UNEVICTABLE LRU INFRASTRUCTURE

CONTENTS

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 zone. When a large fraction of these pages are not evictable for any reason [see below], vmscan will spend a lot of time scanning the LRU lists looking for the small fraction of pages that are evictable. This can result in a situation where all CPUs are spending 100% of their time in vmscan for hours or days on end, with the system completely unresponsive.

The unevictable list addresses the following classes of unevictable pages:

(*) Those owned by ramfs.

(*) Those mapped into SHM_LOCK'd shared memory regions.

(*) Those mapped into VM_LOCKED [mlock()ed] VMAs.

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

THE UNEVICTABLE PAGE LIST

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

The PG_unevictable flag is analogous to, and mutually exclusive with, the PG_active flag in that it indicates on which LRU list a page resides when PG_lru is set. The unevictable list is compile-time configurable based on the UNEVICTABLE_LRU Kconfig option.

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

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

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

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

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

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

MEMORY CONTROL GROUP INTERACTION

The unevictable LRU facility interacts with the memory control group [aka memory controller; see Documentation/cgroups/memory.txt] by extending the lru_list enum.

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

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

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

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

MARKING ADDRESS SPACES UNEVICTABLE

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

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

Mark the address space as being completely unevictable.

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

Mark the address space as being evictable.

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

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

These are currently used in two 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.

DETECTING UNEVICTABLE PAGES

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

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

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

page_evictable() also checks for mlocked pages by testing an additional page flag, PG_mlocked (as wrapped by PageMlocked()). If the page is NOT mlocked, and a non-NULL VMA is supplied, page_evictable() will check whether the VMA is VM_LOCKED via is_mlocked_vma(). is_mlocked_vma() will SetPageMlocked() and update the appropriate statistics if the vma is VM_LOCKED. This method allows efficient "culling" of pages in the fault path that are being faulted in to VM_LOCKED VMAs.

VMSCAN'S HANDLING OF UNEVICTABLE PAGES

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

There may be situations where a page is mapped into a VM_LOCKED VMA, but the page is not marked as PG_mlocked. Such pages will make it all the way to shrink_page_list() where they will be detected when vmscan walks the reverse

map in try_to_unmap(). If try_to_unmap() returns SWAP_MLOCK,
shrink_page_list() will cull the page at that point.

To "cull" an unevictable page, vmscan simply puts the page back on the LRU list
using putback_lru_page() - the inverse operation to isolate_lru_page() - after
dropping the page lock. Because the condition which makes the page unevictable
may change once the page is unlocked, putback_lru_page() will recheck the
unevictable state of a page that it places on the unevictable list. If the
page has become unevictable, putback_lru_page() removes it from the list and
retries, including the page_unevictable() test. Because such a race is a rare
event and movement of pages onto the unevictable list should be rare, these
extra evictabilty checks should not occur in the majority of calls to
putback_lru_page().

=============
MLOCKED PAGES
=============

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

HISTORY

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

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

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

BASIC MANAGEMENT
----

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

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

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

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

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

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

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

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

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

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

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

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

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

mlock()/mlockall() SYSTEM CALL HANDLING

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

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

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

Also note that a page returned by get_user_pages() could be truncated or migrated out from under us, while we're trying to mlock it. To detect this, __mlock_vma_pages_range() checks page_mapping() after acquiring the page lock. If the page is still associated with its mapping, we'll go ahead and call

mlock_vma_page(). If the mapping is gone, we just unlock the page and move on. In the worst case, this will result in a page mapped in a VM_LOCKED VMA remaining on a normal LRU list without being PageMlocked(). Again, vmscan will detect and cull such pages.

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

If the page was NOT already mlocked, mlock_vma_page() attempts to isolate the page from the LRU, as it is likely on the appropriate active or inactive list at that time. If the isolate_lru_page() succeeds, mlock_vma_page() will put back the page - by calling putback_lru_page() - which will notice that the page is now mlocked and divert the page to the zone's unevictable list. If mlock_vma_page() is unable to isolate the page from the LRU, vmscan will handle it later if and when it attempts to reclaim the page.

FILTERING SPECIAL VMAS

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

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

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

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

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

munlock()/munlockall() SYSTEM CALL HANDLING

The munlock() and munlockall() system calls are handled by the same functions - do_mlock[all]() - as the mlock() and mlockall() system calls with the unlock vs lock operation indicated by an argument. So, these system calls are also

handled by mlock_fixup(). Again, if called for an already munlocked VMA, mlock_fixup() simply returns. Because of the VMA filtering discussed above, VM_LOCKED will not be set in any "special" VMAs. So, these VMAs will be ignored for munlock.

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

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

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

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

MIGRATING MLOCKED PAGES

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

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

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

mmap(MAP_LOCKED) SYSTEM CALL HANDLING

In addition the the mlock()/mlockall() system calls, an application can request that a region of memory be mlocked supplying the MAP_LOCKED flag to the mmap() call. 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 mlock_vma_pages_range() specifying the vma and the address range to mlock. mlock_vma_pages_range() filters VMAs like mlock_fixup(), as described above in "Filtering Special VMAs". It will clear the VM_LOCKED flag, which will have already been set by the caller, in filtered VMAs. Thus these VMA's need not be visited for munlock when the region is unmapped.

For "normal" VMAs, mlock_vma_pages_range() calls __mlock_vma_pages_range() to fault/allocate the pages and mlock them. Again, like mlock_fixup(), mlock_vma_pages_range() downgrades the mmap semaphore to read mode before attempting to fault/allocate and mlock the pages and "upgrades" the semaphore back to write mode before returning.

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

munmap()/exit()/exec() SYSTEM CALL HANDLING

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

To munlock a range of memory under the unevictable/mlock infrastructure, the munmap() handler and task address space call tear down function munlock_vma_pages_all(). The name reflects the observation that one always

specifies the entire VMA range when munlock()ing during unmap of a region. Because of the VMA filtering when mlocking() regions, only "normal" VMAs that actually contain mlocked pages will be passed to munlock_vma_pages_all().

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

try_to_unmap()

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

try_to_unmap() is always called, by either vmscan for reclaim or for page migration, with the argument page locked and isolated from the LRU. Separate functions handle anonymous and mapped file pages, as these types of pages have different reverse map mechanisms.

(*) try_to_unmap_anon()

    To unmap anonymous pages, each VMA in the list anchored in the anon_vma must be visited - at least until a VM_LOCKED VMA is encountered. If the page is being unmapped for migration, VM_LOCKED VMAs do not stop the process because mlocked pages are migratable. However, for reclaim, if the page is mapped into a VM_LOCKED VMA, the scan stops.

    try_to_unmap_anon() attempts to acquire in read mode the mmap semphore of the mm_struct to which the VMA belongs. If this is successful, it will mlock the page via mlock_vma_page() - we wouldn't have gotten to try_to_unmap_anon() if the page were already mlocked - and will return SWAP_MLOCK, indicating that the page is unevictable.

    If the mmap semaphore cannot be acquired, we are not sure whether the page is really unevictable or not. In this case, try_to_unmap_anon() will return SWAP_AGAIN.

(*) try_to_unmap_file() - linear mappings

    Unmapping of a mapped file page works the same as for anonymous mappings, except that the scan visits all VMAs that map the page's index/page offset in the page's mapping's reverse map priority search tree. It also visits each VMA in the page's mapping's non-linear list, if the list is non-empty.

    As for anonymous pages, on encountering a VM_LOCKED VMA for a mapped file

page, try_to_unmap_file() will attempt to acquire the associated mm_struct's mmap semaphore to mlock the page, returning SWAP_MLOCK if this is successful, and SWAP_AGAIN, if not.

(*) try_to_unmap_file() - non-linear mappings

If a page's mapping contains a non-empty non-linear mapping VMA list, then try_to_un{map|lock}() must also visit each VMA in that list to determine whether the page is mapped in a VM_LOCKED VMA. Again, the scan must visit all VMAs in the non-linear list to ensure that the pages is not/should not be mlocked.

If a VM_LOCKED VMA is found in the list, the scan could terminate. However, there is no easy way to determine whether the page is actually mapped in a given VMA - either for unmapping or testing whether the VM_LOCKED VMA actually pins the page.

try_to_unmap_file() handles non-linear mappings by scanning a certain number of pages - a "cluster" - in each non-linear VMA associated with the page's mapping, for each file mapped page that vmscan tries to unmap. If this happens to unmap the page we're trying to unmap, try_to_unmap() will notice this on return (page_mapcount(page) will be 0) and return SWAP_SUCCESS. Otherwise, it will return SWAP_AGAIN, causing vmscan to recirculate this page. We take advantage of the cluster scan in try_to_unmap_cluster() as follows:

For each non-linear VMA, try_to_unmap_cluster() attempts to acquire the mmap semaphore of the associated mm_struct for read without blocking.

If this attempt is successful and the VMA is VM_LOCKED, try_to_unmap_cluster() will retain the mmap semaphore for the scan; otherwise it drops it here.

Then, for each page in the cluster, if we're holding the mmap semaphore for a locked VMA, try_to_unmap_cluster() calls mlock_vma_page() to mlock the page. This call is a no-op if the page is already locked, but will mlock any pages in the non-linear mapping that happen to be unlocked.

If one of the pages so mlocked is the page passed in to try_to_unmap(), try_to_unmap_cluster() will return SWAP_MLOCK, rather than the default SWAP_AGAIN. This will allow vmscan to cull the page, rather than recirculating it on the inactive list.

Again, if try_to_unmap_cluster() cannot acquire the VMA's mmap sem, it returns SWAP_AGAIN, indicating that the page is mapped by a VM_LOCKED VMA, but couldn't be mlocked.

try_to_munlock() REVERSE MAP SCAN

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

When munlock_vma_page() [see section "munlock()/munlockall() System Call

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

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

If try_to_unmap() is unable to acquire a VM_LOCKED VMA's associated mmap semaphore, it will return SWAP_AGAIN. This will allow shrink_page_list() to recycle the page on the inactive list and hope that it has better luck with the page next time.

For file pages mapped into non-linear VMAs, the try_to_munlock() logic works slightly differently. On encountering a VM_LOCKED non-linear VMA that might map the page, try_to_munlock() returns SWAP_AGAIN without actually mlocking the page. munlock_vma_page() will just leave the page unlocked and let vmscan deal with it - the usual fallback position.

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

PAGE RECLAIM IN shrink_*_list()
----

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

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

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

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

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

(4) Pages mapped into multiple VM_LOCKED VMAs, but try_to_munlock() couldn't acquire the VMA's mmap semaphore to test the flags and set PageMlocked. munlock_vma_page() was forced to let the page back on to the normal LRU list for vmscan to handle.

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

shrink_inactive_list() should only see SHM_LOCK'd pages that became SHM_LOCK'd after shrink_active_list() had moved them to the inactive list, or pages mapped into VM_LOCKED VMAs that munlock_vma_page() couldn't isolate from the LRU to recheck via try_to_munlock(). shrink_inactive_list() won't notice the latter, but will pass on to shrink_page_list().

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