Linux Vm Documentation

The kernel development community

Jul 14, 2020

CONTENTS

This is a collection of documents about the Linux memory management (mm) subsystem. If you are looking for advice on simply allocating memory, see the memory_allocation.

CHAPTER

ONE

USER GUIDES FOR MM FEATURES

The following documents provide guides for controlling and tuning various features of the Linux memory management

1.1 Automatically bind swap device to numa node

If the system has more than one swap device and swap device has the node information, we can make use of this information to decide which swap device to use in get_swap_pages() to get better performance.

1.1.1 How to use this feature

Swap device has priority and that decides the order of it to be used. To make use of automatically binding, there is no need to manipulate priority settings for swap devices. e.g. on a 2 node machine, assume 2 swap devices swapA and swapB, with swapA attached to node 0 and swapB attached to node 1, are going to be swapped on. Simply swapping them on by doing:

```
# swapon /dev/swapA
# swapon /dev/swapB
```

Then node 0 will use the two swap devices in the order of swapA then swapB and node 1 will use the two swap devices in the order of swapB then swapA. Note that the order of them being swapped on doesn't matter.

A more complex example on a 4 node machine. Assume 6 swap devices are going to be swapped on: swapA and swapB are attached to node 0, swapC is attached to node 1, swapD and swapE are attached to node 2 and swapF is attached to node3. The way to swap them on is the same as above:

```
# swapon /dev/swapA
# swapon /dev/swapB
# swapon /dev/swapC
# swapon /dev/swapD
# swapon /dev/swapE
# swapon /dev/swapF
```

Then node 0 will use them in the order of:

```
swapA/swapB -> swapC -> swapD -> swapE -> swapF
```

swapA and swapB will be used in a round robin mode before any other swap device.

node 1 will use them in the order of:

```
swapC -> swapA -> swapB -> swapD -> swapE -> swapF
```

node 2 will use them in the order of:

```
swapD/swapE -> swapA -> swapB -> swapC -> swapF
```

Similaly, swapD and swapE will be used in a round robin mode before any other swap devices.

node 3 will use them in the order of:

```
swapF -> swapA -> swapB -> swapC -> swapD -> swapE
```

1.1.2 Implementation details

The current code uses a priority based list, swap_avail_list, to decide which swap device to use and if multiple swap devices share the same priority, they are used round robin. This change here replaces the single global swap_avail_list with a per-numa-node list, i.e. for each numa node, it sees its own priority based list of available swap devices. Swap device's priority can be promoted on its matching node's swap_avail_list.

The current swap device's priority is set as: user can set a >=0 value, or the system will pick one starting from -1 then downwards. The priority value in the swap_avail_list is the negated value of the swap device's due to plist being sorted from low to high. The new policy doesn't change the semantics for priority >=0 cases, the previous starting from -1 then downwards now becomes starting from -2 then downwards and -1 is reserved as the promoted value. So if multiple swap devices are attached to the same node, they will all be promoted to priority -1 on that node's plist and will be used round robin before any other swap devices.

1.2 zswap

1.2.1 Overview

Zswap is a lightweight compressed cache for swap pages. It takes pages that are in the process of being swapped out and attempts to compress them into a dynamically allocated RAM-based memory pool. zswap basically trades CPU cycles for potentially reduced swap I/O. This trade-off can also result in a significant performance improvement if reads from the compressed cache are faster than reads from a swap device.

Note: Zswap is a new feature as of v3.11 and interacts heavily with memory reclaim. This interaction has not been fully explored on the large set of poten-

tial configurations and workloads that exist. For this reason, zswap is a work in progress and should be considered experimental.

Some potential benefits:

- Desktop/laptop users with limited RAM capacities can mitigate the performance impact of swapping.
- Overcommitted guests that share a common I/O resource can dramatically reduce their swap I/O pressure, avoiding heavy handed I/O throttling by the hypervisor. This allows more work to get done with less impact to the guest workload and guests sharing the I/O subsystem
- Users with SSDs as swap devices can extend the life of the device by drastically reducing life-shortening writes.

Zswap evicts pages from compressed cache on an LRU basis to the backing swap device when the compressed pool reaches its size limit. This requirement had been identified in prior community discussions.

Whether Zswap is enabled at the boot time depends on whether the `CONFIG_ZSWAP_DEFAULT_ON` Kconfig option is enabled or not. This setting can then be overridden by providing the kernel command line `zswap.enabled=` option, for example `zswap.enabled=0`. Zswap can also be enabled and disabled at runtime using the sysfs interface. An example command to enable zswap at runtime, assuming sysfs is mounted at `/sys`, is:

```
echo 1 > /sys/module/zswap/parameters/enabled
```

When zswap is disabled at runtime it will stop storing pages that are being swapped out. However, it will _not_ immediately write out or fault back into memory all of the pages stored in the compressed pool. The pages stored in zswap will remain in the compressed pool until they are either invalidated or faulted back into memory. In order to force all pages out of the compressed pool, a swapoff on the swap device(s) will fault back into memory all swapped out pages, including those in the compressed pool.

1.2.2 Design

Zswap receives pages for compression through the Frontswap API and is able to evict pages from its own compressed pool on an LRU basis and write them back to the backing swap device in the case that the compressed pool is full.

Zswap makes use of zpool for the managing the compressed memory pool. Each allocation in zpool is not directly accessible by address. Rather, a handle is returned by the allocation routine and that handle must be mapped before being accessed. The compressed memory pool grows on demand and shrinks as compressed pages are freed. The pool is not preallocated. By default, a zpool of type selected in `CONFIG_ZSWAP_ZPOOL_DEFAULT` Kconfig option is created, but it can be overridden at boot time by setting the `zpool` attribute, e.g. `zswap.zpool=zbud`. It can also be changed at runtime using the sysfs `zpool` attribute, e.g.:

```
echo zbud > /sys/module/zswap/parameters/zpool
```

The zbud type zpool allocates exactly 1 page to store 2 compressed pages, which means the compression ratio will always be 2:1 or worse (because of half-full zbud pages). The zsmalloc type zpool has a more complex compressed page storage method, and it can achieve greater storage densities. However, zsmalloc does not implement compressed page eviction, so once zswap fills it cannot evict the oldest page, it can only reject new pages.

When a swap page is passed from frontswap to zswap, zswap maintains a mapping of the swap entry, a combination of the swap type and swap offset, to the zpool handle that references that compressed swap page. This mapping is achieved with a red-black tree per swap type. The swap offset is the search key for the tree nodes.

During a page fault on a PTE that is a swap entry, frontswap calls the zswap load function to decompress the page into the page allocated by the page fault handler.

Once there are no PTEs referencing a swap page stored in zswap (i.e. the count in the swap_map goes to 0) the swap code calls the zswap invalidate function, via frontswap, to free the compressed entry.

Zswap seeks to be simple in its policies. Sysfs attributes allow for one user controlled policy:

- max_pool_percent - The maximum percentage of memory that the compressed pool can occupy.

The default compressor is selected in `CONFIG_ZSWAP_COMPRESSOR_DEFAULT` Kconfig option, but it can be overridden at boot time by setting the `compressor` attribute, e.g. `zswap.compressor=lzo`. It can also be changed at runtime using the sysfs "compressor" attribute, e.g.:

```
echo lzo > /sys/module/zswap/parameters/compressor
```

When the zpool and/or compressor parameter is changed at runtime, any existing compressed pages are not modified; they are left in their own zpool. When a request is made for a page in an old zpool, it is uncompressed using its original compressor. Once all pages are removed from an old zpool, the zpool and its compressor are freed.

Some of the pages in zswap are same-value filled pages (i.e. contents of the page have same value or repetitive pattern). These pages include zero-filled pages and they are handled differently. During store operation, a page is checked if it is a same-value filled page before compressing it. If true, the compressed length of the page is set to zero and the pattern or same-filled value is stored.

Same-value filled pages identification feature is enabled by default and can be disabled at boot time by setting the `same_filled_pages_enabled` attribute to 0, e.g. `zswap.same_filled_pages_enabled=0`. It can also be enabled and disabled at runtime using the sysfs `same_filled_pages_enabled` attribute, e.g.:

```
echo 1 > /sys/module/zswap/parameters/same_filled_pages_enabled
```

When zswap same-filled page identification is disabled at runtime, it will stop checking for the same-value filled pages during store operation. However, the existing pages which are marked as same-value filled pages remain stored unchanged in zswap until they are either loaded or invalidated.

To prevent zswap from shrinking pool when zswap is full and there's a high pressure on swap (this will result in flipping pages in and out zswap pool without any real benefit but with a performance drop for the system), a special parameter has been introduced to implement a sort of hysteresis to refuse taking pages into zswap pool until it has sufficient space if the limit has been hit. To set the threshold at which zswap would start accepting pages again after it became full, use the sysfs `accept_threshold_percent` attribute, e. g.:

```
echo 80 > /sys/module/zswap/parameters/accept_threshold_percent
```

Setting this parameter to 100 will disable the hysteresis.

A debugfs interface is provided for various statistic about pool size, number of pages stored, same-value filled pages and various counters for the reasons pages are rejected.

CHAPTER

TWO

KERNEL DEVELOPERS MM DOCUMENTATION

The below documents describe MM internals with different level of details ranging from notes and mailing list responses to elaborate descriptions of data structures and algorithms.

2.1 Active MM

```
List:       linux-kernel
Subject:    Re: active_mm
From:       Linus Torvalds <torvalds () transmeta ! com>
Date:       1999-07-30 21:36:24

Cc'd to linux-kernel, because I don't write explanations all that often,
and when I do I feel better about more people reading them.

On Fri, 30 Jul 1999, David Mosberger wrote:
>
> Is there a brief description someplace on how "mm" vs. "active_mm" in
> the task_struct are supposed to be used?  (My apologies if this was
> discussed on the mailing lists---I just returned from vacation and
> wasn't able to follow linux-kernel for a while).

Basically, the new setup is:

 - we have "real address spaces" and "anonymous address spaces". The
   difference is that an anonymous address space doesn't care about the
   user-level page tables at all, so when we do a context switch into an
   anonymous address space we just leave the previous address space
   active.

   The obvious use for a "anonymous address space" is any thread that
   doesn't need any user mappings - all kernel threads basically fall into
   this category, but even "real" threads can temporarily say that for
   some amount of time they are not going to be interested in user space,
   and that the scheduler might as well try to avoid wasting time on
   switching the VM state around. Currently only the old-style bdflush
   sync does that.

 - "tsk->mm" points to the "real address space". For an anonymous process,
   tsk->mm will be NULL, for the logical reason that an anonymous process
   really doesn't _have_ a real address space at all.
```

(continues on next page)

(continued from previous page)

```
 - however, we obviously need to keep track of which address space we
   "stole" for such an anonymous user. For that, we have "tsk->active_mm",
   which shows what the currently active address space is.

   The rule is that for a process with a real address space (ie tsk->mm is
   non-NULL) the active_mm obviously always has to be the same as the real
   one.

   For a anonymous process, tsk->mm == NULL, and tsk->active_mm is the
   "borrowed" mm while the anonymous process is running. When the
   anonymous process gets scheduled away, the borrowed address space is
   returned and cleared.

To support all that, the "struct mm_struct" now has two counters: a
"mm_users" counter that is how many "real address space users" there are,
and a "mm_count" counter that is the number of "lazy" users (ie anonymous
users) plus one if there are any real users.

Usually there is at least one real user, but it could be that the real
user exited on another CPU while a lazy user was still active, so you do
actually get cases where you have a address space that is _only_ used by
lazy users. That is often a short-lived state, because once that thread
gets scheduled away in favour of a real thread, the "zombie" mm gets
released because "mm_users" becomes zero.

Also, a new rule is that _nobody_ ever has "init_mm" as a real MM any
more. "init_mm" should be considered just a "lazy context when no other
context is available", and in fact it is mainly used just at bootup when
no real VM has yet been created. So code that used to check

       if (current->mm == &init_mm)

should generally just do

       if (!current->mm)

instead (which makes more sense anyway - the test is basically one of "do
we have a user context", and is generally done by the page fault handler
and things like that).

Anyway, I put a pre-patch-2.3.13-1 on ftp.kernel.org just a moment ago,
because it slightly changes the interfaces to accommodate the alpha (who
would have thought it, but the alpha actually ends up having one of the
ugliest context switch codes - unlike the other architectures where the MM
and register state is separate, the alpha PALcode joins the two, and you
need to switch both together).

(From http://marc.info/?l=linux-kernel&m=93337278602211&w=2)
```

2.2 Memory Balancing

Started Jan 2000 by Kanoj Sarcar <kanoj@sgi.com>

Memory balancing is needed for !__GFP_ATOMIC and !__GFP_KSWAPD_RECLAIM as well as for non __GFP_IO allocations.

The first reason why a caller may avoid reclaim is that the caller can not sleep due to holding a spinlock or is in interrupt context. The second may be that the caller is willing to fail the allocation without incurring the overhead of page reclaim. This may happen for opportunistic high-order allocation requests that have order-0 fallback options. In such cases, the caller may also wish to avoid waking kswapd.

__GFP_IO allocation requests are made to prevent file system deadlocks.

In the absence of non sleepable allocation requests, it seems detrimental to be doing balancing. Page reclamation can be kicked off lazily, that is, only when needed (aka zone free memory is 0), instead of making it a proactive process.

That being said, the kernel should try to fulfill requests for direct mapped pages from the direct mapped pool, instead of falling back on the dma pool, so as to keep the dma pool filled for dma requests (atomic or not). A similar argument applies to highmem and direct mapped pages. OTOH, if there is a lot of free dma pages, it is preferable to satisfy regular memory requests by allocating one from the dma pool, instead of incurring the overhead of regular zone balancing.

In 2.2, memory balancing/page reclamation would kick off only when the _total_ number of free pages fell below 1/64 th of total memory. With the right ratio of dma and regular memory, it is quite possible that balancing would not be done even when the dma zone was completely empty. 2.2 has been running production machines of varying memory sizes, and seems to be doing fine even with the presence of this problem. In 2.3, due to HIGHMEM, this problem is aggravated.

In 2.3, zone balancing can be done in one of two ways: depending on the zone size (and possibly of the size of lower class zones), we can decide at init time how many free pages we should aim for while balancing any zone. The good part is, while balancing, we do not need to look at sizes of lower class zones, the bad part is, we might do too frequent balancing due to ignoring possibly lower usage in the lower class zones. Also, with a slight change in the allocation routine, it is possible to reduce the memclass() macro to be a simple equality.

Another possible solution is that we balance only when the free memory of a zone _and_ all its lower class zones falls below 1/64th of the total memory in the zone and its lower class zones. This fixes the 2.2 balancing problem, and stays as close to 2.2 behavior as possible. Also, the balancing algorithm works the same way on the various architectures, which have different numbers and types of zones. If we wanted to get fancy, we could assign different weights to free pages in different zones in the future.

Note that if the size of the regular zone is huge compared to dma zone, it becomes less significant to consider the free dma pages while deciding whether to balance the regular zone. The first solution becomes more attractive then.

The appended patch implements the second solution. It also “fixes” two problems: first, kswapd is woken up as in 2.2 on low memory conditions for non-sleepable allocations. Second, the HIGHMEM zone is also balanced, so as to give a fighting

chance for replace_with_highmem() to get a HIGHMEM page, as well as to ensure that HIGHMEM allocations do not fall back into regular zone. This also makes sure that HIGHMEM pages are not leaked (for example, in situations where a HIGHMEM page is in the swapcache but is not being used by anyone)

kswapd also needs to know about the zones it should balance. kswapd is primarily needed in a situation where balancing can not be done, probably because all allocation requests are coming from intr context and all process contexts are sleeping. For 2.3, kswapd does not really need to balance the highmem zone, since intr context does not request highmem pages. kswapd looks at the zone_wake_kswapd field in the zone structure to decide whether a zone needs balancing.

Page stealing from process memory and shm is done if stealing the page would alleviate memory pressure on any zone in the page's node that has fallen below its watermark.

watemark[WMARK_MIN/WMARK_LOW/WMARK_HIGH]/low_on_memory/zone_wake_kswapd: These are per-zone fields, used to determine when a zone needs to be balanced. When the number of pages falls below watermark[WMARK_MIN], the hysteric field low_on_memory gets set. This stays set till the number of free pages becomes watermark[WMARK_HIGH]. When low_on_memory is set, page allocation requests will try to free some pages in the zone (providing GFP_WAIT is set in the request). Orthogonal to this, is the decision to poke kswapd to free some zone pages. That decision is not hysteresis based, and is done when the number of free pages is below watermark[WMARK_LOW]; in which case zone_wake_kswapd is also set.

(Good) Ideas that I have heard:

1. Dynamic experience should influence balancing: number of failed requests for a zone can be tracked and fed into the balancing scheme (jalvo@mbay.net)
2. Implement a replace_with_highmem()-like replace_with_regular() to preserve dma pages. (lkd@tantalophile.demon.co.uk)

2.3 Cleancache

2.3.1 Motivation

Cleancache is a new optional feature provided by the VFS layer that potentially dramatically increases page cache effectiveness for many workloads in many environments at a negligible cost.

Cleancache can be thought of as a page-granularity victim cache for clean pages that the kernel's pageframe replacement algorithm (PFRA) would like to keep around, but can't since there isn't enough memory. So when the PFRA "evicts" a page, it first attempts to use cleancache code to put the data contained in that page into "transcendent memory", memory that is not directly accessible or addressable by the kernel and is of unknown and possibly time-varying size.

Later, when a cleancache-enabled filesystem wishes to access a page in a file on disk, it first checks cleancache to see if it already contains it; if it does, the page of data is copied into the kernel and a disk access is avoided.

Transcendent memory "drivers" for cleancache are currently implemented in Xen (using hypervisor memory) and zcache (using in-kernel compressed memory) and other implementations are in development.

FAQs are included below.

2.3.2 Implementation Overview

A cleancache "backend" that provides transcendent memory registers itself to the kernel's cleancache "frontend" by calling cleancache_register_ops, passing a pointer to a cleancache_ops structure with funcs set appropriately. The functions provided must conform to certain semantics as follows:

Most important, cleancache is "ephemeral". Pages which are copied into cleancache have an indefinite lifetime which is completely unknowable by the kernel and so may or may not still be in cleancache at any later time. Thus, as its name implies, cleancache is not suitable for dirty pages. Cleancache has complete discretion over what pages to preserve and what pages to discard and when.

Mounting a cleancache-enabled filesystem should call "init_fs" to obtain a pool id which, if positive, must be saved in the filesystem's superblock; a negative return value indicates failure. A "put_page" will copy a (presumably about-to-be-evicted) page into cleancache and associate it with the pool id, a file key, and a page index into the file. (The combination of a pool id, a file key, and an index is sometimes called a "handle".) A "get_page" will copy the page, if found, from cleancache into kernel memory. An "invalidate_page" will ensure the page no longer is present in cleancache; an "invalidate_inode" will invalidate all pages associated with the specified file; and, when a filesystem is unmounted, an "invalidate_fs" will invalidate all pages in all files specified by the given pool id and also surrender the pool id.

An "init_shared_fs", like init_fs, obtains a pool id but tells cleancache to treat the pool as shared using a 128-bit UUID as a key. On systems that may run multiple kernels (such as hard partitioned or virtualized systems) that may share a clustered filesystem, and where cleancache may be shared among those kernels, calls to init_shared_fs that specify the same UUID will receive the same pool id, thus allowing the pages to be shared. Note that any security requirements must be imposed outside of the kernel (e.g. by "tools" that control cleancache). Or a cleancache implementation can simply disable shared_init by always returning a negative value.

If a get_page is successful on a non-shared pool, the page is invalidated (thus making cleancache an "exclusive" cache). On a shared pool, the page is NOT invalidated on a successful get_page so that it remains accessible to other sharers. The kernel is responsible for ensuring coherency between cleancache (shared or not), the page cache, and the filesystem, using cleancache invalidate operations as required.

Note that cleancache must enforce put-put-get coherency and get-get coherency. For the former, if two puts are made to the same handle but with different data, say AAA by the first put and BBB by the second, a subsequent get can never return the stale data (AAA). For get-get coherency, if a get for a given handle fails, subsequent gets for that handle will never succeed unless preceded by a successful put with that handle.

Last, cleancache provides no SMP serialization guarantees; if two different Linux threads are simultaneously putting and invalidating a page with the same handle, the results are indeterminate. Callers must lock the page to ensure serial behavior.

2.3.3 Cleancache Performance Metrics

If properly configured, monitoring of cleancache is done via debugfs in the /sys/kernel/debug/cleancache directory. The effectiveness of cleancache can be measured (across all filesystems) with:

succ_gets number of gets that were successful

failed_gets number of gets that failed

puts number of puts attempted (all "succeed")

invalidates number of invalidates attempted

A backend implementation may provide additional metrics.

2.3.4 FAQ

- Where's the value? (Andrew Morton)

Cleancache provides a significant performance benefit to many workloads in many environments with negligible overhead by improving the effectiveness of the pagecache. Clean pagecache pages are saved in transcendent memory (RAM that is otherwise not directly addressable to the kernel); fetching those pages later avoids "refaults" and thus disk reads.

Cleancache (and its sister code "frontswap") provide interfaces for this transcendent memory (aka "tmem"), which conceptually lies between fast kernel-directly-addressable RAM and slower DMA/asynchronous devices. Disallowing direct kernel or userland reads/writes to tmem is ideal when data is transformed to a different form and size (such as with compression) or secretly moved (as might be useful for write- balancing for some RAM-like devices). Evicted page-cache pages (and swap pages) are a great use for this kind of slower-than-RAM-but-much- faster-than-disk transcendent memory, and the cleancache (and frontswap) "page-object-oriented" specification provides a nice way to read and write – and indirectly "name" – the pages.

In the virtual case, the whole point of virtualization is to statistically multiplex physical resources across the varying demands of multiple virtual machines. This is really hard to do with RAM and efforts to do it well with no kernel change have essentially failed (except in some well-publicized special-case workloads). Cleancache – and frontswap – with a fairly small impact on the kernel, provide a huge amount of flexibility for more dynamic, flexible RAM multiplexing. Specifically, the Xen Transcendent Memory backend allows otherwise "fallow" hypervisor-owned RAM to not only be "time-shared"between multiple virtual machines, but the pages can be compressed and deduplicated to optimize RAM utilization. And when guest OS's are induced to surrender underutilized RAM (e.g. with "self-ballooning"), page cache pages are the first to go, and cleancache allows those pages to be saved and reclaimed if overall host system memory conditions allow.

And the identical interface used for cleancache can be used in physical systems as well. The zcache driver acts as a memory-hungry device that stores pages of data in a compressed state. And the proposed "RAMster" driver shares RAM across multiple physical systems.

- Why does cleancache have its sticky fingers so deep inside the filesystems and VFS? (Andrew Morton and Christoph Hellwig)

The core hooks for cleancache in VFS are in most cases a single line and the minimum set are placed precisely where needed to maintain coherency (via cleancache_invalidate operations) between cleancache, the page cache, and disk. All hooks compile into nothingness if cleancache is config'ed off and turn into a function-pointer- compare-to-NULL if config'ed on but no backend claims the ops functions, or to a compare-struct-element-to-negative if a backend claims the ops functions but a filesystem doesn't enable cleancache.

Some filesystems are built entirely on top of VFS and the hooks in VFS are sufficient, so don't require an "init_fs" hook; the initial implementation of cleancache didn't provide this hook. But for some filesystems (such as btrfs), the VFS hooks are incomplete and one or more hooks in fs-specific code are required. And for some other filesystems, such as tmpfs, cleancache may be counterproductive. So it seemed prudent to require a filesystem to "opt in" to use cleancache, which requires adding a hook in each filesystem. Not all filesystems are supported by cleancache only because they haven't been tested. The existing set should be sufficient to validate the concept, the opt-in approach means that untested filesystems are not affected, and the hooks in the existing filesystems should make it very easy to add more filesystems in the future.

The total impact of the hooks to existing fs and mm files is only about 40 lines added (not counting comments and blank lines).

- Why not make cleancache asynchronous and batched so it can more easily interface with real devices with DMA instead of copying each individual page? (Minchan Kim)

The one-page-at-a-time copy semantics simplifies the implementation on both the frontend and backend and also allows the backend to do fancy things on-the-fly like page compression and page deduplication. And since the data is "gone" (copied into/out of the pageframe) before the cleancache get/put call returns, a great deal of race conditions and potential coherency issues are avoided. While the interface seems odd for a "real device" or for real kernel-addressable RAM, it makes perfect sense for transcendent memory.

- Why is non-shared cleancache "exclusive"? And where is the page "invalidated" after a "get"? (Minchan Kim)

The main reason is to free up space in transcendent memory and to avoid unnecessary cleancache_invalidate calls. If you want inclusive, the page can be "put" immediately following the "get". If put-after-get for inclusive becomes common, the interface could be easily extended to add a "get_no_invalidate" call.

The invalidate is done by the cleancache backend implementation.

- What's the performance impact?

Performance analysis has been presented at OLS'09 and LCA'10. Briefly, performance gains can be significant on most workloads, especially when memory pres-

sure is high (e.g. when RAM is overcommitted in a virtual workload); and because the hooks are invoked primarily in place of or in addition to a disk read/write, overhead is negligible even in worst case workloads. Basically cleancache replaces I/O with memory-copy-CPU-overhead; on older single-core systems with slow memory-copy speeds, cleancache has little value, but in newer multicore machines, especially consolidated/virtualized machines, it has great value.

- How do I add cleancache support for filesystem X? (Boaz Harrash)

Filesystems that are well-behaved and conform to certain restrictions can utilize cleancache simply by making a call to cleancache_init_fs at mount time. Unusual, misbehaving, or poorly layered filesystems must either add additional hooks and/or undergo extensive additional testing…or should just not enable the optional cleancache.

Some points for a filesystem to consider:

- The FS should be block-device-based (e.g. a ram-based FS such as tmpfs should not enable cleancache)
- To ensure coherency/correctness, the FS must ensure that all file removal or truncation operations either go through VFS or add hooks to do the equivalent cleancache "invalidate" operations
- To ensure coherency/correctness, either inode numbers must be unique across the lifetime of the on-disk file OR the FS must provide an "encode_fh" function.
- The FS must call the VFS superblock alloc and deactivate routines or add hooks to do the equivalent cleancache calls done there.
- To maximize performance, all pages fetched from the FS should go through the do_mpag_readpage routine or the FS should add hooks to do the equivalent (cf. btrfs)
- Currently, the FS blocksize must be the same as PAGESIZE. This is not an architectural restriction, but no backends currently support anything different.
- A clustered FS should invoke the "shared_init_fs" cleancache hook to get best performance for some backends.
- Why not use the KVA of the inode as the key? (Christoph Hellwig)

If cleancache would use the inode virtual address instead of inode/filehandle, the pool id could be eliminated. But, this won't work because cleancache retains pagecache data pages persistently even when the inode has been pruned from the inode unused list, and only invalidates the data page if the file gets removed/truncated. So if cleancache used the inode kva, there would be potential coherency issues if/when the inode kva is reused for a different file. Alternately, if cleancache invalidated the pages when the inode kva was freed, much of the value of cleancache would be lost because the cache of pages in cleanache is potentially much larger than the kernel pagecache and is most useful if the pages survive inode cache removal.

- Why is a global variable required?

The cleancache_enabled flag is checked in all of the frequently-used cleancache hooks. The alternative is a function call to check a static variable. Since clean-

cache is enabled dynamically at runtime, systems that don't enable cleancache would suffer thousands (possibly tens-of-thousands) of unnecessary function calls per second. So the global variable allows cleancache to be enabled by default at compile time, but have insignificant performance impact when cleancache remains disabled at runtime.

- Does cleanache work with KVM?

The memory model of KVM is sufficiently different that a cleancache backend may have less value for KVM. This remains to be tested, especially in an overcommitted system.

- Does cleancache work in userspace? It sounds useful for memory hungry caches like web browsers. (Jamie Lokier)

No plans yet, though we agree it sounds useful, at least for apps that bypass the page cache (e.g. O_DIRECT).

Last updated: Dan Magenheimer, April 13 2011

2.4 Free Page Reporting

Free page reporting is an API by which a device can register to receive lists of pages that are currently unused by the system. This is useful in the case of virtualization where a guest is then able to use this data to notify the hypervisor that it is no longer using certain pages in memory.

For the driver, typically a balloon driver, to use of this functionality it will allocate and initialize a page_reporting_dev_info structure. The field within the structure it will populate is the "report"function pointer used to process the scatterlist. It must also guarantee that it can handle at least PAGE_REPORTING_CAPACITY worth of scatterlist entries per call to the function. A call to page_reporting_register will register the page reporting interface with the reporting framework assuming no other page reporting devices are already registered.

Once registered the page reporting API will begin reporting batches of pages to the driver. The API will start reporting pages 2 seconds after the interface is registered and will continue to do so 2 seconds after any page of a sufficiently high order is freed.

Pages reported will be stored in the scatterlist passed to the reporting function with the final entry having the end bit set in entry nent - 1. While pages are being processed by the report function they will not be accessible to the allocator. Once the report function has been completed the pages will be returned to the free area from which they were obtained.

Prior to removing a driver that is making use of free page reporting it is necessary to call page_reporting_unregister to have the page_reporting_dev_info structure that is currently in use by free page reporting removed. Doing this will prevent further reports from being issued via the interface. If another driver or the same driver is registered it is possible for it to resume where the previous driver had left off in terms of reporting free pages.

Alexander Duyck, Dec 04, 2019

2.5 Frontswap

Frontswap provides a "transcendent memory" interface for swap pages. In some environments, dramatic performance savings may be obtained because swapped pages are saved in RAM (or a RAM-like device) instead of a swap disk.

(Note, frontswap – and Cleancache (merged at 3.0) – are the "frontends" and the only necessary changes to the core kernel for transcendent memory; all other supporting code – the "backends" – is implemented as drivers. See the LWN.net article Transcendent memory in a nutshell for a detailed overview of frontswap and related kernel parts)

Frontswap is so named because it can be thought of as the opposite of a "backing" store for a swap device. The storage is assumed to be a synchronous concurrency-safe page-oriented "pseudo-RAM device" conforming to the requirements of transcendent memory (such as Xen's "tmem", or in-kernel compressed memory, aka "zcache", or future RAM-like devices); this pseudo-RAM device is not directly accessible or addressable by the kernel and is of unknown and possibly time-varying size. The driver links itself to frontswap by calling frontswap_register_ops to set the frontswap_ops funcs appropriately and the functions it provides must conform to certain policies as follows:

An "init" prepares the device to receive frontswap pages associated with the specified swap device number (aka "type"). A "store" will copy the page to transcendent memory and associate it with the type and offset associated with the page. A "load" will copy the page, if found, from transcendent memory into kernel memory, but will NOT remove the page from transcendent memory. An "invalidate_page" will remove the page from transcendent memory and an "invalidate_area" will remove ALL pages associated with the swap type (e.g., like swapoff) and notify the "device" to refuse further stores with that swap type.

Once a page is successfully stored, a matching load on the page will normally succeed. So when the kernel finds itself in a situation where it needs to swap out a page, it first attempts to use frontswap. If the store returns success, the data has been successfully saved to transcendent memory and a disk write and, if the data is later read back, a disk read are avoided. If a store returns failure, transcendent memory has rejected the data, and the page can be written to swap as usual.

If a backend chooses, frontswap can be configured as a "writethrough cache" by calling frontswap_writethrough(). In this mode, the reduction in swap device writes is lost (and also a non-trivial performance advantage) in order to allow the backend to arbitrarily "reclaim" space used to store frontswap pages to more completely manage its memory usage.

Note that if a page is stored and the page already exists in transcendent memory (a "duplicate" store), either the store succeeds and the data is overwritten, or the store fails AND the page is invalidated. This ensures stale data may never be obtained from frontswap.

If properly configured, monitoring of frontswap is done via debugfs in the /sys/kernel/debug/frontswap directory. The effectiveness of frontswap can be measured (across all swap devices) with:

failed_stores how many store attempts have failed

loads how many loads were attempted (all should succeed)

succ_stores how many store attempts have succeeded

invalidates how many invalidates were attempted

A backend implementation may provide additional metrics.

2.5.1 FAQ

- Where’s the value?

When a workload starts swapping, performance falls through the floor. Frontswap significantly increases performance in many such workloads by providing a clean, dynamic interface to read and write swap pages to “transcendent memory” that is otherwise not directly addressable to the kernel. This interface is ideal when data is transformed to a different form and size (such as with compression) or secretly moved (as might be useful for write-balancing for some RAM-like devices). Swap pages (and evicted page-cache pages) are a great use for this kind of slower-than-RAM- but-much-faster-than-disk “pseudo-RAM device” and the frontswap (and cleancache) interface to transcendent memory provides a nice way to read and write – and indirectly “name” – the pages.

Frontswap – and cleancache – with a fairly small impact on the kernel, provides a huge amount of flexibility for more dynamic, flexible RAM utilization in various system configurations:

In the single kernel case, aka “zcache”, pages are compressed and stored in local memory, thus increasing the total anonymous pages that can be safely kept in RAM. Zcache essentially trades off CPU cycles used in compression/decompression for better memory utilization. Benchmarks have shown little or no impact when memory pressure is low while providing a significant performance improvement (25%+) on some workloads under high memory pressure.

“RAMster”builds on zcache by adding “peer-to-peer”transcendent memory support for clustered systems. Frontswap pages are locally compressed as in zcache, but then “remotified” to another system’s RAM. This allows RAM to be dynamically load-balanced back-and-forth as needed, i.e. when system A is overcommitted, it can swap to system B, and vice versa. RAMster can also be configured as a memory server so many servers in a cluster can swap, dynamically as needed, to a single server configured with a large amount of RAM⋯without pre-configuring how much of the RAM is available for each of the clients!

In the virtual case, the whole point of virtualization is to statistically multiplex physical resources across the varying demands of multiple virtual machines. This is really hard to do with RAM and efforts to do it well with no kernel changes have essentially failed (except in some well-publicized special-case workloads). Specifically, the Xen Transcendent Memory backend allows otherwise “fallow” hypervisor-owned RAM to not only be “time-shared” between multiple virtual machines, but the pages can be compressed and deduplicated to optimize RAM utilization. And when guest OS’s are induced to surrender underutilized RAM (e.g. with “selfballooning”), sudden unexpected memory pressure may result in swapping; frontswap allows those pages to be swapped to and from hypervisor RAM (if overall host system memory conditions allow), thus mitigating the potentially awful performance impact of unplanned swapping.

A KVM implementation is underway and has been RFC'ed to lkml. And, using frontswap, investigation is also underway on the use of NVM as a memory extension technology.

- Sure there may be performance advantages in some situations, but what's the space/time overhead of frontswap?

If CONFIG_FRONTSWAP is disabled, every frontswap hook compiles into nothingness and the only overhead is a few extra bytes per swapon'ed swap device. If CONFIG_FRONTSWAP is enabled but no frontswap "backend" registers, there is one extra global variable compared to zero for every swap page read or written. If CONFIG_FRONTSWAP is enabled AND a frontswap backend registers AND the backend fails every "store" request (i.e. provides no memory despite claiming it might), CPU overhead is still negligible – and since every frontswap fail precedes a swap page write-to-disk, the system is highly likely to be I/O bound and using a small fraction of a percent of a CPU will be irrelevant anyway.

As for space, if CONFIG_FRONTSWAP is enabled AND a frontswap backend registers, one bit is allocated for every swap page for every swap device that is swapon'd. This is added to the EIGHT bits (which was sixteen until about 2.6.34) that the kernel already allocates for every swap page for every swap device that is swapon'd. (Hugh Dickins has observed that frontswap could probably steal one of the existing eight bits, but let's worry about that minor optimization later.) For very large swap disks (which are rare) on a standard 4K pagesize, this is 1MB per 32GB swap.

When swap pages are stored in transcendent memory instead of written out to disk, there is a side effect that this may create more memory pressure that can potentially outweigh the other advantages. A backend, such as zcache, must implement policies to carefully (but dynamically) manage memory limits to ensure this doesn't happen.

- OK, how about a quick overview of what this frontswap patch does in terms that a kernel hacker can grok?

Let's assume that a frontswap "backend" has registered during kernel initialization; this registration indicates that this frontswap backend has access to some "memory" that is not directly accessible by the kernel. Exactly how much memory it provides is entirely dynamic and random.

Whenever a swap-device is swapon'd frontswap_init() is called, passing the swap device number (aka "type") as a parameter. This notifies frontswap to expect attempts to "store" swap pages associated with that number.

Whenever the swap subsystem is readying a page to write to a swap device (c.f swap_writepage()), frontswap_store is called. Frontswap consults with the frontswap backend and if the backend says it does NOT have room, frontswap_store returns -1 and the kernel swaps the page to the swap device as normal. Note that the response from the frontswap backend is unpredictable to the kernel; it may choose to never accept a page, it could accept every ninth page, or it might accept every page. But if the backend does accept a page, the data from the page has already been copied and associated with the type and offset, and the backend guarantees the persistence of the data. In this case, frontswap sets a bit in the "frontswap_map" for the swap device corresponding to the page offset on the swap device to which it would otherwise have written the data.

When the swap subsystem needs to swap-in a page (swap_readpage()), it first calls frontswap_load() which checks the frontswap_map to see if the page was earlier accepted by the frontswap backend. If it was, the page of data is filled from the frontswap backend and the swap-in is complete. If not, the normal swap-in code is executed to obtain the page of data from the real swap device.

So every time the frontswap backend accepts a page, a swap device read and (potentially) a swap device write are replaced by a "frontswap backend store" and (possibly) a "frontswap backend loads", which are presumably much faster.

- Can't frontswap be configured as a "special" swap device that is just higher priority than any real swap device (e.g. like zswap, or maybe swap-over-nbd/NFS)?

No. First, the existing swap subsystem doesn't allow for any kind of swap hierarchy. Perhaps it could be rewritten to accommodate a hierarchy, but this would require fairly drastic changes. Even if it were rewritten, the existing swap subsystem uses the block I/O layer which assumes a swap device is fixed size and any page in it is linearly addressable. Frontswap barely touches the existing swap subsystem, and works around the constraints of the block I/O subsystem to provide a great deal of flexibility and dynamicity.

For example, the acceptance of any swap page by the frontswap backend is entirely unpredictable. This is critical to the definition of frontswap backends because it grants completely dynamic discretion to the backend. In zcache, one cannot know a priori how compressible a page is. "Poorly" compressible pages can be rejected, and "poorly" can itself be defined dynamically depending on current memory constraints.

Further, frontswap is entirely synchronous whereas a real swap device is, by definition, asynchronous and uses block I/O. The block I/O layer is not only unnecessary, but may perform "optimizations" that are inappropriate for a RAM-oriented device including delaying the write of some pages for a significant amount of time. Synchrony is required to ensure the dynamicity of the backend and to avoid thorny race conditions that would unnecessarily and greatly complicate frontswap and/or the block I/O subsystem. That said, only the initial "store" and "load" operations need be synchronous. A separate asynchronous thread is free to manipulate the pages stored by frontswap. For example, the "remotification" thread in RAMster uses standard asynchronous kernel sockets to move compressed frontswap pages to a remote machine. Similarly, a KVM guest-side implementation could do in-guest compression and use "batched" hypercalls.

In a virtualized environment, the dynamicity allows the hypervisor (or host OS) to do "intelligent overcommit". For example, it can choose to accept pages only until host-swapping might be imminent, then force guests to do their own swapping.

There is a downside to the transcendent memory specifications for frontswap: Since any "store" might fail, there must always be a real slot on a real swap device to swap the page. Thus frontswap must be implemented as a "shadow" to every swapon'd device with the potential capability of holding every page that the swap device might have held and the possibility that it might hold no pages at all. This means that frontswap cannot contain more pages than the total of swapon'd swap devices. For example, if NO swap device is configured on some installation, frontswap is useless. Swapless portable devices can still use frontswap but a backend for such devices must configure some kind of "ghost" swap device and

ensure that it is never used.

- Why this weird definition about “duplicate stores”? If a page has been previously successfully stored, can’t it always be successfully overwritten?

Nearly always it can, but no, sometimes it cannot. Consider an example where data is compressed and the original 4K page has been compressed to 1K. Now an attempt is made to overwrite the page with data that is non-compressible and so would take the entire 4K. But the backend has no more space. In this case, the store must be rejected. Whenever frontswap rejects a store that would overwrite, it also must invalidate the old data and ensure that it is no longer accessible. Since the swap subsystem then writes the new data to the read swap device, this is the correct course of action to ensure coherency.

- What is frontswap_shrink for?

When the (non-frontswap) swap subsystem swaps out a page to a real swap device, that page is only taking up low-value pre-allocated disk space. But if frontswap has placed a page in transcendent memory, that page may be taking up valuable real estate. The frontswap_shrink routine allows code outside of the swap subsystem to force pages out of the memory managed by frontswap and back into kernel-addressable memory. For example, in RAMster, a “suction driver” thread will attempt to “repatriate” pages sent to a remote machine back to the local machine; this is driven using the frontswap_shrink mechanism when memory pressure subsides.

- Why does the frontswap patch create the new include file swapfile.h?

The frontswap code depends on some swap-subsystem-internal data structures that have, over the years, moved back and forth between static and global. This seemed a reasonable compromise: Define them as global but declare them in a new include file that isn’t included by the large number of source files that include swap.h.

Dan Magenheimer, last updated April 9, 2012

2.6 High Memory Handling

By: Peter Zijlstra <a.p.zijlstra@chello.nl>

2.6.1 What Is High Memory?

High memory (highmem) is used when the size of physical memory approaches or exceeds the maximum size of virtual memory. At that point it becomes impossible for the kernel to keep all of the available physical memory mapped at all times. This means the kernel needs to start using temporary mappings of the pieces of physical memory that it wants to access.

The part of (physical) memory not covered by a permanent mapping is what we refer to as 'highmem'. There are various architecture dependent constraints on where exactly that border lies.

In the i386 arch, for example, we choose to map the kernel into every process's VM space so that we don't have to pay the full TLB invalidation costs for kernel entry/exit. This means the available virtual memory space (4GiB on i386) has to be divided between user and kernel space.

The traditional split for architectures using this approach is 3:1, 3GiB for userspace and the top 1GiB for kernel space:

```
+--------+ 0xffffffff
| Kernel |
+--------+ 0xc0000000
|        |
| User   |
|        |
+--------+ 0x00000000
```

This means that the kernel can at most map 1GiB of physical memory at any one time, but because we need virtual address space for other things - including temporary maps to access the rest of the physical memory - the actual direct map will typically be less (usually around ~896MiB).

Other architectures that have mm context tagged TLBs can have separate kernel and user maps. Some hardware (like some ARMs), however, have limited virtual space when they use mm context tags.

2.6.2 Temporary Virtual Mappings

The kernel contains several ways of creating temporary mappings:

- vmap(). This can be used to make a long duration mapping of multiple physical pages into a contiguous virtual space. It needs global synchronization to unmap.
- kmap(). This permits a short duration mapping of a single page. It needs global synchronization, but is amortized somewhat. It is also prone to deadlocks when using in a nested fashion, and so it is not recommended for new code.
- kmap_atomic(). This permits a very short duration mapping of a single page. Since the mapping is restricted to the CPU that issued it, it performs well, but the issuing task is therefore required to stay on that CPU until it has finished, lest some other task displace its mappings.

kmap_atomic() may also be used by interrupt contexts, since it is does not sleep and the caller may not sleep until after kunmap_atomic() is called.

It may be assumed that k[un]map_atomic() won't fail.

2.6.3 Using kmap_atomic

When and where to use kmap_atomic() is straightforward. It is used when code wants to access the contents of a page that might be allocated from high memory (see __GFP_HIGHMEM), for example a page in the pagecache. The API has two functions, and they can be used in a manner similar to the following:

```
/* Find the page of interest. */
struct page *page = find_get_page(mapping, offset);

/* Gain access to the contents of that page. */
void *vaddr = kmap_atomic(page);

/* Do something to the contents of that page. */
memset(vaddr, 0, PAGE_SIZE);

/* Unmap that page. */
kunmap_atomic(vaddr);
```

Note that the kunmap_atomic() call takes the result of the kmap_atomic() call not the argument.

If you need to map two pages because you want to copy from one page to another you need to keep the kmap_atomic calls strictly nested, like:

```
vaddr1 = kmap_atomic(page1);
vaddr2 = kmap_atomic(page2);

memcpy(vaddr1, vaddr2, PAGE_SIZE);

kunmap_atomic(vaddr2);
kunmap_atomic(vaddr1);
```

2.6.4 Cost of Temporary Mappings

The cost of creating temporary mappings can be quite high. The arch has to manipulate the kernel's page tables, the data TLB and/or the MMU's registers.

If CONFIG_HIGHMEM is not set, then the kernel will try and create a mapping simply with a bit of arithmetic that will convert the page struct address into a pointer to the page contents rather than juggling mappings about. In such a case, the unmap operation may be a null operation.

If CONFIG_MMU is not set, then there can be no temporary mappings and no highmem. In such a case, the arithmetic approach will also be used.

2.6.5 i386 PAE

The i386 arch, under some circumstances, will permit you to stick up to 64GiB of RAM into your 32-bit machine. This has a number of consequences:

- Linux needs a page-frame structure for each page in the system and the page-frames need to live in the permanent mapping, which means:
- you can have 896M/sizeof(struct page) page-frames at most; with struct page being 32-bytes that would end up being something in the order of 112G worth of pages; the kernel, however, needs to store more than just page-frames in that memory…
- PAE makes your page tables larger - which slows the system down as more data has to be accessed to traverse in TLB fills and the like. One advantage is that PAE has more PTE bits and can provide advanced features like NX and PAT.

The general recommendation is that you don't use more than 8GiB on a 32-bit machine - although more might work for you and your workload, you're pretty much on your own - don't expect kernel developers to really care much if things come apart.

2.7 Heterogeneous Memory Management (HMM)

Provide infrastructure and helpers to integrate non-conventional memory (device memory like GPU on board memory) into regular kernel path, with the cornerstone of this being specialized struct page for such memory (see sections 5 to 7 of this document).

HMM also provides optional helpers for SVM (Share Virtual Memory), i.e., allowing a device to transparently access program addresses coherently with the CPU meaning that any valid pointer on the CPU is also a valid pointer for the device. This is becoming mandatory to simplify the use of advanced heterogeneous computing where GPU, DSP, or FPGA are used to perform various computations on behalf of a process.

This document is divided as follows: in the first section I expose the problems related to using device specific memory allocators. In the second section, I expose the hardware limitations that are inherent to many platforms. The third section gives an overview of the HMM design. The fourth section explains how CPU page-table mirroring works and the purpose of HMM in this context. The fifth section deals with how device memory is represented inside the kernel. Finally, the last section presents a new migration helper that allows leveraging the device DMA engine.

- Problems of using a device specific memory allocator
- I/O bus, device memory characteristics
- Shared address space and migration
- Address space mirroring implementation and API

- Leverage default_flags and pfn_flags_mask
- Represent and manage device memory from core kernel point of view
- Migration to and from device memory
- Memory cgroup (memcg) and rss accounting

2.7.1 Problems of using a device specific memory allocator

Devices with a large amount of on board memory (several gigabytes) like GPUs have historically managed their memory through dedicated driver specific APIs. This creates a disconnect between memory allocated and managed by a device driver and regular application memory (private anonymous, shared memory, or regular file backed memory). From here on I will refer to this aspect as split address space. I use shared address space to refer to the opposite situation: i.e., one in which any application memory region can be used by a device transparently.

Split address space happens because devices can only access memory allocated through a device specific API. This implies that all memory objects in a program are not equal from the device point of view which complicates large programs that rely on a wide set of libraries.

Concretely, this means that code that wants to leverage devices like GPUs needs to copy objects between generically allocated memory (malloc, mmap private, mmap share) and memory allocated through the device driver API (this still ends up with an mmap but of the device file).

For flat data sets (array, grid, image, ⋯) this isn’t too hard to achieve but for complex data sets (list, tree, ⋯) it’s hard to get right. Duplicating a complex data set needs to re-map all the pointer relations between each of its elements. This is error prone and programs get harder to debug because of the duplicate data set and addresses.

Split address space also means that libraries cannot transparently use data they are getting from the core program or another library and thus each library might have to duplicate its input data set using the device specific memory allocator. Large projects suffer from this and waste resources because of the various memory copies.

Duplicating each library API to accept as input or output memory allocated by each device specific allocator is not a viable option. It would lead to a combinatorial explosion in the library entry points.

Finally, with the advance of high level language constructs (in C++ but in other languages too) it is now possible for the compiler to leverage GPUs and other devices without programmer knowledge. Some compiler identified patterns are only do-able with a shared address space. It is also more reasonable to use a shared address space for all other patterns.

2.7.2 I/O bus, device memory characteristics

I/O buses cripple shared address spaces due to a few limitations. Most I/O buses only allow basic memory access from device to main memory; even cache coherency is often optional. Access to device memory from a CPU is even more limited. More often than not, it is not cache coherent.

If we only consider the PCIE bus, then a device can access main memory (often through an IOMMU) and be cache coherent with the CPUs. However, it only allows a limited set of atomic operations from the device on main memory. This is worse in the other direction: the CPU can only access a limited range of the device memory and cannot perform atomic operations on it. Thus device memory cannot be considered the same as regular memory from the kernel point of view.

Another crippling factor is the limited bandwidth (~32GBytes/s with PCIE 4.0 and 16 lanes). This is 33 times less than the fastest GPU memory (1 TBytes/s). The final limitation is latency. Access to main memory from the device has an order of magnitude higher latency than when the device accesses its own memory.

Some platforms are developing new I/O buses or additions/modifications to PCIE to address some of these limitations (OpenCAPI, CCIX). They mainly allow two-way cache coherency between CPU and device and allow all atomic operations the architecture supports. Sadly, not all platforms are following this trend and some major architectures are left without hardware solutions to these problems.

So for shared address space to make sense, not only must we allow devices to access any memory but we must also permit any memory to be migrated to device memory while the device is using it (blocking CPU access while it happens).

2.7.3 Shared address space and migration

HMM intends to provide two main features. The first one is to share the address space by duplicating the CPU page table in the device page table so the same address points to the same physical memory for any valid main memory address in the process address space.

To achieve this, HMM offers a set of helpers to populate the device page table while keeping track of CPU page table updates. Device page table updates are not as easy as CPU page table updates. To update the device page table, you must allocate a buffer (or use a pool of pre-allocated buffers) and write GPU specific commands in it to perform the update (unmap, cache invalidations, and flush, ⋯). This cannot be done through common code for all devices. Hence why HMM provides helpers to factor out everything that can be while leaving the hardware specific details to the device driver.

The second mechanism HMM provides is a new kind of ZONE_DEVICE memory that allows allocating a struct page for each page of device memory. Those pages are special because the CPU cannot map them. However, they allow migrating main memory to device memory using existing migration mechanisms and everything looks like a page that is swapped out to disk from the CPU point of view. Using a struct page gives the easiest and cleanest integration with existing mm mechanisms. Here again, HMM only provides helpers, first to hotplug new ZONE_DEVICE memory for the device memory and second to perform migration. Policy decisions of what and when to migrate is left to the device driver.

Note that any CPU access to a device page triggers a page fault and a migration back to main memory. For example, when a page backing a given CPU address A is migrated from a main memory page to a device page, then any CPU access to address A triggers a page fault and initiates a migration back to main memory.

With these two features, HMM not only allows a device to mirror process address space and keeps both CPU and device page tables synchronized, but also leverages device memory by migrating the part of the data set that is actively being used by the device.

2.7.4 Address space mirroring implementation and API

Address space mirroring' s main objective is to allow duplication of a range of CPU page table into a device page table; HMM helps keep both synchronized. A device driver that wants to mirror a process address space must start with the registration of a mmu_interval_notifier:

```
int mmu_interval_notifier_insert(struct mmu_interval_notifier *interval_
↪sub,
                                 struct mm_struct *mm, unsigned long start,
                                 unsigned long length,
                                 const struct mmu_interval_notifier_ops␣
↪*ops);
```

During the ops->invalidate() callback the device driver must perform the update action to the range (mark range read only, or fully unmap, etc.). The device must complete the update before the driver callback returns.

When the device driver wants to populate a range of virtual addresses, it can use:

```
int hmm_range_fault(struct hmm_range *range);
```

It will trigger a page fault on missing or read-only entries if write access is requested (see below). Page faults use the generic mm page fault code path just like a CPU page fault.

Both functions copy CPU page table entries into their pfns array argument. Each entry in that array corresponds to an address in the virtual range. HMM provides a set of flags to help the driver identify special CPU page table entries.

Locking within the sync_cpu_device_pagetables() callback is the most important aspect the driver must respect in order to keep things properly synchronized. The usage pattern is:

```
int driver_populate_range(...)
{
     struct hmm_range range;
     ...

     range.notifier = &interval_sub;
     range.start = ...;
     range.end = ...;
     range.hmm_pfns = ...;

     if (!mmget_not_zero(interval_sub->notifier.mm))
```

(continues on next page)

(continued from previous page)

```
        return -EFAULT;

again:
     range.notifier_seq = mmu_interval_read_begin(&interval_sub);
     mmap_read_lock(mm);
     ret = hmm_range_fault(&range);
     if (ret) {
         mmap_read_unlock(mm);
         if (ret == -EBUSY)
                goto again;
         return ret;
     }
     mmap_read_unlock(mm);

     take_lock(driver->update);
     if (mmu_interval_read_retry(&ni, range.notifier_seq) {
         release_lock(driver->update);
         goto again;
     }

     /* Use pfns array content to update device page table,
      * under the update lock */

     release_lock(driver->update);
     return 0;
}
```

The driver->update lock is the same lock that the driver takes inside its invalidate() callback. That lock must be held before calling mmu_interval_read_retry() to avoid any race with a concurrent CPU page table update.

2.7.5 Leverage default_flags and pfn_flags_mask

The hmm_range struct has 2 fields, default_flags and pfn_flags_mask, that specify fault or snapshot policy for the whole range instead of having to set them for each entry in the pfns array.

For instance if the device driver wants pages for a range with at least read permission, it sets:

```
range->default_flags = HMM_PFN_REQ_FAULT;
range->pfn_flags_mask = 0;
```

and calls hmm_range_fault() as described above. This will fill fault all pages in the range with at least read permission.

Now let’ s say the driver wants to do the same except for one page in the range for which it wants to have write permission. Now driver set:

```
range->default_flags = HMM_PFN_REQ_FAULT;
range->pfn_flags_mask = HMM_PFN_REQ_WRITE;
range->pfns[index_of_write] = HMM_PFN_REQ_WRITE;
```

With this, HMM will fault in all pages with at least read (i.e., valid) and for the address == range->start + (index_of_write << PAGE_SHIFT) it will fault with write

permission i.e., if the CPU pte does not have write permission set then HMM will call handle_mm_fault().

After hmm_range_fault completes the flag bits are set to the current state of the page tables, ie HMM_PFN_VALID | HMM_PFN_WRITE will be set if the page is writable.

2.7.6 Represent and manage device memory from core kernel point of view

Several different designs were tried to support device memory. The first one used a device specific data structure to keep information about migrated memory and HMM hooked itself in various places of mm code to handle any access to addresses that were backed by device memory. It turns out that this ended up replicating most of the fields of struct page and also needed many kernel code paths to be updated to understand this new kind of memory.

Most kernel code paths never try to access the memory behind a page but only care about struct page contents. Because of this, HMM switched to directly using struct page for device memory which left most kernel code paths unaware of the difference. We only need to make sure that no one ever tries to map those pages from the CPU side.

2.7.7 Migration to and from device memory

Because the CPU cannot access device memory, migration must use the device DMA engine to perform copy from and to device memory. For this we need to use migrate_vma_setup(), migrate_vma_pages(), and migrate_vma_finalize() helpers.

2.7.8 Memory cgroup (memcg) and rss accounting

For now, device memory is accounted as any regular page in rss counters (either anonymous if device page is used for anonymous, file if device page is used for file backed page, or shmem if device page is used for shared memory). This is a deliberate choice to keep existing applications, that might start using device memory without knowing about it, running unimpacted.

A drawback is that the OOM killer might kill an application using a lot of device memory and not a lot of regular system memory and thus not freeing much system memory. We want to gather more real world experience on how applications and system react under memory pressure in the presence of device memory before deciding to account device memory differently.

Same decision was made for memory cgroup. Device memory pages are accounted against same memory cgroup a regular page would be accounted to. This does simplify migration to and from device memory. This also means that migration back from device memory to regular memory cannot fail because it would go above memory cgroup limit. We might revisit this choice latter on once we get more experience in how device memory is used and its impact on memory resource control.

Note that device memory can never be pinned by a device driver nor through GUP and thus such memory is always free upon process exit. Or when last reference is dropped in case of shared memory or file backed memory.

2.8 hwpoison

2.8.1 What is hwpoison?

Upcoming Intel CPUs have support for recovering from some memory errors (`MCA recovery`). This requires the OS to declare a page "poisoned", kill the processes associated with it and avoid using it in the future.

This patchkit implements the necessary infrastructure in the VM.

To quote the overview comment:

```
High level machine check handler. Handles pages reported by the
hardware as being corrupted usually due to a 2bit ECC memory or cache
failure.

This focusses on pages detected as corrupted in the background.
When the current CPU tries to consume corruption the currently
running process can just be killed directly instead. This implies
that if the error cannot be handled for some reason it's safe to
just ignore it because no corruption has been consumed yet. Instead
when that happens another machine check will happen.

Handles page cache pages in various states. The tricky part
here is that we can access any page asynchronous to other VM
users, because memory failures could happen anytime and anywhere,
possibly violating some of their assumptions. This is why this code
has to be extremely careful. Generally it tries to use normal locking
rules, as in get the standard locks, even if that means the
error handling takes potentially a long time.

Some of the operations here are somewhat inefficient and have non
linear algorithmic complexity, because the data structures have not
been optimized for this case. This is in particular the case
for the mapping from a vma to a process. Since this case is expected
to be rare we hope we can get away with this.
```

The code consists of a the high level handler in mm/memory-failure.c, a new page poison bit and various checks in the VM to handle poisoned pages.

The main target right now is KVM guests, but it works for all kinds of applications. KVM support requires a recent qemu-kvm release.

For the KVM use there was need for a new signal type so that KVM can inject the machine check into the guest with the proper address. This in theory allows other applications to handle memory failures too. The expection is that near all applications won't do that, but some very specialized ones might.

2.8.2 Failure recovery modes

There are two (actually three) modes memory failure recovery can be in:

vm.memory_failure_recovery sysctl set to zero: All memory failures cause a panic. Do not attempt recovery. (on x86 this can be also affected by the tolerant level of the MCE subsystem)

early kill (can be controlled globally and per process) Send SIGBUS to the application as soon as the error is detected This allows applications who can process memory errors in a gentle way (e.g. drop affected object) This is the mode used by KVM qemu.

late kill Send SIGBUS when the application runs into the corrupted page. This is best for memory error unaware applications and default Note some pages are always handled as late kill.

2.8.3 User control

vm.memory_failure_recovery See sysctl.txt

vm.memory_failure_early_kill Enable early kill mode globally

PR_MCE_KILL Set early/late kill mode/revert to system default

arg1: PR_MCE_KILL_CLEAR: Revert to system default

arg1: PR_MCE_KILL_SET: arg2 defines thread specific mode

PR_MCE_KILL_EARLY: Early kill

PR_MCE_KILL_LATE: Late kill

PR_MCE_KILL_DEFAULT Use system global default

Note that if you want to have a dedicated thread which handles the SIGBUS(BUS_MCEERR_AO) on behalf of the process, you should call prctl(PR_MCE_KILL_EARLY) on the designated thread. Otherwise, the SIGBUS is sent to the main thread.

PR_MCE_KILL_GET return current mode

2.8.4 Testing

- madvise(MADV_HWPOISON, ⋯.) (as root) - Poison a page in the process for testing
- hwpoison-inject module through debugfs `/sys/kernel/debug/hwpoison/`

corrupt-pfn Inject hwpoison fault at PFN echoed into this file. This does some early filtering to avoid corrupted unintended pages in test suites.

unpoison-pfn Software-unpoison page at PFN echoed into this file. This way a page can be reused again. This only works for Linux injected failures, not for real memory failures.

Note these injection interfaces are not stable and might change between kernel versions

corrupt-filter-dev-major, corrupt-filter-dev-minor Only handle memory failures to pages associated with the file system defined by block device major/minor. -1U is the wildcard value. This should be only used for testing with artificial injection.

corrupt-filter-memcg Limit injection to pages owned by memgroup. Specified by inode number of the memcg.

Example:

```
mkdir /sys/fs/cgroup/mem/hwpoison

usemem -m 100 -s 1000 &
echo `jobs -p` > /sys/fs/cgroup/mem/hwpoison/tasks

memcg_ino=$(ls -id /sys/fs/cgroup/mem/hwpoison | cut -f1 -d' ')
echo $memcg_ino > /debug/hwpoison/corrupt-filter-memcg

page-types -p `pidof init`   --hwpoison  # shall do nothing
page-types -p `pidof usemem` --hwpoison  # poison its pages
```

corrupt-filter-flags-mask, corrupt-filter-flags-value When specified, only poison pages if ((page_flags & mask) == value). This allows stress testing of many kinds of pages. The page_flags are the same as in /proc/kpageflags. The flag bits are defined in include/linux/kernel-page-flags.h and documented in Documentation/admin-guide/mm/pagemap.rst

- Architecture specific MCE injector

  x86 has mce-inject, mce-test

  Some portable hwpoison test programs in mce-test, see below.

2.8.5 References

http://halobates.de/mce-lc09-2.pdf Overview presentation from LinuxCon 09

git://git.kernel.org/pub/scm/utils/cpu/mce/mce-test.git Test suite (hwpoison specific portable tests in tsrc)

git://git.kernel.org/pub/scm/utils/cpu/mce/mce-inject.git x86 specific injector

2.8.6 Limitations

- Not all page types are supported and never will. Most kernel internal objects cannot be recovered, only LRU pages for now.
- Right now hugepage support is missing.

—Andi Kleen, Oct 2009

2.9 Hugetlbfs Reservation

2.9.1 Overview

Huge pages as described at hugetlbpage are typically preallocated for application use. These huge pages are instantiated in a task's address space at page fault time if the VMA indicates huge pages are to be used. If no huge page exists at page fault time, the task is sent a SIGBUS and often dies an unhappy death. Shortly after huge page support was added, it was determined that it would be better to detect a shortage of huge pages at mmap() time. The idea is that if there were not enough huge pages to cover the mapping, the mmap() would fail. This was first done with a simple check in the code at mmap() time to determine if there were enough free huge pages to cover the mapping. Like most things in the kernel, the code has evolved over time. However, the basic idea was to 'reserve' huge pages at mmap() time to ensure that huge pages would be available for page faults in that mapping. The description below attempts to describe how huge page reserve processing is done in the v4.10 kernel.

2.9.2 Audience

This description is primarily targeted at kernel developers who are modifying hugetlbfs code.

2.9.3 The Data Structures

resv_huge_pages This is a global (per-hstate) count of reserved huge pages. Reserved huge pages are only available to the task which reserved them. Therefore, the number of huge pages generally available is computed as (`free_huge_pages - resv_huge_pages`).

Reserve Map A reserve map is described by the structure:

```
struct resv_map {
        struct kref refs;
        spinlock_t lock;
        struct list_head regions;
        long adds_in_progress;
        struct list_head region_cache;
        long region_cache_count;
};
```

There is one reserve map for each huge page mapping in the system. The regions list within the resv_map describes the regions within the mapping. A region is described as:

```
struct file_region {
        struct list_head link;
        long from;
        long to;
};
```

The ‘from’ and ‘to’ fields of the file region structure are huge page indices into the mapping. Depending on the type of mapping, a region in the reserv_map may indicate reservations exist for the range, or reservations do not exist.

Flags for MAP_PRIVATE Reservations These are stored in the bottom bits of the reservation map pointer.

`#define HPAGE_RESV_OWNER (1UL << 0)` Indicates this task is the owner of the reservations associated with the mapping.

`#define HPAGE_RESV_UNMAPPED (1UL << 1)` Indicates task originally mapping this range (and creating reserves) has unmapped a page from this task (the child) due to a failed COW.

Page Flags The PagePrivate page flag is used to indicate that a huge page reservation must be restored when the huge page is freed. More details will be discussed in the “Freeing huge pages” section.

2.9.4 Reservation Map Location (Private or Shared)

A huge page mapping or segment is either private or shared. If private, it is typically only available to a single address space (task). If shared, it can be mapped into multiple address spaces (tasks). The location and semantics of the reservation map is significantly different for the two types of mappings. Location differences are:

- For private mappings, the reservation map hangs off the VMA structure. Specifically, vma->vm_private_data. This reserve map is created at the time the mapping (mmap(MAP_PRIVATE)) is created.
- For shared mappings, the reservation map hangs off the inode. Specifically, inode->i_mapping->private_data. Since shared mappings are always backed by files in the hugetlbfs filesystem, the hugetlbfs code ensures each inode contains a reservation map. As a result, the reservation map is allocated when the inode is created.

2.9.5 Creating Reservations

Reservations are created when a huge page backed shared memory segment is created (shmget(SHM_HUGETLB)) or a mapping is created via mmap(MAP_HUGETLB). These operations result in a call to the routine hugetlb_reserve_pages():

```
int hugetlb_reserve_pages(struct inode *inode,
                          long from, long to,
                          struct vm_area_struct *vma,
                          vm_flags_t vm_flags)
```

The first thing hugetlb_reserve_pages() does is check if the NORESERVE flag was specified in either the shmget() or mmap() call. If NORESERVE was specified, then this routine returns immediately as no reservations are desired.

The arguments ‘from’ and ‘to’ are huge page indices into the mapping or underlying file. For shmget(), ‘from’ is always 0 and ‘to’ corresponds to the length of the

segment/mapping. For mmap(), the offset argument could be used to specify the offset into the underlying file. In such a case, the 'from' and 'to' arguments have been adjusted by this offset.

One of the big differences between PRIVATE and SHARED mappings is the way in which reservations are represented in the reservation map.

- For shared mappings, an entry in the reservation map indicates a reservation exists or did exist for the corresponding page. As reservations are consumed, the reservation map is not modified.
- For private mappings, the lack of an entry in the reservation map indicates a reservation exists for the corresponding page. As reservations are consumed, entries are added to the reservation map. Therefore, the reservation map can also be used to determine which reservations have been consumed.

For private mappings, hugetlb_reserve_pages() creates the reservation map and hangs it off the VMA structure. In addition, the HPAGE_RESV_OWNER flag is set to indicate this VMA owns the reservations.

The reservation map is consulted to determine how many huge page reservations are needed for the current mapping/segment. For private mappings, this is always the value (to - from). However, for shared mappings it is possible that some reservations may already exist within the range (to - from). See the section Reservation Map Modifications for details on how this is accomplished.

The mapping may be associated with a subpool. If so, the subpool is consulted to ensure there is sufficient space for the mapping. It is possible that the subpool has set aside reservations that can be used for the mapping. See the section Subpool Reservations for more details.

After consulting the reservation map and subpool, the number of needed new reservations is known. The routine hugetlb_acct_memory() is called to check for and take the requested number of reservations. hugetlb_acct_memory() calls into routines that potentially allocate and adjust surplus page counts. However, within those routines the code is simply checking to ensure there are enough free huge pages to accommodate the reservation. If there are, the global reservation count resv_huge_pages is adjusted something like the following:

```
if (resv_needed <= (resv_huge_pages - free_huge_pages))
        resv_huge_pages += resv_needed;
```

Note that the global lock hugetlb_lock is held when checking and adjusting these counters.

If there were enough free huge pages and the global count resv_huge_pages was adjusted, then the reservation map associated with the mapping is modified to reflect the reservations. In the case of a shared mapping, a file_region will exist that includes the range 'from' - 'to'. For private mappings, no modifications are made to the reservation map as lack of an entry indicates a reservation exists.

If hugetlb_reserve_pages() was successful, the global reservation count and reservation map associated with the mapping will be modified as required to ensure reservations exist for the range 'from' - 'to'.

2.9.6 Consuming Reservations/Allocating a Huge Page

Reservations are consumed when huge pages associated with the reservations are allocated and instantiated in the corresponding mapping. The allocation is performed within the routine alloc_huge_page():

```
struct page *alloc_huge_page(struct vm_area_struct *vma,
                             unsigned long addr, int avoid_reserve)
```

alloc_huge_page is passed a VMA pointer and a virtual address, so it can consult the reservation map to determine if a reservation exists. In addition, alloc_huge_page takes the argument avoid_reserve which indicates reserves should not be used even if it appears they have been set aside for the specified address. The avoid_reserve argument is most often used in the case of Copy on Write and Page Migration where additional copies of an existing page are being allocated.

The helper routine vma_needs_reservation() is called to determine if a reservation exists for the address within the mapping(vma). See the section Reservation Map Helper Routines for detailed information on what this routine does. The value returned from vma_needs_reservation() is generally 0 or 1. 0 if a reservation exists for the address, 1 if no reservation exists. If a reservation does not exist, and there is a subpool associated with the mapping the subpool is consulted to determine if it contains reservations. If the subpool contains reservations, one can be used for this allocation. However, in every case the avoid_reserve argument overrides the use of a reservation for the allocation. After determining whether a reservation exists and can be used for the allocation, the routine dequeue_huge_page_vma() is called. This routine takes two arguments related to reservations:

- avoid_reserve, this is the same value/argument passed to alloc_huge_page()
- chg, even though this argument is of type long only the values 0 or 1 are passed to dequeue_huge_page_vma. If the value is 0, it indicates a reservation exists (see the section "Memory Policy and Reservations" for possible issues). If the value is 1, it indicates a reservation does not exist and the page must be taken from the global free pool if possible.

The free lists associated with the memory policy of the VMA are searched for a free page. If a page is found, the value free_huge_pages is decremented when the page is removed from the free list. If there was a reservation associated with the page, the following adjustments are made:

```
SetPagePrivate(page);   /* Indicates allocating this page consumed
                         * a reservation, and if an error is
                         * encountered such that the page must be
                         * freed, the reservation will be restored. */
resv_huge_pages--;      /* Decrement the global reservation count */
```

Note, if no huge page can be found that satisfies the VMA's memory policy an attempt will be made to allocate one using the buddy allocator. This brings up the issue of surplus huge pages and overcommit which is beyond the scope reservations. Even if a surplus page is allocated, the same reservation based adjustments as above will be made: SetPagePrivate(page) and resv_huge_pages–.

After obtaining a new huge page, (page)->private is set to the value of the subpool associated with the page if it exists. This will be used for subpool accounting when

the page is freed.

The routine vma_commit_reservation() is then called to adjust the reserve map based on the consumption of the reservation. In general, this involves ensuring the page is represented within a file_region structure of the region map. For shared mappings where the reservation was present, an entry in the reserve map already existed so no change is made. However, if there was no reservation in a shared mapping or this was a private mapping a new entry must be created.

It is possible that the reserve map could have been changed between the call to vma_needs_reservation() at the beginning of alloc_huge_page() and the call to vma_commit_reservation() after the page was allocated. This would be possible if hugetlb_reserve_pages was called for the same page in a shared mapping. In such cases, the reservation count and subpool free page count will be off by one. This rare condition can be identified by comparing the return value from vma_needs_reservation and vma_commit_reservation. If such a race is detected, the subpool and global reserve counts are adjusted to compensate. See the section Reservation Map Helper Routines for more information on these routines.

2.9.7 Instantiate Huge Pages

After huge page allocation, the page is typically added to the page tables of the allocating task. Before this, pages in a shared mapping are added to the page cache and pages in private mappings are added to an anonymous reverse mapping. In both cases, the PagePrivate flag is cleared. Therefore, when a huge page that has been instantiated is freed no adjustment is made to the global reservation count (resv_huge_pages).

2.9.8 Freeing Huge Pages

Huge page freeing is performed by the routine free_huge_page(). This routine is the destructor for hugetlbfs compound pages. As a result, it is only passed a pointer to the page struct. When a huge page is freed, reservation accounting may need to be performed. This would be the case if the page was associated with a subpool that contained reserves, or the page is being freed on an error path where a global reserve count must be restored.

The page->private field points to any subpool associated with the page. If the PagePrivate flag is set, it indicates the global reserve count should be adjusted (see the section Consuming Reservations/Allocating a Huge Page for information on how these are set).

The routine first calls hugepage_subpool_put_pages() for the page. If this routine returns a value of 0 (which does not equal the value passed 1) it indicates reserves are associated with the subpool, and this newly free page must be used to keep the number of subpool reserves above the minimum size. Therefore, the global resv_huge_pages counter is incremented in this case.

If the PagePrivate flag was set in the page, the global resv_huge_pages counter will always be incremented.

2.9.9 Subpool Reservations

There is a struct hstate associated with each huge page size. The hstate tracks all huge pages of the specified size. A subpool represents a subset of pages within a hstate that is associated with a mounted hugetlbfs filesystem.

When a hugetlbfs filesystem is mounted a min_size option can be specified which indicates the minimum number of huge pages required by the filesystem. If this option is specified, the number of huge pages corresponding to min_size are reserved for use by the filesystem. This number is tracked in the min_hpages field of a struct hugepage_subpool. At mount time, hugetlb_acct_memory(min_hpages) is called to reserve the specified number of huge pages. If they can not be reserved, the mount fails.

The routines hugepage_subpool_get/put_pages() are called when pages are obtained from or released back to a subpool. They perform all subpool accounting, and track any reservations associated with the subpool. hugepage_subpool_get/put_pages are passed the number of huge pages by which to adjust the subpool 'used page' count (down for get, up for put). Normally, they return the same value that was passed or an error if not enough pages exist in the subpool.

However, if reserves are associated with the subpool a return value less than the passed value may be returned. This return value indicates the number of additional global pool adjustments which must be made. For example, suppose a subpool contains 3 reserved huge pages and someone asks for 5. The 3 reserved pages associated with the subpool can be used to satisfy part of the request. But, 2 pages must be obtained from the global pools. To relay this information to the caller, the value 2 is returned. The caller is then responsible for attempting to obtain the additional two pages from the global pools.

2.9.10 COW and Reservations

Since shared mappings all point to and use the same underlying pages, the biggest reservation concern for COW is private mappings. In this case, two tasks can be pointing at the same previously allocated page. One task attempts to write to the page, so a new page must be allocated so that each task points to its own page.

When the page was originally allocated, the reservation for that page was consumed. When an attempt to allocate a new page is made as a result of COW, it is possible that no free huge pages are free and the allocation will fail.

When the private mapping was originally created, the owner of the mapping was noted by setting the HPAGE_RESV_OWNER bit in the pointer to the reservation map of the owner. Since the owner created the mapping, the owner owns all the reservations associated with the mapping. Therefore, when a write fault occurs and there is no page available, different action is taken for the owner and non-owner of the reservation.

In the case where the faulting task is not the owner, the fault will fail and the task will typically receive a SIGBUS.

If the owner is the faulting task, we want it to succeed since it owned the original reservation. To accomplish this, the page is unmapped from the non-owning

task. In this way, the only reference is from the owning task. In addition, the HPAGE_RESV_UNMAPPED bit is set in the reservation map pointer of the non-owning task. The non-owning task may receive a SIGBUS if it later faults on a non-present page. But, the original owner of the mapping/reservation will behave as expected.

2.9.11 Reservation Map Modifications

The following low level routines are used to make modifications to a reservation map. Typically, these routines are not called directly. Rather, a reservation map helper routine is called which calls one of these low level routines. These low level routines are fairly well documented in the source code (mm/hugetlb.c). These routines are:

```
long region_chg(struct resv_map *resv, long f, long t);
long region_add(struct resv_map *resv, long f, long t);
void region_abort(struct resv_map *resv, long f, long t);
long region_count(struct resv_map *resv, long f, long t);
```

Operations on the reservation map typically involve two operations:

1) region_chg() is called to examine the reserve map and determine how many pages in the specified range [f, t) are NOT currently represented.

   The calling code performs global checks and allocations to determine if there are enough huge pages for the operation to succeed.

2) a) If the operation can succeed, region_add() is called to actually modify the reservation map for the same range [f, t) previously passed to region_chg().

   b) If the operation can not succeed, region_abort is called for the same range [f, t) to abort the operation.

Note that this is a two step process where region_add() and region_abort() are guaranteed to succeed after a prior call to region_chg() for the same range. region_chg() is responsible for pre-allocating any data structures necessary to ensure the subsequent operations (specifically region_add())) will succeed.

As mentioned above, region_chg() determines the number of pages in the range which are NOT currently represented in the map. This number is returned to the caller. region_add() returns the number of pages in the range added to the map. In most cases, the return value of region_add() is the same as the return value of region_chg(). However, in the case of shared mappings it is possible for changes to the reservation map to be made between the calls to region_chg() and region_add(). In this case, the return value of region_add() will not match the return value of region_chg(). It is likely that in such cases global counts and subpool accounting will be incorrect and in need of adjustment. It is the responsibility of the caller to check for this condition and make the appropriate adjustments.

The routine region_del() is called to remove regions from a reservation map. It is typically called in the following situations:

- When a file in the hugetlbfs filesystem is being removed, the inode will be released and the reservation map freed. Before freeing the reservation map,

all the individual file_region structures must be freed. In this case region_del is passed the range [0, LONG_MAX).

- When a hugetlbfs file is being truncated. In this case, all allocated pages after the new file size must be freed. In addition, any file_region entries in the reservation map past the new end of file must be deleted. In this case, region_del is passed the range [new_end_of_file, LONG_MAX).
- When a hole is being punched in a hugetlbfs file. In this case, huge pages are removed from the middle of the file one at a time. As the pages are removed, region_del() is called to remove the corresponding entry from the reservation map. In this case, region_del is passed the range [page_idx, page_idx + 1).

In every case, region_del() will return the number of pages removed from the reservation map. In VERY rare cases, region_del() can fail. This can only happen in the hole punch case where it has to split an existing file_region entry and can not allocate a new structure. In this error case, region_del() will return -ENOMEM. The problem here is that the reservation map will indicate that there is a reservation for the page. However, the subpool and global reservation counts will not reflect the reservation. To handle this situation, the routine hugetlb_fix_reserve_counts() is called to adjust the counters so that they correspond with the reservation map entry that could not be deleted.

region_count() is called when unmapping a private huge page mapping. In private mappings, the lack of a entry in the reservation map indicates that a reservation exists. Therefore, by counting the number of entries in the reservation map we know how many reservations were consumed and how many are outstanding (outstanding = (end - start) - region_count(resv, start, end)). Since the mapping is going away, the subpool and global reservation counts are decremented by the number of outstanding reservations.

2.9.12 Reservation Map Helper Routines

Several helper routines exist to query and modify the reservation maps. These routines are only interested with reservations for a specific huge page, so they just pass in an address instead of a range. In addition, they pass in the associated VMA. From the VMA, the type of mapping (private or shared) and the location of the reservation map (inode or VMA) can be determined. These routines simply call the underlying routines described in the section “Reservation Map Modifications” . However, they do take into account the ‘opposite’ meaning of reservation map entries for private and shared mappings and hide this detail from the caller:

```
long vma_needs_reservation(struct hstate *h,
                           struct vm_area_struct *vma,
                           unsigned long addr)
```

This routine calls region_chg() for the specified page. If no reservation exists, 1 is returned. If a reservation exists, 0 is returned:

```
long vma_commit_reservation(struct hstate *h,
                            struct vm_area_struct *vma,
                            unsigned long addr)
```

This calls region_add() for the specified page. As in the case of region_chg and region_add, this routine is to be called after a previous call to vma_needs_reservation. It will add a reservation entry for the page. It returns 1 if the reservation was added and 0 if not. The return value should be compared with the return value of the previous call to vma_needs_reservation. An unexpected difference indicates the reservation map was modified between calls:

```
void vma_end_reservation(struct hstate *h,
                         struct vm_area_struct *vma,
                         unsigned long addr)
```

This calls region_abort() for the specified page. As in the case of region_chg and region_abort, this routine is to be called after a previous call to vma_needs_reservation. It will abort/end the in progress reservation add operation:

```
long vma_add_reservation(struct hstate *h,
                         struct vm_area_struct *vma,
                         unsigned long addr)
```

This is a special wrapper routine to help facilitate reservation cleanup on error paths. It is only called from the routine restore_reserve_on_error(). This routine is used in conjunction with vma_needs_reservation in an attempt to add a reservation to the reservation map. It takes into account the different reservation map semantics for private and shared mappings. Hence, region_add is called for shared mappings (as an entry present in the map indicates a reservation), and region_del is called for private mappings (as the absence of an entry in the map indicates a reservation). See the section "Reservation cleanup in error paths" for more information on what needs to be done on error paths.

2.9.13 Reservation Cleanup in Error Paths

As mentioned in the section Reservation Map Helper Routines, reservation map modifications are performed in two steps. First vma_needs_reservation is called before a page is allocated. If the allocation is successful, then vma_commit_reservation is called. If not, vma_end_reservation is called. Global and subpool reservation counts are adjusted based on success or failure of the operation and all is well.

Additionally, after a huge page is instantiated the PagePrivate flag is cleared so that accounting when the page is ultimately freed is correct.

However, there are several instances where errors are encountered after a huge page is allocated but before it is instantiated. In this case, the page allocation has consumed the reservation and made the appropriate subpool, reservation map and global count adjustments. If the page is freed at this time (before instantiation and clearing of PagePrivate), then free_huge_page will increment the global reservation count. However, the reservation map indicates the reservation was consumed. This resulting inconsistent state will cause the 'leak' of a reserved huge page. The global reserve count will be higher than it should and prevent allocation of a pre-allocated page.

The routine restore_reserve_on_error() attempts to handle this situation. It is fairly

well documented. The intention of this routine is to restore the reservation map to the way it was before the page allocation. In this way, the state of the reservation map will correspond to the global reservation count after the page is freed.

The routine restore_reserve_on_error itself may encounter errors while attempting to restore the reservation map entry. In this case, it will simply clear the PagePrivate flag of the page. In this way, the global reserve count will not be incremented when the page is freed. However, the reservation map will continue to look as though the reservation was consumed. A page can still be allocated for the address, but it will not use a reserved page as originally intended.

There is some code (most notably userfaultfd) which can not call restore_reserve_on_error. In this case, it simply modifies the PagePrivate so that a reservation will not be leaked when the huge page is freed.

2.9.14 Reservations and Memory Policy

Per-node huge page lists existed in struct hstate when git was first used to manage Linux code. The concept of reservations was added some time later. When reservations were added, no attempt was made to take memory policy into account. While cpusets are not exactly the same as memory policy, this comment in hugetlb_acct_memory sums up the interaction between reservations and cpusets/memory policy:

```
/*
 * When cpuset is configured, it breaks the strict hugetlb page
 * reservation as the accounting is done on a global variable. Such
 * reservation is completely rubbish in the presence of cpuset because
 * the reservation is not checked against page availability for the
 * current cpuset. Application can still potentially OOM'ed by kernel
 * with lack of free htlb page in cpuset that the task is in.
 * Attempt to enforce strict accounting with cpuset is almost
 * impossible (or too ugly) because cpuset is too fluid that
 * task or memory node can be dynamically moved between cpusets.
 *
 * The change of semantics for shared hugetlb mapping with cpuset is
 * undesirable. However, in order to preserve some of the semantics,
 * we fall back to check against current free page availability as
 * a best attempt and hopefully to minimize the impact of changing
 * semantics that cpuset has.
 */
```

Huge page reservations were added to prevent unexpected page allocation failures (OOM) at page fault time. However, if an application makes use of cpusets or memory policy there is no guarantee that huge pages will be available on the required nodes. This is true even if there are a sufficient number of global reservations.

2.9.15 Hugetlbfs regression testing

The most complete set of hugetlb tests are in the libhugetlbfs repository. If you modify any hugetlb related code, use the libhugetlbfs test suite to check for regressions. In addition, if you add any new hugetlb functionality, please add appropriate tests to libhugetlbfs.

– Mike Kravetz, 7 April 2017

2.10 Kernel Samepage Merging

KSM is a memory-saving de-duplication feature, enabled by CONFIG_KSM=y, added to the Linux kernel in 2.6.32. See `mm/ksm.c` for its implementation, and http://lwn.net/Articles/306704/ and https://lwn.net/Articles/330589/

The userspace interface of KSM is described in Documentation/admin-guide/mm/ksm.rst

2.10.1 Design

Overview

A few notes about the KSM scanning process, to make it easier to understand the data structures below:

In order to reduce excessive scanning, KSM sorts the memory pages by their contents into a data structure that holds pointers to the pages' locations.

Since the contents of the pages may change at any moment, KSM cannot just insert the pages into a normal sorted tree and expect it to find anything. Therefore KSM uses two data structures - the stable and the unstable tree.

The stable tree holds pointers to all the merged pages (ksm pages), sorted by their contents. Because each such page is write-protected, searching on this tree is fully assured to be working (except when pages are unmapped), and therefore this tree is called the stable tree.

The stable tree node includes information required for reverse mapping from a KSM page to virtual addresses that map this page.

In order to avoid large latencies of the rmap walks on KSM pages, KSM maintains two types of nodes in the stable tree:

- the regular nodes that keep the reverse mapping structures in a linked list
- the "chains" that link nodes ("dups") that represent the same write protected memory content, but each "dup" corresponds to a different KSM page copy of that content

Internally, the regular nodes, "dups" and "chains" are represented using the same `struct stable_node` structure.

In addition to the stable tree, KSM uses a second data structure called the unstable tree: this tree holds pointers to pages which have been found to be "unchanged for a period of time". The unstable tree sorts these pages by their contents, but

since they are not write-protected, KSM cannot rely upon the unstable tree to work correctly - the unstable tree is liable to be corrupted as its contents are modified, and so it is called unstable.

KSM solves this problem by several techniques:

1) The unstable tree is flushed every time KSM completes scanning all memory areas, and then the tree is rebuilt again from the beginning.
2) KSM will only insert into the unstable tree, pages whose hash value has not changed since the previous scan of all memory areas.
3) The unstable tree is a RedBlack Tree - so its balancing is based on the colors of the nodes and not on their contents, assuring that even when the tree gets "corrupted" it won't get out of balance, so scanning time remains the same (also, searching and inserting nodes in an rbtree uses the same algorithm, so we have no overhead when we flush and rebuild).
4) KSM never flushes the stable tree, which means that even if it were to take 10 attempts to find a page in the unstable tree, once it is found, it is secured in the stable tree. (When we scan a new page, we first compare it against the stable tree, and then against the unstable tree.)

If the merge_across_nodes tunable is unset, then KSM maintains multiple stable trees and multiple unstable trees: one of each for each NUMA node.

Reverse mapping

KSM maintains reverse mapping information for KSM pages in the stable tree.

If a KSM page is shared between less than `max_page_sharing` VMAs, the node of the stable tree that represents such KSM page points to a list of `struct rmap_item` and the `page->mapping` of the KSM page points to the stable tree node.

When the sharing passes this threshold, KSM adds a second dimension to the stable tree. The tree node becomes a "chain" that links one or more "dups". Each "dup" keeps reverse mapping information for a KSM page with `page->mapping` pointing to that "dup".

Every "chain" and all "dups" linked into a "chain" enforce the invariant that they represent the same write protected memory content, even if each "dup" will be pointed by a different KSM page copy of that content.

This way the stable tree lookup computational complexity is unaffected if compared to an unlimited list of reverse mappings. It is still enforced that there cannot be KSM page content duplicates in the stable tree itself.

The deduplication limit enforced by `max_page_sharing` is required to avoid the virtual memory rmap lists to grow too large. The rmap walk has O(N) complexity where N is the number of rmap_items (i.e. virtual mappings) that are sharing the page, which is in turn capped by `max_page_sharing`. So this effectively spreads the linear O(N) computational complexity from rmap walk context over different KSM pages. The ksmd walk over the stable_node "chains" is also O(N), but N is the number of stable_node "dups", not the number of rmap_items, so it has not a significant impact on ksmd performance. In practice the best stable_node "dup" candidate will be kept and found at the head of the "dups" list.

High values of `max_page_sharing` result in faster memory merging (because there will be fewer stable_node dups queued into the stable_node chain->hlist to check for pruning) and higher deduplication factor at the expense of slower worst case for rmap walks for any KSM page which can happen during swapping, compaction, NUMA balancing and page migration.

The `stable_node_dups/stable_node_chains` ratio is also affected by the `max_page_sharing` tunable, and an high ratio may indicate fragmentation in the stable_node dups, which could be solved by introducing fragmentation algorithms in ksmd which would refile rmap_items from one stable_node dup to another stable_node dup, in order to free up stable_node "dups" with few rmap_items in them, but that may increase the ksmd CPU usage and possibly slowdown the readonly computations on the KSM pages of the applications.

The whole list of stable_node "dups" linked in the stable_node "chains" is scanned periodically in order to prune stale stable_nodes. The frequency of such scans is defined by `stable_node_chains_prune_millisecs` sysfs tunable.

Reference

struct **mm_slot**
ksm information per mm that is being scanned

Definition

```
struct mm_slot {
  struct hlist_node link;
  struct list_head mm_list;
  struct rmap_item *rmap_list;
  struct mm_struct *mm;
};
```

Members

link link to the mm_slots hash list

mm_list link into the mm_slots list, rooted in ksm_mm_head

rmap_list head for this mm_slot's singly-linked list of rmap_items

mm the mm that this information is valid for

struct **ksm_scan**
cursor for scanning

Definition

```
struct ksm_scan {
  struct mm_slot *mm_slot;
  unsigned long address;
  struct rmap_item **rmap_list;
  unsigned long seqnr;
};
```

Members

mm_slot the current mm_slot we are scanning

address the next address inside that to be scanned

rmap_list link to the next rmap to be scanned in the rmap_list

seqnr count of completed full scans (needed when removing unstable node)

Description

There is only the one ksm_scan instance of this cursor structure.

struct **stable_node**
node of the stable rbtree

Definition

```
struct stable_node {
  union {
    struct rb_node node;
    struct {
      struct list_head *head;
      struct {
        struct hlist_node hlist_dup;
        struct list_head list;
      };
    };
  };
  struct hlist_head hlist;
  union {
    unsigned long kpfn;
    unsigned long chain_prune_time;
  };
#define STABLE_NODE_CHAIN -1024;
  int rmap_hlist_len;
#ifdef CONFIG_NUMA;
  int nid;
#endif;
};
```

Members

{unnamed_union} anonymous

node rb node of this ksm page in the stable tree

{unnamed_struct} anonymous

head (overlaying parent) `migrate_nodes` indicates temporarily on that list

{unnamed_struct} anonymous

hlist_dup linked into the stable_node->hlist with a stable_node chain

list linked into migrate_nodes, pending placement in the proper node tree

hlist hlist head of rmap_items using this ksm page

{unnamed_union} anonymous

kpfn page frame number of this ksm page (perhaps temporarily on wrong nid)

chain_prune_time time of the last full garbage collection

rmap_hlist_len number of rmap_item entries in hlist or STABLE_NODE_CHAIN

nid NUMA node id of stable tree in which linked (may not match kpfn)

struct **rmap_item**
reverse mapping item for virtual addresses

Definition

```
struct rmap_item {
  struct rmap_item *rmap_list;
  union {
    struct anon_vma *anon_vma;
#ifdef CONFIG_NUMA;
    int nid;
#endif;
  };
  struct mm_struct *mm;
  unsigned long address;
  unsigned int oldchecksum;
  union {
    struct rb_node node;
    struct {
      struct stable_node *head;
      struct hlist_node hlist;
    };
  };
};
```

Members

rmap_list next rmap_item in mm_slot's singly-linked rmap_list

{unnamed_union} anonymous

anon_vma pointer to anon_vma for this mm,address, when in stable tree

nid NUMA node id of unstable tree in which linked (may not match page)

mm the memory structure this rmap_item is pointing into

address the virtual address this rmap_item tracks (+ flags in low bits)

oldchecksum previous checksum of the page at that virtual address

{unnamed_union} anonymous

node rb node of this rmap_item in the unstable tree

{unnamed_struct} anonymous

head pointer to stable_node heading this list in the stable tree

hlist link into hlist of rmap_items hanging off that stable_node

– Izik Eidus, Hugh Dickins, 17 Nov 2009

2.11 Physical Memory Model

Physical memory in a system may be addressed in different ways. The simplest case is when the physical memory starts at address 0 and spans a contiguous range up to the maximal address. It could be, however, that this range contains small holes that are not accessible for the CPU. Then there could be several contiguous ranges at completely distinct addresses. And, don't forget about NUMA, where different memory banks are attached to different CPUs.

Linux abstracts this diversity using one of the three memory models: FLATMEM, DISCONTIGMEM and SPARSEMEM. Each architecture defines what memory models it supports, what the default memory model is and whether it is possible to manually override that default.

Note: At time of this writing, DISCONTIGMEM is considered deprecated, although it is still in use by several architectures.

All the memory models track the status of physical page frames using `struct page` arranged in one or more arrays.

Regardless of the selected memory model, there exists one-to-one mapping between the physical page frame number (PFN) and the corresponding struct page.

Each memory model defines `pfn_to_page()` and `page_to_pfn()` helpers that allow the conversion from PFN to struct page and vice versa.

2.11.1 FLATMEM

The simplest memory model is FLATMEM. This model is suitable for non-NUMA systems with contiguous, or mostly contiguous, physical memory.

In the FLATMEM memory model, there is a global mem_map array that maps the entire physical memory. For most architectures, the holes have entries in the mem_map array. The struct page objects corresponding to the holes are never fully initialized.

To allocate the mem_map array, architecture specific setup code should call `free_area_init()` function. Yet, the mappings array is not usable until the call to `memblock_free_all()` that hands all the memory to the page allocator.

If an architecture enables CONFIG_ARCH_HAS_HOLES_MEMORYMODEL option, it may free parts of the mem_map array that do not cover the actual physical pages. In such case, the architecture specific `pfn_valid()` implementation should take the holes in the mem_map into account.

With FLATMEM, the conversion between a PFN and the struct page is straightforward: PFN - ARCH_PFN_OFFSET is an index to the mem_map array.

The ARCH_PFN_OFFSET defines the first page frame number for systems with physical memory starting at address different from 0.

2.11.2 DISCONTIGMEM

The DISCONTIGMEM model treats the physical memory as a collection of nodes similarly to how Linux NUMA support does. For each node Linux constructs an independent memory management subsystem represented by struct pglist_data (or pg_data_t for short). Among other things, pg_data_t holds the node_mem_map array that maps physical pages belonging to that node. The node_start_pfn field of pg_data_t is the number of the first page frame belonging to that node.

The architecture setup code should call `free_area_init_node()` for each node in the system to initialize the pg_data_t object and its node_mem_map.

Every node_mem_map behaves exactly as FLATMEM' s mem_map - every physical page frame in a node has a struct page entry in the node_mem_map array. When DISCONTIGMEM is enabled, a portion of the flags field of the struct page encodes the node number of the node hosting that page.

The conversion between a PFN and the struct page in the DISCONTIGMEM model became slightly more complex as it has to determine which node hosts the physical page and which pg_data_t object holds the struct page.

Architectures that support DISCONTIGMEM provide `pfn_to_nid()` to convert PFN to the node number. The opposite conversion helper `page_to_nid()` is generic as it uses the node number encoded in page->flags.

Once the node number is known, the PFN can be used to index appropriate node_mem_map array to access the struct page and the offset of the struct page from the node_mem_map plus node_start_pfn is the PFN of that page.

2.11.3 SPARSEMEM

SPARSEMEM is the most versatile memory model available in Linux and it is the only memory model that supports several advanced features such as hot-plug and hot-remove of the physical memory, alternative memory maps for non-volatile memory devices and deferred initialization of the memory map for larger systems.

The SPARSEMEM model presents the physical memory as a collection of sections. A section is represented with `struct mem_section` that contains section_mem_map that is, logically, a pointer to an array of struct pages. However, it is stored with some other magic that aids the sections management. The section size and maximal number of section is specified using SECTION_SIZE_BITS and MAX_PHYSMEM_BITS constants defined by each architecture that supports SPARSEMEM. While MAX_PHYSMEM_BITS is an actual width of a physical address that an architecture supports, the SECTION_SIZE_BITS is an arbitrary value.

The maximal number of sections is denoted NR_MEM_SECTIONS and defined as

$$NR\_MEM\_SECTIONS = 2^{(MAX\_PHYSMEM\_BITS - SECTION\_SIZE\_BITS)}$$

The mem_section objects are arranged in a two-dimensional array called mem_sections. The size and placement of this array depend on CONFIG_SPARSEMEM_EXTREME and the maximal possible number of sections:

- When CONFIG_SPARSEMEM_EXTREME is disabled, the mem_sections array is static and has NR_MEM_SECTIONS rows. Each row holds a single mem_section object.
- When CONFIG_SPARSEMEM_EXTREME is enabled, the mem_sections array is dynamically allocated. Each row contains PAGE_SIZE worth of mem_section objects and the number of rows is calculated to fit all the memory sections.

The architecture setup code should call `memory_present()` for each active memory range or use `memblocks_present()` or `sparse_memory_present_with_active_regions()` wrappers to initialize the memory sections. Next, the actual memory maps should be set up using `sparse_init()`.

With SPARSEMEM there are two possible ways to convert a PFN to the corresponding struct page - a “classic sparse” and “sparse vmemmap” . The selection is made at build time and it is determined by the value of CONFIG_SPARSEMEM_VMEMMAP.

The classic sparse encodes the section number of a page in page->flags and uses high bits of a PFN to access the section that maps that page frame. Inside a section, the PFN is the index to the array of pages.

The sparse vmemmap uses a virtually mapped memory map to optimize pfn_to_page and page_to_pfn operations. There is a global struct page *vmemmap pointer that points to a virtually contiguous array of struct page objects. A PFN is an index to that array and the the offset of the struct page from vmemmap is the PFN of that page.

To use vmemmap, an architecture has to reserve a range of virtual addresses that will map the physical pages containing the memory map and make sure that vmemmap points to that range. In addition, the architecture should implement `vmemmap_populate()` method that will allocate the physical memory and create page tables for the virtual memory map. If an architecture does not have any special requirements for the vmemmap mappings, it can use default `vmemmap_populate_basepages()` provided by the generic memory management.

The virtually mapped memory map allows storing struct page objects for persistent memory devices in pre-allocated storage on those devices. This storage is represented with `struct vmem_altmap` that is eventually passed to vmemmap_populate() through a long chain of function calls. The vmemmap_populate() implementation may use the vmem_altmap along with `altmap_alloc_block_buf()` helper to allocate memory map on the persistent memory device.

2.11.4 ZONE_DEVICE

The ZONE_DEVICE facility builds upon SPARSEMEM_VMEMMAP to offer struct page mem_map services for device driver identified physical address ranges. The “device” aspect of ZONE_DEVICE relates to the fact that the page objects for these address ranges are never marked online, and that a reference must be taken against the device, not just the page to keep the memory pinned for active use. ZONE_DEVICE, via `devm_memremap_pages()`, performs just enough memory hotplug to turn on `pfn_to_page()`, `page_to_pfn()`, and `get_user_pages()` service for the given range of pfns. Since the page reference count never drops below 1 the page is never tracked as free memory and the page’s struct list_head lru space is repurposed for back referencing to the host device / driver that mapped the memory.

While SPARSEMEM presents memory as a collection of sections, optionally collected into memory blocks, ZONE_DEVICE users have a need for smaller granularity of populating the mem_map. Given that ZONE_DEVICE memory is never marked online it is subsequently never subject to its memory ranges being exposed through the sysfs memory hotplug api on memory block boundaries. The implementation relies on this lack of user-api constraint to allow sub-section sized memory ranges to be specified to `arch_add_memory()`, the top-half of memory hotplug. Sub-section support allows for 2MB as the cross-arch common alignment granularity for `devm_memremap_pages()`.

The users of ZONE_DEVICE are:

- pmem: Map platform persistent memory to be used as a direct-I/O target via DAX mappings.
- hmm: Extend ZONE_DEVICE with ->page_fault() and ->page_free() event callbacks to allow a device-driver to coordinate memory management events related to device-memory, typically GPU memory. See Documentation/vm/hmm.rst.
- p2pdma: Create struct page objects to allow peer devices in a PCI/-E topology to coordinate direct-DMA operations between themselves, i.e. bypass host memory.

2.12 When do you need to notify inside page table lock ?

When clearing a pte/pmd we are given a choice to notify the event through (notify version of *_clear_flush call mmu_notifier_invalidate_range) under the page table lock. But that notification is not necessary in all cases.

For secondary TLB (non CPU TLB) like IOMMU TLB or device TLB (when device use thing like ATS/PASID to get the IOMMU to walk the CPU page table to access a process virtual address space). There is only 2 cases when you need to notify those secondary TLB while holding page table lock when clearing a pte/pmd:

A) page backing address is free before mmu_notifier_invalidate_range_end()

B) a page table entry is updated to point to a new page (COW, write fault on zero page, __replace_page(), ⋯)

Case A is obvious you do not want to take the risk for the device to write to a page that might now be used by some completely different task.

Case B is more subtle. For correctness it requires the following sequence to happen:

- take page table lock
- clear page table entry and notify ([pmd/pte]p_huge_clear_flush_notify())
- set page table entry to point to new page

If clearing the page table entry is not followed by a notify before setting the new pte/pmd value then you can break memory model like C11 or C++11 for the device.

Consider the following scenario (device use a feature similar to ATS/PASID):

Two address addrA and addrB such that |addrA - addrB| >= PAGE_SIZE we assume they are write protected for COW (other case of B apply too).

```
[Time N] ----------------------------------------------------------------
↪--
CPU-thread-0  {try to write to addrA}
CPU-thread-1  {try to write to addrB}
CPU-thread-2  {}
CPU-thread-3  {}
DEV-thread-0  {read addrA and populate device TLB}
DEV-thread-2  {read addrB and populate device TLB}
[Time N+1] ----------------------------------------------------------------
↪--
CPU-thread-0  {COW_step0: {mmu_notifier_invalidate_range_start(addrA)}}
CPU-thread-1  {COW_step0: {mmu_notifier_invalidate_range_start(addrB)}}
CPU-thread-2  {}
CPU-thread-3  {}
DEV-thread-0  {}
DEV-thread-2  {}
[Time N+2] ----------------------------------------------------------------
↪--
CPU-thread-0  {COW_step1: {update page table to point to new page for␣
↪addrA}}
CPU-thread-1  {COW_step1: {update page table to point to new page for␣
↪addrB}}
CPU-thread-2  {}
CPU-thread-3  {}
DEV-thread-0  {}
DEV-thread-2  {}
[Time N+3] ----------------------------------------------------------------
↪--
CPU-thread-0  {preempted}
CPU-thread-1  {preempted}
CPU-thread-2  {write to addrA which is a write to new page}
CPU-thread-3  {}
DEV-thread-0  {}
DEV-thread-2  {}
[Time N+3] ----------------------------------------------------------------
↪--
CPU-thread-0  {preempted}
CPU-thread-1  {preempted}
CPU-thread-2  {}
```

(continues on next page)

(continued from previous page)

```
CPU-thread-3  {write to addrB which is a write to new page}
DEV-thread-0  {}
DEV-thread-2  {}
[Time N+4] ------------------------------------------------------------
↪--
CPU-thread-0  {preempted}
CPU-thread-1  {COW_step3: {mmu_notifier_invalidate_range_end(addrB)}}
CPU-thread-2  {}
CPU-thread-3  {}
DEV-thread-0  {}
DEV-thread-2  {}
[Time N+5] ------------------------------------------------------------
↪--
CPU-thread-0  {preempted}
CPU-thread-1  {}
CPU-thread-2  {}
CPU-thread-3  {}
DEV-thread-0  {read addrA from old page}
DEV-thread-2  {read addrB from new page}
```

So here because at time N+2 the clear page table entry was not pair with a notification to invalidate the secondary TLB, the device see the new value for addrB before seing the new value for addrA. This break total memory ordering for the device.

When changing a pte to write protect or to point to a new write protected page with same content (KSM) it is fine to delay the mmu_notifier_invalidate_range call to mmu_notifier_invalidate_range_end() outside the page table lock. This is true even if the thread doing the page table update is preempted right after releasing page table lock but before call mmu_notifier_invalidate_range_end(). Started Nov 1999 by Kanoj Sarcar <kanoj@sgi.com>

2.13 What is NUMA?

This question can be answered from a couple of perspectives: the hardware view and the Linux software view.

From the hardware perspective, a NUMA system is a computer platform that comprises multiple components or assemblies each of which may contain 0 or more CPUs, local memory, and/or IO buses. For brevity and to disambiguate the hardware view of these physical components/assemblies from the software abstraction thereof, we'll call the components/assemblies 'cells' in this document.

Each of the 'cells' may be viewed as an SMP [symmetric multi-processor] subset of the system–although some components necessary for a stand-alone SMP system may not be populated on any given cell. The cells of the NUMA system are connected together with some sort of system interconnect–e.g., a crossbar or point-to-point link are common types of NUMA system interconnects. Both of these types of interconnects can be aggregated to create NUMA platforms with cells at multiple distances from other cells.

For Linux, the NUMA platforms of interest are primarily what is known as Cache Coherent NUMA or ccNUMA systems. With ccNUMA systems, all memory is vis-

ible to and accessible from any CPU attached to any cell and cache coherency is handled in hardware by the processor caches and/or the system interconnect.

Memory access time and effective memory bandwidth varies depending on how far away the cell containing the CPU or IO bus making the memory access is from the cell containing the target memory. For example, access to memory by CPUs attached to the same cell will experience faster access times and higher bandwidths than accesses to memory on other, remote cells. NUMA platforms can have cells at multiple remote distances from any given cell.

Platform vendors don't build NUMA systems just to make software developers' lives interesting. Rather, this architecture is a means to provide scalable memory bandwidth. However, to achieve scalable memory bandwidth, system and application software must arrange for a large majority of the memory references [cache misses] to be to "local" memory–memory on the same cell, if any–or to the closest cell with memory.

This leads to the Linux software view of a NUMA system:

Linux divides the system's hardware resources into multiple software abstractions called "nodes". Linux maps the nodes onto the physical cells of the hardware platform, abstracting away some of the details for some architectures. As with physical cells, software nodes may contain 0 or more CPUs, memory and/or IO buses. And, again, memory accesses to memory on "closer" nodes–nodes that map to closer cells–will generally experience faster access times and higher effective bandwidth than accesses to more remote cells.

For some architectures, such as x86, Linux will "hide" any node representing a physical cell that has no memory attached, and reassign any CPUs attached to that cell to a node representing a cell that does have memory. Thus, on these architectures, one cannot assume that all CPUs that Linux associates with a given node will see the same local memory access times and bandwidth.

In addition, for some architectures, again x86 is an example, Linux supports the emulation of additional nodes. For NUMA emulation, linux will carve up the existing nodes–or the system memory for non-NUMA platforms–into multiple nodes. Each emulated node will manage a fraction of the underlying cells' physical memory. NUMA emluation is useful for testing NUMA kernel and application features on non-NUMA platforms, and as a sort of memory resource management mechanism when used together with cpusets. [see Documentation/admin-guide/cgroup-v1/cpusets.rst]

For each node with memory, Linux constructs an independent memory management subsystem, complete with its own free page lists, in-use page lists, usage statistics and locks to mediate access. In addition, Linux constructs for each memory zone [one or more of DMA, DMA32, NORMAL, HIGH_MEMORY, MOVABLE], an ordered "zonelist". A zonelist specifies the zones/nodes to visit when a selected zone/node cannot satisfy the allocation request. This situation, when a zone has no available memory to satisfy a request, is called "overflow" or "fallback".

Because some nodes contain multiple zones containing different types of memory, Linux must decide whether to order the zonelists such that allocations fall back to the same zone type on a different node, or to a different zone type on the same node. This is an important consideration because some zones, such as DMA or DMA32, represent relatively scarce resources. Linux chooses a default Node or-

dered zonelist. This means it tries to fallback to other zones from the same node before using remote nodes which are ordered by NUMA distance.

By default, Linux will attempt to satisfy memory allocation requests from the node to which the CPU that executes the request is assigned. Specifically, Linux will attempt to allocate from the first node in the appropriate zonelist for the node where the request originates. This is called "local allocation." If the "local" node cannot satisfy the request, the kernel will examine other nodes' zones in the selected zonelist looking for the first zone in the list that can satisfy the request.

Local allocation will tend to keep subsequent access to the allocated memory "local" to the underlying physical resources and off the system interconnect– as long as the task on whose behalf the kernel allocated some memory does not later migrate away from that memory. The Linux scheduler is aware of the NUMA topology of the platform–embodied in the "scheduling domains" data structures [see Documentation/scheduler/sched-domains.rst]–and the scheduler attempts to minimize task migration to distant scheduling domains. However, the scheduler does not take a task's NUMA footprint into account directly. Thus, under sufficient imbalance, tasks can migrate between nodes, remote from their initial node and kernel data structures.

System administrators and application designers can restrict a task's migration to improve NUMA locality using various CPU affinity command line interfaces, such as taskset(1) and numactl(1), and program interfaces such as sched_setaffinity(2). Further, one can modify the kernel's default local allocation behavior using Linux NUMA memory policy. [see Documentation/admin-guide/mm/numa_memory_policy.rst].

System administrators can restrict the CPUs and nodes' memories that a non-privileged user can specify in the scheduling or NUMA commands and functions using control groups and CPUsets. [see Documentation/admin-guide/cgroup-v1/cpusets.rst]

On architectures that do not hide memoryless nodes, Linux will include only zones [nodes] with memory in the zonelists. This means that for a memoryless node the "local memory node" –the node of the first zone in CPU's node's zonelist–will not be the node itself. Rather, it will be the node that the kernel selected as the nearest node with memory when it built the zonelists. So, default, local allocations will succeed with the kernel supplying the closest available memory. This is a consequence of the same mechanism that allows such allocations to fallback to other nearby nodes when a node that does contain memory overflows.

Some kernel allocations do not want or cannot tolerate this allocation fallback behavior. Rather they want to be sure they get memory from the specified node or get notified that the node has no free memory. This is usually the case when a subsystem allocates per CPU memory resources, for example.

A typical model for making such an allocation is to obtain the node id of the node to which the "current CPU" is attached using one of the kernel's numa_node_id() or CPU_to_node() functions and then request memory from only the node id returned. When such an allocation fails, the requesting subsystem may revert to its own fallback path. The slab kernel memory allocator is an example of this. Or, the subsystem may choose to disable or not to enable itself on allocation failure. The kernel profiling subsystem is an example of this.

If the architecture supports–does not hide–memoryless nodes, then CPUs attached to memoryless nodes would always incur the fallback path overhead or some subsystems would fail to initialize if they attempted to allocated memory exclusively from a node without memory. To support such architectures transparently, kernel subsystems can use the numa_mem_id() or cpu_to_mem() function to locate the "local memory node" for the calling or specified CPU. Again, this is the same node from which default, local page allocations will be attempted.

2.14 Overcommit Accounting

The Linux kernel supports the following overcommit handling modes

0 Heuristic overcommit handling. Obvious overcommits of address space are refused. Used for a typical system. It ensures a seriously wild allocation fails while allowing overcommit to reduce swap usage. root is allowed to allocate slightly more memory in this mode. This is the default.

1 Always overcommit. Appropriate for some scientific applications. Classic example is code using sparse arrays and just relying on the virtual memory consisting almost entirely of zero pages.

2 Don' t overcommit. The total address space commit for the system is not permitted to exceed swap + a configurable amount (default is 50%) of physical RAM. Depending on the amount you use, in most situations this means a process will not be killed while accessing pages but will receive errors on memory allocation as appropriate.

Useful for applications that want to guarantee their memory allocations will be available in the future without having to initialize every page.

The overcommit policy is set via the sysctl `vm.overcommit_memory`.

The overcommit amount can be set via `vm.overcommit_ratio` (percentage) or `vm.overcommit_kbytes` (absolute value).

The current overcommit limit and amount committed are viewable in `/proc/meminfo` as CommitLimit and Committed_AS respectively.

2.14.1 Gotchas

The C language stack growth does an implicit mremap. If you want absolute guarantees and run close to the edge you MUST mmap your stack for the largest size you think you will need. For typical stack usage this does not matter much but it' s a corner case if you really really care

In mode 2 the MAP_NORESERVE flag is ignored.

2.14.2 How It Works

The overcommit is based on the following rules

For a file backed map

SHARED or READ-only - 0 cost (the file is the map not swap)
PRIVATE WRITABLE - size of mapping per instance

For an anonymous or `/dev/zero` map

SHARED - size of mapping
PRIVATE READ-only - 0 cost (but of little use)
PRIVATE WRITABLE - size of mapping per instance

Additional accounting

Pages made writable copies by mmap
shmfs memory drawn from the same pool

2.14.3 Status

- We account mmap memory mappings
- We account mprotect changes in commit
- We account mremap changes in size
- We account brk
- We account munmap
- We report the commit status in /proc
- Account and check on fork
- Review stack handling/building on exec
- SHMfs accounting
- Implement actual limit enforcement

2.14.4 To Do

- Account ptrace pages (this is hard)

2.15 Page migration

Page migration allows the moving of the physical location of pages between nodes in a numa system while the process is running. This means that the virtual addresses that the process sees do not change. However, the system rearranges the physical location of those pages.

The main intend of page migration is to reduce the latency of memory access by moving pages near to the processor where the process accessing that memory is running.

Page migration allows a process to manually relocate the node on which its pages are located through the MF_MOVE and MF_MOVE_ALL options while setting a new memory policy via mbind(). The pages of process can also be relocated from another process using the sys_migrate_pages() function call. The migrate_pages function call takes two sets of nodes and moves pages of a process that are located on the from nodes to the destination nodes. Page migration functions are provided by the numactl package by Andi Kleen (a version later than 0.9.3 is required. Get it from ftp://oss.sgi.com/www/projects/libnuma/download/). numactl provides libnuma which provides an interface similar to other numa functionality for page migration. cat `/proc/<pid>/numa_maps` allows an easy review of where the pages of a process are located. See also the numa_maps documentation in the proc(5) man page.

Manual migration is useful if for example the scheduler has relocated a process to a processor on a distant node. A batch scheduler or an administrator may detect the situation and move the pages of the process nearer to the new processor. The kernel itself does only provide manual page migration support. Automatic page migration may be implemented through user space processes that move pages. A special function call "move_pages" allows the moving of individual pages within a process. A NUMA profiler may f.e. obtain a log showing frequent off node accesses and may use the result to move pages to more advantageous locations.

Larger installations usually partition the system using cpusets into sections of nodes. Paul Jackson has equipped cpusets with the ability to move pages when a task is moved to another cpuset (See Documentation/admin-guide/cgroup-v1/cpusets.rst). Cpusets allows the automation of process locality. If a task is moved to a new cpuset then also all its pages are moved with it so that the performance of the process does not sink dramatically. Also the pages of processes in a cpuset are moved if the allowed memory nodes of a cpuset are changed.

Page migration allows the preservation of the relative location of pages within a group of nodes for all migration techniques which will preserve a particular memory allocation pattern generated even after migrating a process. This is necessary in order to preserve the memory latencies. Processes will run with similar performance after migration.

Page migration occurs in several steps. First a high level description for those trying to use migrate_pages() from the kernel (for userspace usage see the Andi Kleen' s numactl package mentioned above) and then a low level description of how the low level details work.

2.15.1 In kernel use of migrate_pages()

1. Remove pages from the LRU.

   Lists of pages to be migrated are generated by scanning over pages and moving them into lists. This is done by calling isolate_lru_page(). Calling isolate_lru_page increases the references to the page so that it cannot vanish while the page migration occurs. It also prevents the swapper or other scans to encounter the page.

2. We need to have a function of type new_page_t that can be passed to migrate_pages(). This function should figure out how to allocate the correct new page given the old page.

3. The migrate_pages() function is called which attempts to do the migration. It will call the function to allocate the new page for each page that is considered for moving.

2.15.2 How migrate_pages() works

migrate_pages() does several passes over its list of pages. A page is moved if all references to a page are removable at the time. The page has already been removed from the LRU via isolate_lru_page() and the refcount is increased so that the page cannot be freed while page migration occurs.

Steps:

1. Lock the page to be migrated
2. Ensure that writeback is complete.
3. Lock the new page that we want to move to. It is locked so that accesses to this (not yet uptodate) page immediately lock while the move is in progress.
4. All the page table references to the page are converted to migration entries. This decreases the mapcount of a page. If the resulting mapcount is not zero then we do not migrate the page. All user space processes that attempt to access the page will now wait on the page lock.
5. The i_pages lock is taken. This will cause all processes trying to access the page via the mapping to block on the spinlock.
6. The refcount of the page is examined and we back out if references remain otherwise we know that we are the only one referencing this page.
7. The radix tree is checked and if it does not contain the pointer to this page then we back out because someone else modified the radix tree.
8. The new page is prepped with some settings from the old page so that accesses to the new page will discover a page with the correct settings.
9. The radix tree is changed to point to the new page.
10. The reference count of the old page is dropped because the address space reference is gone. A reference to the new page is established because the new page is referenced by the address space.
11. The i_pages lock is dropped. With that lookups in the mapping become possible again. Processes will move from spinning on the lock to sleeping on the locked new page.
12. The page contents are copied to the new page.
13. The remaining page flags are copied to the new page.
14. The old page flags are cleared to indicate that the page does not provide any information anymore.
15. Queued up writeback on the new page is triggered.
16. If migration entries were page then replace them with real ptes. Doing so will enable access for user space processes not already waiting for the page lock.

19. The page locks are dropped from the old and new page. Processes waiting on the page lock will redo their page faults and will reach the new page.
20. The new page is moved to the LRU and can be scanned by the swapper etc again.

2.15.3 Non-LRU page migration

Although original migration aimed for reducing the latency of memory access for NUMA, compaction who want to create high-order page is also main customer.

Current problem of the implementation is that it is designed to migrate only LRU pages. However, there are potential non-lru pages which can be migrated in drivers, for example, zsmalloc, virtio-balloon pages.

For virtio-balloon pages, some parts of migration code path have been hooked up and added virtio-balloon specific functions to intercept migration logics. It's too specific to a driver so other drivers who want to make their pages movable would have to add own specific hooks in migration path.

To overclome the problem, VM supports non-LRU page migration which provides generic functions for non-LRU movable pages without driver specific hooks migration path.

If a driver want to make own pages movable, it should define three functions which are function pointers of struct address_space_operations.

1. `bool (*isolate_page) (struct page *page, isolate_mode_t mode);`

   What VM expects on isolate_page function of driver is to return true if driver isolates page successfully. On returing true, VM marks the page as PG_isolated so concurrent isolation in several CPUs skip the page for isolation. If a driver cannot isolate the page, it should return false.

   Once page is successfully isolated, VM uses page.lru fields so driver shouldn't expect to preserve values in that fields.

2. `int (*migratepage) (struct address_space *mapping, | struct page *newpage, struct page *oldpage, enum migrate_mode);`

   After isolation, VM calls migratepage of driver with isolated page. The function of migratepage is to move content of the old page to new page and set up fields of struct page newpage. Keep in mind that you should indicate to the VM the oldpage is no longer movable via __ClearPageMovable() under page_lock if you migrated the oldpage successfully and returns MIGRATEPAGE_SUCCESS. If driver cannot migrate the page at the moment, driver can return -EAGAIN. On -EAGAIN, VM will retry page migration in a short time because VM interprets -EAGAIN as "temporal migration failure". On returning any error except -EAGAIN, VM will give up the page migration without retrying in this time.

   Driver shouldn't touch page.lru field VM using in the functions.

3. `void (*putback_page)(struct page *);`

If migration fails on isolated page, VM should return the isolated page to the driver so VM calls driver's putback_page with migration failed page. In this function, driver should put the isolated page back to the own data structure.

4. non-lru movable page flags

   There are two page flags for supporting non-lru movable page.

   - PG_movable

     Driver should use the below function to make page movable under page_lock:

     ```
     void __SetPageMovable(struct page *page, struct address_space␣
     ↪*mapping)
     ```

     It needs argument of address_space for registering migration family functions which will be called by VM. Exactly speaking, PG_movable is not a real flag of struct page. Rather than, VM reuses page->mapping's lower bits to represent it.

::

     #define PAGE_MAPPING_MOVABLE 0x2 page->mapping = page->mapping | PAGE_MAPPING_MOVABLE;

     so driver shouldn't access page->mapping directly. Instead, driver should use page_mapping which mask off the low two bits of page->mapping under page lock so it can get right struct address_space.

     For testing of non-lru movable page, VM supports __PageMovable function. However, it doesn't guarantee to identify non-lru movable page because page->mapping field is unified with other variables in struct page. As well, if driver releases the page after isolation by VM, page->mapping doesn't have stable value although it has PAGE_MAPPING_MOVABLE (Look at __ClearPageMovable). But __PageMovable is cheap to catch whether page is LRU or non-lru movable once the page has been isolated. Because LRU pages never can have PAGE_MAPPING_MOVABLE in page->mapping. It is also good for just peeking to test non-lru movable pages before more expensive checking with lock_page in pfn scanning to select victim.

     For guaranteeing non-lru movable page, VM provides PageMovable function. Unlike __PageMovable, PageMovable functions validates page->mapping and mapping->a_ops->isolate_page under lock_page. The lock_page prevents sudden destroying of page->mapping.

     Driver using __SetPageMovable should clear the flag via __ClearMovablePage under page_lock before the releasing the page.

   - PG_isolated

     To prevent concurrent isolation among several CPUs, VM marks isolated page as PG_isolated under lock_page. So if a CPU encounters PG_isolated non-lru movable page, it can skip it. Driver doesn't need to manipulate the flag because VM will set/clear it automatically. Keep in mind that if

driver sees PG_isolated page, it means the page have been isolated by VM so it shouldn't touch page.lru field. PG_isolated is alias with PG_reclaim flag so driver shouldn't use the flag for own purpose.

Christoph Lameter, May 8, 2006. Minchan Kim, Mar 28, 2016.

2.16 Page fragments

A page fragment is an arbitrary-length arbitrary-offset area of memory which resides within a 0 or higher order compound page. Multiple fragments within that page are individually refcounted, in the page's reference counter.

The page_frag functions, page_frag_alloc and page_frag_free, provide a simple allocation framework for page fragments. This is used by the network stack and network device drivers to provide a backing region of memory for use as either an sk_buff->head, or to be used in the "frags" portion of skb_shared_info.

In order to make use of the page fragment APIs a backing page fragment cache is needed. This provides a central point for the fragment allocation and tracks allows multiple calls to make use of a cached page. The advantage to doing this is that multiple calls to get_page can be avoided which can be expensive at allocation time. However due to the nature of this caching it is required that any calls to the cache be protected by either a per-cpu limitation, or a per-cpu limitation and forcing interrupts to be disabled when executing the fragment allocation.

The network stack uses two separate caches per CPU to handle fragment allocation. The netdev_alloc_cache is used by callers making use of the netdev_alloc_frag and __netdev_alloc_skb calls. The napi_alloc_cache is used by callers of the __napi_alloc_frag and __napi_alloc_skb calls. The main difference between these two calls is the context in which they may be called. The "netdev" prefixed functions are usable in any context as these functions will disable interrupts, while the "napi" prefixed functions are only usable within the softirq context.

Many network device drivers use a similar methodology for allocating page fragments, but the page fragments are cached at the ring or descriptor level. In order to enable these cases it is necessary to provide a generic way of tearing down a page cache. For this reason __page_frag_cache_drain was implemented. It allows for freeing multiple references from a single page via a single call. The advantage to doing this is that it allows for cleaning up the multiple references that were added to a page in order to avoid calling get_page per allocation.

Alexander Duyck, Nov 29, 2016.

2.17 page owner: Tracking about who allocated each page

2.17.1 Introduction

page owner is for the tracking about who allocated each page. It can be used to debug memory leak or to find a memory hogger. When allocation happens, information about allocation such as call stack and order of pages is stored into certain storage for each page. When we need to know about status of all pages, we can get and analyze this information.

Although we already have tracepoint for tracing page allocation/free, using it for analyzing who allocate each page is rather complex. We need to enlarge the trace buffer for preventing overlapping until userspace program launched. And, launched program continually dump out the trace buffer for later analysis and it would change system behviour with more possibility rather than just keeping it in memory, so bad for debugging.

page owner can also be used for various purposes. For example, accurate fragmentation statistics can be obtained through gfp flag information of each page. It is already implemented and activated if page owner is enabled. Other usages are more than welcome.

page owner is disabled in default. So, if you'd like to use it, you need to add "page_owner=on" into your boot cmdline. If the kernel is built with page owner and page owner is disabled in runtime due to no enabling boot option, runtime overhead is marginal. If disabled in runtime, it doesn't require memory to store owner information, so there is no runtime memory overhead. And, page owner inserts just two unlikely branches into the page allocator hotpath and if not enabled, then allocation is done like as the kernel without page owner. These two unlikely branches should not affect to allocation performance, especially if the static keys jump label patching functionality is available. Following is the kernel's code size change due to this facility.

- Without page owner:

```
text    data     bss     dec     hex filename
40662   1493     644   42799    a72f mm/page_alloc.o
```

- With page owner:

```
text    data     bss     dec     hex filename
40892   1493     644   43029    a815 mm/page_alloc.o
1427      24       8    1459     5b3 mm/page_ext.o
2722      50       0    2772     ad4 mm/page_owner.o
```

Although, roughly, 4 KB code is added in total, page_alloc.o increase by 230 bytes and only half of it is in hotpath. Building the kernel with page owner and turning it on if needed would be great option to debug kernel memory problem.

There is one notice that is caused by implementation detail. page owner stores information into the memory from struct page extension. This memory is initialized some time later than that page allocator starts in sparse memory system, so, until initialization, many pages can be allocated and they would have no owner

information. To fix it up, these early allocated pages are investigated and marked as allocated in initialization phase. Although it doesn't mean that they have the right owner information, at least, we can tell whether the page is allocated or not, more accurately. On 2GB memory x86-64 VM box, 13343 early allocated pages are catched and marked, although they are mostly allocated from struct page extension feature. Anyway, after that, no page is left in un-tracking state.

2.17.2 Usage

1) Build user-space helper:

```
cd tools/vm
make page_owner_sort
```

2) Enable page owner: add "page_owner=on" to boot cmdline.

3) Do the job what you want to debug

4) Analyze information from page owner:

```
cat /sys/kernel/debug/page_owner > page_owner_full.txt
./page_owner_sort page_owner_full.txt sorted_page_owner.txt
```

See the result about who allocated each page in the `sorted_page_owner.txt`.

2.18 remap_file_pages() system call

The remap_file_pages() system call is used to create a nonlinear mapping, that is, a mapping in which the pages of the file are mapped into a nonsequential order in memory. The advantage of using remap_file_pages() over using repeated calls to mmap(2) is that the former approach does not require the kernel to create additional VMA (Virtual Memory Area) data structures.

Supporting of nonlinear mapping requires significant amount of non-trivial code in kernel virtual memory subsystem including hot paths. Also to get nonlinear mapping work kernel need a way to distinguish normal page table entries from entries with file offset (pte_file). Kernel reserves flag in PTE for this purpose. PTE flags are scarce resource especially on some CPU architectures. It would be nice to free up the flag for other usage.

Fortunately, there are not many users of remap_file_pages() in the wild. It's only known that one enterprise RDBMS implementation uses the syscall on 32-bit systems to map files bigger than can linearly fit into 32-bit virtual address space. This use-case is not critical anymore since 64-bit systems are widely available.

The syscall is deprecated and replaced it with an emulation now. The emulation creates new VMAs instead of nonlinear mappings. It's going to work slower for rare users of remap_file_pages() but ABI is preserved.

One side effect of emulation (apart from performance) is that user can hit vm.max_map_count limit more easily due to additional VMAs. See comment for DEFAULT_MAX_MAP_COUNT for more details on the limit.

2.19 Short users guide for SLUB

The basic philosophy of SLUB is very different from SLAB. SLAB requires rebuilding the kernel to activate debug options for all slab caches. SLUB always includes full debugging but it is off by default. SLUB can enable debugging only for selected slabs in order to avoid an impact on overall system performance which may make a bug more difficult to find.

In order to switch debugging on one can add an option `slub_debug` to the kernel command line. That will enable full debugging for all slabs.

Typically one would then use the `slabinfo` command to get statistical data and perform operation on the slabs. By default `slabinfo` only lists slabs that have data in them. See "slabinfo -h" for more options when running the command. `slabinfo` can be compiled with

```
gcc -o slabinfo tools/vm/slabinfo.c
```

Some of the modes of operation of `slabinfo` require that slub debugging be enabled on the command line. F.e. no tracking information will be available without debugging on and validation can only partially be performed if debugging was not switched on.

2.19.1 Some more sophisticated uses of slub_debug:

Parameters may be given to `slub_debug`. If none is specified then full debugging is enabled. Format:

slub_debug=<Debug-Options> Enable options for all slabs

slub_debug=<Debug-Options>,<slab name1>,<slab name2>,… Enable options only for select slabs (no spaces after a comma)

Possible debug options are:

```
F               Sanity checks on (enables SLAB_DEBUG_CONSISTENCY_CHECKS
                Sorry SLAB legacy issues)
Z               Red zoning
P               Poisoning (object and padding)
U               User tracking (free and alloc)
T               Trace (please only use on single slabs)
A               Enable failslab filter mark for the cache
O               Switch debugging off for caches that would have
                caused higher minimum slab orders
-               Switch all debugging off (useful if the kernel is
                configured with CONFIG_SLUB_DEBUG_ON)
```

F.e. in order to boot just with sanity checks and red zoning one would specify:

```
slub_debug=FZ
```

Trying to find an issue in the dentry cache? Try:

```
slub_debug=,dentry
```

to only enable debugging on the dentry cache. You may use an asterisk at the end of the slab name, in order to cover all slabs with the same prefix. For example, here's how you can poison the dentry cache as well as all kmalloc slabs:

```
slub_debug=P,kmalloc-*,dentry
```

Red zoning and tracking may realign the slab. We can just apply sanity checks to the dentry cache with:

```
slub_debug=F,dentry
```

Debugging options may require the minimum possible slab order to increase as a result of storing the metadata (for example, caches with PAGE_SIZE object sizes). This has a higher liklihood of resulting in slab allocation errors in low memory situations or if there's high fragmentation of memory. To switch off debugging for such caches by default, use:

```
slub_debug=O
```

In case you forgot to enable debugging on the kernel command line: It is possible to enable debugging manually when the kernel is up. Look at the contents of:

```
/sys/kernel/slab/<slab name>/
```

Look at the writable files. Writing 1 to them will enable the corresponding debug option. All options can be set on a slab that does not contain objects. If the slab already contains objects then sanity checks and tracing may only be enabled. The other options may cause the realignment of objects.

Careful with tracing: It may spew out lots of information and never stop if used on the wrong slab.

Slab merging

If no debug options are specified then SLUB may merge similar slabs together in order to reduce overhead and increase cache hotness of objects. `slabinfo -a` displays which slabs were merged together.

Slab validation

SLUB can validate all object if the kernel was booted with slub_debug. In order to do so you must have the `slabinfo` tool. Then you can do

```
slabinfo -v
```

which will test all objects. Output will be generated to the syslog.

This also works in a more limited way if boot was without slab debug. In that case `slabinfo -v` simply tests all reachable objects. Usually these are in the cpu slabs and the partial slabs. Full slabs are not tracked by SLUB in a non debug situation.

Getting more performance

To some degree SLUB's performance is limited by the need to take the list_lock once in a while to deal with partial slabs. That overhead is governed by the order of the allocation for each slab. The allocations can be influenced by kernel parameters:

slub_min_objects allows to specify how many objects must at least fit into one slab in order for the allocation order to be acceptable. In general slub will be able to perform this number of allocations on a slab without consulting centralized resources (list_lock) where contention may occur.

slub_min_order specifies a minimum order of slabs. A similar effect like `slub_min_objects`.

slub_max_order specified the order at which `slub_min_objects` should no longer be checked. This is useful to avoid SLUB trying to generate super large order pages to fit `slub_min_objects` of a slab cache with large object sizes into one high order page. Setting command line parameter `debug_guardpage_minorder=N` (N > 0), forces setting `slub_max_order` to 0, what cause minimum possible order of slabs allocation.

SLUB Debug output

Here is a sample of slub debug output:

```
====================================================================
BUG kmalloc-8: Redzone overwritten
--------------------------------------------------------------------

INFO: 0xc90f6d28-0xc90f6d2b. First byte 0x00 instead of 0xcc
INFO: Slab 0xc528c530 flags=0x400000c3 inuse=61 fp=0xc90f6d58
INFO: Object 0xc90f6d20 @offset=3360 fp=0xc90f6d58
INFO: Allocated in get_modalias+0x61/0xf5 age=53 cpu=1 pid=554

Bytes b4 0xc90f6d10:  00 00 00 00 00 00 00 00 5a 5a 5a 5a 5a 5a 5a 5a .....
↪...ZZZZZZZZ
  Object 0xc90f6d20:  31 30 31 39 2e 30 30 35                         1019.
↪005
 Redzone 0xc90f6d28:  00 cc cc cc                                     .
 Padding 0xc90f6d50:  5a 5a 5a 5a 5a 5a 5a 5a                         ␣
↪ZZZZZZZZ

  [<c010523d>] dump_trace+0x63/0x1eb
  [<c01053df>] show_trace_log_lvl+0x1a/0x2f
  [<c010601d>] show_trace+0x12/0x14
  [<c0106035>] dump_stack+0x16/0x18
  [<c017e0fa>] object_err+0x143/0x14b
  [<c017e2cc>] check_object+0x66/0x234
  [<c017eb43>] __slab_free+0x239/0x384
  [<c017f446>] kfree+0xa6/0xc6
  [<c02e2335>] get_modalias+0xb9/0xf5
  [<c02e23b7>] dmi_dev_uevent+0x27/0x3c
  [<c027866a>] dev_uevent+0x1ad/0x1da
  [<c0205024>] kobject_uevent_env+0x20a/0x45b
```

(continues on next page)

(continued from previous page)

```
  [<c020527f>] kobject_uevent+0xa/0xf
  [<c02779f1>] store_uevent+0x4f/0x58
  [<c027758e>] dev_attr_store+0x29/0x2f
  [<c01bec4f>] sysfs_write_file+0x16e/0x19c
  [<c0183ba7>] vfs_write+0xd1/0x15a
  [<c01841d7>] sys_write+0x3d/0x72
  [<c0104112>] sysenter_past_esp+0x5f/0x99
  [<b7f7b410>] 0xb7f7b410
  =======================

FIX kmalloc-8: Restoring Redzone 0xc90f6d28-0xc90f6d2b=0xcc
```

If SLUB encounters a corrupted object (full detection requires the kernel to be booted with slub_debug) then the following output will be dumped into the syslog:

1. Description of the problem encountered

   This will be a message in the system log starting with:

   ```
   ============================================
   BUG <slab cache affected>: <What went wrong>
   --------------------------------------------

   INFO: <corruption start>-<corruption_end> <more info>
   INFO: Slab <address> <slab information>
   INFO: Object <address> <object information>
   INFO: Allocated in <kernel function> age=<jiffies since alloc> cpu=
    ↪<allocated by
      cpu> pid=<pid of the process>
   INFO: Freed in <kernel function> age=<jiffies since free> cpu=<freed␣
    ↪by cpu>
      pid=<pid of the process>
   ```

   (Object allocation / free information is only available if SLAB_STORE_USER is set for the slab. slub_debug sets that option)

2. The object contents if an object was involved.

   Various types of lines can follow the BUG SLUB line:

   Bytes b4 <address> [<bytes>] Shows a few bytes before the object where the problem was detected. Can be useful if the corruption does not stop with the start of the object.

   Object <address> [<bytes>] The bytes of the object. If the object is inactive then the bytes typically contain poison values. Any non-poison value shows a corruption by a write after free.

   Redzone <address> [<bytes>] The Redzone following the object. The Redzone is used to detect writes after the object. All bytes should always have the same value. If there is any deviation then it is due to a write after the object boundary.

   (Redzone information is only available if SLAB_RED_ZONE is set. slub_debug sets that option)

   Padding <address> [<bytes>] Unused data to fill up the space in order to get the next object properly aligned. In the debug case we make sure

that there are at least 4 bytes of padding. This allows the detection of writes before the object.

3. A stackdump

   The stackdump describes the location where the error was detected. The cause of the corruption is may be more likely found by looking at the function that allocated or freed the object.

4. Report on how the problem was dealt with in order to ensure the continued operation of the system.

   These are messages in the system log beginning with:

   ```
   FIX <slab cache affected>: <corrective action taken>
   ```

   In the above sample SLUB found that the Redzone of an active object has been overwritten. Here a string of 8 characters was written into a slab that has the length of 8 characters. However, a 8 character string needs a terminating 0. That zero has overwritten the first byte of the Redzone field. After reporting the details of the issue encountered the FIX SLUB message tells us that SLUB has restored the Redzone to its proper value and then system operations continue.

Emergency operations

Minimal debugging (sanity checks alone) can be enabled by booting with:

```
slub_debug=F
```

This will be generally be enough to enable the resiliency features of slub which will keep the system running even if a bad kernel component will keep corrupting objects. This may be important for production systems. Performance will be impacted by the sanity checks and there will be a continual stream of error messages to the syslog but no additional memory will be used (unlike full debugging).

No guarantees. The kernel component still needs to be fixed. Performance may be optimized further by locating the slab that experiences corruption and enabling debugging only for that cache

I.e.:

```
slub_debug=F,dentry
```

If the corruption occurs by writing after the end of the object then it may be advisable to enable a Redzone to avoid corrupting the beginning of other objects:

```
slub_debug=FZ,dentry
```

Extended slabinfo mode and plotting

The `slabinfo` tool has a special 'extended' ('-X') mode that includes:

- Slabcache Totals
- Slabs sorted by size (up to -N <num> slabs, default 1)
- Slabs sorted by loss (up to -N <num> slabs, default 1)

Additionally, in this mode `slabinfo` does not dynamically scale sizes (G/M/K) and reports everything in bytes (this functionality is also available to other slabinfo modes via '-B' option) which makes reporting more precise and accurate. Moreover, in some sense the -X' mode also simplifies the analysis of slabs' behaviour, because its output can be plotted using the ``slabinfo-gnuplot.sh` script. So it pushes the analysis from looking through the numbers (tons of numbers) to something easier – visual analysis.

To generate plots:

a) collect slabinfo extended records, for example:

```
while [ 1 ]; do slabinfo -X >> FOO_STATS; sleep 1; done
```

b) pass stats file(-s) to `slabinfo-gnuplot.sh` script:

```
slabinfo-gnuplot.sh FOO_STATS [FOO_STATS2 .. FOO_STATSN]
```

The `slabinfo-gnuplot.sh` script will pre-processes the collected records and generates 3 png files (and 3 pre-processing cache files) per STATS file: - Slabcache Totals: FOO_STATS-totals.png - Slabs sorted by size: FOO_STATS-slabs-by-size.png - Slabs sorted by loss: FOO_STATS-slabs-by-loss.png

Another use case, when `slabinfo-gnuplot.sh` can be useful, is when you need to compare slabs' behaviour "prior to" and "after" some code modification. To help you out there, `slabinfo-gnuplot.sh` script can 'merge' the Slabcache Totals sections from different measurements. To visually compare N plots:

a) Collect as many STATS1, STATS2, .. STATSN files as you need:

```
while [ 1 ]; do slabinfo -X >> STATS<X>; sleep 1; done
```

b) Pre-process those STATS files:

```
slabinfo-gnuplot.sh STATS1 STATS2 .. STATSN
```

c) Execute `slabinfo-gnuplot.sh` in '-t' mode, passing all of the generated pre-processed *-totals:

```
slabinfo-gnuplot.sh -t STATS1-totals STATS2-totals .. STATSN-totals
```

This will produce a single plot (png file).

Plots, expectedly, can be large so some fluctuations or small spikes can go unnoticed. To deal with that, `slabinfo-gnuplot.sh` has two options to 'zoom-in' / 'zoom-out' :

a) `-s %d,%d` – overwrites the default image width and heigh

b) `-r %d,%d` – specifies a range of samples to use (for example, in `slabinfo -X >> FOO_STATS; sleep 1;` case, using a `-r 40,60` range will plot only samples collected between 40th and 60th seconds).

Christoph Lameter, May 30, 2007 Sergey Senozhatsky, October 23, 2015

2.20 Split page table lock

Originally, mm->page_table_lock spinlock protected all page tables of the mm_struct. But this approach leads to poor page fault scalability of multi-threaded applications due high contention on the lock. To improve scalability, split page table lock was introduced.

With split page table lock we have separate per-table lock to serialize access to the table. At the moment we use split lock for PTE and PMD tables. Access to higher level tables protected by mm->page_table_lock.

There are helpers to lock/unlock a table and other accessor functions:

- **pte_offset_map_lock()** maps pte and takes PTE table lock, returns pointer to the taken lock;
- **pte_unmap_unlock()** unlocks and unmaps PTE table;
- **pte_alloc_map_lock()** allocates PTE table if needed and take the lock, returns pointer to taken lock or NULL if allocation failed;
- **pte_lockptr()** returns pointer to PTE table lock;
- **pmd_lock()** takes PMD table lock, returns pointer to taken lock;
- **pmd_lockptr()** returns pointer to PMD table lock;

Split page table lock for PTE tables is enabled compile-time if CONFIG_SPLIT_PTLOCK_CPUS (usually 4) is less or equal to NR_CPUS. If split lock is disabled, all tables guaded by mm->page_table_lock.

Split page table lock for PMD tables is enabled, if it’s enabled for PTE tables and the architecture supports it (see below).

2.20.1 Hugetlb and split page table lock

Hugetlb can support several page sizes. We use split lock only for PMD level, but not for PUD.

Hugetlb-specific helpers:

- **huge_pte_lock()** takes pmd split lock for PMD_SIZE page, mm->page_table_lock otherwise;
- **huge_pte_lockptr()** returns pointer to table lock;

2.20.2 Support of split page table lock by an architecture

There’s no need in special enabling of PTE split page table lock: everything required is done by pgtable_pte_page_ctor() and pgtable_pte_page_dtor(), which must be called on PTE table allocation / freeing.

Make sure the architecture doesn’t use slab allocator for page table allocation: slab uses page->slab_cache for its pages. This field shares storage with page->ptl.

PMD split lock only makes sense if you have more than two page table levels.

PMD split lock enabling requires pgtable_pmd_page_ctor() call on PMD table allocation and pgtable_pmd_page_dtor() on freeing.

Allocation usually happens in pmd_alloc_one(), freeing in pmd_free() and pmd_free_tlb(), but make sure you cover all PMD table allocation / freeing paths: i.e X86_PAE preallocate few PMDs on pgd_alloc().

With everything in place you can set CONFIG_ARCH_ENABLE_SPLIT_PMD_PTLOCK.

NOTE: pgtable_pte_page_ctor() and pgtable_pmd_page_ctor() can fail – it must be handled properly.

2.20.3 page->ptl

page->ptl is used to access split page table lock, where ‘page’ is struct page of page containing the table. It shares storage with page->private (and few other fields in union).

To avoid increasing size of struct page and have best performance, we use a trick:

- if spinlock_t fits into long, we use page->ptr as spinlock, so we can avoid indirect access and save a cache line.
- if size of spinlock_t is bigger then size of long, we use page->ptl as pointer to spinlock_t and allocate it dynamically. This allows to use split lock with enabled DEBUG_SPINLOCK or DEBUG_LOCK_ALLOC, but costs one more cache line for indirect access;

The spinlock_t allocated in pgtable_pte_page_ctor() for PTE table and in pgtable_pmd_page_ctor() for PMD table.

Please, never access page->ptl directly – use appropriate helper.

2.21 Transparent Hugepage Support

This document describes design principles for Transparent Hugepage (THP) support and its interaction with other parts of the memory management system.

2.21.1 Design principles

- “graceful fallback”: mm components which don’t have transparent hugepage knowledge fall back to breaking huge pmd mapping into table of ptes and, if necessary, split a transparent hugepage. Therefore these components can continue working on the regular pages or regular pte mappings.
- if a hugepage allocation fails because of memory fragmentation, regular pages should be gracefully allocated instead and mixed in the same vma without any failure or significant delay and without userland noticing
- if some task quits and more hugepages become available (either immediately in the buddy or through the VM), guest physical memory backed by regular pages should be relocated on hugepages automatically (with khugepaged)
- it doesn’t require memory reservation and in turn it uses hugepages whenever possible (the only possible reservation here is kernelcore= to avoid unmovable pages to fragment all the memory but such a tweak is not specific to transparent hugepage support and it’s a generic feature that applies to all dynamic high order allocations in the kernel)

2.21.2 get_user_pages and follow_page

get_user_pages and follow_page if run on a hugepage, will return the head or tail pages as usual (exactly as they would do on hugetlbfs). Most GUP users will only care about the actual physical address of the page and its temporary pinning to release after the I/O is complete, so they won’t ever notice the fact the page is huge. But if any driver is going to mangle over the page structure of the tail page (like for checking page->mapping or other bits that are relevant for the head page and not the tail page), it should be updated to jump to check head page instead. Taking a reference on any head/tail page would prevent the page from being split by anyone.

Note: these aren’t new constraints to the GUP API, and they match the same constraints that apply to hugetlbfs too, so any driver capable of handling GUP on hugetlbfs will also work fine on transparent hugepage backed mappings.

In case you can’t handle compound pages if they’re returned by follow_page, the FOLL_SPLIT bit can be specified as a parameter to follow_page, so that it will split the hugepages before returning them.

2.21.3 Graceful fallback

Code walking pagetables but unaware about huge pmds can simply call split_huge_pmd(vma, pmd, addr) where the pmd is the one returned by pmd_offset. It’s trivial to make the code transparent hugepage aware by just grepping for “pmd_offset” and adding split_huge_pmd where missing after pmd_offset returns the pmd. Thanks to the graceful fallback design, with a one liner change, you can avoid to write hundreds if not thousands of lines of complex code to make your code hugepage aware.

If you're not walking pagetables but you run into a physical hugepage that you can't handle natively in your code, you can split it by calling split_huge_page(page). This is what the Linux VM does before it tries to swapout the hugepage for example. split_huge_page() can fail if the page is pinned and you must handle this correctly.

Example to make mremap.c transparent hugepage aware with a one liner change:

```
diff --git a/mm/mremap.c b/mm/mremap.c
--- a/mm/mremap.c
+++ b/mm/mremap.c
@@ -41,6 +41,7 @@ static pmd_t *get_old_pmd(struct mm_stru
                return NULL;

        pmd = pmd_offset(pud, addr);
+       split_huge_pmd(vma, pmd, addr);
        if (pmd_none_or_clear_bad(pmd))
                return NULL;
```

2.21.4 Locking in hugepage aware code

We want as much code as possible hugepage aware, as calling split_huge_page() or split_huge_pmd() has a cost.

To make pagetable walks huge pmd aware, all you need to do is to call pmd_trans_huge() on the pmd returned by pmd_offset. You must hold the mmap_lock in read (or write) mode to be sure a huge pmd cannot be created from under you by khugepaged (khugepaged collapse_huge_page takes the mmap_lock in write mode in addition to the anon_vma lock). If pmd_trans_huge returns false, you just fallback in the old code paths. If instead pmd_trans_huge returns true, you have to take the page table lock (pmd_lock()) and re-run pmd_trans_huge. Taking the page table lock will prevent the huge pmd being converted into a regular pmd from under you (split_huge_pmd can run in parallel to the pagetable walk). If the second pmd_trans_huge returns false, you should just drop the page table lock and fallback to the old code as before. Otherwise, you can proceed to process the huge pmd and the hugepage natively. Once finished, you can drop the page table lock.

2.21.5 Refcounts and transparent huge pages

Refcounting on THP is mostly consistent with refcounting on other compound pages:

- get_page()/put_page() and GUP operate on head page's ->_refcount.
- ->_refcount in tail pages is always zero: get_page_unless_zero() never succeeds on tail pages.
- map/unmap of the pages with PTE entry increment/decrement ->_mapcount on relevant sub-page of the compound page.
- map/unmap of the whole compound page is accounted for in compound_mapcount (stored in first tail page). For file huge pages, we also increment ->_mapcount of all sub-pages in order to have race-free detection of last unmap of subpages.

PageDoubleMap() indicates that the page is possibly mapped with PTEs.

For anonymous pages, PageDoubleMap() also indicates ->_mapcount in all subpages is offset up by one. This additional reference is required to get race-free detection of unmap of subpages when we have them mapped with both PMDs and PTEs.

This optimization is required to lower the overhead of per-subpage mapcount tracking. The alternative is to alter ->_mapcount in all subpages on each map/unmap of the whole compound page.

For anonymous pages, we set PG_double_map when a PMD of the page is split for the first time, but still have a PMD mapping. The additional references go away with the last compound_mapcount.

File pages get PG_double_map set on the first map of the page with PTE and goes away when the page gets evicted from the page cache.

split_huge_page internally has to distribute the refcounts in the head page to the tail pages before clearing all PG_head/tail bits from the page structures. It can be done easily for refcounts taken by page table entries, but we don't have enough information on how to distribute any additional pins (i.e. from get_user_pages). split_huge_page() fails any requests to split pinned huge pages: it expects page count to be equal to the sum of mapcount of all sub-pages plus one (split_huge_page caller must have a reference to the head page).

split_huge_page uses migration entries to stabilize page->_refcount and page->_mapcount of anonymous pages. File pages just get unmapped.

We are safe against physical memory scanners too: the only legitimate way a scanner can get a reference to a page is get_page_unless_zero().

All tail pages have zero ->_refcount until atomic_add(). This prevents the scanner from getting a reference to the tail page up to that point. After the atomic_add() we don't care about the ->_refcount value. We already know how many references should be uncharged from the head page.

For head page get_page_unless_zero() will succeed and we don't mind. It's clear where references should go after split: it will stay on the head page.

Note that split_huge_pmd() doesn't have any limitations on refcounting: pmd can be split at any point and never fails.

2.21.6 Partial unmap and deferred_split_huge_page()

Unmapping part of THP (with munmap() or other way) is not going to free memory immediately. Instead, we detect that a subpage of THP is not in use in page_remove_rmap() and queue the THP for splitting if memory pressure comes. Splitting will free up unused subpages.

Splitting the page right away is not an option due to locking context in the place where we can detect partial unmap. It also might be counterproductive since in many cases partial unmap happens during exit(2) if a THP crosses a VMA boundary.

The function deferred_split_huge_page() is used to queue a page for splitting. The splitting itself will happen when we get memory pressure via shrinker interface.

2.22 Unevictable LRU Infrastructure

2.22.1 Introduction

This document describes the Linux memory manager's "Unevictable LRU" infrastructure and the use of this to manage several types of "unevictable" pages.

The document attempts to provide the overall rationale behind this mechanism and the rationale for some of the design decisions that drove the implementation. The latter design rationale is discussed in the context of an implementation description. Admittedly, one can obtain the implementation details - the "what does it do?" - by reading the code. One hopes that the descriptions below add value by provide the answer to "why does it do that?".

2.22.2 The Unevictable LRU

The Unevictable LRU facility adds an additional LRU list to track unevictable pages and to hide these pages from vmscan. This mechanism is based on a patch by Larry Woodman of Red Hat to address several scalability problems with page reclaim in Linux. The problems have been observed at customer sites on large memory x86_64 systems.

To illustrate this with an example, a non-NUMA x86_64 platform with 128GB of main memory will have over 32 million 4k pages in a single zone. When a large fraction of these pages are not evictable for any reason [see below], vmscan will spend a lot of time scanning the LRU lists looking for the small fraction of pages that are evictable. This can result in a situation where all CPUs are spending 100% of their time in vmscan for hours or days on end, with the system completely unresponsive.

The unevictable list addresses the following classes of unevictable pages:

- Those owned by ramfs.
- Those mapped into SHM_LOCK'd shared memory regions.
- Those mapped into VM_LOCKED [mlock()ed] VMAs.

The infrastructure may also be able to handle other conditions that make pages unevictable, either by definition or by circumstance, in the future.

The Unevictable Page List

The Unevictable LRU infrastructure consists of an additional, per-zone, LRU list called the "unevictable" list and an associated page flag, PG_unevictable, to indicate that the page is being managed on the unevictable list.

The PG_unevictable flag is analogous to, and mutually exclusive with, the PG_active flag in that it indicates on which LRU list a page resides when PG_lru is set.

The Unevictable LRU infrastructure maintains unevictable pages on an additional LRU list for a few reasons:

(1) We get to "treat unevictable pages just like we treat other pages in the system - which means we get to use the same code to manipulate them, the same code to isolate them (for migrate, etc.), the same code to keep track of the statistics, etc…" [Rik van Riel]

(2) We want to be able to migrate unevictable pages between nodes for memory defragmentation, workload management and memory hotplug. The linux kernel can only migrate pages that it can successfully isolate from the LRU lists. If we were to maintain pages elsewhere than on an LRU-like list, where they can be found by isolate_lru_page(), we would prevent their migration, unless we reworked migration code to find the unevictable pages itself.

The unevictable list does not differentiate between file-backed and anonymous, swap-backed pages. This differentiation is only important while the pages are, in fact, evictable.

The unevictable list benefits from the "arrayification" of the per-zone LRU lists and statistics originally proposed and posted by Christoph Lameter.

The unevictable list does not use the LRU pagevec mechanism. Rather, unevictable pages are placed directly on the page's zone's unevictable list under the zone lru_lock. This allows us to prevent the stranding of pages on the unevictable list when one task has the page isolated from the LRU and other tasks are changing the "evictability" state of the page.

Memory Control Group Interaction

The unevictable LRU facility interacts with the memory control group [aka memory controller; see Documentation/admin-guide/cgroup-v1/memory.rst] by extending the lru_list enum.

The memory controller data structure automatically gets a per-zone unevictable list as a result of the "arrayification" of the per-zone LRU lists (one per lru_list enum element). The memory controller tracks the movement of pages to and from the unevictable list.

When a memory control group comes under memory pressure, the controller will not attempt to reclaim pages on the unevictable list. This has a couple of effects:

(1) Because the pages are "hidden" from reclaim on the unevictable list, the reclaim process can be more efficient, dealing only with pages that have a chance of being reclaimed.

(2) On the other hand, if too many of the pages charged to the control group are unevictable, the evictable portion of the working set of the tasks in the control group may not fit into the available memory. This can cause the control group to thrash or to OOM-kill tasks.

Marking Address Spaces Unevictable

For facilities such as ramfs none of the pages attached to the address space may be evicted. To prevent eviction of any such pages, the AS_UNEVICTABLE address space flag is provided, and this can be manipulated by a filesystem using a number of wrapper functions:

- `void mapping_set_unevictable(struct address_space *mapping);`

  Mark the address space as being completely unevictable.

- `void mapping_clear_unevictable(struct address_space *mapping);`

  Mark the address space as being evictable.

- `int mapping_unevictable(struct address_space *mapping);`

  Query the address space, and return true if it is completely unevictable.

These are currently used in three places in the kernel:

(1) By ramfs to mark the address spaces of its inodes when they are created, and this mark remains for the life of the inode.

(2) By SYSV SHM to mark SHM_LOCK'd address spaces until SHM_UNLOCK is called.

Note that SHM_LOCK is not required to page in the locked pages if they're swapped out; the application must touch the pages manually if it wants to ensure they're in memory.

(3) By the i915 driver to mark pinned address space until it's unpinned. The amount of unevictable memory marked by i915 driver is roughly the bounded object size in debugfs/dri/0/i915_gem_objects.

Detecting Unevictable Pages

The function page_evictable() in vmscan.c determines whether a page is evictable or not using the query function outlined above [see section Marking address spaces unevictable] to check the AS_UNEVICTABLE flag.

For address spaces that are so marked after being populated (as SHM regions might be), the lock action (eg: SHM_LOCK) can be lazy, and need not populate the page tables for the region as does, for example, mlock(), nor need it make any special effort to push any pages in the SHM_LOCK'd area to the unevictable list. Instead, vmscan will do this if and when it encounters the pages during a reclamation scan.

On an unlock action (such as SHM_UNLOCK), the unlocker (eg: shmctl()) must scan the pages in the region and "rescue" them from the unevictable list if no other condition is keeping them unevictable. If an unevictable region is destroyed, the pages are also "rescued" from the unevictable list in the process of freeing them.

page_evictable() also checks for mlocked pages by testing an additional page flag, PG_mlocked (as wrapped by PageMlocked()), which is set when a page is faulted into a VM_LOCKED vma, or found in a vma being VM_LOCKED.

Vmscan's Handling of Unevictable Pages

If unevictable pages are culled in the fault path, or moved to the unevictable list at mlock() or mmap() time, vmscan will not encounter the pages until they have become evictable again (via munlock() for example) and have been "rescued"from the unevictable list. However, there may be situations where we decide, for the sake of expediency, to leave a unevictable page on one of the regular active/inactive LRU lists for vmscan to deal with. vmscan checks for such pages in all of the shrink_{active|inactive|page}_list() functions and will "cull" such pages that it encounters: that is, it diverts those pages to the unevictable list for the zone being scanned.

There may be situations where a page is mapped into a VM_LOCKED VMA, but the page is not marked as PG_mlocked. Such pages will make it all the way to shrink_page_list() where they will be detected when vmscan walks the reverse map in try_to_unmap(). If try_to_unmap() returns SWAP_MLOCK, shrink_page_list() will cull the page at that point.

To "cull" an unevictable page, vmscan simply puts the page back on the LRU list using putback_lru_page() - the inverse operation to isolate_lru_page() - after

dropping the page lock. Because the condition which makes the page unevictable may change once the page is unlocked, putback_lru_page() will recheck the unevictable state of a page that it places on the unevictable list. If the page has become unevictable, putback_lru_page() removes it from the list and retries, including the page_unevictable() test. Because such a race is a rare event and movement of pages onto the unevictable list should be rare, these extra evictabilty checks should not occur in the majority of calls to putback_lru_page().

2.22.3 MLOCKED Pages

The unevictable page list is also useful for mlock(), in addition to ramfs and SYSV SHM. Note that mlock() is only available in CONFIG_MMU=y situations; in NOMMU situations, all mappings are effectively mlocked.

History

The "Unevictable mlocked Pages" infrastructure is based on work originally posted by Nick Piggin in an RFC patch entitled "mm: mlocked pages off LRU". Nick posted his patch as an alternative to a patch posted by Christoph Lameter to achieve the same objective: hiding mlocked pages from vmscan.

In Nick's patch, he used one of the struct page LRU list link fields as a count of VM_LOCKED VMAs that map the page. This use of the link field for a count prevented the management of the pages on an LRU list, and thus mlocked pages were not migratable as isolate_lru_page() could not find them, and the LRU list link field was not available to the migration subsystem.

Nick resolved this by putting mlocked pages back on the lru list before attempting to isolate them, thus abandoning the count of VM_LOCKED VMAs. When Nick's patch was integrated with the Unevictable LRU work, the count was replaced by walking the reverse map to determine whether any VM_LOCKED VMAs mapped the page. More on this below.

Basic Management

mlocked pages - pages mapped into a VM_LOCKED VMA - are a class of unevictable pages. When such a page has been "noticed" by the memory management subsystem, the page is marked with the PG_mlocked flag. This can be manipulated using the PageMlocked() functions.

A PG_mlocked page will be placed on the unevictable list when it is added to the LRU. Such pages can be "noticed" by memory management in several places:

(1) in the mlock()/mlockall() system call handlers;

(2) in the mmap() system call handler when mmapping a region with the MAP_LOCKED flag;

(3) mmapping a region in a task that has called mlockall() with the MCL_FUTURE flag

(4) in the fault path, if mlocked pages are "culled" in the fault path, and when a VM_LOCKED stack segment is expanded; or

(5) as mentioned above, in vmscan:shrink_page_list() when attempting to reclaim a page in a VM_LOCKED VMA via try_to_unmap()

all of which result in the VM_LOCKED flag being set for the VMA if it doesn't already have it set.

mlocked pages become unlocked and rescued from the unevictable list when:

(1) mapped in a range unlocked via the munlock()/munlockall() system calls;

(2) munmap()'d out of the last VM_LOCKED VMA that maps the page, including unmapping at task exit;

(3) when the page is truncated from the last VM_LOCKED VMA of an mmapped file; or

(4) before a page is COW'd in a VM_LOCKED VMA.

mlock()/mlockall() System Call Handling

Both [do_]mlock() and [do_]mlockall() system call handlers call mlock_fixup() for each VMA in the range specified by the call. In the case of mlockall(), this is the entire active address space of the task. Note that mlock_fixup() is used for both mlocking and munlocking a range of memory. A call to mlock() an already VM_LOCKED VMA, or to munlock() a VMA that is not VM_LOCKED is treated as a no-op, and mlock_fixup() simply returns.

If the VMA passes some filtering as described in "Filtering Special Vmas" below, mlock_fixup() will attempt to merge the VMA with its neighbors or split off a subset of the VMA if the range does not cover the entire VMA. Once the VMA has been merged or split or neither, mlock_fixup() will call populate_vma_page_range() to fault in the pages via get_user_pages() and to mark the pages as mlocked via mlock_vma_page().

Note that the VMA being mlocked might be mapped with PROT_NONE. In this case, get_user_pages() will be unable to fault in the pages. That's okay. If pages do end up getting faulted into this VM_LOCKED VMA, we'll handle them in the fault path or in vmscan.

Also note that a page returned by get_user_pages() could be truncated or migrated out from under us, while we're trying to mlock it. To detect this, populate_vma_page_range() checks page_mapping() after acquiring the page lock. If the page is still associated with its mapping, we'll go ahead and call mlock_vma_page(). If the mapping is gone, we just unlock the page and move on. In the worst case, this will result in a page mapped in a VM_LOCKED VMA remaining on a normal LRU list without being PageMlocked(). Again, vmscan will detect and cull such pages.

mlock_vma_page() will call TestSetPageMlocked() for each page returned by get_user_pages(). We use TestSetPageMlocked() because the page might already be mlocked by another task/VMA and we don't want to do extra work. We especially do not want to count an mlocked page more than once in the statistics. If the page was already mlocked, mlock_vma_page() need do nothing more.

If the page was NOT already mlocked, mlock_vma_page() attempts to isolate the page from the LRU, as it is likely on the appropriate active or inactive list at that

time. If the isolate_lru_page() succeeds, mlock_vma_page() will put back the page - by calling putback_lru_page() - which will notice that the page is now mlocked and divert the page to the zone's unevictable list. If mlock_vma_page() is unable to isolate the page from the LRU, vmscan will handle it later if and when it attempts to reclaim the page.

Filtering Special VMAs

mlock_fixup() filters several classes of "special" VMAs:

1) VMAs with VM_IO or VM_PFNMAP set are skipped entirely. The pages behind these mappings are inherently pinned, so we don't need to mark them as mlocked. In any case, most of the pages have no struct page in which to so mark the page. Because of this, get_user_pages() will fail for these VMAs, so there is no sense in attempting to visit them.

2) VMAs mapping hugetlbfs page are already effectively pinned into memory. We neither need nor want to mlock() these pages. However, to preserve the prior behavior of mlock() - before the unevictable/mlock changes - mlock_fixup() will call make_pages_present() in the hugetlbfs VMA range to allocate the huge pages and populate the ptes.

3) VMAs with VM_DONTEXPAND are generally userspace mappings of kernel pages, such as the VDSO page, relay channel pages, etc. These pages are inherently unevictable and are not managed on the LRU lists. mlock_fixup() treats these VMAs the same as hugetlbfs VMAs. It calls make_pages_present() to populate the ptes.

Note that for all of these special VMAs, mlock_fixup() does not set the VM_LOCKED flag. Therefore, we won't have to deal with them later during munlock(), munmap() or task exit. Neither does mlock_fixup() account these VMAs against the task's "locked_vm".

munlock()/munlockall() System Call Handling

The munlock() and munlockall() system calls are handled by the same functions - do_mlock[all]() - as the mlock() and mlockall() system calls with the unlock vs lock operation indicated by an argument. So, these system calls are also handled by mlock_fixup(). Again, if called for an already munlocked VMA, mlock_fixup() simply returns. Because of the VMA filtering discussed above, VM_LOCKED will not be set in any "special" VMAs. So, these VMAs will be ignored for munlock.

If the VMA is VM_LOCKED, mlock_fixup() again attempts to merge or split off the specified range. The range is then munlocked via the function populate_vma_page_range() - the same function used to mlock a VMA range - passing a flag to indicate that munlock() is being performed.

Because the VMA access protections could have been changed to PROT_NONE after faulting in and mlocking pages, get_user_pages() was unreliable for visiting these pages for munlocking. Because we don't want to leave pages mlocked, get_user_pages() was enhanced to accept a flag to ignore the permissions when fetching the pages - all of which should be resident as a result of previous mlocking.

For munlock(), populate_vma_page_range() unlocks individual pages by calling munlock_vma_page(). munlock_vma_page() unconditionally clears the PG_mlocked flag using TestClearPageMlocked(). As with mlock_vma_page(), munlock_vma_page() use the Test*PageMlocked() function to handle the case where the page might have already been unlocked by another task. If the page was mlocked, munlock_vma_page() updates that zone statistics for the number of mlocked pages. Note, however, that at this point we haven't checked whether the page is mapped by other VM_LOCKED VMAs.

We can't call try_to_munlock(), the function that walks the reverse map to check for other VM_LOCKED VMAs, without first isolating the page from the LRU. try_to_munlock() is a variant of try_to_unmap() and thus requires that the page not be on an LRU list [more on these below]. However, the call to isolate_lru_page() could fail, in which case we couldn't try_to_munlock(). So, we go ahead and clear PG_mlocked up front, as this might be the only chance we have. If we can successfully isolate the page, we go ahead and try_to_munlock(), which will restore the PG_mlocked flag and update the zone page statistics if it finds another VMA holding the page mlocked. If we fail to isolate the page, we'll have left a potentially mlocked page on the LRU. This is fine, because we'll catch it later if and if vmscan tries to reclaim the page. This should be relatively rare.

Migrating MLOCKED Pages

A page that is being migrated has been isolated from the LRU lists and is held locked across unmapping of the page, updating the page's address space entry and copying the contents and state, until the page table entry has been replaced with an entry that refers to the new page. Linux supports migration of mlocked pages and other unevictable pages. This involves simply moving the PG_mlocked and PG_unevictable states from the old page to the new page.

Note that page migration can race with mlocking or munlocking of the same page. This has been discussed from the mlock/munlock perspective in the respective sections above. Both processes (migration and m[un]locking) hold the page locked. This provides the first level of synchronization. Page migration zeros out the page_mapping of the old page before unlocking it, so m[un]lock can skip these pages by testing the page mapping under page lock.

To complete page migration, we place the new and old pages back onto the LRU after dropping the page lock. The "unneeded" page - old page on success, new page on failure - will be freed when the reference count held by the migration process is released. To ensure that we don't strand pages on the unevictable list because of a race between munlock and migration, page migration uses the putback_lru_page() function to add migrated pages back to the LRU.

Compacting MLOCKED Pages

The unevictable LRU can be scanned for compactable regions and the default behavior is to do so. /proc/sys/vm/compact_unevictable_allowed controls this behavior (see Documentation/admin-guide/sysctl/vm.rst). Once scanning of the unevictable LRU is enabled, the work of compaction is mostly handled by the page migration code and the same work flow as described in MIGRATING MLOCKED PAGES will apply.

MLOCKING Transparent Huge Pages

A transparent huge page is represented by a single entry on an LRU list. Therefore, we can only make unevictable an entire compound page, not individual subpages.

If a user tries to mlock() part of a huge page, we want the rest of the page to be reclaimable.

We cannot just split the page on partial mlock() as split_huge_page() can fail and new intermittent failure mode for the syscall is undesirable.

We handle this by keeping PTE-mapped huge pages on normal LRU lists: the PMD on border of VM_LOCKED VMA will be split into PTE table.

This way the huge page is accessible for vmscan. Under memory pressure the page will be split, subpages which belong to VM_LOCKED VMAs will be moved to unevictable LRU and the rest can be reclaimed.

See also comment in follow_trans_huge_pmd().

mmap(MAP_LOCKED) System Call Handling

In addition the mlock()/mlockall() system calls, an application can request that a region of memory be mlocked supplying the MAP_LOCKED flag to the mmap() call. There is one important and subtle difference here, though. mmap() + mlock() will fail if the range cannot be faulted in (e.g. because mm_populate fails) and returns with ENOMEM while mmap(MAP_LOCKED) will not fail. The mmaped area will still have properties of the locked area - aka. pages will not get swapped out - but major page faults to fault memory in might still happen.

Furthermore, any mmap() call or brk() call that expands the heap by a task that has previously called mlockall() with the MCL_FUTURE flag will result in the newly mapped memory being mlocked. Before the unevictable/mlock changes, the kernel simply called make_pages_present() to allocate pages and populate the page table.

To mlock a range of memory under the unevictable/mlock infrastructure, the mmap() handler and task address space expansion functions call populate_vma_page_range() specifying the vma and the address range to mlock.

The callers of populate_vma_page_range() will have already added the memory range to be mlocked to the task's "locked_vm". To account for filtered VMAs, populate_vma_page_range() returns the number of pages NOT mlocked. All of the callers then subtract a non-negative return value from the task's locked_vm. A negative return value represent an error - for example, from get_user_pages() attempting to fault in a VMA with PROT_NONE access. In this case, we leave the

memory range accounted as locked_vm, as the protections could be changed later and pages allocated into that region.

munmap()/exit()/exec() System Call Handling

When unmapping an mlocked region of memory, whether by an explicit call to munmap() or via an internal unmap from exit() or exec() processing, we must munlock the pages if we're removing the last VM_LOCKED VMA that maps the pages. Before the unevictable/mlock changes, mlocking did not mark the pages in any way, so unmapping them required no processing.

To munlock a range of memory under the unevictable/mlock infrastructure, the munmap() handler and task address space call tear down function munlock_vma_pages_all(). The name reflects the observation that one always specifies the entire VMA range when munlock()ing during unmap of a region. Because of the VMA filtering when mlocking() regions, only "normal" VMAs that actually contain mlocked pages will be passed to munlock_vma_pages_all().

munlock_vma_pages_all() clears the VM_LOCKED VMA flag and, like mlock_fixup() for the munlock case, calls __munlock_vma_pages_range() to walk the page table for the VMA's memory range and munlock_vma_page() each resident page mapped by the VMA. This effectively munlocks the page, only if this is the last VM_LOCKED VMA that maps the page.

try_to_unmap()

Pages can, of course, be mapped into multiple VMAs. Some of these VMAs may have VM_LOCKED flag set. It is possible for a page mapped into one or more VM_LOCKED VMAs not to have the PG_mlocked flag set and therefore reside on one of the active or inactive LRU lists. This could happen if, for example, a task in the process of munlocking the page could not isolate the page from the LRU. As a result, vmscan/shrink_page_list() might encounter such a page as described in section "vmscan's handling of unevictable pages". To handle this situation, try_to_unmap() checks for VM_LOCKED VMAs while it is walking a page's reverse map.

try_to_unmap() is always called, by either vmscan for reclaim or for page migration, with the argument page locked and isolated from the LRU. Separate functions handle anonymous and mapped file and KSM pages, as these types of pages have different reverse map lookup mechanisms, with different locking. In each case, whether rmap_walk_anon() or rmap_walk_file() or rmap_walk_ksm(), it will call try_to_unmap_one() for every VMA which might contain the page.

When trying to reclaim, if try_to_unmap_one() finds the page in a VM_LOCKED VMA, it will then mlock the page via mlock_vma_page() instead of unmapping it, and return SWAP_MLOCK to indicate that the page is unevictable: and the scan stops there.

mlock_vma_page() is called while holding the page table's lock (in addition to the page lock, and the rmap lock): to serialize against concurrent mlock or munlock or munmap system calls, mm teardown (munlock_vma_pages_all), reclaim, holepunching, and truncation of file pages and their anonymous COWed pages.

try_to_munlock() Reverse Map Scan

Warning: [!] TODO/FIXME: a better name might be page_mlocked() - analogous to the page_referenced() reverse map walker.

When munlock_vma_page() [see section munlock()/munlockall() System Call Handling above] tries to munlock a page, it needs to determine whether or not the page is mapped by any VM_LOCKED VMA without actually attempting to unmap all PTEs from the page. For this purpose, the unevictable/mlock infrastructure introduced a variant of try_to_unmap() called try_to_munlock().

try_to_munlock() calls the same functions as try_to_unmap() for anonymous and mapped file and KSM pages with a flag argument specifying unlock versus unmap processing. Again, these functions walk the respective reverse maps looking for VM_LOCKED VMAs. When such a VMA is found, as in the try_to_unmap() case, the functions mlock the page via mlock_vma_page() and return SWAP_MLOCK. This undoes the pre-clearing of the page's PG_mlocked done by munlock_vma_page.

Note that try_to_munlock()'s reverse map walk must visit every VMA in a page's reverse map to determine that a page is NOT mapped into any VM_LOCKED VMA. However, the scan can terminate when it encounters a VM_LOCKED VMA. Although try_to_munlock() might be called a great many times when munlocking a large region or tearing down a large address space that has been mlocked via mlockall(), overall this is a fairly rare event.

Page Reclaim in shrink_*_list()

shrink_active_list() culls any obviously unevictable pages - i.e. !page_evictable(page) - diverting these to the unevictable list. However, shrink_active_list() only sees unevictable pages that made it onto the active/inactive lru lists. Note that these pages do not have PageUnevictable set - otherwise they would be on the unevictable list and shrink_active_list would never see them.

Some examples of these unevictable pages on the LRU lists are:

(1) ramfs pages that have been placed on the LRU lists when first allocated.

(2) SHM_LOCK'd shared memory pages. shmctl(SHM_LOCK) does not attempt to allocate or fault in the pages in the shared memory region. This happens when an application accesses the page the first time after SHM_LOCK'ing the segment.

(3) mlocked pages that could not be isolated from the LRU and moved to the unevictable list in mlock_vma_page().

shrink_inactive_list() also diverts any unevictable pages that it finds on the inactive lists to the appropriate zone's unevictable list.

shrink_inactive_list() should only see SHM_LOCK'd pages that became SHM_LOCK'd after shrink_active_list() had moved them to the inactive list, or

pages mapped into VM_LOCKED VMAs that munlock_vma_page() couldn't isolate from the LRU to recheck via try_to_munlock(). shrink_inactive_list() won't notice the latter, but will pass on to shrink_page_list().

shrink_page_list() again culls obviously unevictable pages that it could encounter for similar reason to shrink_inactive_list(). Pages mapped into VM_LOCKED VMAs but without PG_mlocked set will make it all the way to try_to_unmap(). shrink_page_list() will divert them to the unevictable list when try_to_unmap() returns SWAP_MLOCK, as discussed above.

2.23 z3fold

z3fold is a special purpose allocator for storing compressed pages. It is designed to store up to three compressed pages per physical page. It is a zbud derivative which allows for higher compression ratio keeping the simplicity and determinism of its predecessor.

The main differences between z3fold and zbud are:

- unlike zbud, z3fold allows for up to PAGE_SIZE allocations
- z3fold can hold up to 3 compressed pages in its page
- z3fold doesn't export any API itself and is thus intended to be used via the zpool API.

To keep the determinism and simplicity, z3fold, just like zbud, always stores an integral number of compressed pages per page, but it can store up to 3 pages unlike zbud which can store at most 2. Therefore the compression ratio goes to around 2.7x while zbud's one is around 1.7x.

Unlike zbud (but like zsmalloc for that matter) z3fold_alloc() does not return a dereferenceable pointer. Instead, it returns an unsigned long handle which encodes actual location of the allocated object.

Keeping effective compression ratio close to zsmalloc's, z3fold doesn't depend on MMU enabled and provides more predictable reclaim behavior which makes it a better fit for small and response-critical systems.

2.24 zsmalloc

This allocator is designed for use with zram. Thus, the allocator is supposed to work well under low memory conditions. In particular, it never attempts higher order page allocation which is very likely to fail under memory pressure. On the other hand, if we just use single (0-order) pages, it would suffer from very high fragmentation – any object of size PAGE_SIZE/2 or larger would occupy an entire page. This was one of the major issues with its predecessor (xvmalloc).

To overcome these issues, zsmalloc allocates a bunch of 0-order pages and links them together using various 'struct page' fields. These linked pages act as a single higher-order page i.e. an object can span 0-order page boundaries. The code refers to these linked pages as a single entity called zspage.

For simplicity, zsmalloc can only allocate objects of size up to PAGE_SIZE since this satisfies the requirements of all its current users (in the worst case, page is incompressible and is thus stored “as-is”i.e. in uncompressed form). For allocation requests larger than this size, failure is returned (see zs_malloc).

Additionally, zs_malloc() does not return a dereferenceable pointer. Instead, it returns an opaque handle (unsigned long) which encodes actual location of the allocated object. The reason for this indirection is that zsmalloc does not keep zspages permanently mapped since that would cause issues on 32-bit systems where the VA region for kernel space mappings is very small. So, before using the allocating memory, the object has to be mapped using zs_map_object() to get a usable pointer and subsequently unmapped using zs_unmap_object().

2.24.1 stat

With CONFIG_ZSMALLOC_STAT, we could see zsmalloc internal information via `/sys/kernel/debug/zsmalloc/<user name>`. Here is a sample of stat output:

```
# cat /sys/kernel/debug/zsmalloc/zram0/classes

class  size almost_full almost_empty obj_allocated   obj_used pages_used␣
↪pages_per_zspage
   ...
   ...
    9   176           0            1          186        129           8  ␣
↪              4
   10   192           1            0         2880       2872         135  ␣
↪              3
   11   208           0            1          819        795          42  ␣
↪              2
   12   224           0            1          219        159          12  ␣
↪              4
   ...
   ...
```

class index

size object size zspage stores

almost_empty the number of ZS_ALMOST_EMPTY zspages(see below)

almost_full the number of ZS_ALMOST_FULL zspages(see below)

obj_allocated the number of objects allocated

obj_used the number of objects allocated to the user

pages_used the number of pages allocated for the class

pages_per_zspage the number of 0-order pages to make a zspage

We assign a zspage to ZS_ALMOST_EMPTY fullness group when n <= N / f, where

- n = number of allocated objects
- N = total number of objects zspage can store
- f = fullness_threshold_frac(ie, 4 at the moment)

Similarly, we assign zspage to:

- ZS_ALMOST_FULL when n > N / f
- ZS_EMPTY when n == 0
- ZS_FULL when n == N