Linux Mm Documentation

Release 6.8.0

The kernel development community

Jan 16, 2026

CONTENTS

CHAPTER

ONE

MEMORY MANAGEMENT GUIDE

This is a guide to understanding the memory management subsystem of Linux. If you are looking for advice on simply allocating memory, see the memory_allocation. For controlling and tuning guides, see the admin guide.

1.1 Physical Memory

Linux is available for a wide range of architectures so there is a need for an architecture-independent abstraction to represent the physical memory. This chapter describes the structures used to manage physical memory in a running system.

The first principal concept prevalent in the memory management is Non-Uniform Memory Access (NUMA). With multi-core and multi-socket machines, memory may be arranged into banks that incur a different cost to access depending on the "distance" from the processor. For example, there might be a bank of memory assigned to each CPU or a bank of memory very suitable for DMA near peripheral devices.

Each bank is called a node and the concept is represented under Linux by a `struct pglist_data` even if the architecture is UMA. This structure is always referenced by its typedef `pg_data_t`. A `pg_data_t` structure for a particular node can be referenced by `NODE_DATA(nid)` macro where `nid` is the ID of that node.

For NUMA architectures, the node structures are allocated by the architecture specific code early during boot. Usually, these structures are allocated locally on the memory bank they represent. For UMA architectures, only one static `pg_data_t` structure called `contig_page_data` is used. Nodes will be discussed further in Section *Nodes*

The entire physical address space is partitioned into one or more blocks called zones which represent ranges within memory. These ranges are usually determined by architectural constraints for accessing the physical memory. The memory range within a node that corresponds to a particular zone is described by a `struct zone`, typedeffed to `zone_t`. Each zone has one of the types described below.

- `ZONE_DMA` and `ZONE_DMA32` historically represented memory suitable for DMA by peripheral devices that cannot access all of the addressable memory. For many years there are better more and robust interfaces to get memory with DMA specific requirements (Documentation/core-api/dma-api.rst), but `ZONE_DMA` and `ZONE_DMA32` still represent memory ranges that have restrictions on how they can be accessed. Depending on the architecture, either of these zone types or even they both can be disabled at build time using `CONFIG_ZONE_DMA` and `CONFIG_ZONE_DMA32` configuration options. Some 64-bit platforms

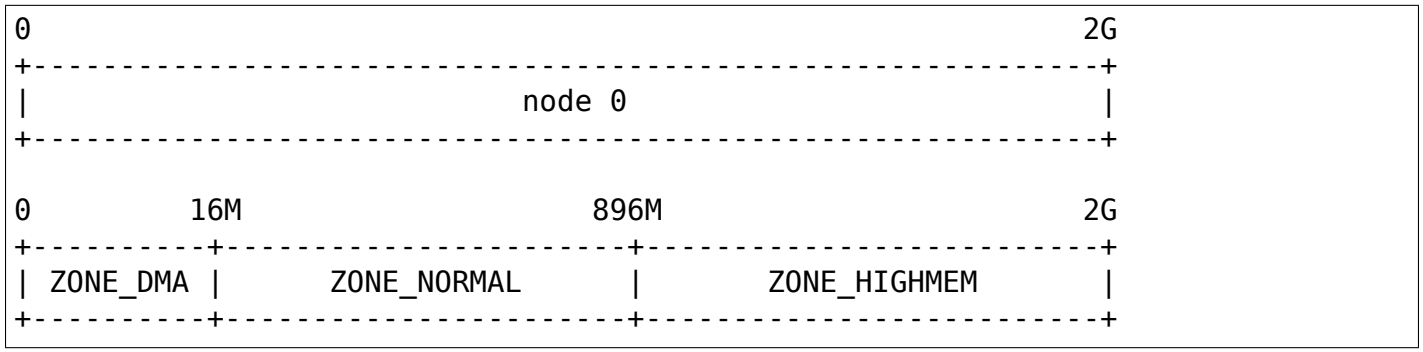
may need both zones as they support peripherals with different DMA addressing limitations.

- ZONE_NORMAL is for normal memory that can be accessed by the kernel all the time. DMA operations can be performed on pages in this zone if the DMA devices support transfers to all addressable memory. ZONE_NORMAL is always enabled.
- ZONE_HIGHMEM is the part of the physical memory that is not covered by a permanent mapping in the kernel page tables. The memory in this zone is only accessible to the kernel using temporary mappings. This zone is available only on some 32-bit architectures and is enabled with CONFIG_HIGHMEM.
- ZONE_MOVABLE is for normal accessible memory, just like ZONE_NORMAL. The difference is that the contents of most pages in ZONE_MOVABLE is movable. That means that while virtual addresses of these pages do not change, their content may move between different physical pages. Often ZONE_MOVABLE is populated during memory hotplug, but it may be also populated on boot using one of kernelcore, movablecore and movable_node kernel command line parameters. See *Page migration* and Documentation/admin-guide/mm/memory-hotplug.rst for additional details.
- ZONE_DEVICE represents memory residing on devices such as PMEM and GPU. It has different characteristics than RAM zone types and it exists to provide *struct page* and memory map services for device driver identified physical address ranges. ZONE_DEVICE is enabled with configuration option CONFIG_ZONE_DEVICE.

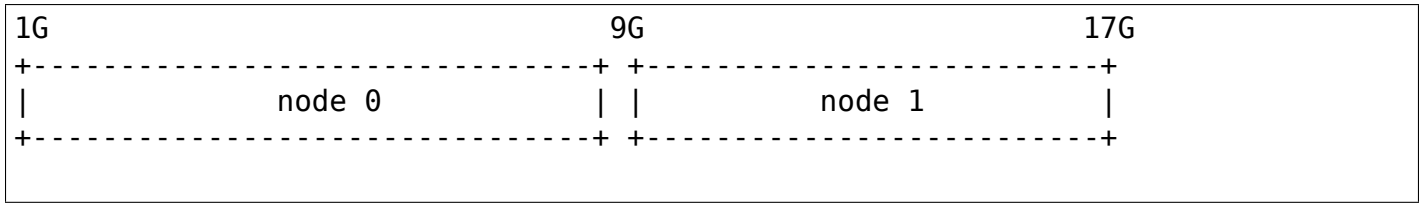
It is important to note that many kernel operations can only take place using ZONE_NORMAL so it is the most performance critical zone. Zones are discussed further in Section *Zones*.

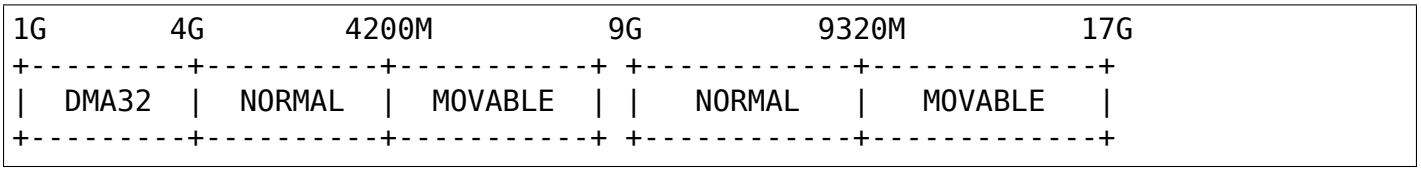
The relation between node and zone extents is determined by the physical memory map reported by the firmware, architectural constraints for memory addressing and certain parameters in the kernel command line.

For example, with 32-bit kernel on an x86 UMA machine with 2 Gbytes of RAM the entire memory will be on node 0 and there will be three zones: ZONE_DMA, ZONE_NORMAL and ZONE_HIGHMEM:

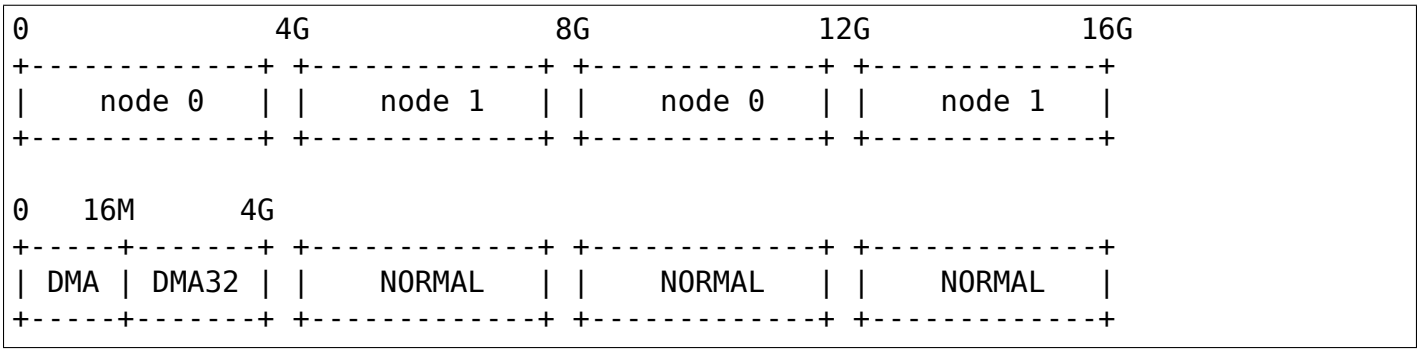


With a kernel built with ZONE_DMA disabled and ZONE_DMA32 enabled and booted with movablecore=80% parameter on an arm64 machine with 16 Gbytes of RAM equally split between two nodes, there will be ZONE_DMA32, ZONE_NORMAL and ZONE_MOVABLE on node 0, and ZONE_NORMAL and ZONE_MOVABLE on node 1:





Memory banks may belong to interleaving nodes. In the example below an x86 machine has 16 Gbytes of RAM in 4 memory banks, even banks belong to node 0 and odd banks belong to node 1:



In this case node 0 will span from 0 to 12 Gbytes and node 1 will span from 4 to 16 Gbytes.

1.1.1 Nodes

As we have mentioned, each node in memory is described by a `pg_data_t` which is a typedef for a `struct pglist_data`. When allocating a page, by default Linux uses a node-local allocation policy to allocate memory from the node closest to the running CPU. As processes tend to run on the same CPU, it is likely the memory from the current node will be used. The allocation policy can be controlled by users as described in Documentation/admin-guide/mm/numa_memory_policy.rst.

Most NUMA architectures maintain an array of pointers to the node structures. The actual structures are allocated early during boot when architecture specific code parses the physical memory map reported by the firmware. The bulk of the node initialization happens slightly later in the boot process by free_area_init() function, described later in Section *Initialization*.

Along with the node structures, kernel maintains an array of `nodemask_t` bitmasks called `node_states`. Each bitmask in this array represents a set of nodes with particular properties as defined by `enum node_states`:

N_POSSIBLE
: The node could become online at some point.

N_ONLINE
: The node is online.

N_NORMAL_MEMORY
: The node has regular memory.

N_HIGH_MEMORY
: The node has regular or high memory. When `CONFIG_HIGHMEM` is disabled aliased to `N_NORMAL_MEMORY`.

N_MEMORY
: The node has memory(regular, high, movable)

N_CPU
: The node has one or more CPUs

For each node that has a property described above, the bit corresponding to the node ID in the `node_states[<property>]` bitmask is set.

For example, for node 2 with normal memory and CPUs, bit 2 will be set in

```
node_states[N_POSSIBLE]
node_states[N_ONLINE]
node_states[N_NORMAL_MEMORY]
node_states[N_HIGH_MEMORY]
node_states[N_MEMORY]
node_states[N_CPU]
```

For various operations possible with nodemasks please refer to `include/linux/nodemask.h`.

Among other things, nodemasks are used to provide macros for node traversal, namely `for_each_node()` and `for_each_online_node()`.

For instance, to call a function foo() for each online node:

```
for_each_online_node(nid) {
        pg_data_t *pgdat = NODE_DATA(nid);

        foo(pgdat);
}
```

Node structure

The nodes structure `struct pglist_data` is declared in `include/linux/mmzone.h`. Here we briefly describe fields of this structure:

General

node_zones
: The zones for this node. Not all of the zones may be populated, but it is the full list. It is referenced by this node's node_zonelists as well as other node's node_zonelists.

node_zonelists
: The list of all zones in all nodes. This list defines the order of zones that allocations are preferred from. The `node_zonelists` is set up by `build_zonelists()` in `mm/page_alloc.c` during the initialization of core memory management structures.

nr_zones
: Number of populated zones in this node.

node_mem_map
: For UMA systems that use FLATMEM memory model the 0's node `node_mem_map` is array of struct pages representing each physical frame.

node_page_ext
: For UMA systems that use FLATMEM memory model the 0's node `node_page_ext`

is array of extensions of struct pages. Available only in the kernels built with `CONFIG_PAGE_EXTENSION` enabled.

node_start_pfn
The page frame number of the starting page frame in this node.

node_present_pages
Total number of physical pages present in this node.

node_spanned_pages
Total size of physical page range, including holes.

node_size_lock
A lock that protects the fields defining the node extents. Only defined when at least one of `CONFIG_MEMORY_HOTPLUG` or `CONFIG_DEFERRED_STRUCT_PAGE_INIT` configuration options are enabled. `pgdat_resize_lock()` and `pgdat_resize_unlock()` are provided to manipulate `node_size_lock` without checking for `CONFIG_MEMORY_HOTPLUG` or `CONFIG_DEFERRED_STRUCT_PAGE_INIT`.

node_id
The Node ID (NID) of the node, starts at 0.

totalreserve_pages
This is a per-node reserve of pages that are not available to userspace allocations.

first_deferred_pfn
If memory initialization on large machines is deferred then this is the first PFN that needs to be initialized. Defined only when `CONFIG_DEFERRED_STRUCT_PAGE_INIT` is enabled

deferred_split_queue
Per-node queue of huge pages that their split was deferred. Defined only when `CONFIG_TRANSPARENT_HUGEPAGE` is enabled.

__lruvec
Per-node lruvec holding LRU lists and related parameters. Used only when memory cgroups are disabled. It should not be accessed directly, use `mem_cgroup_lruvec()` to look up lruvecs instead.

Reclaim control

See also *Page Reclaim*.

kswapd
Per-node instance of kswapd kernel thread.

kswapd_wait, pfmemalloc_wait, reclaim_wait
Workqueues used to synchronize memory reclaim tasks

nr_writeback_throttled
Number of tasks that are throttled waiting on dirty pages to clean.

nr_reclaim_start
Number of pages written while reclaim is throttled waiting for writeback.

kswapd_order
Controls the order kswapd tries to reclaim

kswapd_highest_zoneidx
The highest zone index to be reclaimed by kswapd

kswapd_failures
Number of runs kswapd was unable to reclaim any pages

min_unmapped_pages
Minimal number of unmapped file backed pages that cannot be reclaimed. Determined by `vm.min_unmapped_ratio` sysctl. Only defined when `CONFIG_NUMA` is enabled.

min_slab_pages
Minimal number of SLAB pages that cannot be reclaimed. Determined by `vm.min_slab_ratio sysctl`. Only defined when `CONFIG_NUMA` is enabled

flags
Flags controlling reclaim behavior.

Compaction control

kcompactd_max_order
Page order that kcompactd should try to achieve.

kcompactd_highest_zoneidx
The highest zone index to be compacted by kcompactd.

kcompactd_wait
Workqueue used to synchronize memory compaction tasks.

kcompactd
Per-node instance of kcompactd kernel thread.

proactive_compact_trigger
Determines if proactive compaction is enabled. Controlled by `vm.compaction_proactiveness` sysctl.

Statistics

per_cpu_nodestats
Per-CPU VM statistics for the node

vm_stat
VM statistics for the node.

1.1.2 Zones

Stub

This section is incomplete. Please list and describe the appropriate fields.

1.1.3 Pages

Stub

This section is incomplete. Please list and describe the appropriate fields.

1.1.4 Folios

Stub

This section is incomplete. Please list and describe the appropriate fields.

1.1.5 Initialization

Stub

This section is incomplete. Please list and describe the appropriate fields.

1.2 Page Tables

Paged virtual memory was invented along with virtual memory as a concept in 1962 on the Ferranti Atlas Computer which was the first computer with paged virtual memory. The feature migrated to newer computers and became a de facto feature of all Unix-like systems as time went by. In 1985 the feature was included in the Intel 80386, which was the CPU Linux 1.0 was developed on.

Page tables map virtual addresses as seen by the CPU into physical addresses as seen on the external memory bus.

Linux defines page tables as a hierarchy which is currently five levels in height. The architecture code for each supported architecture will then map this to the restrictions of the hardware.

The physical address corresponding to the virtual address is often referenced by the underlying physical page frame. The **page frame number** or **pfn** is the physical address of the page (as seen on the external memory bus) divided by *PAGE_SIZE*.

Physical memory address 0 will be *pfn 0* and the highest pfn will be the last page of physical memory the external address bus of the CPU can address.

With a page granularity of 4KB and a address range of 32 bits, pfn 0 is at address 0x00000000, pfn 1 is at address 0x00001000, pfn 2 is at 0x00002000 and so on until we reach pfn 0xfffff at 0xfffff000. With 16KB pages pfs are at 0x00004000, 0x00008000 ... 0xffffc000 and pfn goes from 0 to 0x3fffff.

As you can see, with 4KB pages the page base address uses bits 12-31 of the address, and this is why *PAGE_SHIFT* in this case is defined as 12 and *PAGE_SIZE* is usually defined in terms of the page shift as *(1 << PAGE_SHIFT)*

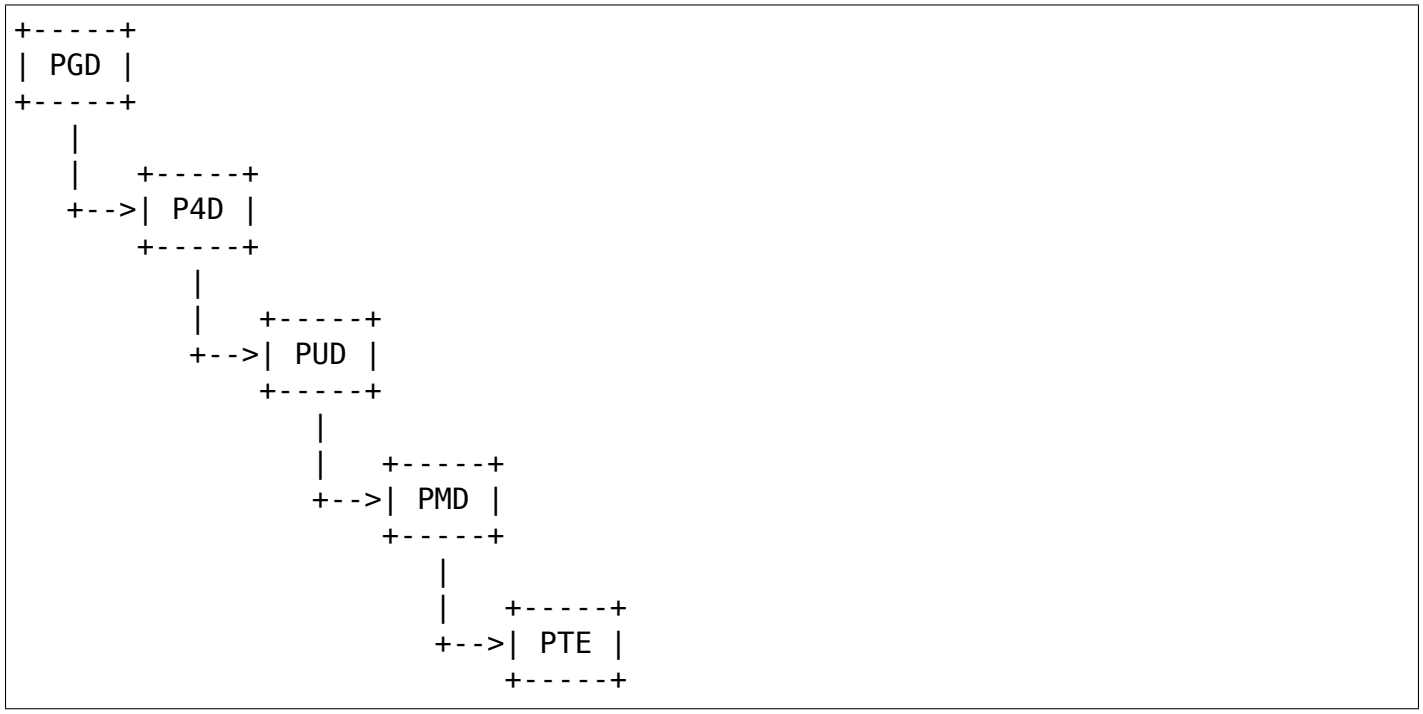
Over time a deeper hierarchy has been developed in response to increasing memory sizes. When Linux was created, 4KB pages and a single page table called *swapper_pg_dir* with 1024 entries was used, covering 4MB which coincided with the fact that Torvald's first computer had 4MB of physical memory. Entries in this single table were referred to as *PTE*:s - page table entries.

The software page table hierarchy reflects the fact that page table hardware has become hierarchical and that in turn is done to save page table memory and speed up mapping.

One could of course imagine a single, linear page table with enormous amounts of entries, breaking down the whole memory into single pages. Such a page table would be very sparse, because large portions of the virtual memory usually remains unused. By using hierarchical page tables large holes in the virtual address space does not waste valuable page table memory, because it will suffice to mark large areas as unmapped at a higher level in the page table hierarchy.

Additionally, on modern CPUs, a higher level page table entry can point directly to a physical memory range, which allows mapping a contiguous range of several megabytes or even gigabytes in a single high-level page table entry, taking shortcuts in mapping virtual memory to physical memory: there is no need to traverse deeper in the hierarchy when you find a large mapped range like this.

The page table hierarchy has now developed into this:



Symbols on the different levels of the page table hierarchy have the following meaning beginning from the bottom:

- **pte**, *pte_t*, *pteval_t* = **Page Table Entry** - mentioned earlier. The *pte* is an array of *PTRS_PER_PTE* elements of the *pteval_t* type, each mapping a single page of virtual memory to a single page of physical memory. The architecture defines the size and contents of *pteval_t*.

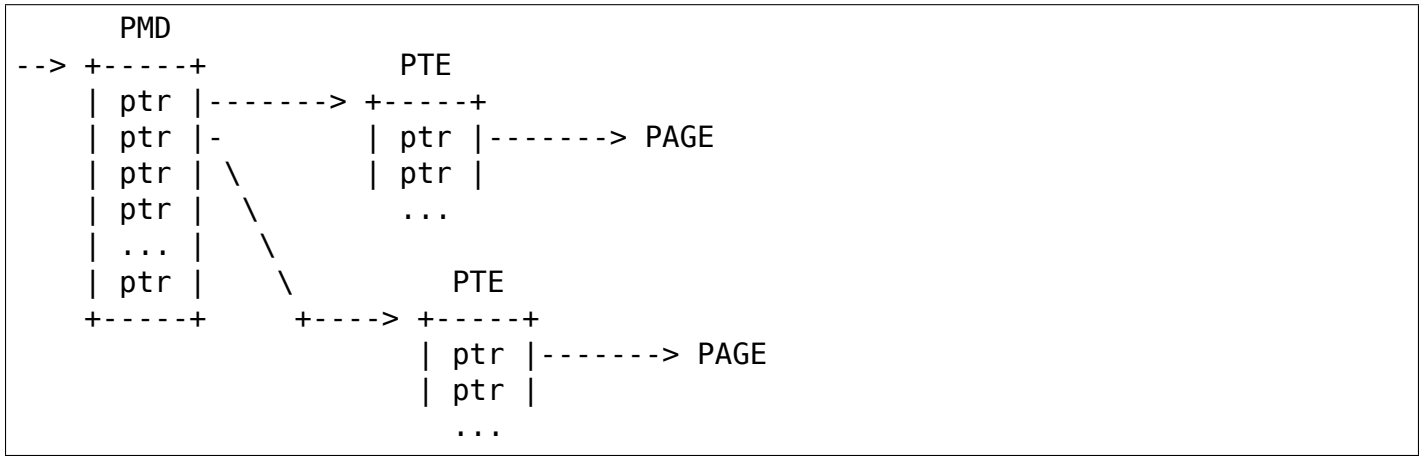
  A typical example is that the *pteval_t* is a 32- or 64-bit value with the upper bits being a **pfn** (page frame number), and the lower bits being some architecture-specific bits such as

memory protection.

The **entry** part of the name is a bit confusing because while in Linux 1.0 this did refer to a single page table entry in the single top level page table, it was retrofitted to be an array of mapping elements when two-level page tables were first introduced, so the *pte* is the lowermost page *table*, not a page table *entry*.

- **pmd**, *pmd_t*, *pmdval_t* = **Page Middle Directory**, the hierarchy right above the *pte*, with *PTRS_PER_PMD* references to the *pte*:s.
- **pud**, *pud_t*, *pudval_t* = **Page Upper Directory** was introduced after the other levels to handle 4-level page tables. It is potentially unused, or *folded* as we will discuss later.
- **p4d**, *p4d_t*, *p4dval_t* = **Page Level 4 Directory** was introduced to handle 5-level page tables after the *pud* was introduced. Now it was clear that we needed to replace *pgd*, *pmd*, *pud* etc with a figure indicating the directory level and that we cannot go on with ad hoc names any more. This is only used on systems which actually have 5 levels of page tables, otherwise it is folded.
- **pgd**, *pgd_t*, *pgdval_t* = **Page Global Directory** - the Linux kernel main page table handling the PGD for the kernel memory is still found in *swapper_pg_dir*, but each userspace process in the system also has its own memory context and thus its own *pgd*, found in *struct mm_struct* which in turn is referenced to in each *struct task_struct*. So tasks have memory context in the form of a *struct mm_struct* and this in turn has a *struct pgt_t *pgd* pointer to the corresponding page global directory.

To repeat: each level in the page table hierarchy is a *array of pointers*, so the **pgd** contains *PTRS_PER_PGD* pointers to the next level below, **p4d** contains *PTRS_PER_P4D* pointers to **pud** items and so on. The number of pointers on each level is architecture-defined.:



1.2.1 Page Table Folding

If the architecture does not use all the page table levels, they can be *folded* which means skipped, and all operations performed on page tables will be compile-time augmented to just skip a level when accessing the next lower level.

Page table handling code that wishes to be architecture-neutral, such as the virtual memory manager, will need to be written so that it traverses all of the currently five levels. This style should also be preferred for architecture-specific code, so as to be robust to future changes.

1.2.2 MMU, TLB, and Page Faults

The *Memory Management Unit (MMU)* is a hardware component that handles virtual to physical address translations. It may use relatively small caches in hardware called *Translation Lookaside Buffers (TLBs)* and *Page Walk Caches* to speed up these translations.

When CPU accesses a memory location, it provides a virtual address to the MMU, which checks if there is the existing translation in the TLB or in the Page Walk Caches (on architectures that support them). If no translation is found, MMU uses the page walks to determine the physical address and create the map.

The dirty bit for a page is set (i.e., turned on) when the page is written to. Each page of memory has associated permission and dirty bits. The latter indicate that the page has been modified since it was loaded into memory.

If nothing prevents it, eventually the physical memory can be accessed and the requested operation on the physical frame is performed.

There are several reasons why the MMU can't find certain translations. It could happen because the CPU is trying to access memory that the current task is not permitted to, or because the data is not present into physical memory.

When these conditions happen, the MMU triggers page faults, which are types of exceptions that signal the CPU to pause the current execution and run a special function to handle the mentioned exceptions.

There are common and expected causes of page faults. These are triggered by process management optimization techniques called ”Lazy Allocation” and ”Copy-on-Write”. Page faults may also happen when frames have been swapped out to persistent storage (swap partition or file) and evicted from their physical locations.

These techniques improve memory efficiency, reduce latency, and minimize space occupation. This document won't go deeper into the details of ”Lazy Allocation” and ”Copy-on-Write” because these subjects are out of scope as they belong to Process Address Management.

Swapping differentiates itself from the other mentioned techniques because it's undesirable since it's performed as a means to reduce memory under heavy pressure.

Swapping can't work for memory mapped by kernel logical addresses. These are a subset of the kernel virtual space that directly maps a contiguous range of physical memory. Given any logical address, its physical address is determined with simple arithmetic on an offset. Accesses to logical addresses are fast because they avoid the need for complex page table lookups at the expenses of frames not being evictable and pageable out.

If the kernel fails to make room for the data that must be present in the physical frames, the kernel invokes the out-of-memory (OOM) killer to make room by terminating lower priority processes until pressure reduces under a safe threshold.

Additionally, page faults may be also caused by code bugs or by maliciously crafted addresses that the CPU is instructed to access. A thread of a process could use instructions to address (non-shared) memory which does not belong to its own address space, or could try to execute an instruction that want to write to a read-only location.

If the above-mentioned conditions happen in user-space, the kernel sends a *Segmentation Fault* (SIGSEGV) signal to the current thread. That signal usually causes the termination of the thread and of the process it belongs to.

This document is going to simplify and show an high altitude view of how the Linux kernel handles these page faults, creates tables and tables' entries, check if memory is present and, if not, requests to load data from persistent storage or from other devices, and updates the MMU and its caches.

The first steps are architecture dependent. Most architectures jump to *do_page_fault()*, whereas the x86 interrupt handler is defined by the *DEFINE_IDTENTRY_RAW_ERRORCODE()* macro which calls *handle_page_fault()*.

Whatever the routes, all architectures end up to the invocation of *handle_mm_fault()* which, in turn, (likely) ends up calling *__handle_mm_fault()* to carry out the actual work of allocating the page tables.

The unfortunate case of not being able to call *__handle_mm_fault()* means that the virtual address is pointing to areas of physical memory which are not permitted to be accessed (at least from the current context). This condition resolves to the kernel sending the above-mentioned SIGSEGV signal to the process and leads to the consequences already explained.

__handle_mm_fault() carries out its work by calling several functions to find the entry's offsets of the upper layers of the page tables and allocate the tables that it may need.

The functions that look for the offset have names like **_offset()*, where the ”*” is for pgd, p4d, pud, pmd, pte; instead the functions to allocate the corresponding tables, layer by layer, are called **_alloc*, using the above-mentioned convention to name them after the corresponding types of tables in the hierarchy.

The page table walk may end at one of the middle or upper layers (PMD, PUD).

Linux supports larger page sizes than the usual 4KB (i.e., the so called *huge pages*). When using these kinds of larger pages, higher level pages can directly map them, with no need to use lower level page entries (PTE). Huge pages contain large contiguous physical regions that usually span from 2MB to 1GB. They are respectively mapped by the PMD and PUD page entries.

The huge pages bring with them several benefits like reduced TLB pressure, reduced page table overhead, memory allocation efficiency, and performance improvement for certain workloads. However, these benefits come with trade-offs, like wasted memory and allocation challenges.

At the very end of the walk with allocations, if it didn't return errors, *__handle_mm_fault()* finally calls *handle_pte_fault()*, which via *do_fault()* performs one of *do_read_fault()*, *do_cow_fault()*, *do_shared_fault()*. ”read”, ”cow”, ”shared” give hints about the reasons and the kind of fault it's handling.

The actual implementation of the workflow is very complex. Its design allows Linux to handle page faults in a way that is tailored to the specific characteristics of each architecture, while still sharing a common overall structure.

To conclude this high altitude view of how Linux handles page faults, let's add that the page faults handler can be disabled and enabled respectively with *pagefault_disable()* and *pagefault_enable()*.

Several code path make use of the latter two functions because they need to disable traps into the page faults handler, mostly to prevent deadlocks.

1.3 Process Addresses

1.4 Boot Memory

1.5 Page Allocation

1.6 Virtually Contiguous Memory Allocation

1.7 Slab Allocation

1.8 High Memory Handling

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

1.8.1 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.

1.8.2 Temporary Virtual Mappings

The kernel contains several ways of creating temporary mappings. The following list shows them in order of preference of use.

- kmap_local_page(), *kmap_local_folio()* - These functions are used to create short term mappings. They can be invoked from any context (including interrupts) but the mappings can only be used in the context which acquired them. The only differences between them consist in the first taking a pointer to a struct page and the second taking a pointer to struct folio and the byte offset within the folio which identifies the page.

  These functions should always be used, whereas kmap_atomic() and kmap() have been deprecated.

  These mappings are thread-local and CPU-local, meaning that the mapping can only be accessed from within this thread and the thread is bound to the CPU while the mapping is active. Although preemption is never disabled by this function, the CPU can not be unplugged from the system via CPU-hotplug until the mapping is disposed.

  It's valid to take pagefaults in a local kmap region, unless the context in which the local mapping is acquired does not allow it for other reasons.

  As said, pagefaults and preemption are never disabled. There is no need to disable preemption because, when context switches to a different task, the maps of the outgoing task are saved and those of the incoming one are restored.

  kmap_local_page(), as well as *kmap_local_folio()* always returns valid virtual kernel addresses and it is assumed that kunmap_local() will never fail.

  On CONFIG_HIGHMEM=n kernels and for low memory pages they return the virtual address of the direct mapping. Only real highmem pages are temporarily mapped. Therefore, users may call a plain page_address() for pages which are known to not come from ZONE_HIGHMEM. However, it is always safe to use kmap_local_{page,folio}() / kunmap_local().

  While they are significantly faster than kmap(), for the highmem case they come with restrictions about the pointers validity. Contrary to kmap() mappings, the local mappings are only valid in the context of the caller and cannot be handed to other contexts. This implies that users must be absolutely sure to keep the use of the return address local to the thread which mapped it.

  Most code can be designed to use thread local mappings. User should therefore try to design their code to avoid the use of kmap() by mapping pages in the same thread the address will be used and prefer kmap_local_page() or *kmap_local_folio()*.

  Nesting kmap_local_page() and kmap_atomic() mappings is allowed to a certain extent (up to KMAP_TYPE_NR) but their invocations have to be strictly ordered because the map

implementation is stack based. See kmap_local_page() kdocs (included in the "Functions" section) for details on how to manage nested mappings.

- kmap_atomic(). This function has been deprecated; use kmap_local_page().

  NOTE: Conversions to kmap_local_page() must take care to follow the mapping restrictions imposed on kmap_local_page(). Furthermore, the code between calls to kmap_atomic() and kunmap_atomic() may implicitly depend on the side effects of atomic mappings, i.e. disabling page faults or preemption, or both. In that case, explicit calls to pagefault_disable() or preempt_disable() or both must be made in conjunction with the use of kmap_local_page().

  [Legacy documentation]

  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 does not sleep and the callers too may not sleep until after kunmap_atomic() is called.

  Each call of kmap_atomic() in the kernel creates a non-preemptible section and disable pagefaults. This could be a source of unwanted latency. Therefore users should prefer kmap_local_page() instead of kmap_atomic().

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

- kmap(). This function has been deprecated; use kmap_local_page().

  NOTE: Conversions to kmap_local_page() must take care to follow the mapping restrictions imposed on kmap_local_page(). In particular, it is necessary to make sure that the kernel virtual memory pointer is only valid in the thread that obtained it.

  [Legacy documentation]

  This should be used to make short duration mapping of a single page with no restrictions on preemption or migration. It comes with an overhead as mapping space is restricted and protected by a global lock for synchronization. When mapping is no longer needed, the address that the page was mapped to must be released with kunmap().

  Mapping changes must be propagated across all the CPUs. kmap() also requires global TLB invalidation when the kmap's pool wraps and it might block when the mapping space is fully utilized until a slot becomes available. Therefore, kmap() is only callable from preemptible context.

  All the above work is necessary if a mapping must last for a relatively long time but the bulk of high-memory mappings in the kernel are short-lived and only used in one place. This means that the cost of kmap() is mostly wasted in such cases. kmap() was not intended for long term mappings but it has morphed in that direction and its use is strongly discouraged in newer code and the set of the preceding functions should be preferred.

  On 64-bit systems, calls to kmap_local_page(), kmap_atomic() and kmap() have no real work to do because a 64-bit address space is more than sufficient to address all the physical memory whose pages are permanently mapped.

- 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.

1.8.3 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.

1.8.4 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 pageframes 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.

1.8.5 Functions

void *`kmap`(struct *page* *page)

Map a page for long term usage

Parameters

`struct page *page`
Pointer to the page to be mapped

Return

The virtual address of the mapping

Description

Can only be invoked from preemptible task context because on 32bit systems with CONFIG_HIGHMEM enabled this function might sleep.

For systems with CONFIG_HIGHMEM=n and for pages in the low memory area this returns the virtual address of the direct kernel mapping.

The returned virtual address is globally visible and valid up to the point where it is unmapped via kunmap(). The pointer can be handed to other contexts.

For highmem pages on 32bit systems this can be slow as the mapping space is limited and protected by a global lock. In case that there is no mapping slot available the function blocks until a slot is released via kunmap().

void **kunmap**(struct *page* *page)

Unmap the virtual address mapped by kmap()

Parameters

struct page *page

Pointer to the page which was mapped by kmap()

Description

Counterpart to kmap(). A NOOP for CONFIG_HIGHMEM=n and for mappings of pages in the low memory area.

struct page ***kmap_to_page**(void *addr)

Get the page for a kmap'ed address

Parameters

void *addr

The address to look up

Return

The page which is mapped to **addr**.

void **kmap_flush_unused**(void)

Flush all unused kmap mappings in order to remove stray mappings

Parameters

void

no arguments

void ***kmap_local_page**(struct *page* *page)

Map a page for temporary usage

Parameters

struct page *page

Pointer to the page to be mapped

Return

The virtual address of the mapping

Description

Can be invoked from any context, including interrupts.

Requires careful handling when nesting multiple mappings because the map management is stack based. The unmap has to be in the reverse order of the map operation:

addr1 = kmap_local_page(page1); addr2 = kmap_local_page(page2); … kunmap_local(addr2); kunmap_local(addr1);

Unmapping addr1 before addr2 is invalid and causes malfunction.

Contrary to kmap() mappings the mapping is only valid in the context of the caller and cannot be handed to other contexts.

On CONFIG_HIGHMEM=n kernels and for low memory pages this returns the virtual address of the direct mapping. Only real highmem pages are temporarily mapped.

While kmap_local_page() is significantly faster than kmap() for the highmem case it comes with restrictions about the pointer validity.

On HIGHMEM enabled systems mapping a highmem page has the side effect of disabling migration in order to keep the virtual address stable across preemption. No caller of kmap_local_page() can rely on this side effect.

void *__kmap_local_folio__(struct *folio* *folio, size_t offset)

Map a page in this folio for temporary usage

Parameters

struct folio *folio

The folio containing the page.

size_t offset

The byte offset within the folio which identifies the page.

Description

Requires careful handling when nesting multiple mappings because the map management is stack based. The unmap has to be in the reverse order of the map operation:

```
addr1 = kmap_local_folio(folio1, offset1);
addr2 = kmap_local_folio(folio2, offset2);
...
kunmap_local(addr2);
kunmap_local(addr1);
```

Unmapping addr1 before addr2 is invalid and causes malfunction.

Contrary to kmap() mappings the mapping is only valid in the context of the caller and cannot be handed to other contexts.

On CONFIG_HIGHMEM=n kernels and for low memory pages this returns the virtual address of the direct mapping. Only real highmem pages are temporarily mapped.

While it is significantly faster than kmap() for the highmem case it comes with restrictions about the pointer validity.

On HIGHMEM enabled systems mapping a highmem page has the side effect of disabling migration in order to keep the virtual address stable across preemption. No caller of *kmap_local_folio()* can rely on this side effect.

Context

Can be invoked from any context.

Return

The virtual address of **offset**.

void *__kmap_atomic__(struct *page* *page)

Atomically map a page for temporary usage - Deprecated!

Parameters

struct page *page
Pointer to the page to be mapped

Return

The virtual address of the mapping

Description

In fact a wrapper around kmap_local_page() which also disables pagefaults and, depending on PREEMPT_RT configuration, also CPU migration and preemption. Therefore users should not count on the latter two side effects.

Mappings should always be released by kunmap_atomic().

Do not use in new code. Use kmap_local_page() instead.

It is used in atomic context 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);

struct folio ***vma_alloc_zeroed_movable_folio**(struct vm_area_struct *vma, unsigned long vaddr)

Allocate a zeroed page for a VMA.

Parameters

struct vm_area_struct *vma
The VMA the page is to be allocated for.

unsigned long vaddr
The virtual address the page will be inserted into.

Description

This function will allocate a page suitable for inserting into this VMA at this virtual address. It may be allocated from highmem or the movable zone. An architecture may provide its own implementation.

Return

A folio containing one allocated and zeroed page or NULL if we are out of memory.

void *`folio_zero_tail`(struct *folio* *folio, size_t offset, void *kaddr)

Zero the tail of a folio.

Parameters

`struct folio *folio`
The folio to zero.

`size_t offset`
The byte offset in the folio to start zeroing at.

`void *kaddr`
The address the folio is currently mapped to.

Description

If you have already used *kmap_local_folio()* to map a folio, written some data to it and now need to zero the end of the folio (and flush the dcache), you can use this function. If you do not have the folio kmapped (eg the folio has been partially populated by DMA), use *folio_zero_range()* or *folio_zero_segment()* instead.

Return

An address which can be passed to kunmap_local().

void `folio_fill_tail`(struct *folio* *folio, size_t offset, const char *from, size_t len)

Copy some data to a folio and pad with zeroes.

Parameters

`struct folio *folio`
The destination folio.

`size_t offset`
The offset into **folio** at which to start copying.

`const char *from`
The data to copy.

`size_t len`
How many bytes of data to copy.

Description

This function is most useful for filesystems which support inline data. When they want to copy data from the inode into the page cache, this function does everything for them. It supports large folios even on HIGHMEM configurations.

size_t `memcpy_from_file_folio`(char *to, struct *folio* *folio, loff_t pos, size_t len)

Copy some bytes from a file folio.

Parameters

char *to
The destination buffer.

struct folio *folio
The folio to copy from.

loff_t pos
The position in the file.

size_t len
The maximum number of bytes to copy.

Description

Copy up to **len** bytes from this folio. This may be limited by PAGE_SIZE if the folio comes from HIGHMEM, and by the size of the folio.

Return

The number of bytes copied from the folio.

void **folio_zero_segments**(struct *folio* *folio, size_t start1, size_t xend1, size_t start2, size_t xend2)
Zero two byte ranges in a folio.

Parameters

struct folio *folio
The folio to write to.

size_t start1
The first byte to zero.

size_t xend1
One more than the last byte in the first range.

size_t start2
The first byte to zero in the second range.

size_t xend2
One more than the last byte in the second range.

void **folio_zero_segment**(struct *folio* *folio, size_t start, size_t xend)
Zero a byte range in a folio.

Parameters

struct folio *folio
The folio to write to.

size_t start
The first byte to zero.

size_t xend
One more than the last byte to zero.

void **folio_zero_range**(struct *folio* *folio, size_t start, size_t length)
Zero a byte range in a folio.

Parameters

struct folio *folio
The folio to write to.

size_t start
The first byte to zero.

size_t length
The number of bytes to zero.

void **folio_release_kmap**(struct *folio* *folio, void *addr)
Unmap a folio and drop a refcount.

Parameters

struct folio *folio
The folio to release.

void *addr
The address previously returned by a call to *kmap_local_folio()*.

Description

It is common, eg in directory handling to kmap a folio. This function unmaps the folio and drops the refcount that was being held to keep the folio alive while we accessed it.

void ***kmap_high**(struct *page* *page)
map a highmem page into memory

Parameters

struct page *page
`struct page` to map

Description

Returns the page's virtual memory address.

We cannot call this from interrupts, as it may block.

void ***kmap_high_get**(struct *page* *page)
pin a highmem page into memory

Parameters

struct page *page
`struct page` to pin

Description

Returns the page's current virtual memory address, or NULL if no mapping exists. If and only if a non null address is returned then a matching call to *kunmap_high()* is necessary.

This can be called from any context.

void **kunmap_high**(struct *page* *page)
unmap a highmem page into memory

Parameters

struct page *page
`struct page` to unmap

Description

If ARCH_NEEDS_KMAP_HIGH_GET is not defined then this may be called only from user context.

void ***page_address**(const struct *page* *page)
get the mapped virtual address of a page

Parameters

const struct page *page
`struct page` to get the virtual address of

Description

Returns the page's virtual address.

void **set_page_address**(struct *page* *page, void *virtual)
set a page's virtual address

Parameters

struct page *page
`struct page` to set

void *virtual
virtual address to use

kunmap_atomic

`kunmap_atomic (__addr)`

Unmap the virtual address mapped by kmap_atomic() - deprecated!

Parameters

__addr
Virtual address to be unmapped

Description

Unmaps an address previously mapped by kmap_atomic() and re-enables pagefaults. Depending on PREEMP_RT configuration, re-enables also migration and preemption. Users should not count on these side effects.

Mappings should be unmapped in the reverse order that they were mapped. See kmap_local_page() for details on nesting.

__addr can be any address within the mapped page, so there is no need to subtract any offset that has been added. In contrast to kunmap(), this function takes the address returned from kmap_atomic(), not the page passed to it. The compiler will warn you if you pass the page.

kunmap_local

`kunmap_local (__addr)`

Unmap a page mapped via kmap_local_page().

Parameters

__addr
An address within the page mapped

Description

__addr can be any address within the mapped page. Commonly it is the address return from kmap_local_page(), but it can also include offsets.

Unmapping should be done in the reverse order of the mapping. See kmap_local_page() for details.

1.9 Page Reclaim

1.10 Swap

1.11 Page Cache

1.12 Shared Memory Filesystem

1.13 Out Of Memory Handling

CHAPTER

TWO

LEGACY DOCUMENTATION

This is a collection of older documents about the Linux memory management (MM) subsystem internals with different level of details ranging from notes and mailing list responses for elaborating descriptions of data structures and algorithms. It should all be integrated nicely into the above structured documentation, or deleted if it has served its purpose.

2.1 Active MM

Note, the mm_count refcount may no longer include the ”lazy” users (running tasks with ->active_mm == mm && ->mm == NULL) on kernels with CONFIG_MMU_LAZY_TLB_REFCOUNT=n. Taking and releasing these lazy references must be done with mmgrab_lazy_tlb() and mmdrop_lazy_tlb() helpers, which abstract this config option.

```
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.

 - 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_count" 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 Architecture Page Table Helpers

Generic MM expects architectures (with MMU) to provide helpers to create, access and modify page table entries at various level for different memory functions. These page table helpers need to conform to a common semantics across platforms. Following tables describe the expected semantics which can also be tested during boot via CONFIG_DEBUG_VM_PGTABLE option. All future changes in here or the debug test need to be in sync.

2.2.1 PTE Page Table Helpers

pte_same	Tests whether both PTE entries are the same
pte_present	Tests a valid mapped PTE
pte_young	Tests a young PTE
pte_dirty	Tests a dirty PTE
pte_write	Tests a writable PTE
pte_special	Tests a special PTE
pte_protnone	Tests a PROT_NONE PTE
pte_devmap	Tests a ZONE_DEVICE mapped PTE
pte_soft_dirty	Tests a soft dirty PTE
pte_swp_soft_dirty	Tests a soft dirty swapped PTE
pte_mkyoung	Creates a young PTE
pte_mkold	Creates an old PTE
pte_mkdirty	Creates a dirty PTE
pte_mkclean	Creates a clean PTE
pte_mkwrite	Creates a writable PTE of the type specified by the VMA.
pte_mkwrite_novma	Creates a writable PTE, of the conventional type of writable.
pte_wrprotect	Creates a write protected PTE
pte_mkspecial	Creates a special PTE
pte_mkdevmap	Creates a ZONE_DEVICE mapped PTE
pte_mksoft_dirty	Creates a soft dirty PTE
pte_clear_soft_dirty	Clears a soft dirty PTE
pte_swp_mksoft_dirty	Creates a soft dirty swapped PTE
pte_swp_clear_soft_dirty	Clears a soft dirty swapped PTE
pte_mknotpresent	Invalidates a mapped PTE
ptep_clear	Clears a PTE
ptep_get_and_clear	Clears and returns PTE
ptep_get_and_clear_full	Clears and returns PTE (batched PTE unmap)
ptep_test_and_clear_young	Clears young from a PTE
ptep_set_wrprotect	Converts into a write protected PTE
ptep_set_access_flags	Converts into a more permissive PTE

2.2.2 PMD Page Table Helpers

pmd_same	Tests whether both PMD entries are the same
pmd_bad	Tests a non-table mapped PMD
pmd_leaf	Tests a leaf mapped PMD
pmd_huge	Tests a HugeTLB mapped PMD
pmd_trans_huge	Tests a Transparent Huge Page (THP) at PMD
pmd_present	Tests whether pmd_page() points to valid memory
pmd_young	Tests a young PMD
pmd_dirty	Tests a dirty PMD
pmd_write	Tests a writable PMD
pmd_special	Tests a special PMD
pmd_protnone	Tests a PROT_NONE PMD
pmd_devmap	Tests a ZONE_DEVICE mapped PMD

continues on next page

Table 1 – continued from previous page

pmd_soft_dirty	Tests a soft dirty PMD
pmd_swp_soft_dirty	Tests a soft dirty swapped PMD
pmd_mkyoung	Creates a young PMD
pmd_mkold	Creates an old PMD
pmd_mkdirty	Creates a dirty PMD
pmd_mkclean	Creates a clean PMD
pmd_mkwrite	Creates a writable PMD of the type specified by the VMA.
pmd_mkwrite_novma	Creates a writable PMD, of the conventional type of writable.
pmd_wrprotect	Creates a write protected PMD
pmd_mkspecial	Creates a special PMD
pmd_mkdevmap	Creates a ZONE_DEVICE mapped PMD
pmd_mksoft_dirty	Creates a soft dirty PMD
pmd_clear_soft_dirty	Clears a soft dirty PMD
pmd_swp_mksoft_dirty	Creates a soft dirty swapped PMD
pmd_swp_clear_soft_dirty	Clears a soft dirty swapped PMD
pmd_mkinvalid	Invalidates a mapped PMD [1]
pmd_set_huge	Creates a PMD huge mapping
pmd_clear_huge	Clears a PMD huge mapping
pmdp_get_and_clear	Clears a PMD
pmdp_get_and_clear_full	Clears a PMD
pmdp_test_and_clear_young	Clears young from a PMD
pmdp_set_wrprotect	Converts into a write protected PMD
pmdp_set_access_flags	Converts into a more permissive PMD

2.2.3 PUD Page Table Helpers

pud_same	Tests whether both PUD entries are the same
pud_bad	Tests a non-table mapped PUD
pud_leaf	Tests a leaf mapped PUD
pud_huge	Tests a HugeTLB mapped PUD
pud_trans_huge	Tests a Transparent Huge Page (THP) at PUD
pud_present	Tests a valid mapped PUD
pud_young	Tests a young PUD
pud_dirty	Tests a dirty PUD
pud_write	Tests a writable PUD
pud_devmap	Tests a ZONE_DEVICE mapped PUD
pud_mkyoung	Creates a young PUD
pud_mkold	Creates an old PUD
pud_mkdirty	Creates a dirty PUD
pud_mkclean	Creates a clean PUD
pud_mkwrite	Creates a writable PUD
pud_wrprotect	Creates a write protected PUD
pud_mkdevmap	Creates a ZONE_DEVICE mapped PUD
pud_mkinvalid	Invalidates a mapped PUD [1]
pud_set_huge	Creates a PUD huge mapping
pud_clear_huge	Clears a PUD huge mapping
pudp_get_and_clear	Clears a PUD
pudp_get_and_clear_full	Clears a PUD
pudp_test_and_clear_young	Clears young from a PUD
pudp_set_wrprotect	Converts into a write protected PUD
pudp_set_access_flags	Converts into a more permissive PUD

2.2.4 HugeTLB Page Table Helpers

pte_huge	Tests a HugeTLB
arch_make_huge_pte	Creates a HugeTLB
huge_pte_dirty	Tests a dirty HugeTLB
huge_pte_write	Tests a writable HugeTLB
huge_pte_mkdirty	Creates a dirty HugeTLB
huge_pte_mkwrite	Creates a writable HugeTLB
huge_pte_wrprotect	Creates a write protected HugeTLB
huge_ptep_get_and_clear	Clears a HugeTLB
huge_ptep_set_wrprotect	Converts into a write protected HugeTLB
huge_ptep_set_access_flags | Converts into a more permissive HugeTLB	

2.2.5 SWAP Page Table Helpers

__pte_to_swp_entry	Creates a swapped entry (arch) from a mapped PTE
__swp_to_pte_entry	Creates a mapped PTE from a swapped entry (arch)
__pmd_to_swp_entry	Creates a swapped entry (arch) from a mapped PMD
__swp_to_pmd_entry	Creates a mapped PMD from a swapped entry (arch)
is_migration_entry | Tests a migration (read or write) swapped entry	
is_writable_migration_entry	Tests a write migration swapped entry
make_readable_migration_entry	Creates a read migration swapped entry
make_writable_migration_entry	Creates a write migration swapped entry

[1] https://lore.kernel.org/linux-mm/20181017020930.GN30832@redhat.com/

2.3 Memory Balancing

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

Memory balancing is needed for !__GFP_HIGH 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)

2.4 DAMON: Data Access MONitor

DAMON is a Linux kernel subsystem that provides a framework for data access monitoring and the monitoring results based system operations. The core monitoring mechanisms of DAMON (refer to *Design* for the detail) make it

- *accurate* (the monitoring output is useful enough for DRAM level memory management; It might not appropriate for CPU Cache levels, though),
- *light-weight* (the monitoring overhead is low enough to be applied online), and
- *scalable* (the upper-bound of the overhead is in constant range regardless of the size of target workloads).

Using this framework, therefore, the kernel can operate system in an access-aware fashion. Because the features are also exposed to the user space, users who have special information about their workloads can write personalized applications for better understanding and optimizations of their workloads and systems.

For easier development of such systems, DAMON provides a feature called DAMOS (DAMon-based Operation Schemes) in addition to the monitoring. Using the feature, DAMON users in both kernel and user spaces can do access-aware system operations with no code but simple configurations.

2.4.1 Frequently Asked Questions

Does DAMON support virtual memory only?

No. The core of the DAMON is address space independent. The address space specific monitoring operations including monitoring target regions constructions and actual access checks can be implemented and configured on the DAMON core by the users. In this way, DAMON users can monitor any address space with any access check technique.

Nonetheless, DAMON provides vma/rmap tracking and PTE Accessed bit check based implementations of the address space dependent functions for the virtual memory and the physical memory by default, for a reference and convenient use.

Can I simply monitor page granularity?

Yes. You can do so by setting the `min_nr_regions` attribute higher than the working set size divided by the page size. Because the monitoring target regions size is forced to be `>=page size`, the region split will make no effect.

2.4.2 Design

Execution Model and Data Structures

The monitoring-related information including the monitoring request specification and DAMON-based operation schemes are stored in a data structure called DAMON `context`. DAMON executes each context with a kernel thread called `kdamond`. Multiple kdamonds could run in parallel, for different types of monitoring.

Overall Architecture

DAMON subsystem is configured with three layers including

- Operations Set: Implements fundamental operations for DAMON that depends on the given monitoring target address-space and available set of software/hardware primitives,
- Core: Implements core logics including monitoring overhead/accurach control and access-aware system operations on top of the operations set layer, and
- Modules: Implements kernel modules for various purposes that provides interfaces for the user space, on top of the core layer.

Configurable Operations Set

For data access monitoring and additional low level work, DAMON needs a set of implementations for specific operations that are dependent on and optimized for the given target address space. On the other hand, the accuracy and overhead tradeoff mechanism, which is the core logic of DAMON, is in the pure logic space. DAMON separates the two parts in different layers, namely DAMON Operations Set and DAMON Core Logics Layers, respectively. It further defines the interface between the layers to allow various operations sets to be configured with the core logic.

Due to this design, users can extend DAMON for any address space by configuring the core logic to use the appropriate operations set. If any appropriate set is unavailable, users can implement one on their own.

For example, physical memory, virtual memory, swap space, those for specific processes, NUMA nodes, files, and backing memory devices would be supportable. Also, if some architectures or devices supporting special optimized access check primitives, those will be easily configurable.

Programmable Modules

Core layer of DAMON is implemented as a framework, and exposes its application programming interface to all kernel space components such as subsystems and modules. For common use cases of DAMON, DAMON subsystem provides kernel modules that built on top of the core layer using the API, which can be easily used by the user space end users.

Operations Set Layer

The monitoring operations are defined in two parts:

1. Identification of the monitoring target address range for the address space.
2. Access check of specific address range in the target space.

DAMON currently provides the implementations of the operations for the physical and virtual address spaces. Below two subsections describe how those work.

VMA-based Target Address Range Construction

This is only for the virtual address space monitoring operations implementation. That for the physical address space simply asks users to manually set the monitoring target address ranges.

Only small parts in the super-huge virtual address space of the processes are mapped to the physical memory and accessed. Thus, tracking the unmapped address regions is just wasteful. However, because DAMON can deal with some level of noise using the adaptive regions adjustment mechanism, tracking every mapping is not strictly required but could even incur a high overhead in some cases. That said, too huge unmapped areas inside the monitoring target should be removed to not take the time for the adaptive mechanism.

For the reason, this implementation converts the complex mappings to three distinct regions that cover every mapped area of the address space. The two gaps between the three regions are the two biggest unmapped areas in the given address space. The two biggest unmapped areas would be the gap between the heap and the uppermost mmap()-ed region, and the gap between the lowermost mmap()-ed region and the stack in most of the cases. Because these gaps are exceptionally huge in usual address spaces, excluding these will be sufficient to make a reasonable trade-off. Below shows this in detail:

```
<heap>
<BIG UNMAPPED REGION 1>
<uppermost mmap()-ed region>
(small mmap()-ed regions and munmap()-ed regions)
<lowermost mmap()-ed region>
<BIG UNMAPPED REGION 2>
<stack>
```

PTE Accessed-bit Based Access Check

Both of the implementations for physical and virtual address spaces use PTE Accessed-bit for basic access checks. Only one difference is the way of finding the relevant PTE Accessed bit(s) from the address. While the implementation for the virtual address walks the page table for the target task of the address, the implementation for the physical address walks every page table having a mapping to the address. In this way, the implementations find and clear the bit(s) for next sampling target address and checks whether the bit(s) set again after one sampling period. This could disturb other kernel subsystems using the Accessed bits, namely Idle page tracking and the reclaim logic. DAMON does nothing to avoid disturbing Idle page tracking, so handling the interference is the responsibility of sysadmins. However, it solves the conflict with the reclaim logic using `PG_idle` and `PG_young` page flags, as Idle page tracking does.

Core Logics

Monitoring

Below four sections describe each of the DAMON core mechanisms and the five monitoring attributes, `sampling interval`, `aggregation interval`, `update interval`, `minimum number of regions`, and `maximum number of regions`.

Access Frequency Monitoring

The output of DAMON says what pages are how frequently accessed for a given duration. The resolution of the access frequency is controlled by setting `sampling interval` and `aggregation interval`. In detail, DAMON checks access to each page per `sampling interval` and aggregates the results. In other words, counts the number of the accesses to each page. After each `aggregation interval` passes, DAMON calls callback functions that previously registered by users so that users can read the aggregated results and then clears the results. This can be described in below simple pseudo-code:

```
while monitoring_on:
    for page in monitoring_target:
        if accessed(page):
            nr_accesses[page] += 1
    if time() % aggregation_interval == 0:
        for callback in user_registered_callbacks:
            callback(monitoring_target, nr_accesses)
        for page in monitoring_target:
            nr_accesses[page] = 0
    sleep(sampling interval)
```

The monitoring overhead of this mechanism will arbitrarily increase as the size of the target workload grows.

Region Based Sampling

To avoid the unbounded increase of the overhead, DAMON groups adjacent pages that assumed to have the same access frequencies into a region. As long as the assumption (pages in a region have the same access frequencies) is kept, only one page in the region is required to be checked. Thus, for each `sampling interval`, DAMON randomly picks one page in each region, waits for one `sampling interval`, checks whether the page is accessed meanwhile, and increases the access frequency counter of the region if so. The counter is called `nr_regions` of the region. Therefore, the monitoring overhead is controllable by setting the number of regions. DAMON allows users to set the minimum and the maximum number of regions for the trade-off.

This scheme, however, cannot preserve the quality of the output if the assumption is not guaranteed.

Adaptive Regions Adjustment

Even somehow the initial monitoring target regions are well constructed to fulfill the assumption (pages in same region have similar access frequencies), the data access pattern can be dynamically changed. This will result in low monitoring quality. To keep the assumption as much as possible, DAMON adaptively merges and splits each region based on their access frequency.

For each `aggregation interval`, it compares the access frequencies of adjacent regions and merges those if the frequency difference is small. Then, after it reports and clears the aggregated access frequency of each region, it splits each region into two or three regions if the total number of regions will not exceed the user-specified maximum number of regions after the split.

In this way, DAMON provides its best-effort quality and minimal overhead while keeping the bounds users set for their trade-off.

Age Tracking

By analyzing the monitoring results, users can also find how long the current access pattern of a region has maintained. That could be used for good understanding of the access pattern. For example, page placement algorithm utilizing both the frequency and the recency could be implemented using that. To make such access pattern maintained period analysis easier, DAMON maintains yet another counter called `age` in each region. For each `aggregation interval`, DAMON checks if the region's size and access frequency (`nr_accesses`) has significantly changed. If so, the counter is reset to zero. Otherwise, the counter is increased.

Dynamic Target Space Updates Handling

The monitoring target address range could dynamically changed. For example, virtual memory could be dynamically mapped and unmapped. Physical memory could be hot-plugged.

As the changes could be quite frequent in some cases, DAMON allows the monitoring operations to check dynamic changes including memory mapping changes and applies it to monitoring operations-related data structures such as the abstracted monitoring target memory area only for each of a user-specified time interval (`update interval`).

Operation Schemes

One common purpose of data access monitoring is access-aware system efficiency optimizations. For example,

> paging out memory regions that are not accessed for more than two minutes

or

> using THP for memory regions that are larger than 2 MiB and showing a high access frequency for more than one minute.

One straightforward approach for such schemes would be profile-guided optimizations. That is, getting data access monitoring results of the workloads or the system using DAMON, finding

memory regions of special characteristics by profiling the monitoring results, and making system operation changes for the regions. The changes could be made by modifying or providing advice to the software (the application and/or the kernel), or reconfiguring the hardware. Both offline and online approaches could be available.

Among those, providing advice to the kernel at runtime would be flexible and effective, and therefore widely be used. However, implementing such schemes could impose unnecessary redundancy and inefficiency. The profiling could be redundant if the type of interest is common. Exchanging the information including monitoring results and operation advice between kernel and user spaces could be inefficient.

To allow users to reduce such redundancy and inefficiencies by offloading the works, DAMON provides a feature called Data Access Monitoring-based Operation Schemes (DAMOS). It lets users specify their desired schemes at a high level. For such specifications, DAMON starts monitoring, finds regions having the access pattern of interest, and applies the user-desired operation actions to the regions, for every user-specified time interval called `apply_interval`.

Operation Action

The management action that the users desire to apply to the regions of their interest. For example, paging out, prioritizing for next reclamation victim selection, advising `khugepaged` to collapse or split, or doing nothing but collecting statistics of the regions.

The list of supported actions is defined in DAMOS, but the implementation of each action is in the DAMON operations set layer because the implementation normally depends on the monitoring target address space. For example, the code for paging specific virtual address ranges out would be different from that for physical address ranges. And the monitoring operations implementation sets are not mandated to support all actions of the list. Hence, the availability of specific DAMOS action depends on what operations set is selected to be used together.

Applying an action to a region is considered as changing the region's characteristics. Hence, DAMOS resets the age of regions when an action is applied to those.

Target Access Pattern

The access pattern of the schemes' interest. The patterns are constructed with the properties that DAMON's monitoring results provide, specifically the size, the access frequency, and the age. Users can describe their access pattern of interest by setting minimum and maximum values of the three properties. If a region's three properties are in the ranges, DAMOS classifies it as one of the regions that the scheme is having an interest in.

Quotas

DAMOS upper-bound overhead control feature. DAMOS could incur high overhead if the target access pattern is not properly tuned. For example, if a huge memory region having the access pattern of interest is found, applying the scheme's action to all pages of the huge region could consume unacceptably large system resources. Preventing such issues by tuning the access pattern could be challenging, especially if the access patterns of the workloads are highly dynamic.

To mitigate that situation, DAMOS provides an upper-bound overhead control feature called quotas. It lets users specify an upper limit of time that DAMOS can use for applying the action, and/or a maximum bytes of memory regions that the action can be applied within a user-specified time duration.

Prioritization

A mechanism for making a good decision under the quotas. When the action cannot be applied to all regions of interest due to the quotas, DAMOS prioritizes regions and applies the action to only regions having high enough priorities so that it will not exceed the quotas.

The prioritization mechanism should be different for each action. For example, rarely accessed (colder) memory regions would be prioritized for page-out scheme action. In contrast, the colder regions would be deprioritized for huge page collapse scheme action. Hence, the prioritization mechanisms for each action are implemented in each DAMON operations set, together with the actions.

Though the implementation is up to the DAMON operations set, it would be common to calculate the priority using the access pattern properties of the regions. Some users would want the mechanisms to be personalized for their specific case. For example, some users would want the mechanism to weigh the recency (`age`) more than the access frequency (`nr_accesses`). DAMOS allows users to specify the weight of each access pattern property and passes the information to the underlying mechanism. Nevertheless, how and even whether the weight will be respected are up to the underlying prioritization mechanism implementation.

Aim-oriented Feedback-driven Auto-tuning

Automatic feedback-driven quota tuning. Instead of setting the absolute quota value, users can repeatedly provide numbers representing how much of their goal for the scheme is achieved as feedback. DAMOS then automatically tunes the aggressiveness (the quota) of the corresponding scheme. For example, if DAMOS is under achieving the goal, DAMOS automatically increases the quota. If DAMOS is over achieving the goal, it decreases the quota.

Watermarks

Conditional DAMOS (de)activation automation. Users might want DAMOS to run only under certain situations. For example, when a sufficient amount of free memory is guaranteed, running a scheme for proactive reclamation would only consume unnecessary system resources. To avoid such consumption, the user would need to manually monitor some metrics such as free memory ratio, and turn DAMON/DAMOS on or off.

DAMOS allows users to offload such works using three watermarks. It allows the users to configure the metric of their interest, and three watermark values, namely high, middle, and low. If the value of the metric becomes above the high watermark or below the low watermark, the scheme is deactivated. If the metric becomes below the mid watermark but above the low watermark, the scheme is activated. If all schemes are deactivated by the watermarks, the monitoring is also deactivated. In this case, the DAMON worker thread only periodically checks the watermarks and therefore incurs nearly zero overhead.

Filters

Non-access pattern-based target memory regions filtering. If users run self-written programs or have good profiling tools, they could know something more than the kernel, such as future access patterns or some special requirements for specific types of memory. For example, some users may know only anonymous pages can impact their program's performance. They can also have a list of latency-critical processes.

To let users optimize DAMOS schemes with such special knowledge, DAMOS provides a feature called DAMOS filters. The feature allows users to set an arbitrary number of filters for each scheme. Each filter specifies the type of target memory, and whether it should exclude the memory of the type (filter-out), or all except the memory of the type (filter-in).

Currently, anonymous page, memory cgroup, address range, and DAMON monitoring target type filters are supported by the feature. Some filter target types require additional arguments. The memory cgroup filter type asks users to specify the file path of the memory cgroup for the filter. The address range type asks the start and end addresses of the range. The DAMON monitoring target type asks the index of the target from the context's monitoring targets list. Hence, users can apply specific schemes to only anonymous pages, non-anonymous pages, pages of specific cgroups, all pages excluding those of specific cgroups, pages in specific address range, pages in specific DAMON monitoring targets, and any combination of those.

To handle filters efficiently, the address range and DAMON monitoring target type filters are handled by the core layer, while others are handled by operations set. If a memory region is filtered by a core layer-handled filter, it is not counted as the scheme has tried to the region. In contrast, if a memory regions is filtered by an operations set layer-handled filter, it is counted as the scheme has tried. The difference in accounting leads to changes in the statistics.

Application Programming Interface

The programming interface for kernel space data access-aware applications. DAMON is a framework, so it does nothing by itself. Instead, it only helps other kernel components such as subsystems and modules building their data access-aware applications using DAMON's core features. For this, DAMON exposes its all features to other kernel components via its application programming interface, namely `include/linux/damon.h`. Please refer to the API *document* for details of the interface.

Modules

Because the core of DAMON is a framework for kernel components, it doesn't provide any direct interface for the user space. Such interfaces should be implemented by each DAMON API user kernel components, instead. DAMON subsystem itself implements such DAMON API user modules, which are supposed to be used for general purpose DAMON control and special purpose data access-aware system operations, and provides stable application binary interfaces (ABI) for the user space. The user space can build their efficient data access-aware applications using the interfaces.

General Purpose User Interface Modules

DAMON modules that provide user space ABIs for general purpose DAMON usage in runtime.

DAMON user interface modules, namely 'DAMON sysfs interface' and 'DAMON debugfs interface' are DAMON API user kernel modules that provide ABIs to the user-space. Please note that DAMON debugfs interface is currently deprecated.

Like many other ABIs, the modules create files on sysfs and debugfs, allow users to specify their requests to and get the answers from DAMON by writing to and reading from the files. As a response to such I/O, DAMON user interface modules control DAMON and retrieve the results as user requested via the DAMON API, and return the results to the user-space.

The ABIs are designed to be used for user space applications development, rather than human beings' fingers. Human users are recommended to use such user space tools. One such Python-written user space tool is available at Github (https://github.com/awslabs/damo), Pypi (https://pypistats.org/packages/damo), and Fedora (https://packages.fedoraproject.org/pkgs/python-damo/damo/).

Please refer to the ABI document for details of the interfaces.

Special-Purpose Access-aware Kernel Modules

DAMON modules that provide user space ABI for specific purpose DAMON usage.

DAMON sysfs/debugfs user interfaces are for full control of all DAMON features in runtime. For each special-purpose system-wide data access-aware system operations such as proactive reclamation or LRU lists balancing, the interfaces could be simplified by removing unnecessary knobs for the specific purpose, and extended for boot-time and even compile time control. Default values of DAMON control parameters for the usage would also need to be optimized for the purpose.

To support such cases, yet more DAMON API user kernel modules that provide more simple and optimized user space interfaces are available. Currently, two modules for proactive reclamation and LRU lists manipulation are provided. For more detail, please read the usage documents for those (/admin-guide/mm/damon/reclaim and /admin-guide/mm/damon/lru_sort).

2.4.3 API Reference

Kernel space programs can use every feature of DAMON using below APIs. All you need to do is including `damon.h`, which is located in `include/linux/` of the source tree.

Structures

struct **damon_addr_range**

Represents an address region of [**start**, **end**).

Definition:

```
struct damon_addr_range {
    unsigned long start;
```

```
    unsigned long end;
};
```

Members

start
Start address of the region (inclusive).

end
End address of the region (exclusive).

struct **damon_region**
Represents a monitoring target region.

Definition:

```
struct damon_region {
    struct damon_addr_range ar;
    unsigned long sampling_addr;
    unsigned int nr_accesses;
    unsigned int nr_accesses_bp;
    struct list_head list;
    unsigned int age;
};
```

Members

ar
The address range of the region.

sampling_addr
Address of the sample for the next access check.

nr_accesses
Access frequency of this region.

nr_accesses_bp
nr_accesses in basis point (0.01%) that updated for each sampling interval.

list
List head for siblings.

age
Age of this region.

Description

nr_accesses is reset to zero for every *damon_attrs->aggr_interval* and be increased for every *damon_attrs->sample_interval* if an access to the region during the last sampling interval is found. The update of this field should not be done with direct access but with the helper function, *damon_update_region_access_rate()*.

nr_accesses_bp is another representation of **nr_accesses** in basis point (1 in 10,000) that updated for every *damon_attrs->sample_interval* in a manner similar to moving sum. By the algorithm, this value becomes **nr_accesses** * 10000 for every *struct damon_attrs*->aggr_interval. This can be used when the aggregation interval is too huge and therefore cannot wait for it before getting the access monitoring results.

age is initially zero, increased for each aggregation interval, and reset to zero again if the access frequency is significantly changed. If two regions are merged into a new region, both **nr_accesses** and **age** of the new region are set as region size-weighted average of those of the two regions.

struct **damon_target**

Represents a monitoring target.

Definition:

```
struct damon_target {
    struct pid *pid;
    unsigned int nr_regions;
    struct list_head regions_list;
    struct list_head list;
};
```

Members

pid
The PID of the virtual address space to monitor.

nr_regions
Number of monitoring target regions of this target.

regions_list
Head of the monitoring target regions of this target.

list
List head for siblings.

Description

Each monitoring context could have multiple targets. For example, a context for virtual memory address spaces could have multiple target processes. The **pid** should be set for appropriate *struct damon_operations* including the virtual address spaces monitoring operations.

enum **damos_action**

Represents an action of a Data Access Monitoring-based Operation Scheme.

Constants

DAMOS_WILLNEED
Call `madvise()` for the region with MADV_WILLNEED.

DAMOS_COLD
Call `madvise()` for the region with MADV_COLD.

DAMOS_PAGEOUT
Call `madvise()` for the region with MADV_PAGEOUT.

DAMOS_HUGEPAGE
Call `madvise()` for the region with MADV_HUGEPAGE.

DAMOS_NOHUGEPAGE
Call `madvise()` for the region with MADV_NOHUGEPAGE.

DAMOS_LRU_PRIO
Prioritize the region on its LRU lists.

DAMOS_LRU_DEPRIO
Deprioritize the region on its LRU lists.

DAMOS_STAT
Do nothing but count the stat.

NR_DAMOS_ACTIONS
Total number of DAMOS actions

Description

The support of each action is up to running *struct damon_operations*. enum DAMON_OPS_VADDR and enum DAMON_OPS_FVADDR supports all actions except enum DAMOS_LRU_PRIO and enum DAMOS_LRU_DEPRIO. enum DAMON_OPS_PADDR supports only enum DAMOS_PAGEOUT, enum DAMOS_LRU_PRIO, enum DAMOS_LRU_DEPRIO, and DAMOS_STAT.

struct **damos_quota**
Controls the aggressiveness of the given scheme.

Definition:

```
struct damos_quota {
    unsigned long ms;
    unsigned long sz;
    unsigned long reset_interval;
    unsigned int weight_sz;
    unsigned int weight_nr_accesses;
    unsigned int weight_age;
    unsigned long (*get_score)(void *arg);
    void *get_score_arg;
};
```

Members

ms
Maximum milliseconds that the scheme can use.

sz
Maximum bytes of memory that the action can be applied.

reset_interval
Charge reset interval in milliseconds.

weight_sz
Weight of the region's size for prioritization.

weight_nr_accesses
Weight of the region's nr_accesses for prioritization.

weight_age
Weight of the region's age for prioritization.

get_score
Feedback function for self-tuning quota.

get_score_arg
Parameter for **get_score**

Description

To avoid consuming too much CPU time or IO resources for applying the *struct damos*->action to large memory, DAMON allows users to set time and/or size quotas. The quotas can be set by writing non-zero values to `ms` and `sz`, respectively. If the time quota is set, DAMON tries to use only up to `ms` milliseconds within `reset_interval` for applying the action. If the size quota is set, DAMON tries to apply the action only up to `sz` bytes within `reset_interval`.

Internally, the time quota is transformed to a size quota using estimated throughput of the scheme's action. DAMON then compares it against `sz` and uses smaller one as the effective quota.

For selecting regions within the quota, DAMON prioritizes current scheme's target memory regions using the *struct damon_operations*->get_scheme_score. You could customize the prioritization logic by setting `weight_sz`, `weight_nr_accesses`, and `weight_age`, because monitoring operations are encouraged to respect those.

If **get_score** function pointer is set, DAMON calls it back with **get_score_arg** and get the return value of it for every **reset_interval**. Then, DAMON adjusts the effective quota using the return value as a feedback score to the current quota, using its internal feedback loop algorithm.

The feedback loop algorithem assumes the quota input and the feedback score output are in a positive proportional relationship, and the goal of the tuning is getting the feedback screo value of 10,000. If **ms** and/or **sz** are set together, those work as a hard limit quota. If neither **ms** nor **sz** are set, the mechanism starts from the quota of one byte.

enum **damos_wmark_metric**

Represents the watermark metric.

Constants

DAMOS_WMARK_NONE
Ignore the watermarks of the given scheme.

DAMOS_WMARK_FREE_MEM_RATE
Free memory rate of the system in [0,1000].

NR_DAMOS_WMARK_METRICS
Total number of DAMOS watermark metrics

struct **damos_watermarks**

Controls when a given scheme should be activated.

Definition:

```
struct damos_watermarks {
    enum damos_wmark_metric metric;
    unsigned long interval;
    unsigned long high;
    unsigned long mid;
    unsigned long low;
};
```

Members

metric
Metric for the watermarks.

interval
Watermarks check time interval in microseconds.

high
High watermark.

mid
Middle watermark.

low
Low watermark.

Description

If `metric` is `DAMOS_WMARK_NONE`, the scheme is always active. Being active means DAMON does monitoring and applying the action of the scheme to appropriate memory regions. Else, DAMON checks `metric` of the system for at least every `interval` microseconds and works as below.

If `metric` is higher than `high`, the scheme is inactivated. If `metric` is between `mid` and `low`, the scheme is activated. If `metric` is lower than `low`, the scheme is inactivated.

struct **damos_stat**
Statistics on a given scheme.

Definition:

```
struct damos_stat {
    unsigned long nr_tried;
    unsigned long sz_tried;
    unsigned long nr_applied;
    unsigned long sz_applied;
    unsigned long qt_exceeds;
};
```

Members

nr_tried
Total number of regions that the scheme is tried to be applied.

sz_tried
Total size of regions that the scheme is tried to be applied.

nr_applied
Total number of regions that the scheme is applied.

sz_applied
Total size of regions that the scheme is applied.

qt_exceeds
Total number of times the quota of the scheme has exceeded.

enum **damos_filter_type**
Type of memory for *struct damos_filter*

Constants

DAMOS_FILTER_TYPE_ANON
Anonymous pages.

DAMOS_FILTER_TYPE_MEMCG
Specific memcg's pages.

DAMOS_FILTER_TYPE_ADDR
Address range.

DAMOS_FILTER_TYPE_TARGET
Data Access Monitoring target.

NR_DAMOS_FILTER_TYPES
Number of filter types.

Description

The anon pages type and memcg type filters are handled by underlying *struct damon_operations* as a part of scheme action trying, and therefore accounted as 'tried'. In contrast, other types are handled by core layer before trying of the action and therefore not accounted as 'tried'.

The support of the filters that handled by *struct damon_operations* depend on the running *struct damon_operations*. enum DAMON_OPS_PADDR supports both anon pages type and memcg type filters, while enum DAMON_OPS_VADDR and enum DAMON_OPS_FVADDR don't support any of the two types.

struct **damos_filter**
DAMOS action target memory filter.

Definition:

```
struct damos_filter {
    enum damos_filter_type type;
    bool matching;
    union {
        unsigned short memcg_id;
        struct damon_addr_range addr_range;
        int target_idx;
    };
    struct list_head list;
};
```

Members

type
Type of the page.

matching
If the matching page should filtered out or in.

{unnamed_union}
anonymous

memcg_id
Memcg id of the question if **type** is DAMOS_FILTER_MEMCG.

addr_range
Address range if **type** is DAMOS_FILTER_TYPE_ADDR.

target_idx
Index of the *struct damon_target* of *damon_ctx->adaptive_targets* if **type** is DAMOS_FILTER_TYPE_TARGET.

list
List head for siblings.

Description

Before applying the *damos->action* to a memory region, DAMOS checks if each page of the region matches to this and avoid applying the action if so. Support of each filter type depends on the running *struct damon_operations* and the type. Refer to *enum damos_filter_type* for more detai.

struct **damos_access_pattern**
Target access pattern of the given scheme.

Definition:

```
struct damos_access_pattern {
    unsigned long min_sz_region;
    unsigned long max_sz_region;
    unsigned int min_nr_accesses;
    unsigned int max_nr_accesses;
    unsigned int min_age_region;
    unsigned int max_age_region;
};
```

Members

min_sz_region
Minimum size of target regions.

max_sz_region
Maximum size of target regions.

min_nr_accesses
Minimum ->nr_accesses of target regions.

max_nr_accesses
Maximum ->nr_accesses of target regions.

min_age_region
Minimum age of target regions.

max_age_region
Maximum age of target regions.

struct **damos**
Represents a Data Access Monitoring-based Operation Scheme.

Definition:

```
struct damos {
    struct damos_access_pattern pattern;
    enum damos_action action;
    unsigned long apply_interval_us;
    struct damos_quota quota;
```

```
    struct damos_watermarks wmarks;
    struct list_head filters;
    struct damos_stat stat;
    struct list_head list;
};
```

Members

pattern
Access pattern of target regions.

action
damo_action to be applied to the target regions.

apply_interval_us
The time between applying the **action**.

quota
Control the aggressiveness of this scheme.

wmarks
Watermarks for automated (in)activation of this scheme.

filters
Additional set of *struct damos_filter* for action.

stat
Statistics of this scheme.

list
List head for siblings.

Description

For each **apply_interval_us**, DAMON finds regions which fit in the pattern and applies action to those. To avoid consuming too much CPU time or IO resources for the action, quota is used.

If **apply_interval_us** is zero, *damon_attrs->aggr_interval* is used instead.

To do the work only when needed, schemes can be activated for specific system situations using wmarks. If all schemes that registered to the monitoring context are inactive, DAMON stops monitoring either, and just repeatedly checks the watermarks.

Before applying the action to a memory region, *struct damon_operations* implementation could check pages of the region and skip action to respect filters

After applying the action to each region, stat_count and stat_sz is updated to reflect the number of regions and total size of regions that the action is applied.

enum **damon_ops_id**
Identifier for each monitoring operations implementation

Constants

DAMON_OPS_VADDR
Monitoring operations for virtual address spaces

DAMON_OPS_FVADDR
Monitoring operations for only fixed ranges of virtual address spaces

DAMON_OPS_PADDR
Monitoring operations for the physical address space

NR_DAMON_OPS
Number of monitoring operations implementations

struct **damon_operations**
Monitoring operations for given use cases.

Definition:

```
struct damon_operations {
    enum damon_ops_id id;
    void (*init)(struct damon_ctx *context);
    void (*update)(struct damon_ctx *context);
    void (*prepare_access_checks)(struct damon_ctx *context);
    unsigned int (*check_accesses)(struct damon_ctx *context);
    void (*reset_aggregated)(struct damon_ctx *context);
    int (*get_scheme_score)(struct damon_ctx *context,struct damon_target *t,␣
↪struct damon_region *r, struct damos *scheme);
    unsigned long (*apply_scheme)(struct damon_ctx *context,struct damon_
↪target *t, struct damon_region *r, struct damos *scheme);
    bool (*target_valid)(struct damon_target *t);
    void (*cleanup)(struct damon_ctx *context);
};
```

Members

id
Identifier of this operations set.

init
Initialize operations-related data structures.

update
Update operations-related data structures.

prepare_access_checks
Prepare next access check of target regions.

check_accesses
Check the accesses to target regions.

reset_aggregated
Reset aggregated accesses monitoring results.

get_scheme_score
Get the score of a region for a scheme.

apply_scheme
Apply a DAMON-based operation scheme.

target_valid
Determine if the target is valid.

cleanup
Clean up the context.

Description

DAMON can be extended for various address spaces and usages. For this, users should register the low level operations for their target address space and usecase via the *damon_ctx.ops*. Then, the monitoring thread (*damon_ctx.kdamond*) calls **init** and **prepare_access_checks** before starting the monitoring, **update** after each *damon_attrs.ops_update_interval*, and **check_accesses**, **target_valid** and **prepare_access_checks** after each *damon_attrs.sample_interval*. Finally, **reset_aggregated** is called after each *damon_attrs.aggr_interval*.

Each *struct damon_operations* instance having valid **id** can be registered via *damon_register_ops()* and selected by *damon_select_ops()* later. **init** should initialize operations-related data structures. For example, this could be used to construct proper monitoring target regions and link those to **damon_ctx.adaptive_targets**. **update** should update the operations-related data structures. For example, this could be used to update monitoring target regions for current status. **prepare_access_checks** should manipulate the monitoring regions to be prepared for the next access check. **check_accesses** should check the accesses to each region that made after the last preparation and update the number of observed accesses of each region. It should also return max number of observed accesses that made as a result of its update. The value will be used for regions adjustment threshold. **reset_aggregated** should reset the access monitoring results that aggregated by **check_accesses**. **get_scheme_score** should return the priority score of a region for a scheme as an integer in [0, DAMOS_MAX_SCORE]. **apply_scheme** is called from **kdamond** when a region for user provided DAMON-based operation scheme is found. It should apply the scheme's action to the region and return bytes of the region that the action is successfully applied. **target_valid** should check whether the target is still valid for the monitoring. **cleanup** is called from **kdamond** just before its termination.

struct **damon_callback**

Monitoring events notification callbacks.

Definition:

```
struct damon_callback {
    void *private;
    int (*before_start)(struct damon_ctx *context);
    int (*after_wmarks_check)(struct damon_ctx *context);
    int (*after_sampling)(struct damon_ctx *context);
    int (*after_aggregation)(struct damon_ctx *context);
    int (*before_damos_apply)(struct damon_ctx *context,struct damon_target␣
↪*target,struct damon_region *region, struct damos *scheme);
    void (*before_terminate)(struct damon_ctx *context);
};
```

Members

private
User private data.

before_start
Called before starting the monitoring.

after_wmarks_check
Called after each schemes' watermarks check.

after_sampling
Called after each sampling.

after_aggregation
Called after each aggregation.

before_damos_apply
Called before applying DAMOS action.

before_terminate
Called before terminating the monitoring.

Description

The monitoring thread (*damon_ctx.kdamond*) calls **before_start** and **before_terminate** just before starting and finishing the monitoring, respectively. Therefore, those are good places for installing and cleaning **private**.

The monitoring thread calls **after_wmarks_check** after each DAMON-based operation schemes' watermarks check. If users need to make changes to the attributes of the monitoring context while it's deactivated due to the watermarks, this is the good place to do.

The monitoring thread calls **after_sampling** and **after_aggregation** for each of the sampling intervals and aggregation intervals, respectively. Therefore, users can safely access the monitoring results without additional protection. For the reason, users are recommended to use these callback for the accesses to the results.

If any callback returns non-zero, monitoring stops.

struct **damon_attrs**
Monitoring attributes for accuracy/overhead control.

Definition:

```
struct damon_attrs {
    unsigned long sample_interval;
    unsigned long aggr_interval;
    unsigned long ops_update_interval;
    unsigned long min_nr_regions;
    unsigned long max_nr_regions;
};
```

Members

sample_interval
The time between access samplings.

aggr_interval
The time between monitor results aggregations.

ops_update_interval
The time between monitoring operations updates.

min_nr_regions
The minimum number of adaptive monitoring regions.

max_nr_regions
The maximum number of adaptive monitoring regions.

Description

For each **sample_interval**, DAMON checks whether each region is accessed or not during the last **sample_interval**. If such access is found, DAMON aggregates the information by increasing *damon_region->nr_accesses* for **aggr_interval** time. For each **aggr_interval**, the count is reset. DAMON also checks whether the target memory regions need update (e.g., by `mmap()` calls from the application, in case of virtual memory monitoring) and applies the changes for each **ops_update_interval**. All time intervals are in micro-seconds. Please refer to *struct damon_operations* and *struct damon_callback* for more detail.

struct **damon_ctx**

Represents a context for each monitoring. This is the main interface that allows users to set the attributes and get the results of the monitoring.

Definition:

```
struct damon_ctx {
    struct damon_attrs attrs;
    struct task_struct *kdamond;
    struct mutex kdamond_lock;
    struct damon_operations ops;
    struct damon_callback callback;
    struct list_head adaptive_targets;
    struct list_head schemes;
};
```

Members

attrs
Monitoring attributes for accuracy/overhead control.

kdamond
Kernel thread who does the monitoring.

kdamond_lock
Mutex for the synchronizations with **kdamond**.

ops
Set of monitoring operations for given use cases.

callback
Set of callbacks for monitoring events notifications.

adaptive_targets
Head of monitoring targets (*damon_target*) list.

schemes
Head of schemes (*damos*) list.

Description

For each monitoring context, one kernel thread for the monitoring is created. The pointer to the thread is stored in **kdamond**.

Once started, the monitoring thread runs until explicitly required to be terminated or every monitoring target is invalid. The validity of the targets is checked via the *damon_operations.target_valid* of **ops**. The termination can also be explicitly requested by calling *damon_stop()*. The thread sets **kdamond** to NULL when it terminates. Therefore, users can

know whether the monitoring is ongoing or terminated by reading **kdamond**. Reads and writes to **kdamond** from outside of the monitoring thread must be protected by **kdamond_lock**.

Note that the monitoring thread protects only **kdamond** via **kdamond_lock**. Accesses to other fields must be protected by themselves.

Functions

bool **damon_is_registered_ops**(enum *damon_ops_id* id)

Check if a given damon_operations is registered.

Parameters

enum damon_ops_id id
Id of the damon_operations to check if registered.

Return

true if the ops is set, false otherwise.

int **damon_register_ops**(struct *damon_operations* *ops)

Register a monitoring operations set to DAMON.

Parameters

struct damon_operations *ops
monitoring operations set to register.

Description

This function registers a monitoring operations set of valid *struct damon_operations*->id so that others can find and use them later.

Return

0 on success, negative error code otherwise.

int **damon_select_ops**(struct *damon_ctx* *ctx, enum *damon_ops_id* id)

Select a monitoring operations to use with the context.

Parameters

struct damon_ctx *ctx
monitoring context to use the operations.

enum damon_ops_id id
id of the registered monitoring operations to select.

Description

This function finds registered monitoring operations set of **id** and make **ctx** to use it.

Return

0 on success, negative error code otherwise.

int **damon_set_attrs**(struct *damon_ctx* *ctx, struct *damon_attrs* *attrs)

Set attributes for the monitoring.

Parameters

struct damon_ctx *ctx
monitoring context

struct damon_attrs *attrs
monitoring attributes

Description

This function should be called while the kdamond is not running, or an access check results aggregation is not ongoing (e.g., from *struct damon_callback*->after_aggregation or *struct damon_callback*->after_wmarks_check callbacks).

Every time interval is in micro-seconds.

Return

0 on success, negative error code otherwise.

void **damon_set_schemes**(struct *damon_ctx* *ctx, struct *damos* **schemes, ssize_t nr_schemes)
Set data access monitoring based operation schemes.

Parameters

struct damon_ctx *ctx
monitoring context

struct damos **schemes
array of the schemes

ssize_t nr_schemes
number of entries in **schemes**

Description

This function should not be called while the kdamond of the context is running.

int **damon_nr_running_ctxs**(void)
Return number of currently running contexts.

Parameters

void
no arguments

int **damon_start**(struct *damon_ctx* **ctxs, int nr_ctxs, bool exclusive)
Starts the monitorings for a given group of contexts.

Parameters

struct damon_ctx **ctxs
an array of the pointers for contexts to start monitoring

int nr_ctxs
size of **ctxs**

bool exclusive
exclusiveness of this contexts group

Description

This function starts a group of monitoring threads for a group of monitoring contexts. One thread per each context is created and run in parallel. The caller should handle synchronization between the threads by itself. If **exclusive** is true and a group of threads that created by other '*damon_start()*' call is currently running, this function does nothing but returns -EBUSY.

Return

0 on success, negative error code otherwise.

int **damon_stop**(struct *damon_ctx* **ctxs, int nr_ctxs)

Stops the monitorings for a given group of contexts.

Parameters

struct damon_ctx **ctxs
an array of the pointers for contexts to stop monitoring

int nr_ctxs
size of **ctxs**

Return

0 on success, negative error code otherwise.

int **damon_set_region_biggest_system_ram_default**(struct *damon_target* *t, unsigned long *start, unsigned long *end)

Set the region of the given monitoring target as requested, or biggest 'System RAM'.

Parameters

struct damon_target *t
The monitoring target to set the region.

unsigned long *start
The pointer to the start address of the region.

unsigned long *end
The pointer to the end address of the region.

Description

This function sets the region of **t** as requested by **start** and **end**. If the values of **start** and **end** are zero, however, this function finds the biggest 'System RAM' resource and sets the region to cover the resource. In the latter case, this function saves the start and end addresses of the resource in **start** and **end**, respectively.

Return

0 on success, negative error code otherwise.

void **damon_update_region_access_rate**(struct *damon_region* *r, bool accessed, struct *damon_attrs* *attrs)

Update the access rate of a region.

Parameters

struct damon_region *r
The DAMON region to update for its access check result.

bool accessed
Whether the region has accessed during last sampling interval.

struct damon_attrs *attrs
The damon_attrs of the DAMON context.

Description

Update the access rate of a region with the region's last sampling interval access check result.

Usually this will be called by *damon_operations->check_accesses* callback.

2.4.4 DAMON Maintainer Entry Profile

The DAMON subsystem covers the files that are listed in 'DATA ACCESS MONITOR' section of 'MAINTAINERS' file.

The mailing lists for the subsystem are damon@lists.linux.dev and linux-mm@kvack.org. Patches should be made against the mm-unstable tree[1] whenever possible and posted to the mailing lists.

SCM Trees

There are multiple Linux trees for DAMON development. Patches under development or testing are queued in damon/next[2] by the DAMON maintainer. Sufficiently reviewed patches will be queued in mm-unstable[1] by the memory management subsystem maintainer. After more sufficient tests, the patches will be queued in mm-stable[3] , and finally pull-requested to the mainline by the memory management subsystem maintainer.

Note again the patches for review should be made against the mm-unstable tree[1] whenever possible. damon/next is only for preview of others' works in progress.

Submit checklist addendum

When making DAMON changes, you should do below.

- Build changes related outputs including kernel and documents.
- Ensure the builds introduce no new errors or warnings.
- Run and ensure no new failures for DAMON selftests[4] and kunittests[5] .

Further doing below and putting the results will be helpful.

- Run damon-tests/corr[6] for normal changes.
- Run damon-tests/perf[7] for performance changes.

[1] https://git.kernel.org/akpm/mm/h/mm-unstable
[2] https://git.kernel.org/sj/h/damon/next
[3] https://git.kernel.org/akpm/mm/h/mm-stable
[4] https://github.com/awslabs/damon-tests/blob/master/corr/run.sh#L49
[5] https://github.com/awslabs/damon-tests/blob/master/corr/tests/kunit.sh
[6] https://github.com/awslabs/damon-tests/tree/master/corr
[7] https://github.com/awslabs/damon-tests/tree/master/perf

Key cycle dates

Patches can be sent anytime. Key cycle dates of the mm-unstable[1] and mm-stable[3] trees depend on the memory management subsystem maintainer.

Review cadence

The DAMON maintainer does the work on the usual work hour (09:00 to 17:00, Mon-Fri) in PST. The response to patches will occasionally be slow. Do not hesitate to send a ping if you have not heard back within a week of sending a patch.

2.5 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

2.6 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.

2.6.1 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.

2.6.2 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).

2.6.3 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.

2.6.4 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. 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))
         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.

2.6.5 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.

2.6.6 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.

2.6.7 Migration to and from device memory

Because the CPU cannot access device memory directly, the device driver must use hardware DMA or device specific load/store instructions to migrate data. The migrate_vma_setup(), migrate_vma_pages(), and migrate_vma_finalize() functions are designed to make drivers easier to write and to centralize common code across drivers.

Before migrating pages to device private memory, special device private `struct page` need to be created. These will be used as special "swap" page table entries so that a CPU process will fault if it tries to access a page that has been migrated to device private memory.

These can be allocated and freed with:

```
struct resource *res;
struct dev_pagemap pagemap;

res = request_free_mem_region(&iomem_resource, /* number of bytes */,
                              "name of driver resource");
pagemap.type = MEMORY_DEVICE_PRIVATE;
pagemap.range.start = res->start;
pagemap.range.end = res->end;
pagemap.nr_range = 1;
pagemap.ops = &device_devmem_ops;
memremap_pages(&pagemap, numa_node_id());

memunmap_pages(&pagemap);
release_mem_region(pagemap.range.start, range_len(&pagemap.range));
```

There are also devm_request_free_mem_region(), devm_memremap_pages(), devm_memunmap_pages(), and devm_release_mem_region() when the resources can be tied to a `struct device`.

The overall migration steps are similar to migrating NUMA pages within system memory (see *Page migration*) but the steps are split between device driver specific code and shared common code:

1. mmap_read_lock()

   The device driver has to pass a struct vm_area_struct to migrate_vma_setup() so the mmap_read_lock() or mmap_write_lock() needs to be held for the duration of the migration.

2. migrate_vma_setup(struct migrate_vma *args)

   The device driver initializes the struct migrate_vma fields and passes the pointer to migrate_vma_setup(). The args->flags field is used to filter which source pages should be migrated. For example, setting MIGRATE_VMA_SELECT_SYSTEM will only migrate system memory and MIGRATE_VMA_SELECT_DEVICE_PRIVATE will only migrate pages residing in device private memory. If the latter flag is set, the args->pgmap_owner field is used to identify device private pages owned by the driver. This avoids trying to migrate device private pages residing in other devices. Currently only anonymous private VMA ranges can be migrated to or from system memory and device private memory.

   One of the first steps migrate_vma_setup() does is to invalidate other device's MMUs with the mmu_notifier_invalidate_range_start(() and mmu_notifier_invalidate_range_end() calls around the page table walks to fill in the args->src array with PFNs to be migrated. The invalidate_range_start() callback is passed a struct mmu_notifier_range with the event field set to MMU_NOTIFY_MIGRATE and the owner field set to the args->pgmap_owner field passed to migrate_vma_setup(). This is allows the device driver to skip the invalidation callback and only invalidate device private MMU mappings that are actually migrating. This is explained more in the next section.

   While walking the page tables, a pte_none() or is_zero_pfn() entry results in a valid "zero" PFN stored in the args->src array. This lets the driver allocate device private memory and clear it instead of copying a page of zeros. Valid PTE entries to system memory or device private struct pages will be locked with lock_page(), isolated from the LRU (if system memory since device private pages are not on the LRU), unmapped from the process, and a special migration PTE is inserted in place of the original PTE. migrate_vma_setup() also clears the args->dst array.

3. The device driver allocates destination pages and copies source pages to destination pages.

   The driver checks each src entry to see if the MIGRATE_PFN_MIGRATE bit is set and skips entries that are not migrating. The device driver can also choose to skip migrating a page by not filling in the dst array for that page.

   The driver then allocates either a device private struct page or a system memory page, locks the page with lock_page(), and fills in the dst array entry with:

   ```
   dst[i] = migrate_pfn(page_to_pfn(dpage));
   ```

   Now that the driver knows that this page is being migrated, it can invalidate device private MMU mappings and copy device private memory to system memory or another device private page. The core Linux kernel handles CPU page table invalidations so the device driver only has to invalidate its own MMU mappings.

   The driver can use migrate_pfn_to_page(src[i]) to get the struct page of the source and either copy the source page to the destination or clear the destination device private memory if the pointer is NULL meaning the source page was not populated in system memory.

4. migrate_vma_pages()

This step is where the migration is actually "committed".

If the source page was a `pte_none()` or `is_zero_pfn()` page, this is where the newly allocated page is inserted into the CPU's page table. This can fail if a CPU thread faults on the same page. However, the page table is locked and only one of the new pages will be inserted. The device driver will see that the `MIGRATE_PFN_MIGRATE` bit is cleared if it loses the race.

If the source page was locked, isolated, etc. the source `struct page` information is now copied to destination `struct page` finalizing the migration on the CPU side.

5. Device driver updates device MMU page tables for pages still migrating, rolling back pages not migrating.

   If the `src` entry still has `MIGRATE_PFN_MIGRATE` bit set, the device driver can update the device MMU and set the write enable bit if the `MIGRATE_PFN_WRITE` bit is set.

6. `migrate_vma_finalize()`

   This step replaces the special migration page table entry with the new page's page table entry and releases the reference to the source and destination `struct page`.

7. `mmap_read_unlock()`

   The lock can now be released.

2.6.8 Exclusive access memory

Some devices have features such as atomic PTE bits that can be used to implement atomic access to system memory. To support atomic operations to a shared virtual memory page such a device needs access to that page which is exclusive of any userspace access from the CPU. The `make_device_exclusive_range()` function can be used to make a memory range inaccessible from userspace.

This replaces all mappings for pages in the given range with special swap entries. Any attempt to access the swap entry results in a fault which is resovled by replacing the entry with the original mapping. A driver gets notified that the mapping has been changed by MMU notifiers, after which point it will no longer have exclusive access to the page. Exclusive access is guaranteed to last until the driver drops the page lock and page reference, at which point any CPU faults on the page may proceed as described.

2.6.9 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.

2.7 hwpoison

2.7.1 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 expectation is that most applications won't do that, but some very specialized ones might.

2.7.2 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.

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.

2.7.3 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

2.7.4 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. Once any hardware memory failure happens, this feature is disabled.

  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.

2.7.5 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

2.7.6 Limitations

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

--- Andi Kleen, Oct 2009

2.8 Hugetlbfs Reservation

2.8.1 Overview

Huge pages as described at Documentation/admin-guide/mm/hugetlbpage.rst are typically pre-allocated 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.

2.8.2 Audience

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

2.8.3 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.

2.8.4 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.

2.8.5 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'.

2.8.6 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_hugetlb_folio():

```
struct folio *alloc_hugetlb_folio(struct vm_area_struct *vma,
                                  unsigned long addr, int avoid_reserve)
```

alloc_hugetlb_folio 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_hugetlb_folio 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_hugetlb_folio().
- 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 hugetlb folio, (folio)->_hugetlb_subpool is set to the value of the subpool associated with the page if it exists. This will be used for subpool accounting when the folio 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_hugetlb_folio() and the call to vma_commit_reservation() after the folio 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.

2.8.7 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).

2.8.8 Freeing Huge Pages

Huge pages are freed by free_huge_folio(). It is only passed a pointer to the folio as it is called from the generic MM code. 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.

2.8.9 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.

2.8.10 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.

2.8.11 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.

2.8.12 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.

2.8.13 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_folio 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.

2.8.14 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.

2.8.15 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

2.9 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

2.9.1 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 ksm_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 ksm_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 **ksm_scan**
cursor for scanning

Definition:

```
struct ksm_scan {
    struct ksm_mm_slot *mm_slot;
    unsigned long address;
    struct ksm_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.

-- Izik Eidus, Hugh Dickins, 17 Nov 2009

2.10 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 two memory models: FLATMEM 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.

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.

2.10.1 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.

An architecture 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.

2.10.2 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 sparse_init() to initialize the memory sections and the memory maps.

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 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 `vmemmap_alloc_block_buf()` helper to allocate memory map on the persistent memory device.

2.10.3 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 *Heterogeneous Memory Management (HMM)*.

- 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.

2.11 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  {}
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 seeing 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().

2.12 Multi-Gen LRU

The multi-gen LRU is an alternative LRU implementation that optimizes page reclaim and improves performance under memory pressure. Page reclaim decides the kernel's caching policy and ability to overcommit memory. It directly impacts the kswapd CPU usage and RAM efficiency.

2.12.1 Design overview

Objectives

The design objectives are:

- Good representation of access recency
- Try to profit from spatial locality
- Fast paths to make obvious choices
- Simple self-correcting heuristics

The representation of access recency is at the core of all LRU implementations. In the multi-gen LRU, each generation represents a group of pages with similar access recency. Generations establish a (time-based) common frame of reference and therefore help make better choices, e.g., between different memcgs on a computer or different computers in a data center (for job scheduling).

Exploiting spatial locality improves efficiency when gathering the accessed bit. A rmap walk targets a single page and does not try to profit from discovering a young PTE. A page table walk can sweep all the young PTEs in an address space, but the address space can be too sparse to make a profit. The key is to optimize both methods and use them in combination.

Fast paths reduce code complexity and runtime overhead. Unmapped pages do not require TLB flushes; clean pages do not require writeback. These facts are only helpful when other conditions, e.g., access recency, are similar. With generations as a common frame of reference, additional factors stand out. But obvious choices might not be good choices; thus self-correction is necessary.

The benefits of simple self-correcting heuristics are self-evident. Again, with generations as a common frame of reference, this becomes attainable. Specifically, pages in the same generation can be categorized based on additional factors, and a feedback loop can statistically compare the refault percentages across those categories and infer which of them are better choices.

Assumptions

The protection of hot pages and the selection of cold pages are based on page access channels and patterns. There are two access channels:

- Accesses through page tables
- Accesses through file descriptors

The protection of the former channel is by design stronger because:

1. The uncertainty in determining the access patterns of the former channel is higher due to the approximation of the accessed bit.
2. The cost of evicting the former channel is higher due to the TLB flushes required and the likelihood of encountering the dirty bit.
3. The penalty of underprotecting the former channel is higher because applications usually do not prepare themselves for major page faults like they do for blocked I/O. E.g., GUI applications commonly use dedicated I/O threads to avoid blocking rendering threads.

There are also two access patterns:

- Accesses exhibiting temporal locality
- Accesses not exhibiting temporal locality

For the reasons listed above, the former channel is assumed to follow the former pattern unless `VM_SEQ_READ` or `VM_RAND_READ` is present, and the latter channel is assumed to follow the latter pattern unless outlying refaults have been observed.

2.12.2 Workflow overview

Evictable pages are divided into multiple generations for each `lruvec`. The youngest generation number is stored in `lrugen->max_seq` for both anon and file types as they are aged on an equal footing. The oldest generation numbers are stored in `lrugen->min_seq[]` separately for anon and file types as clean file pages can be evicted regardless of swap constraints. These three variables are monotonically increasing.

Generation numbers are truncated into `order_base_2(MAX_NR_GENS+1)` bits in order to fit into the gen counter in `folio->flags`. Each truncated generation number is an index to `lrugen->folios[]`. The sliding window technique is used to track at least `MIN_NR_GENS` and at most `MAX_NR_GENS` generations. The gen counter stores a value within `[1, MAX_NR_GENS]` while a page is on one of `lrugen->folios[]`; otherwise it stores zero.

Each generation is divided into multiple tiers. A page accessed `N` times through file descriptors is in tier `order_base_2(N)`. Unlike generations, tiers do not have dedicated `lrugen->folios[]`. In contrast to moving across generations, which requires the LRU lock, moving across tiers only involves atomic operations on `folio->flags` and therefore has a negligible cost. A feedback loop modeled after the PID controller monitors refaults over all the tiers from anon and file types and decides which tiers from which types to evict or protect. The desired effect is to balance refault percentages between anon and file types proportional to the swappiness level.

There are two conceptually independent procedures: the aging and the eviction. They form a closed-loop system, i.e., the page reclaim.

Aging

The aging produces young generations. Given an `lruvec`, it increments `max_seq` when `max_seq-min_seq+1` approaches `MIN_NR_GENS`. The aging promotes hot pages to the youngest generation when it finds them accessed through page tables; the demotion of cold pages happens consequently when it increments `max_seq`. The aging uses page table walks and rmap walks to find young PTEs. For the former, it iterates `lruvec_memcg()->mm_list` and calls `walk_page_range()` with each `mm_struct` on this list to scan PTEs, and after each iteration, it increments `max_seq`. For the latter, when the eviction walks the rmap and finds a young PTE, the

aging scans the adjacent PTEs. For both, on finding a young PTE, the aging clears the accessed bit and updates the gen counter of the page mapped by this PTE to `(max_seq%MAX_NR_GENS)+1`.

Eviction

The eviction consumes old generations. Given an `lruvec`, it increments `min_seq` when `lrugen->folios[]` indexed by `min_seq%MAX_NR_GENS` becomes empty. To select a type and a tier to evict from, it first compares `min_seq[]` to select the older type. If both types are equally old, it selects the one whose first tier has a lower refault percentage. The first tier contains single-use unmapped clean pages, which are the best bet. The eviction sorts a page according to its gen counter if the aging has found this page accessed through page tables and updated its gen counter. It also moves a page to the next generation, i.e., `min_seq+1`, if this page was accessed multiple times through file descriptors and the feedback loop has detected outlying refaults from the tier this page is in. To this end, the feedback loop uses the first tier as the baseline, for the reason stated earlier.

Working set protection

Each generation is timestamped at birth. If `lru_gen_min_ttl` is set, an `lruvec` is protected from the eviction when its oldest generation was born within `lru_gen_min_ttl` milliseconds. In other words, it prevents the working set of `lru_gen_min_ttl` milliseconds from getting evicted. The OOM killer is triggered if this working set cannot be kept in memory.

This time-based approach has the following advantages:

1. It is easier to configure because it is agnostic to applications and memory sizes.
2. It is more reliable because it is directly wired to the OOM killer.

`mm_struct` list

An `mm_struct` list is maintained for each memcg, and an `mm_struct` follows its owner task to the new memcg when this task is migrated.

A page table walker iterates `lruvec_memcg()->mm_list` and calls `walk_page_range()` with each `mm_struct` on this list to scan PTEs. When multiple page table walkers iterate the same list, each of them gets a unique `mm_struct`, and therefore they can run in parallel.

Page table walkers ignore any misplaced pages, e.g., if an `mm_struct` was migrated, pages left in the previous memcg will be ignored when the current memcg is under reclaim. Similarly, page table walkers will ignore pages from nodes other than the one under reclaim.

This infrastructure also tracks the usage of `mm_struct` between context switches so that page table walkers can skip processes that have been sleeping since the last iteration.

Rmap/PT walk feedback

Searching the rmap for PTEs mapping each page on an LRU list (to test and clear the accessed bit) can be expensive because pages from different VMAs (PA space) are not cache friendly to the rmap (VA space). For workloads mostly using mapped pages, searching the rmap can incur the highest CPU cost in the reclaim path.

`lru_gen_look_around()` exploits spatial locality to reduce the trips into the rmap. It scans the adjacent PTEs of a young PTE and promotes hot pages. If the scan was done cacheline efficiently, it adds the PMD entry pointing to the PTE table to the Bloom filter. This forms a feedback loop between the eviction and the aging.

Bloom filters

Bloom filters are a space and memory efficient data structure for set membership test, i.e., test if an element is not in the set or may be in the set.

In the eviction path, specifically, in `lru_gen_look_around()`, if a PMD has a sufficient number of hot pages, its address is placed in the filter. In the aging path, set membership means that the PTE range will be scanned for young pages.

Note that Bloom filters are probabilistic on set membership. If a test is false positive, the cost is an additional scan of a range of PTEs, which may yield hot pages anyway. Parameters of the filter itself can control the false positive rate in the limit.

PID controller

A feedback loop modeled after the Proportional-Integral-Derivative (PID) controller monitors refaults over anon and file types and decides which type to evict when both types are available from the same generation.

The PID controller uses generations rather than the wall clock as the time domain because a CPU can scan pages at different rates under varying memory pressure. It calculates a moving average for each new generation to avoid being permanently locked in a suboptimal state.

Memcg LRU

An memcg LRU is a per-node LRU of memcgs. It is also an LRU of LRUs, since each node and memcg combination has an LRU of folios (see `mem_cgroup_lruvec()`). Its goal is to improve the scalability of global reclaim, which is critical to system-wide memory overcommit in data centers. Note that memcg LRU only applies to global reclaim.

The basic structure of an memcg LRU can be understood by an analogy to the active/inactive LRU (of folios):

1. It has the young and the old (generations), i.e., the counterparts to the active and the inactive;
2. The increment of `max_seq` triggers promotion, i.e., the counterpart to activation;
3. Other events trigger similar operations, e.g., offlining an memcg triggers demotion, i.e., the counterpart to deactivation.

In terms of global reclaim, it has two distinct features:

1. Sharding, which allows each thread to start at a random memcg (in the old generation) and improves parallelism;
2. Eventual fairness, which allows direct reclaim to bail out at will and reduces latency without affecting fairness over some time.

In terms of traversing memcgs during global reclaim, it improves the best-case complexity from O(n) to O(1) and does not affect the worst-case complexity O(n). Therefore, on average, it has a sublinear complexity.

Summary

The multi-gen LRU (of folios) can be disassembled into the following parts:

- Generations
- Rmap walks
- Page table walks via `mm_struct` list
- Bloom filters for rmap/PT walk feedback
- PID controller for refault feedback

The aging and the eviction form a producer-consumer model; specifically, the latter drives the former by the sliding window over generations. Within the aging, rmap walks drive page table walks by inserting hot densely populated page tables to the Bloom filters. Within the eviction, the PID controller uses refaults as the feedback to select types to evict and tiers to protect.

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 visible 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 emulation 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 ordered 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. 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). These only have an effect when `vm.overcommit_memory` is set to 2.

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 moving 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.

Also see *Heterogeneous Memory Management (HMM)* for migrating pages to or from device private memory.

The main intent of page migration is to reduce the latency of memory accesses 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 a 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 https://github.com/numactl/numactl.git). 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 only provides 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. For example, A NUMA profiler may 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 CPUSETS). Cpusets allow 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 from encountering the page.

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

3. The migrate_pages() function is called which attempts to do the migration. It will call the function to allocate the new folio for each folio 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 up-to-date) page immediately block 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 or wait for the migration page table entry to be removed.

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 inserted into the page table, then replace them with real ptes. Doing so will enable access for user space processes not already waiting for the page lock.
17. 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.
18. 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 migration originally aimed for reducing the latency of memory accesses for NUMA, compaction also uses migration to create high-order pages. For compaction purposes, it is also useful to be able to move non-LRU pages, such as zsmalloc and virtio-balloon pages.

If a driver wants to make its pages movable, it should define a *struct movable_operations*. It then needs to call __SetPageMovable() on each page that it may be able to move. This uses the `page->mapping` field, so this field is not available for the driver to use for other purposes.

2.15.4 Monitoring Migration

The following events (counters) can be used to monitor page migration.

1. PGMIGRATE_SUCCESS: Normal page migration success. Each count means that a page was migrated. If the page was a non-THP and non-hugetlb page, then this counter is increased by one. If the page was a THP or hugetlb, then this counter is increased by the number of THP or hugetlb subpages. For example, migration of a single 2MB THP that has 4KB-size base pages (subpages) will cause this counter to increase by 512.

2. PGMIGRATE_FAIL: Normal page migration failure. Same counting rules as for PGMIGRATE_SUCCESS, above: this will be increased by the number of subpages, if it was a THP or hugetlb.
3. THP_MIGRATION_SUCCESS: A THP was migrated without being split.
4. THP_MIGRATION_FAIL: A THP could not be migrated nor it could be split.
5. THP_MIGRATION_SPLIT: A THP was migrated, but not as such: first, the THP had to be split. After splitting, a migration retry was used for its sub-pages.

THP_MIGRATION_* events also update the appropriate PGMIGRATE_SUCCESS or PGMIGRATE_FAIL events. For example, a THP migration failure will cause both THP_MIGRATION_FAIL and PGMIGRATE_FAIL to increase.

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

struct **movable_operations**

Driver page migration

Definition:

```
struct movable_operations {
    bool (*isolate_page)(struct page *, isolate_mode_t);
    int (*migrate_page)(struct page *dst, struct page *src, enum migrate_mode);
    void (*putback_page)(struct page *);
};
```

Members

`isolate_page`

The VM calls this function to prepare the page to be moved. The page is locked and the driver should not unlock it. The driver should return `true` if the page is movable and `false` if it is not currently movable. After this function returns, the VM uses the page->lru field, so the driver must preserve any information which is usually stored here.

`migrate_page`

After isolation, the VM calls this function with the isolated **src** page. The driver should copy the contents of the **src** page to the **dst** page and set up the fields of **dst** page. Both pages are locked. If page migration is successful, the driver should call __ClearPageMovable(**src**) and return MIGRATEPAGE_SUCCESS. If the driver cannot migrate the page at the moment, it can return -EAGAIN. The VM interprets this as a temporary migration failure and will retry it later. Any other error value is a permanent migration failure and migration will not be retried. The driver shouldn't touch the **src->lru** field while in the migrate_page() function. It may write to **dst->lru**.

`putback_page`

If migration fails on the isolated page, the VM informs the driver that the page is no longer a candidate for migration by calling this function. The driver should put the isolated page back into its own data structure.

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 ref-counted, 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 behaviour 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 by default. So, if you'd like to use it, you need to add ”page_owner=on” to your boot cmdline. If the kernel is built with page owner and page owner is disabled in runtime due to not 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.

Although enabling page owner increases kernel size by several kilobytes, most of this code is outside page allocator and its hot path. 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 caught 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/mm
make page_owner_sort
```

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

3) Do the job that 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
```

The general output of `page_owner_full.txt` is as follows:

```
    Page allocated via order XXX, ...
    PFN XXX ...
    // Detailed stack

    Page allocated via order XXX, ...
    PFN XXX ...
    // Detailed stack
By default, it will do full pfn dump, to start with a given pfn,
page_owner supports fseek.

FILE *fp = fopen("/sys/kernel/debug/page_owner", "r");
fseek(fp, pfn_start, SEEK_SET);
```

The page_owner_sort tool ignores PFN rows, puts the remaining rows in buf, uses regexp to extract the page order value, counts the times and pages of buf, and finally sorts them according to the parameter(s).

See the result about who allocated each page in the sorted_page_owner.txt. General output:

```
XXX times, XXX pages:
Page allocated via order XXX, ...
// Detailed stack
```

By default, page_owner_sort is sorted according to the times of buf. If you want to sort by the page nums of buf, use the -m parameter. The detailed parameters are:

fundamental function:

```
Sort:
        -a              Sort by memory allocation time.
        -m              Sort by total memory.
        -p              Sort by pid.
        -P              Sort by tgid.
        -n              Sort by task command name.
        -r              Sort by memory release time.
        -s              Sort by stack trace.
        -t              Sort by times (default).
        --sort <order>  Specify sorting order.  Sorting syntax is [+|-
↪]key[,[+|-]key[,...]].
                        Choose a key from the **STANDARD FORMAT␣
↪SPECIFIERS** section. The "+" is
                        optional since default direction is increasing␣
↪numerical or lexicographic
                        order. Mixed use of abbreviated and complete-form␣
↪of keys is allowed.

        Examples:
                        ./page_owner_sort <input> <output> --sort=n,+pid,-
↪tgid
                        ./page_owner_sort <input> <output> --sort=at
```

additional function:

```
Cull:
        --cull <rules>
                        Specify culling rules.Culling syntax is key[,key[,.
↪..]].Choose a
                        multi-letter key from the **STANDARD FORMAT␣
↪SPECIFIERS** section.

        <rules> is a single argument in the form of a comma-separated list,
        which offers a way to specify individual culling rules.  The␣
↪recognized
        keywords are described in the **STANDARD FORMAT SPECIFIERS**␣
↪section below.
```

```
        <rules> can be specified by the sequence of keys k1,k2, ..., as␣
↪described in
        the STANDARD SORT KEYS section below. Mixed use of abbreviated and
        complete-form of keys is allowed.

        Examples:
                        ./page_owner_sort <input> <output> --
↪cull=stacktrace
                        ./page_owner_sort <input> <output> --cull=st,pid,
↪name
                        ./page_owner_sort <input> <output> --cull=n,f

Filter:
        -f              Filter out the information of blocks whose memory␣
↪has been released.

Select:
        --pid <pidlist>         Select by pid. This selects the blocks␣
↪whose process ID
                                numbers appear in <pidlist>.
        --tgid <tgidlist>       Select by tgid. This selects the blocks␣
↪whose thread
                                group ID numbers appear in <tgidlist>.
        --name <cmdlist>        Select by task command name. This selects␣
↪the blocks whose
                                task command name appear in <cmdlist>.

        <pidlist>, <tgidlist>, <cmdlist> are single arguments in the form␣
↪of a comma-separated list,
        which offers a way to specify individual selecting rules.


        Examples:
                        ./page_owner_sort <input> <output> --pid=1
                        ./page_owner_sort <input> <output> --tgid=1,2,3
                        ./page_owner_sort <input> <output> --name name1,
↪name2
```

2.17.3 STANDARD FORMAT SPECIFIERS

```
For --sort option:

      KEY             LONG            DESCRIPTION
      p               pid             process ID
      tg              tgid            thread group ID
      n               name            task command name
      st              stacktrace      stack trace of the page allocation
      T               txt             full text of block
      ft              free_ts         timestamp of the page when it was␣
↪released
```

```
      at              alloc_ts        timestamp of the page when it was␣
↪allocated
      ator            allocator       memory allocator for pages

For --cull option:

      KEY             LONG            DESCRIPTION
      p               pid             process ID
      tg              tgid            thread group ID
      n               name            task command name
      f               free            whether the page has been released or not
      st              stacktrace      stack trace of the page allocation
      ator            allocator       memory allocator for pages
```

2.18 Page Table Check

2.18.1 Introduction

Page table check allows to harden the kernel by ensuring that some types of the memory corruptions are prevented.

Page table check performs extra verifications at the time when new pages become accessible from the userspace by getting their page table entries (PTEs PMDs etc.) added into the table.

In case of detected corruption, the kernel is crashed. There is a small performance and memory overhead associated with the page table check. Therefore, it is disabled by default, but can be optionally enabled on systems where the extra hardening outweighs the performance costs. Also, because page table check is synchronous, it can help with debugging double map memory corruption issues, by crashing kernel at the time wrong mapping occurs instead of later which is often the case with memory corruptions bugs.

2.18.2 Double mapping detection logic

Current Mapping	New mapping	Permissions	Rule
Anonymous	Anonymous	Read	Allow
Anonymous	Anonymous	Read / Write	Prohibit
Anonymous	Named	Any	Prohibit
Named	Anonymous	Any	Prohibit
Named	Named	Any	Allow

2.18.3 Enabling Page Table Check

Build kernel with:

- PAGE_TABLE_CHECK=y Note, it can only be enabled on platforms where ARCH_SUPPORTS_PAGE_TABLE_CHECK is available.
- Boot with 'page_table_check=on' kernel parameter.

Optionally, build kernel with PAGE_TABLE_CHECK_ENFORCED in order to have page table support without extra kernel parameter.

2.18.4 Implementation notes

We specifically decided not to use VMA information in order to avoid relying on MM states (except for limited ”struct page” info). The page table check is a separate from Linux-MM state machine that verifies that the user accessible pages are not falsely shared.

PAGE_TABLE_CHECK depends on EXCLUSIVE_SYSTEM_RAM. The reason is that without EXCLUSIVE_SYSTEM_RAM, users are allowed to map arbitrary physical memory regions into the userspace via /dev/mem. At the same time, pages may change their properties (e.g., from anonymous pages to named pages) while they are still being mapped in the userspace, leading to ”corruption” detected by the page table check.

Even with EXCLUSIVE_SYSTEM_RAM, I/O pages may be still allowed to be mapped via /dev/mem. However, these pages are always considered as named pages, so they won't break the logic used in the page table check.

2.19 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.

2.20 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/mm/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.

2.20.1 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)

Multiple blocks of options for all slabs or selected slabs can be given, with blocks of options delimited by ';'. The last of ”all slabs” blocks is applied to all slabs except those that match one of the ”select slabs” block. Options of the first ”select slabs” blocks that matches the slab's name are applied.

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
```

You can apply different options to different list of slab names, using blocks of options. This will enable red zoning for dentry and user tracking for kmalloc. All other slabs will not get any debugging enabled:

```
slub_debug=Z,dentry;U,kmalloc-*
```

You can also enable options (e.g. sanity checks and poisoning) for all caches except some that are deemed too performance critical and don't need to be debugged by specifying global debug options followed by a list of slab names with ”-” as options:

```
slub_debug=FZ;-,zs_handle,zspage
```

The state of each debug option for a slab can be found in the respective files under:

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

If the file contains 1, the option is enabled, 0 means disabled. The debug options from the `slub_debug` parameter translate to the following files:

```
F       sanity_checks
Z       red_zone
P       poison
U       store_user
T       trace
A       failslab
```

failslab file is writable, so writing 1 or 0 will enable or disable the option at runtime. Write returns -EINVAL if cache is an alias. 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: Right 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
  [<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 height

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).

DebugFS files for SLUB

For more information about current state of SLUB caches with the user tracking debug option enabled, debugfs files are available, typically under /sys/kernel/debug/slab/<cache>/ (created only for caches with enabled user tracking). There are 2 types of these files with the following debug information:

1. alloc_traces:

```
Prints information about unique allocation traces of the currently
allocated objects. The output is sorted by frequency of each trace.

Information in the output:
Number of objects, allocating function, possible memory wastage of
kmalloc objects(total/per-object), minimal/average/maximal jiffies
since alloc, pid range of the allocating processes, cpu mask of
allocating cpus, numa node mask of origins of memory, and stack trace.

Example:::

338 pci_alloc_dev+0x2c/0xa0 waste=521872/1544 age=290837/291891/293509␣
↪pid=1 cpus=106 nodes=0-1
    __kmem_cache_alloc_node+0x11f/0x4e0
    kmalloc_trace+0x26/0xa0
    pci_alloc_dev+0x2c/0xa0
```

```
    pci_scan_single_device+0xd2/0x150
    pci_scan_slot+0xf7/0x2d0
    pci_scan_child_bus_extend+0x4e/0x360
    acpi_pci_root_create+0x32e/0x3b0
    pci_acpi_scan_root+0x2b9/0x2d0
    acpi_pci_root_add.cold.11+0x110/0xb0a
    acpi_bus_attach+0x262/0x3f0
    device_for_each_child+0xb7/0x110
    acpi_dev_for_each_child+0x77/0xa0
    acpi_bus_attach+0x108/0x3f0
    device_for_each_child+0xb7/0x110
    acpi_dev_for_each_child+0x77/0xa0
    acpi_bus_attach+0x108/0x3f0
```

2. free_traces:

```
Prints information about unique freeing traces of the currently allocated
objects. The freeing traces thus come from the previous life-cycle of the
objects and are reported as not available for objects allocated for the␣
 ↪first
time. The output is sorted by frequency of each trace.

Information in the output:
Number of objects, freeing function, minimal/average/maximal jiffies since␣
 ↪free,
pid range of the freeing processes, cpu mask of freeing cpus, and stack␣
 ↪trace.

Example:::

1980 <not-available> age=4294912290 pid=0 cpus=0
51 acpi_ut_update_ref_count+0x6a6/0x782 age=236886/237027/237772 pid=1␣
 ↪cpus=1
    kfree+0x2db/0x420
    acpi_ut_update_ref_count+0x6a6/0x782
    acpi_ut_update_object_reference+0x1ad/0x234
    acpi_ut_remove_reference+0x7d/0x84
    acpi_rs_get_prt_method_data+0x97/0xd6
    acpi_get_irq_routing_table+0x82/0xc4
    acpi_pci_irq_find_prt_entry+0x8e/0x2e0
    acpi_pci_irq_lookup+0x3a/0x1e0
    acpi_pci_irq_enable+0x77/0x240
    pcibios_enable_device+0x39/0x40
    do_pci_enable_device.part.0+0x5d/0xe0
    pci_enable_device_flags+0xfc/0x120
    pci_enable_device+0x13/0x20
    virtio_pci_probe+0x9e/0x170
    local_pci_probe+0x48/0x80
    pci_device_probe+0x105/0x1c0
```

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

2.21 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 PTE with pointer to its PTE table lock, or returns NULL if no PTE table;
- **pte_offset_map_nolock()**
  maps PTE, returns pointer to PTE with pointer to its PTE table lock (not taken), or returns NULL if no PTE table;
- **pte_offset_map()**
  maps PTE, returns pointer to PTE, or returns NULL if no PTE table;
- **pte_unmap()**
  unmaps PTE table;
- **pte_unmap_unlock()**
  unlocks and unmaps PTE table;
- **pte_alloc_map_lock()**
  allocates PTE table if needed and takes its lock, returns pointer to PTE with pointer to its lock, or returns NULL if allocation failed;
- **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 are guarded 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).

2.21.1 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;

2.21.2 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 pagetable_pte_ctor() and pagetable_pte_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 pagetable_pmd_ctor() call on PMD table allocation and pagetable_pmd_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: pagetable_pte_ctor() and pagetable_pmd_ctor() can fail -- it must be handled properly.

2.21.3 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 pagetable_pte_ctor() for PTE table and in pagetable_pmd_ctor() for PMD table.

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

2.22 Transparent Hugepage Support

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

2.22.1 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)

2.22.2 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.

2.22.3 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;
```

2.22.4 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.

2.22.5 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 the folio->_refcount.
- ->_refcount in tail pages is always zero: get_page_unless_zero() never succeeds on tail pages.
- map/unmap of a PMD entry for the whole THP increment/decrement folio->_entire_mapcount and also increment/decrement folio->_nr_pages_mapped by ENTIRELY_MAPPED when _entire_mapcount goes from -1 to 0 or 0 to -1.
- map/unmap of individual pages with PTE entry increment/decrement page->_mapcount and also increment/decrement folio->_nr_pages_mapped when page->_mapcount goes from -1 to 0 or 0 to -1 as this counts the number of pages mapped by PTE.

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.

2.22.6 Partial unmap and deferred_split_folio()

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 folio_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_folio() is used to queue a folio for splitting. The splitting itself will happen when we get memory pressure via shrinker interface.

2.23 Unevictable LRU Infrastructure

2.23.1 Introduction

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

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?”.

2.23.2 The Unevictable LRU

The Unevictable LRU facility adds an additional LRU list to track unevictable folios and to hide these folios from vmscan. This mechanism is based on a patch by Larry Woodman of Red Hat to address several scalability problems with folio 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 node. 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 owned by tmpfs with the noswap mount option.
- 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 LRU Folio List

The Unevictable LRU folio list is a lie. It was never an LRU-ordered list, but a companion to the LRU-ordered anonymous and file, active and inactive folio lists; and now it is not even a folio list. But following familiar convention, here in this document and in the source, we often imagine it as a fifth LRU folio list.

The Unevictable LRU infrastructure consists of an additional, per-node, LRU list called the "unevictable" list and an associated folio flag, PG_unevictable, to indicate that the folio 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 folio resides when PG_lru is set.

The Unevictable LRU infrastructure maintains unevictable folios as if they were on an additional LRU list for a few reasons:

(1) We get to "treat unevictable folios just like we treat other folios 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 folios between nodes for memory defragmentation, workload management and memory hotplug. The Linux kernel can only migrate folios that it can successfully isolate from the LRU lists (or "Movable" pages: outside of consideration here). If we were to maintain folios elsewhere than on an LRU-like list, where they can be detected by folio_isolate_lru(), we would prevent their migration.

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

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

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-node unevictable list as a result of the "arrayification" of the per-node 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 folio_evictable() in mm/internal.h determines whether a folio 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 (e.g. 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 folios during a reclamation scan.

On an unlock action (such as SHM_UNLOCK), the unlocker (e.g. 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.

folio_evictable() also checks for mlocked folios by calling folio_test_mlocked(), which is set when a folio is faulted into a VM_LOCKED VMA, or found in a VMA being VM_LOCKED.

Vmscan's Handling of Unevictable Folios

If unevictable folios are culled in the fault path, or moved to the unevictable list at mlock() or mmap() time, vmscan will not encounter the folios 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 an unevictable folio on one of the regular active/inactive LRU lists for vmscan to deal with. vmscan checks for such folios in all of the shrink_{active|inactive|page}_list() functions and will ”cull” such folios that it encounters: that is, it diverts those folios to the unevictable list for the memory cgroup and node being scanned.

There may be situations where a folio is mapped into a VM_LOCKED VMA, but the folio does not have the mlocked flag set. Such folios will make it all the way to shrink_active_list() or shrink_page_list() where they will be detected when vmscan walks the reverse map in folio_referenced() or try_to_unmap(). The folio is culled to the unevictable list when it is released by the shrinker.

To ”cull” an unevictable folio, vmscan simply puts the folio back on the LRU list using folio_putback_lru() - the inverse operation to folio_isolate_lru() - after dropping the folio lock. Because the condition which makes the folio unevictable may change once the folio is unlocked, __pagevec_lru_add_fn() will recheck the unevictable state of a folio before placing it on the unevictable list.

2.23.3 MLOCKED Pages

The unevictable folio 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 (Rik van Riel had the same idea three years earlier). But 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 detect 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 when munlocking, to determine whether any other VM_LOCKED VMAs still mapped the page.

However, walking the reverse map for each page when munlocking was ugly and inefficient, and could lead to catastrophic contention on a file's rmap lock, when many processes which had it mlocked were trying to exit. In 5.18, the idea of keeping mlock_count in Unevictable LRU list link field was revived and put to work, without preventing the migration of mlocked pages.

This is why the ”Unevictable LRU list” cannot be a linked list of pages now; but there was no use for that linked list anyway - though its size is maintained for meminfo.

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()/mlock2()/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 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 by folio_referenced() or try_to_unmap().

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()/mlock2()/mlockall() System Call Handling

mlock(), mlock2() and mlockall() system call handlers proceed to 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. Any pages already present in the VMA are then marked as mlocked by mlock_folio() via mlock_pte_range() via walk_page_range() via mlock_vma_pages_range().

Before returning from the system call, do_mlock() or mlockall() will call __mm_populate() to fault in the remaining pages via get_user_pages() and to mark those pages as mlocked as they are faulted.

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, they will be handled in the fault path - which is also how mlock2()'s MLOCK_ONFAULT areas are handled.

For each PTE (or PMD) being faulted into a VMA, the page add rmap function calls mlock_vma_folio(), which calls mlock_folio() when the VMA is VM_LOCKED (unless it is a PTE mapping of a part of a transparent huge page). Or when it is a newly allocated anonymous page, folio_add_lru_vma() calls mlock_new_folio() instead: similar to mlock_folio(), but can make better judgments, since this page is held exclusively and known not to be on LRU yet.

mlock_folio() sets PG_mlocked immediately, then places the page on the CPU's mlock folio batch, to batch up the rest of the work to be done under lru_lock by __mlock_folio(). __mlock_folio() sets PG_unevictable, initializes mlock_count and moves the page to unevictable state ("the unevictable LRU", but with mlock_count in place of LRU threading). Or if the page was already PG_lru and PG_unevictable and PG_mlocked, it simply increments the mlock_count.

But in practice that may not work ideally: the page may not yet be on an LRU, or it may have been temporarily isolated from LRU. In such cases the mlock_count field cannot be touched, but will be set to 0 later when __munlock_folio() returns the page to "LRU". Races prohibit mlock_count from being set to 1 then: rather than risk stranding a page indefinitely as unevictable, always err with mlock_count on the low side, so that when munlocked the page will be rescued to an evictable LRU, then perhaps be mlocked again later if vmscan finds it in a VM_LOCKED VMA.

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. But __mm_populate() includes hugetlbfs ranges, allocating the huge pages and populating 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. __mm_populate() includes these ranges, populating the PTEs if not already populated.

4) VMAs with VM_MIXEDMAP set are not marked VM_LOCKED, but __mm_populate() includes these ranges, populating the PTEs if not already populated.

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 mlock_fixup() function as mlock(), mlock2() and mlockall() system calls are. If called to munlock 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, those VMAs will be ignored for munlock.

If the VMA is VM_LOCKED, mlock_fixup() again attempts to merge or split off the specified range. All pages in the VMA are then munlocked by munlock_folio() via mlock_pte_range() via

walk_page_range() via mlock_vma_pages_range() - the same function used when mlocking a VMA range, with new flags for the VMA indicating that it is munlock() being performed.

munlock_folio() uses the mlock pagevec to batch up work to be done under lru_lock by __munlock_folio(). __munlock_folio() decrements the folio's mlock_count, and when that reaches 0 it clears the mlocked flag and clears the unevictable flag, moving the folio from unevictable state to the inactive LRU.

But in practice that may not work ideally: the folio may not yet have reached "the unevictable LRU", or it may have been temporarily isolated from it. In those cases its mlock_count field is unusable and must be assumed to be 0: so that the folio will be rescued to an evictable LRU, then perhaps be mlocked again later if vmscan finds it in a VM_LOCKED VMA.

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. PG_mlocked is cleared from the the old page when it is unmapped from the last VM_LOCKED VMA, and set when the new page is mapped in place of migration entry in a VM_LOCKED VMA. If the page was unevictable because mlocked, PG_unevictable follows PG_mlocked; but if the page was unevictable for other reasons, PG_unevictable is copied explicitly.

Note that page migration can race with mlocking or munlocking of the same page. There is mostly no problem since page migration requires unmapping all PTEs of the old page (including munlock where VM_LOCKED), then mapping in the new page (including mlock where VM_LOCKED). The page table locks provide sufficient synchronization.

However, since mlock_vma_pages_range() starts by setting VM_LOCKED on a VMA, before mlocking any pages already present, if one of those pages were migrated before mlock_pte_range() reached it, it would get counted twice in mlock_count. To prevent that, mlock_vma_pages_range() temporarily marks the VMA as VM_IO, so that mlock_vma_folio() will skip it.

To complete page migration, we place the old and new pages back onto the LRU afterwards. The "unneeded" page - old page on success, new page on failure - is freed when the reference count held by the migration process is released.

Compacting MLOCKED Pages

The memory map can be scanned for compactable regions and the default behavior is to let unevictable pages be moved. /proc/sys/vm/compact_unevictable_allowed controls this behavior (see Documentation/admin-guide/sysctl/vm.rst). 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, and no user mlock()s the whole of the 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 a new intermittent failure mode for the syscall is undesirable.

We handle this by keeping PTE-mlocked huge pages on evictable LRU lists: the PMD on the border of a VM_LOCKED VMA will be split into a 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 the unevictable LRU and the rest can be reclaimed.

/proc/meminfo's Unevictable and Mlocked amounts do not include those parts of a transparent huge page which are mapped only by PTEs in VM_LOCKED VMAs.

mmap(MAP_LOCKED) System Call Handling

In addition to the mlock(), mlock2() and mlockall() system calls, an application can request that a region of memory be mlocked by 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 mmapped area will still have properties of the locked area - 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.

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.

For each PTE (or PMD) being unmapped from a VMA, folio_remove_rmap_*() calls munlock_vma_folio(), which calls munlock_folio() when the VMA is VM_LOCKED (unless it was a PTE mapping of a part of a transparent huge page).

munlock_folio() uses the mlock pagevec to batch up work to be done under lru_lock by __munlock_folio(). __munlock_folio() decrements the folio's mlock_count, and when that

reaches 0 it clears the mlocked flag and clears the unevictable flag, moving the folio from unevictable state to the inactive LRU.

But in practice that may not work ideally: the folio may not yet have reached "the unevictable LRU", or it may have been temporarily isolated from it. In those cases its mlock_count field is unusable and must be assumed to be 0: so that the folio will be rescued to an evictable LRU, then perhaps be mlocked again later if vmscan finds it in a VM_LOCKED VMA.

Truncating MLOCKED Pages

File truncation or hole punching forcibly unmaps the deleted pages from userspace; truncation even unmaps and deletes any private anonymous pages which had been Copied-On-Write from the file pages now being truncated.

Mlocked pages can be munlocked and deleted in this way: like with munmap(), for each PTE (or PMD) being unmapped from a VMA, folio_remove_rmap_*() calls munlock_vma_folio(), which calls munlock_folio() when the VMA is VM_LOCKED (unless it was a PTE mapping of a part of a transparent huge page).

However, if there is a racing munlock(), since mlock_vma_pages_range() starts munlocking by clearing VM_LOCKED from a VMA, before munlocking all the pages present, if one of those pages were unmapped by truncation or hole punch before mlock_pte_range() reached it, it would not be recognized as mlocked by this VMA, and would not be counted out of mlock_count. In this rare case, a page may still appear as PG_mlocked after it has been fully unmapped: and it is left to release_pages() (or __page_cache_release()) to clear it and update statistics before freeing (this event is counted in /proc/vmstat unevictable_pgs_cleared, which is usually 0).

Page Reclaim in shrink_*_list()

vmscan's shrink_active_list() culls any obviously unevictable pages - i.e. !page_evictable(page) pages - diverting those 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 PG_unevictable 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) pages still mapped into VM_LOCKED VMAs, which should be marked mlocked, but events left mlock_count too low, so they were munlocked too early.

vmscan's shrink_inactive_list() and shrink_page_list() also divert obviously unevictable pages found on the inactive lists to the appropriate memory cgroup and node unevictable list.

rmap's folio_referenced_one(), called via vmscan's shrink_active_list() or shrink_page_list(), and rmap's try_to_unmap_one() called via shrink_page_list(), check for (3) pages still mapped into VM_LOCKED VMAs, and call mlock_vma_folio() to correct them. Such pages are culled to the unevictable list when released by the shrinker.

2.24 Virtually Mapped Kernel Stack Support

Author
Shuah Khan <skhan@linuxfoundation.org>

2.24.1 Overview

This is a compilation of information from the code and original patch series that introduced the *Virtually Mapped Kernel Stacks feature <https://lwn.net/Articles/694348/>*

2.24.2 Introduction

Kernel stack overflows are often hard to debug and make the kernel susceptible to exploits. Problems could show up at a later time making it difficult to isolate and root-cause.

Virtually-mapped kernel stacks with guard pages causes kernel stack overflows to be caught immediately rather than causing difficult to diagnose corruptions.

HAVE_ARCH_VMAP_STACK and VMAP_STACK configuration options enable support for virtually mapped stacks with guard pages. This feature causes reliable faults when the stack overflows. The usability of the stack trace after overflow and response to the overflow itself is architecture dependent.

Note: As of this writing, arm64, powerpc, riscv, s390, um, and x86 have support for VMAP_STACK.

2.24.3 HAVE_ARCH_VMAP_STACK

Architectures that can support Virtually Mapped Kernel Stacks should enable this bool configuration option. The requirements are:

- vmalloc space must be large enough to hold many kernel stacks. This may rule out many 32-bit architectures.
- Stacks in vmalloc space need to work reliably. For example, if vmap page tables are created on demand, either this mechanism needs to work while the stack points to a virtual address with unpopulated page tables or arch code (switch_to() and switch_mm(), most likely) needs to ensure that the stack's page table entries are populated before running on a possibly unpopulated stack.
- If the stack overflows into a guard page, something reasonable should happen. The definition of ”reasonable” is flexible, but instantly rebooting without logging anything would be unfriendly.

2.24.4 VMAP_STACK

VMAP_STACK bool configuration option when enabled allocates virtually mapped task stacks. This option depends on HAVE_ARCH_VMAP_STACK.

- Enable this if you want the use virtually-mapped kernel stacks with guard pages. This causes kernel stack overflows to be caught immediately rather than causing difficult-to-diagnose corruption.

Note: Using this feature with KASAN requires architecture support for backing virtual mappings with real shadow memory, and KASAN_VMALLOC must be enabled.

Note: VMAP_STACK is enabled, it is not possible to run DMA on stack allocated data.

Kernel configuration options and dependencies keep changing. Refer to the latest code base:

Kconfig <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/Kconfig>

2.24.5 Allocation

When a new kernel thread is created, thread stack is allocated from virtually contiguous memory pages from the page level allocator. These pages are mapped into contiguous kernel virtual space with PAGE_KERNEL protections.

alloc_thread_stack_node() calls __vmalloc_node_range() to allocate stack with PAGE_KERNEL protections.

- Allocated stacks are cached and later reused by new threads, so memcg accounting is performed manually on assigning/releasing stacks to tasks. Hence, __vmalloc_node_range is called without __GFP_ACCOUNT.
- vm_struct is cached to be able to find when thread free is initiated in interrupt context. free_thread_stack() can be called in interrupt context.

- On arm64, all VMAP's stacks need to have the same alignment to ensure that VMAP'd stack overflow detection works correctly. Arch specific vmap stack allocator takes care of this detail.
- This does not address interrupt stacks - according to the original patch

Thread stack allocation is initiated from clone(), fork(), vfork(), kernel_thread() via kernel_clone(). Leaving a few hints for searching the code base to understand when and how thread stack is allocated.

Bulk of the code is in: *kernel/fork.c <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/t*

stack_vm_area pointer in task_struct keeps track of the virtually allocated stack and a non-null stack_vm_area pointer serves as a indication that the virtually mapped kernel stacks are enabled.

```
struct vm_struct *stack_vm_area;
```

2.24.6 Stack overflow handling

Leading and trailing guard pages help detect stack overflows. When stack overflows into the guard pages, handlers have to be careful not overflow the stack again. When handlers are called, it is likely that very little stack space is left.

On x86, this is done by handling the page fault indicating the kernel stack overflow on the double-fault stack.

2.24.7 Testing VMAP allocation with guard pages

How do we ensure that VMAP_STACK is actually allocating with a leading and trailing guard page? The following lkdtm tests can help detect any regressions.

```
void lkdtm_STACK_GUARD_PAGE_LEADING()
void lkdtm_STACK_GUARD_PAGE_TRAILING()
```

2.24.8 Conclusions

- A percpu cache of vmalloced stacks appears to be a bit faster than a high-order stack allocation, at least when the cache hits.
- THREAD_INFO_IN_TASK gets rid of arch-specific thread_info entirely and simply embed the thread_info (containing only flags) and 'int cpu' into task_struct.
- The thread stack can be free'ed as soon as the task is dead (without waiting for RCU) and then, if vmapped stacks are in use, cache the entire stack for reuse on the same cpu.

2.25 A vmemmap diet for HugeTLB and Device DAX

2.25.1 HugeTLB

This section is to explain how HugeTLB Vmemmap Optimization (HVO) works.

The `struct page` structures are used to describe a physical page frame. By default, there is a one-to-one mapping from a page frame to its corresponding `struct page`.

HugeTLB pages consist of multiple base page size pages and is supported by many architectures. See Documentation/admin-guide/mm/hugetlbpage.rst for more details. On the x86-64 architecture, HugeTLB pages of size 2MB and 1GB are currently supported. Since the base page size on x86 is 4KB, a 2MB HugeTLB page consists of 512 base pages and a 1GB HugeTLB page consists of 262144 base pages. For each base page, there is a corresponding `struct page`.

Within the HugeTLB subsystem, only the first 4 `struct page` are used to contain unique information about a HugeTLB page. `__NR_USED_SUBPAGE` provides this upper limit. The only 'useful' information in the remaining `struct page` is the compound_head field, and this field is the same for all tail pages.

By removing redundant `struct page` for HugeTLB pages, memory can be returned to the buddy allocator for other uses.

Different architectures support different HugeTLB pages. For example, the following table is the HugeTLB page size supported by x86 and arm64 architectures. Because arm64 supports 4k, 16k, and 64k base pages and supports contiguous entries, so it supports many kinds of sizes of HugeTLB page.

Architecture	Page Size	HugeTLB Page Size			
x86-64	4KB	2MB	1GB		
arm64	4KB	64KB	2MB	32MB	1GB
	16KB	2MB	32MB	1GB	
	64KB	2MB	512MB	16GB	

When the system boot up, every HugeTLB page has more than one `struct page` structs which size is (unit: pages):

```
struct_size = HugeTLB_Size / PAGE_SIZE * sizeof(struct page) / PAGE_SIZE
```

Where HugeTLB_Size is the size of the HugeTLB page. We know that the size of the HugeTLB page is always n times PAGE_SIZE. So we can get the following relationship:

```
HugeTLB_Size = n * PAGE_SIZE
```

Then:

```
struct_size = n * PAGE_SIZE / PAGE_SIZE * sizeof(struct page) / PAGE_SIZE
            = n * sizeof(struct page) / PAGE_SIZE
```

We can use huge mapping at the pud/pmd level for the HugeTLB page.

For the HugeTLB page of the pmd level mapping, then:

```
struct_size = n * sizeof(struct page) / PAGE_SIZE
            = PAGE_SIZE / sizeof(pte_t) * sizeof(struct page) / PAGE_SIZE
            = sizeof(struct page) / sizeof(pte_t)
            = 64 / 8
            = 8 (pages)
```

Where n is how many pte entries which one page can contains. So the value of n is (PAGE_SIZE / sizeof(pte_t)).

This optimization only supports 64-bit system, so the value of sizeof(pte_t) is 8. And this optimization also applicable only when the size of `struct page` is a power of two. In most cases, the size of `struct page` is 64 bytes (e.g. x86-64 and arm64). So if we use pmd level mapping for a HugeTLB page, the size of `struct page` structs of it is 8 page frames which size depends on the size of the base page.

For the HugeTLB page of the pud level mapping, then:

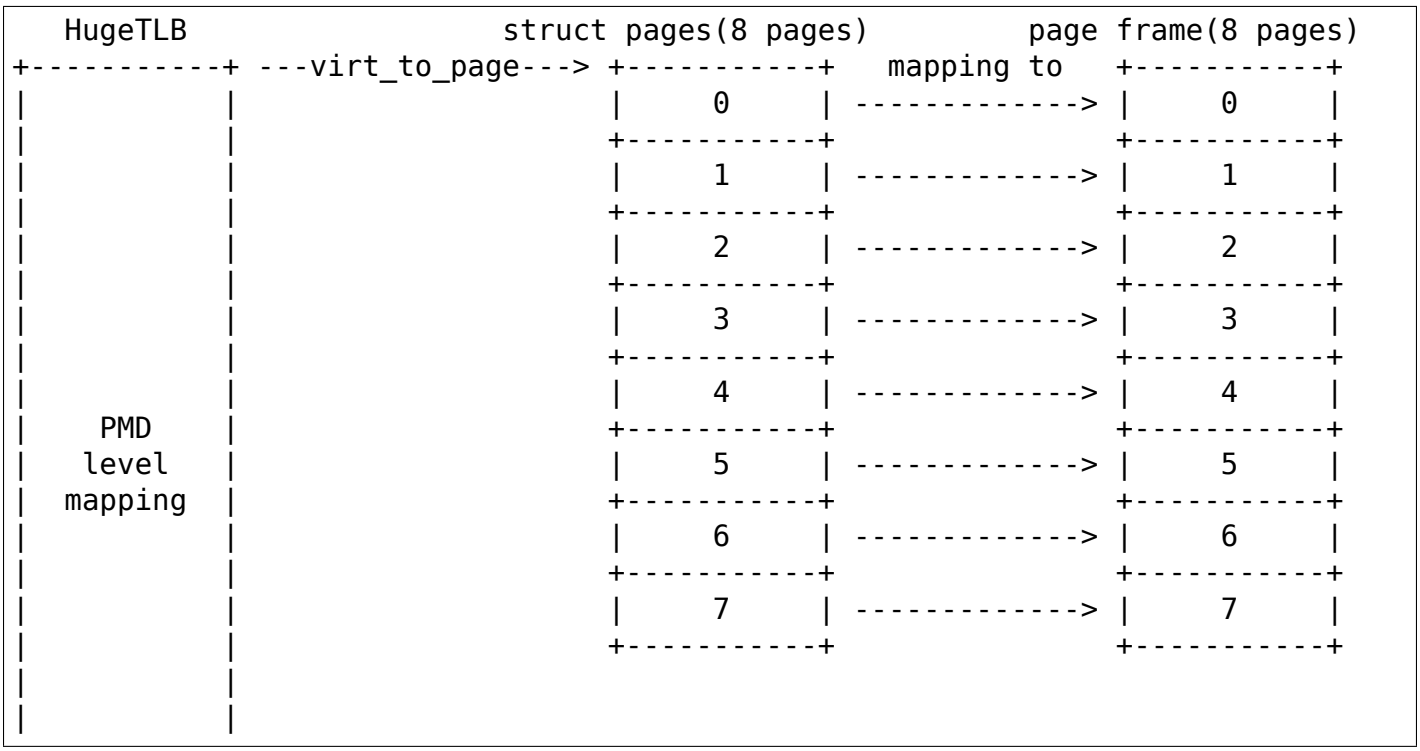
```
struct_size = PAGE_SIZE / sizeof(pmd_t) * struct_size(pmd)
            = PAGE_SIZE / 8 * 8 (pages)
            = PAGE_SIZE (pages)
```

Where the struct_size(pmd) is the size of the `struct page` structs of a HugeTLB page of the pmd level mapping.

E.g.: A 2MB HugeTLB page on x86_64 consists in 8 page frames while 1GB HugeTLB page consists in 4096.

Next, we take the pmd level mapping of the HugeTLB page as an example to show the internal implementation of this optimization. There are 8 pages `struct page` structs associated with a HugeTLB page which is pmd mapped.

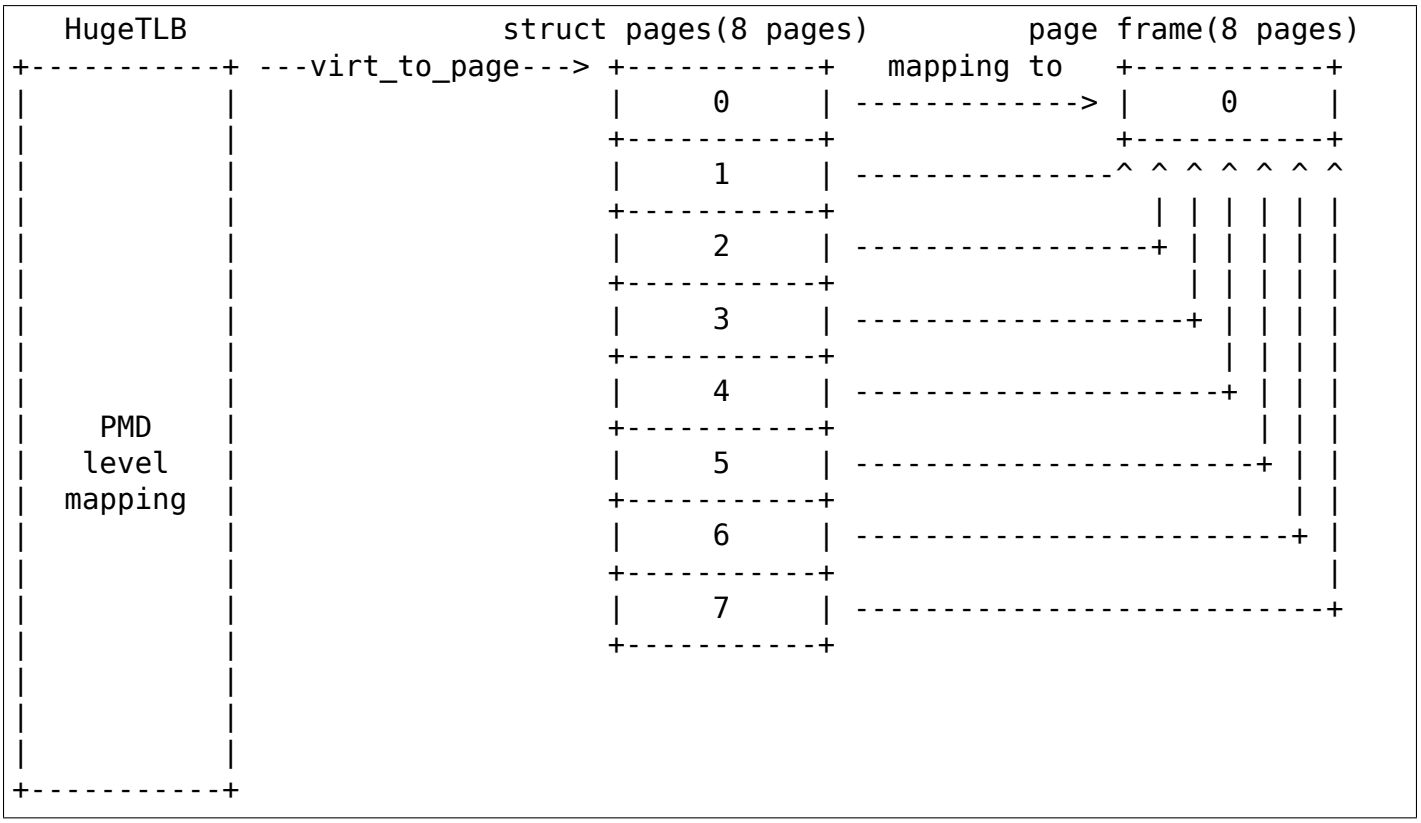
Here is how things look before optimization:



```
|           |
+-----------+
```

The value of page->compound_head is the same for all tail pages. The first page of `struct page` (page 0) associated with the HugeTLB page contains the 4 `struct page` necessary to describe the HugeTLB. The only use of the remaining pages of `struct page` (page 1 to page 7) is to point to page->compound_head. Therefore, we can remap pages 1 to 7 to page 0. Only 1 page of `struct page` will be used for each HugeTLB page. This will allow us to free the remaining 7 pages to the buddy allocator.

Here is how things look after remapping:



When a HugeTLB is freed to the buddy system, we should allocate 7 pages for vmemmap pages and restore the previous mapping relationship.

For the HugeTLB page of the pud level mapping. It is similar to the former. We also can use this approach to free (PAGE_SIZE - 1) vmemmap pages.

Apart from the HugeTLB page of the pmd/pud level mapping, some architectures (e.g. aarch64) provides a contiguous bit in the translation table entries that hints to the MMU to indicate that it is one of a contiguous set of entries that can be cached in a single TLB entry.

The contiguous bit is used to increase the mapping size at the pmd and pte (last) level. So this type of HugeTLB page can be optimized only when its size of the `struct page` structs is greater than **1** page.

Notice: The head vmemmap page is not freed to the buddy allocator and all tail vmemmap pages are mapped to the head vmemmap page frame. So we can see more than one `struct page` struct with `PG_head` (e.g. 8 per 2 MB HugeTLB page) associated with each HugeTLB page. The `compound_head()` can handle this correctly. There is only **one** head `struct page`, the tail

`struct page` with `PG_head` are fake head `struct page`. We need an approach to distinguish between those two different types of `struct page` so that `compound_head()` can return the real head `struct page` when the parameter is the tail `struct page` but with `PG_head`. The following code snippet describes how to distinguish between real and fake head `struct page`.

```
if (test_bit(PG_head, &page->flags)) {
        unsigned long head = READ_ONCE(page[1].compound_head);

        if (head & 1) {
                if (head == (unsigned long)page + 1)
                        /* head struct page */
                else
                        /* tail struct page */
        } else {
                /* head struct page */
        }
}
```

We can safely access the field of the **page[1]** with `PG_head` because the page is a compound page composed with at least two contiguous pages. The implementation refers to `page_fixed_fake_head()`.

2.25.2 Device DAX

The device-dax interface uses the same tail deduplication technique explained in the previous chapter, except when used with the vmemmap in the device (altmap).

The following page sizes are supported in DAX: PAGE_SIZE (4K on x86_64), PMD_SIZE (2M on x86_64) and PUD_SIZE (1G on x86_64). For powerpc equivalent details see Documentation/arch/powerpc/vmemmap_dedup.rst

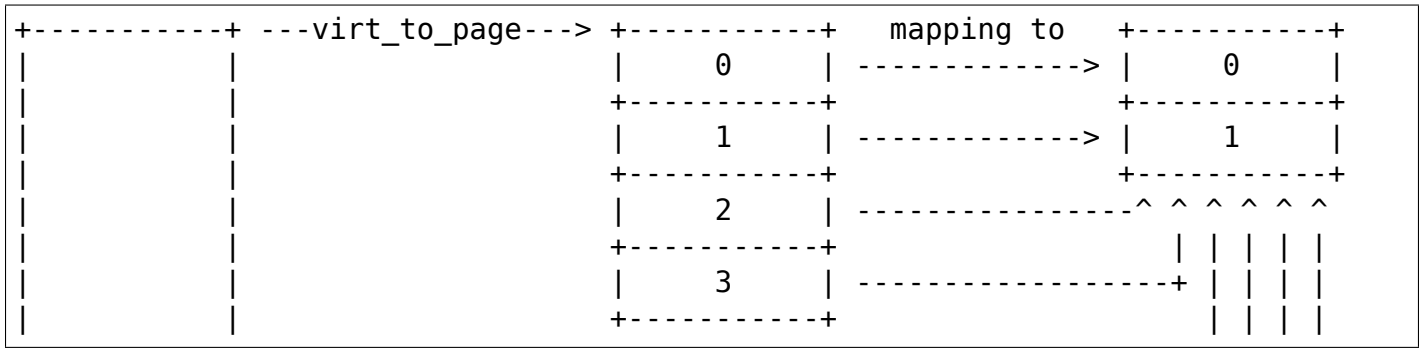
The differences with HugeTLB are relatively minor.

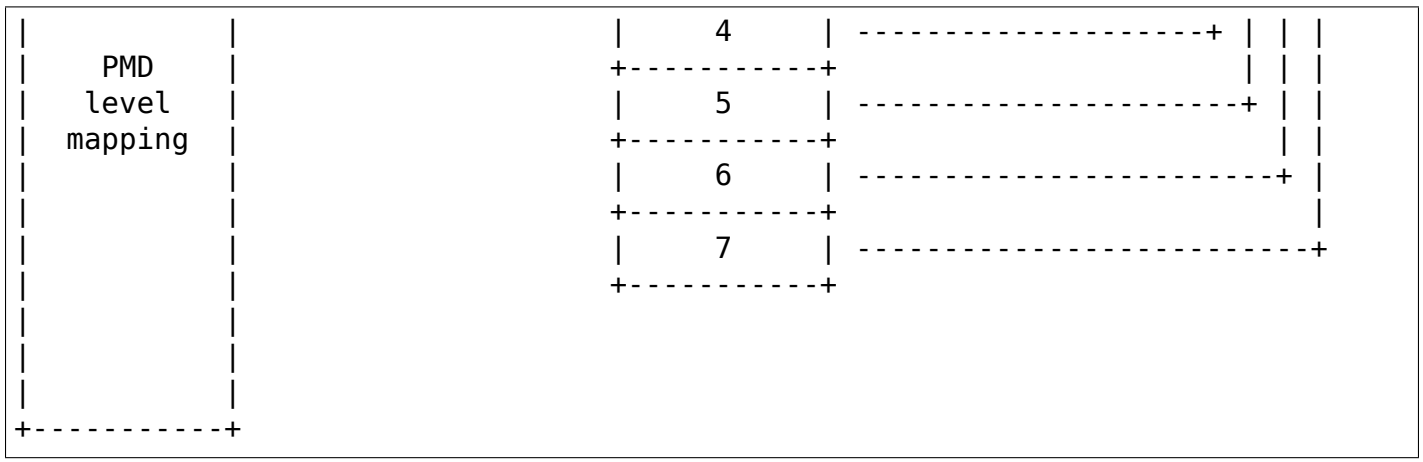
It only use 3 `struct page` for storing all information as opposed to 4 on HugeTLB pages.

There's no remapping of vmemmap given that device-dax memory is not part of System RAM ranges initialized at boot. Thus the tail page deduplication happens at a later stage when we populate the sections. HugeTLB reuses the the head vmemmap page representing, whereas device-dax reuses the tail vmemmap page. This results in only half of the savings compared to HugeTLB.

Deduplicated tail pages are not mapped read-only.

Here's how things look like on device-dax after the sections are populated:





2.26 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.

2.27 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().

2.27.1 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       10%       20%       30%       40%       50%       60%
 ↪70%       80%       90%       99%      100% obj_allocated   obj_used pages_
 ↪used pages_per_zspage freeable
   ...
   ...
   30   512         0        12         4         1         0         1
 ↪ 0         0         1         0       414         3464       3346
 ↪433                1        14
   31   528         2         7         2         2         1         0
 ↪ 1         0         0         2       117         4154       3793
 ↪536                4        44
   32   544         6         3         4         1         2         1
 ↪ 0         0         0         1       260         4170       3965
 ↪556                2        26
   ...
   ...
```

class
: index

size
: object size zspage stores

10%
: the number of zspages with usage ratio less than 10% (see below)

20%
: the number of zspages with usage ratio between 10% and 20%

30%
: the number of zspages with usage ratio between 20% and 30%

40%
: the number of zspages with usage ratio between 30% and 40%

50%
the number of zspages with usage ratio between 40% and 50%

60%
the number of zspages with usage ratio between 50% and 60%

70%
the number of zspages with usage ratio between 60% and 70%

80%
the number of zspages with usage ratio between 70% and 80%

90%
the number of zspages with usage ratio between 80% and 90%

99%
the number of zspages with usage ratio between 90% and 99%

100%
the number of zspages with usage ratio 100%

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

freeable
the approximate number of pages class compaction can free

Each zspage maintains inuse counter which keeps track of the number of objects stored in the zspage. The inuse counter determines the zspage's ”fullness group” which is calculated as the ratio of the ”inuse” objects to the total number of objects the zspage can hold (objs_per_zspage). The closer the inuse counter is to objs_per_zspage, the better.

2.27.2 Internals

zsmalloc has 255 size classes, each of which can hold a number of zspages. Each zspage can contain up to ZSMALLOC_CHAIN_SIZE physical (0-order) pages. The optimal zspage chain size for each size class is calculated during the creation of the zsmalloc pool (see calculate_zspage_chain_size()).

As an optimization, zsmalloc merges size classes that have similar characteristics in terms of the number of pages per zspage and the number of objects that each zspage can store.

For instance, consider the following size classes::

```
class  size       10%   ....    100% obj_allocated   obj_used pages_used pages_
↪per_zspage freeable
...
   94  1536         0    ....       0              0          0          0      ␣
↪          3         0
```

```
  100  1632        0    ....       0             0          0          0
↪          2        0
...
```

Size classes #95-99 are merged with size class #100. This means that when we need to store an object of size, say, 1568 bytes, we end up using size class #100 instead of size class #96. Size class #100 is meant for objects of size 1632 bytes, so each object of size 1568 bytes wastes 1632-1568=64 bytes.

Size class #100 consists of zspages with 2 physical pages each, which can hold a total of 5 objects. If we need to store 13 objects of size 1568, we end up allocating three zspages, or 6 physical pages.

However, if we take a closer look at size class #96 (which is meant for objects of size 1568 bytes) and trace *calculate_zspage_chain_size()*, we find that the most optimal zspage configuration for this class is a chain of 5 physical pages::

```
pages per zspage      wasted bytes     used%
       1                  960          76
       2                  352          95
       3                 1312          89
       4                  704          95
       5                   96          99
```

This means that a class #96 configuration with 5 physical pages can store 13 objects of size 1568 in a single zspage, using a total of 5 physical pages. This is more efficient than the class #100 configuration, which would use 6 physical pages to store the same number of objects.

As the zspage chain size for class #96 increases, its key characteristics such as pages per-zspage and objects per-zspage also change. This leads to dewer class mergers, resulting in a more compact grouping of classes, which reduces memory wastage.

Let's take a closer look at the bottom of */sys/kernel/debug/zsmalloc/zramX/classes*::

```
class  size       10%   ....    100% obj_allocated   obj_used pages_used pages_
↪per_zspage freeable

...
  202  3264         0   ..         0             0          0          0
↪          4        0
  254  4096         0   ..         0             0          0          0
↪          1        0
...
```

Size class #202 stores objects of size 3264 bytes and has a maximum of 4 pages per zspage. Any object larger than 3264 bytes is considered huge and belongs to size class #254, which stores each object in its own physical page (objects in huge classes do not share pages).

Increasing the size of the chain of zspages also results in a higher watermark for the huge size class and fewer huge classes overall. This allows for more efficient storage of large objects.

For zspage chain size of 8, huge class watermark becomes 3632 bytes::

```
class  size        10%   ....    100% obj_allocated   obj_used pages_used pages_
↪per_zspage freeable

...
  202  3264          0   ..         0              0          0          0
↪          4         0
  211  3408          0   ..         0              0          0          0
↪          5         0
  217  3504          0   ..         0              0          0          0
↪          6         0
  222  3584          0   ..         0              0          0          0
↪          7         0
  225  3632          0   ..         0              0          0          0
↪          8         0
  254  4096          0   ..         0              0          0          0
↪          1         0
...
```

For zspage chain size of 16, huge class watermark becomes 3840 bytes::

```
class  size        10%   ....    100% obj_allocated   obj_used pages_used pages_
↪per_zspage freeable

...
  202  3264          0   ..         0              0          0          0
↪          4         0
  206  3328          0   ..         0              0          0          0
↪         13         0
  207  3344          0   ..         0              0          0          0
↪          9         0
  208  3360          0   ..         0              0          0          0
↪         14         0
  211  3408          0   ..         0              0          0          0
↪          5         0
  212  3424          0   ..         0              0          0          0
↪         16         0
  214  3456          0   ..         0              0          0          0
↪         11         0
  217  3504          0   ..         0              0          0          0
↪          6         0
  219  3536          0   ..         0              0          0          0
↪         13         0
  222  3584          0   ..         0              0          0          0
↪          7         0
  223  3600          0   ..         0              0          0          0
↪         15         0
  225  3632          0   ..         0              0          0          0
↪          8         0
  228  3680          0   ..         0              0          0          0
↪          9         0
  230  3712          0   ..         0              0          0          0
↪         10         0
```

```
  232  3744           0   ..          0                0           0           0
↪          11          0
  234  3776           0   ..          0                0           0           0
↪          12          0
  235  3792           0   ..          0                0           0           0
↪          13          0
  236  3808           0   ..          0                0           0           0
↪          14          0
  238  3840           0   ..          0                0           0           0
↪          15          0
  254  4096           0   ..          0                0           0           0
↪           1          0
...
```

Overall the combined zspage chain size effect on zsmalloc pool configuration::

```
pages per zspage   number of size classes (clusters)   huge size class
↪watermark
      4                         69                              3264
      5                         86                              3408
      6                         93                              3504
      7                        112                              3584
      8                        123                              3632
      9                        140                              3680
     10                        143                              3712
     11                        159                              3744
     12                        164                              3776
     13                        180                              3792
     14                        183                              3808
     15                        188                              3840
     16                        191                              3840
```

A synthetic test

zram as a build artifacts storage (Linux kernel compilation).

- *CONFIG_ZSMALLOC_CHAIN_SIZE=4*

  zsmalloc classes stats::

```
class  size       10%   ....    100% obj_allocated   obj_used pages_used
↪pages_per_zspage freeable

...
Total              13   ..        51        413836     412973     159955
↪                    3
```

  zram mm_stat::

```
1691783168 628083717 655175680        0 655175680       60        0
↪34048    34049
```

- *CONFIG_ZSMALLOC_CHAIN_SIZE=8*

  zsmalloc classes stats::

```
class  size       10%   ....    100% obj_allocated   obj_used pages_used␣
↪pages_per_zspage freeable

...
Total              18   ..        87        414852     412978     156666  ␣
↪                       0
```

  zram mm_stat::

```
1691803648 627793930 641703936        0 641703936       60        0    ␣
↪33591    33591
```

Using larger zspage chains may result in using fewer physical pages, as seen in the example where the number of physical pages used decreased from 159955 to 156666, at the same time maximum zsmalloc pool memory usage went down from 655175680 to 641703936 bytes.

However, this advantage may be offset by the potential for increased system memory pressure (as some zspages have larger chain sizes) in cases where there is heavy internal fragmentation and zspool compaction is unable to relocate objects and release zspages. In these cases, it is recommended to decrease the limit on the size of the zspage chains (as specified by the CONFIG_ZSMALLOC_CHAIN_SIZE option).

2.27.3 Functions

void **obj_to_location**(unsigned long obj, struct *page* **page, unsigned int *obj_idx)
get (<page>, <obj_idx>) from encoded object value

Parameters

unsigned long obj
the encoded object value

struct page **page
page object resides in zspage

unsigned int *obj_idx
object index

unsigned long **location_to_obj**(struct *page* *page, unsigned int obj_idx)
get obj value encoded from (<page>, <obj_idx>)

Parameters

struct page *page
page object resides in zspage

unsigned int obj_idx
object index

unsigned int **zs_lookup_class_index**(struct zs_pool *pool, unsigned int size)
Returns index of the zsmalloc `size_class` that hold objects of the provided size.

Parameters

struct zs_pool *pool
zsmalloc pool to use

unsigned int size
object size

Context

Any context.

Return

the index of the zsmalloc `size_class` that hold objects of the provided size.

void ***zs_map_object**(struct zs_pool *pool, unsigned long handle, enum zs_mapmode mm)
get address of allocated object from handle.

Parameters

struct zs_pool *pool
pool from which the object was allocated

unsigned long handle
handle returned from zs_malloc

enum zs_mapmode mm
mapping mode to use

Description

Before using an object allocated from zs_malloc, it must be mapped using this function. When done with the object, it must be unmapped using zs_unmap_object.

Only one object can be mapped per cpu at a time. There is no protection against nested mappings.

This function returns with preemption and page faults disabled.

size_t **zs_huge_class_size**(struct zs_pool *pool)
Returns the size (in bytes) of the first huge zsmalloc `size_class`.

Parameters

struct zs_pool *pool
zsmalloc pool to use

Description

The function returns the size of the first huge class - any object of equal or bigger size will be stored in zspage consisting of a single physical page.

Context

Any context.

Return

the size (in bytes) of the first huge zsmalloc `size_class`.

unsigned long **zs_malloc**(struct zs_pool *pool, size_t size, gfp_t gfp)

Allocate block of given size from pool.

Parameters

struct zs_pool *pool
pool to allocate from

size_t size
size of block to allocate

gfp_t gfp
gfp flags when allocating object

Description

On success, handle to the allocated object is returned, otherwise an ERR_PTR(). Allocation requests with size > ZS_MAX_ALLOC_SIZE will fail.

struct zs_pool ***zs_create_pool**(const char *name)

Creates an allocation pool to work from.

Parameters

const char *name
pool name to be created

Description

This function must be called before anything when using the zsmalloc allocator.

On success, a pointer to the newly created pool is returned, otherwise NULL.

INDEX