come as a fatal surprise if some other CPU might have stored to variable 'a' in the meantime.

Use WRITE_ONCE() to prevent the compiler from making this sort of wrong guess:

```
WRITE_ONCE(a, 0);
... Code that does not store to variable a ...
WRITE_ONCE(a, 0);
```

(*) The compiler is within its rights to reorder memory accesses unless you tell it not to. For example, consider the following interaction between process-level code and an interrupt handler:

```
void process_level(void)
{
        msg = get_message();
        flag = true;
}

void interrupt_handler(void)
{
        if (flag)
                process_message(msg);
}
```

There is nothing to prevent the compiler from transforming process_level() to the following, in fact, this might well be a win for single-threaded code:

```
void process_level(void)
{
        flag = true;
        msg = get_message();
```

```
   }
```

If the interrupt occurs between these two statement, then interrupt_handler() might be passed a garbled msg. Use WRITE_ONCE() to prevent this as follows:

```
   void process_level(void)
   {
           WRITE_ONCE(msg, get_message());
           WRITE_ONCE(flag, true);
   }

   void interrupt_handler(void)
   {
           if (READ_ONCE(flag))
                   process_message(READ_ONCE(msg));
   }
```

Note that the READ_ONCE() and WRITE_ONCE() wrappers in interrupt_handler() are needed if this interrupt handler can itself be interrupted by something that also accesses 'flag' and 'msg', for example, a nested interrupt or an NMI. Otherwise, READ_ONCE() and WRITE_ONCE() are not needed in interrupt_handler() other than for documentation purposes. (Note also that nested interrupts do not typically occur in modern Linux kernels, in fact, if an interrupt handler returns with interrupts enabled, you will get a WARN_ONCE() splat.)

You should assume that the compiler can move READ_ONCE() and WRITE_ONCE() past code not containing READ_ONCE(), WRITE_ONCE(), barrier(), or similar primitives.

This effect could also be achieved using barrier(), but READ_ONCE() and WRITE_ONCE() are more selective: With READ_ONCE() and WRITE_ONCE(), the compiler need only forget the contents of the indicated memory locations, while with barrier() the compiler must discard the value of all memory locations that it has currently cached in any machine registers. Of course, the compiler must also respect the order in which the READ_ONCE()s and WRITE_ONCE()s occur, though the CPU of course need not do so.

(*) The compiler is within its rights to invent stores to a variable, as in the following example:

```
   if (a)
           b = a;
   else
           b = 42;
```

The compiler might save a branch by optimizing this as follows:

```
   b = 42;
   if (a)
           b = a;
```

In single-threaded code, this is not only safe, but also saves a branch. Unfortunately, in concurrent code, this optimization could cause some other CPU to see a spurious value of 42 -- even if variable 'a' was never zero -- when loading variable 'b'. Use WRITE_ONCE() to prevent this as follows:

```
   if (a)
           WRITE_ONCE(b, a);
```

```
	else
		WRITE_ONCE(b, 42);
```

The compiler can also invent loads. These are usually less damaging, but they can result in cache-line bouncing and thus in poor performance and scalability. Use READ_ONCE() to prevent invented loads.

(*) For aligned memory locations whose size allows them to be accessed with a single memory-reference instruction, prevents "load tearing" and "store tearing," in which a single large access is replaced by multiple smaller accesses. For example, given an architecture having 16-bit store instructions with 7-bit immediate fields, the compiler might be tempted to use two 16-bit store-immediate instructions to implement the following 32-bit store:

```
	p = 0x00010002;
```

Please note that GCC really does use this sort of optimization, which is not surprising given that it would likely take more than two instructions to build the constant and then store it. This optimization can therefore be a win in single-threaded code. In fact, a recent bug (since fixed) caused GCC to incorrectly use this optimization in a volatile store. In the absence of such bugs, use of WRITE_ONCE() prevents store tearing in the following example:

```
	WRITE_ONCE(p, 0x00010002);
```

Use of packed structures can also result in load and store tearing, as in this example:

```
	struct __attribute__((__packed__)) foo {
		short a;
		int b;
		short c;
	};
	struct foo foo1, foo2;
	...

	foo2.a = foo1.a;
	foo2.b = foo1.b;
	foo2.c = foo1.c;
```

Because there are no READ_ONCE() or WRITE_ONCE() wrappers and no volatile markings, the compiler would be well within its rights to implement these three assignment statements as a pair of 32-bit loads followed by a pair of 32-bit stores. This would result in load tearing on 'foo1.b' and store tearing on 'foo2.b'. READ_ONCE() and WRITE_ONCE() again prevent tearing in this example:

```
	foo2.a = foo1.a;
	WRITE_ONCE(foo2.b, READ_ONCE(foo1.b));
	foo2.c = foo1.c;
```

All that aside, it is never necessary to use READ_ONCE() and WRITE_ONCE() on a variable that has been marked volatile. For example, because 'jiffies' is marked volatile, it is never necessary to say READ_ONCE(jiffies). The reason for this is that READ_ONCE() and WRITE_ONCE() are implemented as volatile casts, which has no effect when its argument is already marked volatile.

Please note that these compiler barriers have no direct effect on the CPU, which may then reorder things however it wishes.

CPU MEMORY BARRIERS

The Linux kernel has eight basic CPU memory barriers:

TYPE	MANDATORY	SMP CONDITIONAL
GENERAL	mb()	smp_mb()
WRITE	wmb()	smp_wmb()
READ	rmb()	smp_rmb()
DATA DEPENDENCY		READ_ONCE()

All memory barriers except the data dependency barriers imply a compiler barrier. Data dependencies do not impose any additional compiler ordering.

Aside: In the case of data dependencies, the compiler would be expected to issue the loads in the correct order (eg. `a[b]` would have to load the value of b before loading a[b]), however there is no guarantee in the C specification that the compiler may not speculate the value of b (eg. is equal to 1) and load a[b] before b (eg. tmp = a[1]; if (b != 1) tmp = a[b];). There is also the problem of a compiler reloading b after having loaded a[b], thus having a newer copy of b than a[b]. A consensus has not yet been reached about these problems, however the READ_ONCE() macro is a good place to start looking.

SMP memory barriers are reduced to compiler barriers on uniprocessor compiled systems because it is assumed that a CPU will appear to be self-consistent, and will order overlapping accesses correctly with respect to itself. However, see the subsection on "Virtual Machine Guests" below.

[!] Note that SMP memory barriers _must_ be used to control the ordering of references to shared memory on SMP systems, though the use of locking instead is sufficient.

Mandatory barriers should not be used to control SMP effects, since mandatory barriers impose unnecessary overhead on both SMP and UP systems. They may, however, be used to control MMIO effects on accesses through relaxed memory I/O windows. These barriers are required even on non-SMP systems as they affect the order in which memory operations appear to a device by prohibiting both the compiler and the CPU from reordering them.

There are some more advanced barrier functions:

(*) smp_store_mb(var, value)

This assigns the value to the variable and then inserts a full memory barrier after it. It isn't guaranteed to insert anything more than a compiler barrier in a UP compilation.

(*) smp_mb__before_atomic();
(*) smp_mb__after_atomic();

These are for use with atomic RMW functions that do not imply memory barriers, but where the code needs a memory barrier. Examples for atomic RMW functions that do not imply are memory barrier are e.g. add, subtract, (failed) conditional operations, _relaxed functions, but not atomic_read or atomic_set. A common example where a memory barrier may be required is when atomic ops are used for reference counting.

These are also used for atomic RMW bitop functions that do not imply a memory barrier (such as set_bit and clear_bit).

As an example, consider a piece of code that marks an object as being dead and then decrements the object's reference count:

```
obj->dead = 1;
smp_mb__before_atomic();
atomic_dec(&obj->ref_count);
```

This makes sure that the death mark on the object is perceived to be set *before* the reference counter is decremented.

See Documentation/atomic_{t,bitops}.txt for more information.

(*) dma_wmb();
(*) dma_rmb();

These are for use with consistent memory to guarantee the ordering of writes or reads of shared memory accessible to both the CPU and a DMA capable device.

For example, consider a device driver that shares memory with a device and uses a descriptor status value to indicate if the descriptor belongs to the device or the CPU, and a doorbell to notify it when new descriptors are available:

```
if (desc->status != DEVICE_OWN) {
        /* do not read data until we own descriptor */
        dma_rmb();

        /* read/modify data */
        read_data = desc->data;
        desc->data = write_data;

        /* flush modifications before status update */
        dma_wmb();

        /* assign ownership */
        desc->status = DEVICE_OWN;

        /* notify device of new descriptors */
        writel(DESC_NOTIFY, doorbell);
}
```

The dma_rmb() allows us guarantee the device has released ownership before we read the data from the descriptor, and the dma_wmb() allows us to guarantee the data is written to the descriptor before the device can see it now has ownership. Note that, when using writel(), a prior wmb() is not needed to guarantee that the cache coherent memory writes have completed before writing to the MMIO region. The cheaper writel_relaxed() does not provide this guarantee and must not be used here.

See the subsection "Kernel I/O barrier effects" for more information on relaxed I/O accessors and the Documentation/core-api/dma-api.rst file for more information on consistent memory.

(*) pmem_wmb();

```
     This is for use with persistent memory to ensure that stores for which
     modifications are written to persistent storage reached a platform
     durability domain.

     For example, after a non-temporal write to pmem region, we use pmem_wmb()
     to ensure that stores have reached a platform durability domain. This ensures
     that stores have updated persistent storage before any data access or
     data transfer caused by subsequent instructions is initiated. This is
     in addition to the ordering done by wmb().

     For load from persistent memory, existing read memory barriers are sufficient
     to ensure read ordering.

===============================
IMPLICIT KERNEL MEMORY BARRIERS
===============================

Some of the other functions in the linux kernel imply memory barriers, amongst
which are locking and scheduling functions.

This specification is a _minimum_ guarantee; any particular architecture may
provide more substantial guarantees, but these may not be relied upon outside
of arch specific code.


LOCK ACQUISITION FUNCTIONS
--------------------------

The Linux kernel has a number of locking constructs:

 (*) spin locks
 (*) R/W spin locks
 (*) mutexes
 (*) semaphores
 (*) R/W semaphores

In all cases there are variants on "ACQUIRE" operations and "RELEASE" operations
for each construct.  These operations all imply certain barriers:

 (1) ACQUIRE operation implication:

     Memory operations issued after the ACQUIRE will be completed after the
     ACQUIRE operation has completed.

     Memory operations issued before the ACQUIRE may be completed after
     the ACQUIRE operation has completed.

 (2) RELEASE operation implication:

     Memory operations issued before the RELEASE will be completed before the
     RELEASE operation has completed.

     Memory operations issued after the RELEASE may be completed before the
     RELEASE operation has completed.

 (3) ACQUIRE vs ACQUIRE implication:

     All ACQUIRE operations issued before another ACQUIRE operation will be
     completed before that ACQUIRE operation.

 (4) ACQUIRE vs RELEASE implication:

     All ACQUIRE operations issued before a RELEASE operation will be
```

completed before the RELEASE operation.

(5) Failed conditional ACQUIRE implication:

Certain locking variants of the ACQUIRE operation may fail, either due to being unable to get the lock immediately, or due to receiving an unblocked signal while asleep waiting for the lock to become available. Failed locks do not imply any sort of barrier.

[!] Note: one of the consequences of lock ACQUIREs and RELEASEs being only one-way barriers is that the effects of instructions outside of a critical section may seep into the inside of the critical section.

An ACQUIRE followed by a RELEASE may not be assumed to be full memory barrier because it is possible for an access preceding the ACQUIRE to happen after the ACQUIRE, and an access following the RELEASE to happen before the RELEASE, and the two accesses can themselves then cross:

```
        *A = a;
        ACQUIRE M
        RELEASE M
        *B = b;
```

may occur as:

```
        ACQUIRE M, STORE *B, STORE *A, RELEASE M
```

When the ACQUIRE and RELEASE are a lock acquisition and release, respectively, this same reordering can occur if the lock's ACQUIRE and RELEASE are to the same lock variable, but only from the perspective of another CPU not holding that lock. In short, a ACQUIRE followed by an RELEASE may -not- be assumed to be a full memory barrier.

Similarly, the reverse case of a RELEASE followed by an ACQUIRE does not imply a full memory barrier. Therefore, the CPU's execution of the critical sections corresponding to the RELEASE and the ACQUIRE can cross, so that:

```
        *A = a;
        RELEASE M
        ACQUIRE N
        *B = b;
```

could occur as:

```
        ACQUIRE N, STORE *B, STORE *A, RELEASE M
```

It might appear that this reordering could introduce a deadlock. However, this cannot happen because if such a deadlock threatened, the RELEASE would simply complete, thereby avoiding the deadlock.

Why does this work?

One key point is that we are only talking about the CPU doing the reordering, not the compiler. If the compiler (or, for that matter, the developer) switched the operations, deadlock -could- occur.

But suppose the CPU reordered the operations. In this case, the unlock precedes the lock in the assembly code. The CPU simply elected to try executing the later lock operation first. If there is a deadlock, this lock operation will simply spin (or try to sleep, but more on that later). The CPU will eventually

> execute the unlock operation (which preceded the lock operation in the assembly code), which will unravel the potential deadlock, allowing the lock operation to succeed.
>
> But what if the lock is a sleeplock? In that case, the code will try to enter the scheduler, where it will eventually encounter a memory barrier, which will force the earlier unlock operation to complete, again unraveling the deadlock. There might be a sleep-unlock race, but the locking primitive needs to resolve such races properly in any case.

Locks and semaphores may not provide any guarantee of ordering on UP compiled systems, and so cannot be counted on in such a situation to actually achieve anything at all - especially with respect to I/O accesses - unless combined with interrupt disabling operations.

See also the section on "Inter-CPU acquiring barrier effects".

As an example, consider the following:

```
        *A = a;
        *B = b;
        ACQUIRE
        *C = c;
        *D = d;
        RELEASE
        *E = e;
        *F = f;
```

The following sequence of events is acceptable:

```
        ACQUIRE, {*F,*A}, *E, {*C,*D}, *B, RELEASE

        [+] Note that {*F,*A} indicates a combined access.
```

But none of the following are:

```
        {*F,*A}, *B,    ACQUIRE, *C, *D,        RELEASE, *E
        *A, *B, *C,     ACQUIRE, *D,            RELEASE, *E, *F
        *A, *B,         ACQUIRE, *C,            RELEASE, *D, *E, *F
        *B,             ACQUIRE, *C, *D,        RELEASE, {*F,*A}, *E
```

INTERRUPT DISABLING FUNCTIONS

Functions that disable interrupts (ACQUIRE equivalent) and enable interrupts (RELEASE equivalent) will act as compiler barriers only. So if memory or I/O barriers are required in such a situation, they must be provided from some other means.

SLEEP AND WAKE-UP FUNCTIONS

Sleeping and waking on an event flagged in global data can be viewed as an interaction between two pieces of data: the task state of the task waiting for the event and the global data used to indicate the event. To make sure that these appear to happen in the right order, the primitives to begin the process of going to sleep, and the primitives to initiate a wake up imply certain barriers.

Firstly, the sleeper normally follows something like this sequence of events:

```
        for (;;) {
                set_current_state(TASK_UNINTERRUPTIBLE);
                if (event_indicated)
                        break;
                schedule();
        }
```

A general memory barrier is interpolated automatically by set_current_state() after it has altered the task state:

```
        CPU 1
        ===============================
        set_current_state();
          smp_store_mb();
            STORE current->state
            <general barrier>
        LOAD event_indicated
```

set_current_state() may be wrapped by:

```
        prepare_to_wait();
        prepare_to_wait_exclusive();
```

which therefore also imply a general memory barrier after setting the state. The whole sequence above is available in various canned forms, all of which interpolate the memory barrier in the right place:

```
        wait_event();
        wait_event_interruptible();
        wait_event_interruptible_exclusive();
        wait_event_interruptible_timeout();
        wait_event_killable();
        wait_event_timeout();
        wait_on_bit();
        wait_on_bit_lock();
```

Secondly, code that performs a wake up normally follows something like this:

```
        event_indicated = 1;
        wake_up(&event_wait_queue);
```

or:

```
        event_indicated = 1;
        wake_up_process(event_daemon);
```

A general memory barrier is executed by wake_up() if it wakes something up. If it doesn't wake anything up then a memory barrier may or may not be executed; you must not rely on it. The barrier occurs before the task state is accessed, in particular, it sits between the STORE to indicate the event and the STORE to set TASK_RUNNING:

```
        CPU 1 (Sleeper)                 CPU 2 (Waker)
        =============================== ===============================
        set_current_state();            STORE event_indicated
          smp_store_mb();               wake_up();
            STORE current->state          ...
            <general barrier>             <general barrier>
        LOAD event_indicated              if ((LOAD task->state) & TASK_NORMAL)
```

```
                                            STORE task->state
```

where "task" is the thread being woken up and it equals CPU 1's "current".

To repeat, a general memory barrier is guaranteed to be executed by wake_up() if something is actually awakened, but otherwise there is no such guarantee. To see this, consider the following sequence of events, where X and Y are both initially zero:

```
        CPU 1                           CPU 2
        =============================== ===============================
        X = 1;                          Y = 1;
        smp_mb();                       wake_up();
        LOAD Y                          LOAD X
```

If a wakeup does occur, one (at least) of the two loads must see 1. If, on the other hand, a wakeup does not occur, both loads might see 0.

wake_up_process() always executes a general memory barrier. The barrier again occurs before the task state is accessed. In particular, if the wake_up() in the previous snippet were replaced by a call to wake_up_process() then one of the two loads would be guaranteed to see 1.

The available waker functions include:

```
        complete();
        wake_up();
        wake_up_all();
        wake_up_bit();
        wake_up_interruptible();
        wake_up_interruptible_all();
        wake_up_interruptible_nr();
        wake_up_interruptible_poll();
        wake_up_interruptible_sync();
        wake_up_interruptible_sync_poll();
        wake_up_locked();
        wake_up_locked_poll();
        wake_up_nr();
        wake_up_poll();
        wake_up_process();
```

In terms of memory ordering, these functions all provide the same guarantees of a wake_up() (or stronger).

[!] Note that the memory barriers implied by the sleeper and the waker do _not_ order multiple stores before the wake-up with respect to loads of those stored values after the sleeper has called set_current_state(). For instance, if the sleeper does:

```
        set_current_state(TASK_INTERRUPTIBLE);
        if (event_indicated)
                break;
        __set_current_state(TASK_RUNNING);
        do_something(my_data);
```

and the waker does:

```
        my_data = value;
        event_indicated = 1;
        wake_up(&event_wait_queue);
```

there's no guarantee that the change to event_indicated will be perceived by the sleeper as coming after the change to my_data. In such a circumstance, the code on both sides must interpolate its own memory barriers between the separate data accesses. Thus the above sleeper ought to do:

```
        set_current_state(TASK_INTERRUPTIBLE);
        if (event_indicated) {
                smp_rmb();
                do_something(my_data);
        }
```

and the waker should do:

```
        my_data = value;
        smp_wmb();
        event_indicated = 1;
        wake_up(&event_wait_queue);
```

MISCELLANEOUS FUNCTIONS

Other functions that imply barriers:

 (*) schedule() and similar imply full memory barriers.

INTER-CPU ACQUIRING BARRIER EFFECTS

On SMP systems locking primitives give a more substantial form of barrier: one that does affect memory access ordering on other CPUs, within the context of conflict on any particular lock.

ACQUIRES VS MEMORY ACCESSES

Consider the following: the system has a pair of spinlocks (M) and (Q), and three CPUs; then should the following sequence of events occur:

```
        CPU 1                           CPU 2
        =============================== ===============================
        WRITE_ONCE(*A, a);              WRITE_ONCE(*E, e);
        ACQUIRE M                       ACQUIRE Q
        WRITE_ONCE(*B, b);              WRITE_ONCE(*F, f);
        WRITE_ONCE(*C, c);              WRITE_ONCE(*G, g);
        RELEASE M                       RELEASE Q
        WRITE_ONCE(*D, d);              WRITE_ONCE(*H, h);
```

Then there is no guarantee as to what order CPU 3 will see the accesses to *A through *H occur in, other than the constraints imposed by the separate locks on the separate CPUs. It might, for example, see:

```
        *E, ACQUIRE M, ACQUIRE Q, *G, *C, *F, *A, *B, RELEASE Q, *D, *H, RELEASE M
```

But it won't see any of:

```
        *B, *C or *D preceding ACQUIRE M
        *A, *B or *C following RELEASE M
        *F, *G or *H preceding ACQUIRE Q
        *E, *F or *G following RELEASE Q
```

WHERE ARE MEMORY BARRIERS NEEDED?

Under normal operation, memory operation reordering is generally not going to be a problem as a single-threaded linear piece of code will still appear to work correctly, even if it's in an SMP kernel. There are, however, four circumstances in which reordering definitely _could_ be a problem:

(*) Interprocessor interaction.

(*) Atomic operations.

(*) Accessing devices.

(*) Interrupts.

INTERPROCESSOR INTERACTION

When there's a system with more than one processor, more than one CPU in the system may be working on the same data set at the same time. This can cause synchronisation problems, and the usual way of dealing with them is to use locks. Locks, however, are quite expensive, and so it may be preferable to operate without the use of a lock if at all possible. In such a case operations that affect both CPUs may have to be carefully ordered to prevent a malfunction.

Consider, for example, the R/W semaphore slow path. Here a waiting process is queued on the semaphore, by virtue of it having a piece of its stack linked to the semaphore's list of waiting processes:

```
struct rw_semaphore {
        ...
        spinlock_t lock;
        struct list_head waiters;
};

struct rwsem_waiter {
        struct list_head list;
        struct task_struct *task;
};
```

To wake up a particular waiter, the up_read() or up_write() functions have to:

(1) read the next pointer from this waiter's record to know as to where the next waiter record is;

(2) read the pointer to the waiter's task structure;

(3) clear the task pointer to tell the waiter it has been given the semaphore;

(4) call wake_up_process() on the task; and

(5) release the reference held on the waiter's task struct.

In other words, it has to perform this sequence of events:

```
LOAD waiter->list.next;
LOAD waiter->task;
STORE waiter->task;
CALL wakeup
```

```
        RELEASE task
```

and if any of these steps occur out of order, then the whole thing may malfunction.

Once it has queued itself and dropped the semaphore lock, the waiter does not get the lock again; it instead just waits for its task pointer to be cleared before proceeding. Since the record is on the waiter's stack, this means that if the task pointer is cleared _before_ the next pointer in the list is read, another CPU might start processing the waiter and might clobber the waiter's stack before the up*() function has a chance to read the next pointer.

Consider then what might happen to the above sequence of events:

```
        CPU 1                           CPU 2
        =============================== ===============================
                                        down_xxx()
                                        Queue waiter
                                        Sleep
        up_yyy()
        LOAD waiter->task;
        STORE waiter->task;
                                        Woken up by other event
        <preempt>
                                        Resume processing
                                        down_xxx() returns
                                        call foo()
                                        foo() clobbers *waiter
        </preempt>
        LOAD waiter->list.next;
        --- OOPS ---
```

This could be dealt with using the semaphore lock, but then the down_xxx() function has to needlessly get the spinlock again after being woken up.

The way to deal with this is to insert a general SMP memory barrier:

```
        LOAD waiter->list.next;
        LOAD waiter->task;
        smp_mb();
        STORE waiter->task;
        CALL wakeup
        RELEASE task
```

In this case, the barrier makes a guarantee that all memory accesses before the barrier will appear to happen before all the memory accesses after the barrier with respect to the other CPUs on the system. It does _not_ guarantee that all the memory accesses before the barrier will be complete by the time the barrier instruction itself is complete.

On a UP system - where this wouldn't be a problem - the smp_mb() is just a compiler barrier, thus making sure the compiler emits the instructions in the right order without actually intervening in the CPU. Since there's only one CPU, that CPU's dependency ordering logic will take care of everything else.

ATOMIC OPERATIONS

While they are technically interprocessor interaction considerations, atomic operations are noted specially as some of them imply full memory barriers and some don't, but they're very heavily relied on as a group throughout the kernel.

See Documentation/atomic_t.txt for more information.

ACCESSING DEVICES

Many devices can be memory mapped, and so appear to the CPU as if they're just a set of memory locations. To control such a device, the driver usually has to make the right memory accesses in exactly the right order.

However, having a clever CPU or a clever compiler creates a potential problem in that the carefully sequenced accesses in the driver code won't reach the device in the requisite order if the CPU or the compiler thinks it is more efficient to reorder, combine or merge accesses - something that would cause the device to malfunction.

Inside of the Linux kernel, I/O should be done through the appropriate accessor routines - such as inb() or writel() - which know how to make such accesses appropriately sequential. While this, for the most part, renders the explicit use of memory barriers unnecessary, if the accessor functions are used to refer to an I/O memory window with relaxed memory access properties, then _mandatory_ memory barriers are required to enforce ordering.

See Documentation/driver-api/device-io.rst for more information.

INTERRUPTS

A driver may be interrupted by its own interrupt service routine, and thus the two parts of the driver may interfere with each other's attempts to control or access the device.

This may be alleviated - at least in part - by disabling local interrupts (a form of locking), such that the critical operations are all contained within the interrupt-disabled section in the driver. While the driver's interrupt routine is executing, the driver's core may not run on the same CPU, and its interrupt is not permitted to happen again until the current interrupt has been handled, thus the interrupt handler does not need to lock against that.

However, consider a driver that was talking to an ethernet card that sports an address register and a data register. If that driver's core talks to the card under interrupt-disablement and then the driver's interrupt handler is invoked:

```
        LOCAL IRQ DISABLE
        writew(ADDR, 3);
        writew(DATA, y);
        LOCAL IRQ ENABLE
        <interrupt>
        writew(ADDR, 4);
        q = readw(DATA);
        </interrupt>
```

The store to the data register might happen after the second store to the address register if ordering rules are sufficiently relaxed:

```
        STORE *ADDR = 3, STORE *ADDR = 4, STORE *DATA = y, q = LOAD *DATA
```

If ordering rules are relaxed, it must be assumed that accesses done inside an interrupt disabled section may leak outside of it and may interleave with accesses performed in an interrupt - and vice versa - unless implicit or explicit barriers are used.

Normally this won't be a problem because the I/O accesses done inside such sections will include synchronous load operations on strictly ordered I/O registers that form implicit I/O barriers.

A similar situation may occur between an interrupt routine and two routines running on separate CPUs that communicate with each other. If such a case is likely, then interrupt-disabling locks should be used to guarantee ordering.

```
==========================
KERNEL I/O BARRIER EFFECTS
==========================
```

Interfacing with peripherals via I/O accesses is deeply architecture and device specific. Therefore, drivers which are inherently non-portable may rely on specific behaviours of their target systems in order to achieve synchronization in the most lightweight manner possible. For drivers intending to be portable between multiple architectures and bus implementations, the kernel offers a series of accessor functions that provide various degrees of ordering guarantees:

(*) readX(), writeX():

The readX() and writeX() MMIO accessors take a pointer to the peripheral being accessed as an __iomem * parameter. For pointers mapped with the default I/O attributes (e.g. those returned by ioremap()), the ordering guarantees are as follows:

1. All readX() and writeX() accesses to the same peripheral are ordered with respect to each other. This ensures that MMIO register accesses by the same CPU thread to a particular device will arrive in program order.

2. A writeX() issued by a CPU thread holding a spinlock is ordered before a writeX() to the same peripheral from another CPU thread issued after a later acquisition of the same spinlock. This ensures that MMIO register writes to a particular device issued while holding a spinlock will arrive in an order consistent with acquisitions of the lock.

3. A writeX() by a CPU thread to the peripheral will first wait for the completion of all prior writes to memory either issued by, or propagated to, the same thread. This ensures that writes by the CPU to an outbound DMA buffer allocated by dma_alloc_coherent() will be visible to a DMA engine when the CPU writes to its MMIO control register to trigger the transfer.

4. A readX() by a CPU thread from the peripheral will complete before any subsequent reads from memory by the same thread can begin. This ensures that reads by the CPU from an incoming DMA buffer allocated by dma_alloc_coherent() will not see stale data after reading from the DMA engine's MMIO status register to establish that the DMA transfer has completed.

5. A readX() by a CPU thread from the peripheral will complete before any subsequent delay() loop can begin execution on the same thread. This ensures that two MMIO register writes by the CPU to a peripheral will arrive at least 1us apart if the first write is immediately read back with readX() and udelay(1) is called prior to the second writeX():

```
                writel(42, DEVICE_REGISTER_0); // Arrives at the device...
                readl(DEVICE_REGISTER_0);
                udelay(1);
                writel(42, DEVICE_REGISTER_1); // ...at least 1us before this.

        The ordering properties of __iomem pointers obtained with non-default
        attributes (e.g. those returned by ioremap_wc()) are specific to the
        underlying architecture and therefore the guarantees listed above cannot
        generally be relied upon for accesses to these types of mappings.

 (*) readX_relaxed(), writeX_relaxed():

        These are similar to readX() and writeX(), but provide weaker memory
        ordering guarantees. Specifically, they do not guarantee ordering with
        respect to locking, normal memory accesses or delay() loops (i.e.
        bullets 2-5 above) but they are still guaranteed to be ordered with
        respect to other accesses from the same CPU thread to the same
        peripheral when operating on __iomem pointers mapped with the default
        I/O attributes.

 (*) readsX(), writesX():

        The readsX() and writesX() MMIO accessors are designed for accessing
        register-based, memory-mapped FIFOs residing on peripherals that are not
        capable of performing DMA. Consequently, they provide only the ordering
        guarantees of readX_relaxed() and writeX_relaxed(), as documented above.

 (*) inX(), outX():

        The inX() and outX() accessors are intended to access legacy port-mapped
        I/O peripherals, which may require special instructions on some
        architectures (notably x86). The port number of the peripheral being
        accessed is passed as an argument.

        Since many CPU architectures ultimately access these peripherals via an
        internal virtual memory mapping, the portable ordering guarantees
        provided by inX() and outX() are the same as those provided by readX()
        and writeX() respectively when accessing a mapping with the default I/O
        attributes.

        Device drivers may expect outX() to emit a non-posted write transaction
        that waits for a completion response from the I/O peripheral before
        returning. This is not guaranteed by all architectures and is therefore
        not part of the portable ordering semantics.

 (*) insX(), outsX():

        As above, the insX() and outsX() accessors provide the same ordering
        guarantees as readsX() and writesX() respectively when accessing a
        mapping with the default I/O attributes.

 (*) ioreadX(), iowriteX():

        These will perform appropriately for the type of access they're actually
        doing, be it inX()/outX() or readX()/writeX().

With the exception of the string accessors (insX(), outsX(), readsX() and
writesX()), all of the above assume that the underlying peripheral is
little-endian and will therefore perform byte-swapping operations on big-endian
architectures.
```

ASSUMED MINIMUM EXECUTION ORDERING MODEL

It has to be assumed that the conceptual CPU is weakly-ordered but that it will maintain the appearance of program causality with respect to itself. Some CPUs (such as i386 or x86_64) are more constrained than others (such as powerpc or frv), and so the most relaxed case (namely DEC Alpha) must be assumed outside of arch-specific code.

This means that it must be considered that the CPU will execute its instruction stream in any order it feels like - or even in parallel - provided that if an instruction in the stream depends on an earlier instruction, then that earlier instruction must be sufficiently complete[*] before the later instruction may proceed; in other words: provided that the appearance of causality is maintained.

[*] Some instructions have more than one effect - such as changing the condition codes, changing registers or changing memory - and different instructions may depend on different effects.

A CPU may also discard any instruction sequence that winds up having no ultimate effect. For example, if two adjacent instructions both load an immediate value into the same register, the first may be discarded.

Similarly, it has to be assumed that compiler might reorder the instruction stream in any way it sees fit, again provided the appearance of causality is maintained.

THE EFFECTS OF THE CPU CACHE

The way cached memory operations are perceived across the system is affected to a certain extent by the caches that lie between CPUs and memory, and by the memory coherence system that maintains the consistency of state in the system.

As far as the way a CPU interacts with another part of the system through the caches goes, the memory system has to include the CPU's caches, and memory barriers for the most part act at the interface between the CPU and its cache (memory barriers logically act on the dotted line in the following diagram):

```
            <--- CPU --->         :       <----------- Memory ----------->
                                  :
        +--------+    +--------+  :   +--------+    +-----------+
        |        |    |        |  :   |        |    |           |    +--------+
        |  CPU   |    | Memory |  :   | CPU    |    |           |    |        |
        |  Core  |--->| Access |----->| Cache  |<-->|           |    |        |
        |        |    | Queue  |  :   |        |    |           |--->| Memory |
        |        |    |        |  :   |        |    |           |    |        |
        +--------+    +--------+  :   +--------+    |           |    |        |
                                  :                 | Cache     |    +--------+
                                  :                 | Coherency |
                                  :                 | Mechanism |    +--------+
        +--------+    +--------+  :   +--------+    |           |    |        |
        |        |    |        |  :   |        |    |           |    |        |
        |  CPU   |    | Memory |  :   | CPU    |    |           |--->| Device |
        |  Core  |--->| Access |----->| Cache  |<-->|           |    |        |
        |        |    | Queue  |  :   |        |    |           |    |        |
        |        |    |        |  :   |        |    |           |    +--------+
        +--------+    +--------+  :   +--------+    +-----------+
                                  :
                                  :
```

Although any particular load or store may not actually appear outside of the CPU that issued it since it may have been satisfied within the CPU's own cache, it will still appear as if the full memory access had taken place as far as the other CPUs are concerned since the cache coherency mechanisms will migrate the cacheline over to the accessing CPU and propagate the effects upon conflict.

The CPU core may execute instructions in any order it deems fit, provided the expected program causality appears to be maintained. Some of the instructions generate load and store operations which then go into the queue of memory accesses to be performed. The core may place these in the queue in any order it wishes, and continue execution until it is forced to wait for an instruction to complete.

What memory barriers are concerned with is controlling the order in which accesses cross from the CPU side of things to the memory side of things, and the order in which the effects are perceived to happen by the other observers in the system.

[!] Memory barriers are _not_ needed within a given CPU, as CPUs always see their own loads and stores as if they had happened in program order.

[!] MMIO or other device accesses may bypass the cache system. This depends on the properties of the memory window through which devices are accessed and/or the use of any special device communication instructions the CPU may have.

CACHE COHERENCY VS DMA

Not all systems maintain cache coherency with respect to devices doing DMA. In such cases, a device attempting DMA may obtain stale data from RAM because dirty cache lines may be resident in the caches of various CPUs, and may not have been written back to RAM yet. To deal with this, the appropriate part of the kernel must flush the overlapping bits of cache on each CPU (and maybe invalidate them as well).

In addition, the data DMA'd to RAM by a device may be overwritten by dirty cache lines being written back to RAM from a CPU's cache after the device has installed its own data, or cache lines present in the CPU's cache may simply obscure the fact that RAM has been updated, until at such time as the cacheline is discarded from the CPU's cache and reloaded. To deal with this, the appropriate part of the kernel must invalidate the overlapping bits of the cache on each CPU.

See Documentation/core-api/cachetlb.rst for more information on cache management.

CACHE COHERENCY VS MMIO

Memory mapped I/O usually takes place through memory locations that are part of a window in the CPU's memory space that has different properties assigned than the usual RAM directed window.

Amongst these properties is usually the fact that such accesses bypass the caching entirely and go directly to the device buses. This means MMIO accesses may, in effect, overtake accesses to cached memory that were emitted earlier. A memory barrier isn't sufficient in such a case, but rather the cache must be flushed between the cached memory write and the MMIO access if the two are in any way dependent.

THE THINGS CPUS GET UP TO

A programmer might take it for granted that the CPU will perform memory operations in exactly the order specified, so that if the CPU is, for example, given the following piece of code to execute:

```
        a = READ_ONCE(*A);
        WRITE_ONCE(*B, b);
        c = READ_ONCE(*C);
        d = READ_ONCE(*D);
        WRITE_ONCE(*E, e);
```

they would then expect that the CPU will complete the memory operation for each instruction before moving on to the next one, leading to a definite sequence of operations as seen by external observers in the system:

```
        LOAD *A, STORE *B, LOAD *C, LOAD *D, STORE *E.
```

Reality is, of course, much messier. With many CPUs and compilers, the above assumption doesn't hold because:

(*) loads are more likely to need to be completed immediately to permit execution progress, whereas stores can often be deferred without a problem;

(*) loads may be done speculatively, and the result discarded should it prove to have been unnecessary;

(*) loads may be done speculatively, leading to the result having been fetched at the wrong time in the expected sequence of events;

(*) the order of the memory accesses may be rearranged to promote better use of the CPU buses and caches;

(*) loads and stores may be combined to improve performance when talking to memory or I/O hardware that can do batched accesses of adjacent locations, thus cutting down on transaction setup costs (memory and PCI devices may both be able to do this); and

(*) the CPU's data cache may affect the ordering, and while cache-coherency mechanisms may alleviate this - once the store has actually hit the cache - there's no guarantee that the coherency management will be propagated in order to other CPUs.

So what another CPU, say, might actually observe from the above piece of code is:

```
        LOAD *A, ..., LOAD {*C,*D}, STORE *E, STORE *B

        (Where "LOAD {*C,*D}" is a combined load)
```

However, it is guaranteed that a CPU will be self-consistent: it will see its _own_ accesses appear to be correctly ordered, without the need for a memory barrier. For instance with the following code:

```
        U = READ_ONCE(*A);
        WRITE_ONCE(*A, V);
        WRITE_ONCE(*A, W);
        X = READ_ONCE(*A);
        WRITE_ONCE(*A, Y);
        Z = READ_ONCE(*A);
```

and assuming no intervention by an external influence, it can be assumed that the final result will appear to be:

```
        U == the original value of *A
        X == W
        Z == Y
        *A == Y
```

The code above may cause the CPU to generate the full sequence of memory accesses:

```
        U=LOAD *A, STORE *A=V, STORE *A=W, X=LOAD *A, STORE *A=Y, Z=LOAD *A
```

in that order, but, without intervention, the sequence may have almost any combination of elements combined or discarded, provided the program's view of the world remains consistent. Note that READ_ONCE() and WRITE_ONCE() are -not- optional in the above example, as there are architectures where a given CPU might reorder successive loads to the same location. On such architectures, READ_ONCE() and WRITE_ONCE() do whatever is necessary to prevent this, for example, on Itanium the volatile casts used by READ_ONCE() and WRITE_ONCE() cause GCC to emit the special ld.acq and st.rel instructions (respectively) that prevent such reordering.

The compiler may also combine, discard or defer elements of the sequence before the CPU even sees them.

For instance:

```
        *A = V;
        *A = W;
```

may be reduced to:

```
        *A = W;
```

since, without either a write barrier or an WRITE_ONCE(), it can be assumed that the effect of the storage of V to *A is lost. Similarly:

```
        *A = Y;
        Z = *A;
```

may, without a memory barrier or an READ_ONCE() and WRITE_ONCE(), be reduced to:

```
        *A = Y;
        Z = Y;
```

and the LOAD operation never appear outside of the CPU.

AND THEN THERE'S THE ALPHA

The DEC Alpha CPU is one of the most relaxed CPUs there is. Not only that, some versions of the Alpha CPU have a split data cache, permitting them to have two semantically-related cache lines updated at separate times. This is where the data dependency barrier really becomes necessary as this synchronises both caches with the memory coherence system, thus making it seem like pointer

```
changes vs new data occur in the right order.

The Alpha defines the Linux kernel's memory model, although as of v4.15
the Linux kernel's addition of smp_mb() to READ_ONCE() on Alpha greatly
reduced its impact on the memory model.


VIRTUAL MACHINE GUESTS
----------------------

Guests running within virtual machines might be affected by SMP effects even if
the guest itself is compiled without SMP support.  This is an artifact of
interfacing with an SMP host while running an UP kernel.  Using mandatory
barriers for this use-case would be possible but is often suboptimal.

To handle this case optimally, low-level virt_mb() etc macros are available.
These have the same effect as smp_mb() etc when SMP is enabled, but generate
identical code for SMP and non-SMP systems.  For example, virtual machine guests
should use virt_mb() rather than smp_mb() when synchronizing against a
(possibly SMP) host.

These are equivalent to smp_mb() etc counterparts in all other respects,
in particular, they do not control MMIO effects: to control
MMIO effects, use mandatory barriers.


============
EXAMPLE USES
============

CIRCULAR BUFFERS
----------------

Memory barriers can be used to implement circular buffering without the need
of a lock to serialise the producer with the consumer.  See:

        Documentation/core-api/circular-buffers.rst

for details.


==========
REFERENCES
==========

Alpha AXP Architecture Reference Manual, Second Edition (Sites & Witek,
Digital Press)
        Chapter 5.2: Physical Address Space Characteristics
        Chapter 5.4: Caches and Write Buffers
        Chapter 5.5: Data Sharing
        Chapter 5.6: Read/Write Ordering

AMD64 Architecture Programmer's Manual Volume 2: System Programming
        Chapter 7.1: Memory-Access Ordering
        Chapter 7.4: Buffering and Combining Memory Writes

ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)
        Chapter B2: The AArch64 Application Level Memory Model

IA-32 Intel Architecture Software Developer's Manual, Volume 3:
System Programming Guide
        Chapter 7.1: Locked Atomic Operations
        Chapter 7.2: Memory Ordering
        Chapter 7.4: Serializing Instructions

The SPARC Architecture Manual, Version 9
        Chapter 8: Memory Models
        Appendix D: Formal Specification of the Memory Models
        Appendix J: Programming with the Memory Models

Storage in the PowerPC (Stone and Fitzgerald)

UltraSPARC Programmer Reference Manual
        Chapter 5: Memory Accesses and Cacheability
        Chapter 15: Sparc-V9 Memory Models

UltraSPARC III Cu User's Manual
        Chapter 9: Memory Models

UltraSPARC IIIi Processor User's Manual
        Chapter 8: Memory Models

UltraSPARC Architecture 2005
        Chapter 9: Memory
        Appendix D: Formal Specifications of the Memory Models

UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005
        Chapter 8: Memory Models
        Appendix F: Caches and Cache Coherency

Solaris Internals, Core Kernel Architecture, p63-68:
        Chapter 3.3: Hardware Considerations for Locks and
                        Synchronization

Unix Systems for Modern Architectures, Symmetric Multiprocessing and Caching
for Kernel Programmers:
        Chapter 13: Other Memory Models

Intel Itanium Architecture Software Developer's Manual: Volume 1:
        Section 2.6: Speculation
        Section 4.4: Memory Access
```