pin_user_pages() and related calls

Overview

This document describes the following functions:

```
pin_user_pages()
pin_user_pages_fast()
pin_user_pages_remote()
```

Basic description of FOLL_PIN

FOLL_PIN and FOLL_LONGTERM are flags that can be passed to the get_user_pages*() ("gup") family of functions. FOLL_PIN has significant interactions and interdependencies with FOLL_LONGTERM, so both are covered here.

FOLL_PIN is internal to gup, meaning that it should not appear at the gup call sites. This allows the associated wrapper functions (pin_user_pages*() and others) to set the correct combination of these flags, and to check for problems as well.

FOLL_LONGTERM, on the other hand, *is* allowed to be set at the gup call sites. This is in order to avoid creating a large number of wrapper functions to cover all combinations of get*(), pin*(), FOLL_LONGTERM, and more. Also, the pin_user_pages*() APIs are clearly distinct from the get_user_pages*() APIs, so that's a natural dividing line, and a good point to make separate wrapper calls. In other words, use pin_user_pages*() for DMA-pinned pages, and get_user_pages*() for other cases. There are five cases described later on in this document, to further clarify that concept.

FOLL_PIN and FOLL_GET are mutually exclusive for a given gup call. However, multiple threads and call sites are free to pin the same struct pages, via both FOLL_PIN and FOLL_GET. It's just the call site that needs to choose one or the other, not the struct page(s).

The FOLL_PIN implementation is nearly the same as FOLL_GET, except that FOLL_PIN uses a different reference counting technique.

FOLL_PIN is a prerequisite to FOLL_LONGTERM. Another way of saying that is, FOLL_LONGTERM is a specific case, more restrictive case of FOLL_PIN.

Which flags are set by each wrapper

For these pin_user_pages*() functions, FOLL_PIN is OR'd in with whatever gup flags the caller provides. The caller is required to pass in a non-null struct pages* array, and the function then pins pages by incrementing each by a special value: GUP_PIN_COUNTING_BIAS.

For compound pages, the GUP_PIN_COUNTING_BIAS scheme is not used. Instead, an exact form of pin counting is achieved, by using the 2nd struct page in the compound page. A new struct page field, compound_pincount, has been added in order to support this.

This approach for compound pages avoids the counting upper limit problems that are discussed below. Those limitations would have been aggravated severely by huge pages, because each tail page adds a refcount to the head page. And in fact, testing revealed that, without a separate compound_pincount field, page overflows were seen in some huge page stress tests.

This also means that huge pages and compound pages do not suffer from the false positives problem that is mentioned below.:

```
Function
--------
pin_user_pages          FOLL_PIN is always set internally by this function.
pin_user_pages_fast     FOLL_PIN is always set internally by this function.
```

```
pin_user_pages_remote   FOLL_PIN is always set internally by this function.
```

For these get_user_pages*() functions, FOLL_GET might not even be specified. Behavior is a little more complex than above. If FOLL_GET was *not* specified, but the caller passed in a non-null struct pages* array, then the function sets FOLL_GET for you, and proceeds to pin pages by incrementing the refcount of each page by +1.:

```
Function
--------
get_user_pages           FOLL_GET is sometimes set internally by this function.
get_user_pages_fast      FOLL_GET is sometimes set internally by this function.
get_user_pages_remote    FOLL_GET is sometimes set internally by this function.
```

Tracking dma-pinned pages

Some of the key design constraints, and solutions, for tracking dma-pinned pages:

- An actual reference count, per struct page, is required. This is because multiple processes may pin and unpin a page.
- False positives (reporting that a page is dma-pinned, when in fact it is not) are acceptable, but false negatives are not.
- struct page may not be increased in size for this, and all fields are already used.
- Given the above, we can overload the page->_refcount field by using, sort of, the upper bits in that field for a dma-pinned count. "Sort of", means that, rather than dividing page->_refcount into bit fields, we simple add a medium- large value (GUP_PIN_COUNTING_BIAS, initially chosen to be 1024: 10 bits) to page->_refcount. This provides fuzzy behavior: if a page has get_page() called on it 1024 times, then it will appear to have a single dma-pinned count. And again, that's acceptable.

This also leads to limitations: there are only 31-10==21 bits available for a counter that increments 10 bits at a time.

- Callers must specifically request "dma-pinned tracking of pages". In other words, just calling get_user_pages() will not suffice; a new set of functions, pin_user_page() and related, must be used.

FOLL_PIN, FOLL_GET, FOLL_LONGTERM: when to use which flags

Thanks to Jan Kara, Vlastimil Babka and several other -mm people, for describing these categories:

CASE 1: Direct IO (DIO)

There are GUP references to pages that are serving as DIO buffers. These buffers are needed for a relatively short time (so they are not "long term"). No special synchronization with page_mkclean() or munmap() is provided. Therefore, flags to set at the call site are:

```
FOLL_PIN
```

...but rather than setting FOLL_PIN directly, call sites should use one of the pin_user_pages*() routines that set FOLL_PIN.

CASE 2: RDMA

There are GUP references to pages that are serving as DMA buffers. These buffers are needed for a long time ("long term"). No special synchronization with page_mkclean() or munmap() is provided. Therefore, flags to set at the call site are:

```
FOLL_PIN | FOLL_LONGTERM
```

NOTE: Some pages, such as DAX pages, cannot be pinned with longterm pins. That's because DAX pages do not have a separate page cache, and so "pinning" implies locking down file system blocks, which is not (yet) supported in that way.

CASE 3: MMU notifier registration, with or without page faulting hardware

Device drivers can pin pages via get_user_pages*(), and register for mmu notifier callbacks for the memory range. Then, upon receiving a notifier "invalidate range" callback , stop the device from using the range, and unpin the pages. There may be other possible schemes, such as for example explicitly synchronizing against pending IO, that accomplish approximately the same thing.

Or, if the hardware supports replayable page faults, then the device driver can avoid pinning entirely (this is ideal), as follows: register for mmu notifier callbacks as above, but instead of stopping the device and unpinning in the callback, simply remove the range from the device's page tables.

Either way, as long as the driver unpins the pages upon mmu notifier callback, then there is proper synchronization with both filesystem and mm (page_mkclean(), munmap(), etc). Therefore, neither flag needs to be set.

CASE 4: Pinning for struct page manipulation only

If only struct page data (as opposed to the actual memory contents that a page is tracking) is affected, then normal GUP calls are sufficient, and neither flag needs to be set.

CASE 5: Pinning in order to write to the data within the page

Even though neither DMA nor Direct IO is involved, just a simple case of "pin, write to a page's data, unpin" can cause a problem.

Case 5 may be considered a superset of Case 1, plus Case 2, plus anything that invokes that pattern. In other words, if the code is neither Case 1 nor Case 2, it may still require FOLL_PIN, for patterns like this:

Correct (uses FOLL_PIN calls):
  pin_user_pages() write to the data within the pages unpin_user_pages()
INCORRECT (uses FOLL_GET calls):
  get_user_pages() write to the data within the pages put_page()

page_maybe_dma_pinned(): the whole point of pinning

The whole point of marking pages as "DMA-pinned" or "gup-pinned" is to be able to query, "is this page DMA-pinned?" That allows code such as page_mkclean() (and file system writeback code in general) to make informed decisions about what to do when a page cannot be unmapped due to such pins.

What to do in those cases is the subject of a years-long series of discussions and debates (see the References at the end of this document). It's a TODO item here: fill in the details once that's worked out. Meanwhile, it's safe to say that having this available:

```
static inline bool page_maybe_dma_pinned(struct page *page)
```

...is a prerequisite to solving the long-running gup+DMA problem.

Another way of thinking about FOLL_GET, FOLL_PIN, and FOLL_LONGTERM

Another way of thinking about these flags is as a progression of restrictions: FOLL_GET is for struct page manipulation, without affecting the data that the struct page refers to. FOLL_PIN is a *replacement* for FOLL_GET, and is for short term pins on pages whose data *will* get accessed. As such, FOLL_PIN is a "more severe" form of pinning. And finally, FOLL_LONGTERM is an even more restrictive case that has FOLL_PIN as a prerequisite: this is for pages that will be pinned longterm, and whose data will be accessed.

Unit testing

This file:

```
tools/testing/selftests/vm/gup_test.c
```

has the following new calls to exercise the new pin*() wrapper functions:

- PIN_FAST_BENCHMARK (./gup_test -a)
- PIN_BASIC_TEST (./gup_test -b)

You can monitor how many total dma-pinned pages have been acquired and released since the system was booted, via two new /proc/vmstat entries:

```
/proc/vmstat/nr_foll_pin_acquired
/proc/vmstat/nr_foll_pin_released
```

Under normal conditions, these two values will be equal unless there are any long-term [R]DMA pins in place, or during pin/unpin transitions.

- nr_foll_pin_acquired: This is the number of logical pins that have been acquired since the system was powered on. For huge pages, the head page is pinned once for each page (head page and each tail page) within the huge page. This follows the same sort of behavior that get_user_pages() uses for huge pages: the head page is refcounted once for each tail or head page in the huge page, when get_user_pages() is applied to a huge page.
- nr_foll_pin_released: The number of logical pins that have been released since the system was powered on. Note that pages are released (unpinned) on a PAGE_SIZE granularity, even if the original pin was applied to a huge page. Becaused of the pin count behavior described above in "nr_foll_pin_acquired", the accounting balances out, so that after doing this:

  ```
  pin_user_pages(huge_page);
  for (each page in huge_page)
      unpin_user_page(page);
  ```

...the following is expected:

```
nr_foll_pin_released == nr_foll_pin_acquired
```

(...unless it was already out of balance due to a long-term RDMA pin being in place.)

Other diagnostics

dump_page() has been enhanced slightly, to handle these new counting fields, and to better report on compound pages in general. Specifically, for compound pages, the exact (compound_pincount) pincount is reported.

References

- Some slow progress on get_user_pages() (Apr 2, 2019)
- DMA and get_user_pages() (LPC: Dec 12, 2018)
- The trouble with get_user_pages() (Apr 30, 2018)
- LWN kernel index: get_user_pages()

John Hubbard, October, 2019