Linux Staging Documentation

Release 6.8.0

The kernel development community

Jan 16, 2026

CONTENTS

1 Brief tutorial on CRC computation **1**

2 LZO stream format as understood by Linux's LZO decompressor **5**
- 2.1 Introduction . . . 5
- 2.2 Description . . . 5
- 2.3 Byte sequences . . . 6
- 2.4 Authors . . . 9

3 Remote Processor Framework **11**
- 3.1 Introduction . . . 11
- 3.2 User API . . . 11
- 3.3 Typical usage . . . 12
- 3.4 API for implementors . . . 13
- 3.5 Implementation callbacks . . . 14
- 3.6 Binary Firmware Structure . . . 15
- 3.7 Virtio and remoteproc . . . 17

4 Remote Processor Messaging (rpmsg) Framework **19**
- 4.1 Introduction . . . 19
- 4.2 User API . . . 20
- 4.3 Typical usage . . . 23
- 4.4 Allocations of rpmsg channels . . . 24

5 Speculation **25**

6 Mitigating speculation side-channels **27**

7 Static Keys **29**
- 7.1 Abstract . . . 29
- 7.2 Motivation . . . 30
- 7.3 Solution . . . 30
- 7.4 Static key label API, usage and examples . . . 30

8 XZ data compression in Linux **37**
- 8.1 Introduction . . . 37
- 8.2 XZ related components in the kernel . . . 37
- 8.3 Notes on compression options . . . 38
- 8.4 Future plans . . . 38
- 8.5 Conformance to the .xz file format specification . . . 38
- 8.6 Reporting bugs . . . 39

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;
        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++) {
                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 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.

  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).

```
int 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.

Returns 0 on success, and an appropriate error value otherwise. 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.
 */
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 *Remote Processor Messaging (rpmsg) Framework*).

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 *Remote Processor Framework* (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_endpoint *ept, void *data, int len);
```

sends a message across to the remote processor from the given endpoint. The caller should specify the endpoint, the data it wants to send, and its length (in bytes). The message will be sent on the specified endpoint's channel, i.e. its source and destination address fields will be respectively set to the endpoint's src address and its parent channel 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_endpoint *ept, void *data, int len, u32 dst);
```

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

The caller should specify the endpoint, 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 endpoints's channel belongs, using the endpoints'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_endpoint *ept, 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 endpoint, 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 endpoint's channel belongs, but the endpoint's src and channel 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_endpoint *ept, void *data, int len);
```

sends a message across to the remote processor from a given endpoint. The caller should specify the endpoint, the data it wants to send, and its length (in bytes). The message will be sent on the specified endpoint's channel, i.e. its source and destination address fields will be respectively set to the endpoint's src address and its parent channel 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_endpoint *ept, void *data, int len, u32 dst)
```

sends a message across to the remote processor from a given endoint, 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_endpoint *ept, 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->ept, "hello!", 6);
        if (err) {
                pr_err("rpmsg_send failed: %d\n", err);
                return err;
        }

        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);
...
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
- **`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>
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

XZ DATA COMPRESSION IN LINUX

8.1 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/>.

8.2 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.

8.3 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.

8.4 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.

8.5 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.

8.6 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.