Unevictable LRU Infrastructure

Introduction

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

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

The Unevictable LRU

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

To illustrate this with an example, a non-NUMA x86_64 platform with 128GB of main memory will have over 32 million 4k pages in a single 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 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 Page List

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

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

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

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

1. We get to "treat unevictable pages just like we treat other pages in the system - which means we get to use the same code to manipulate them, the same code to isolate them (for migrate, etc.), the same code to keep track of the statistics, etc..." [Rik van Riel]
2. We want to be able to migrate unevictable pages between nodes for memory defragmentation, workload management and memory hotplug. The Linux kernel can only migrate pages that it can successfully isolate from the LRU lists (or "Movable" pages: outside of consideration here). If we were to maintain pages elsewhere than on an LRU-like list, where they can be detected by isolate_lru_page(), we would prevent their migration.

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

The unevictable list benefits from the "arrayification" of the per-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 page_evictable() in mm/internal.h determines whether a page is evictable or not using the query function outlined above [see section :ref:`Marking address spaces unevictable <mark_addr_space_unevict>`] to check the AS_UNEVICTABLE flag.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\[linux-master][Documentation][vm]unevictable-lru.rst`, line 161); *backlink***
>
> Unknown interpreted text role "ref".

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 pages 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.

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

Vmscan's Handling of Unevictable Pages

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

There may be situations where a page is mapped into a VM_LOCKED VMA, but the page is not marked as PG_mlocked. Such pages will make it all the way to shrink_active_list() or shrink_page_list() where they will be detected when vmscan walks the reverse map in page_referenced() or try_to_unmap(). The page is culled to the unevictable list when it is released by the shrinker.

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

MLOCKED Pages

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

History

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

In Nick's patch, he used one of the struct page LRU list link fields as a count of VM_LOCKED VMAs that map the page (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 page_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_page() 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_page(), which calls mlock_page() 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, lru_cache_add_inactive_or_unevictable() calls mlock_new_page() instead: similar to mlock_page(), but can make better judgments, since this page is held exclusively and known not to be on LRU yet.

mlock_page() sets PageMlocked immediately, then places the page on the CPU's mlock pagevec, to batch up the rest of the work to be done under lru_lock by __mlock_page(). __mlock_page() sets PageUnevictable, 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 PageLRU and PageUnevictable and PageMlocked, 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 __pagevec_lru_add_fn() 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_page() 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_page() uses the mlock pagevec to batch up work to be done under lru_lock by __munlock_page(). __munlock_page() decrements the page's mlock_count, and when that reaches 0 it clears PageMlocked and clears PageUnevictable, moving the page from unevictable state to inactive LRU.

But in practice that may not work ideally: the page 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 page 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_page() 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 mmaped 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, page_remove_rmap() calls munlock_vma_page(), which calls

munlock_page() when the VMA is VM_LOCKED (unless it was a PTE mapping of a part of a transparent huge page).

munlock_page() uses the mlock pagevec to batch up work to be done under lru_lock by __munlock_page(). __munlock_page() decrements the page's mlock_count, and when that reaches 0 it clears PageMlocked and clears PageUnevictable, moving the page from unevictable state to inactive LRU.

But in practice that may not work ideally: the page 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 page 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, page_remove_rmap() calls munlock_vma_page(), which calls munlock_page() 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 PageMlocked 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 PageUnevictable set - otherwise they would be on the unevictable list and shrink_active_list() would never see them.

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

1. ramfs pages that have been placed on the LRU lists when first allocated.
2. SHM_LOCK'd shared memory pages. shmctl(SHM_LOCK) does not attempt to allocate or fault in the pages in the shared memory region. This happens when an application accesses the page the first time after SHM_LOCK'ing the segment.
3. 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 page_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_page() to correct them. Such pages are culled to the unevictable list when released by the shrinker.