Started Oct 1999 by Kanoj Sarcar <kanojsarcar@yahoo.com>

The intent of this file is to have an uptodate, running commentary from different people about how locking and synchronization is done in the Linux vm code.

page_table_lock & mmap_sem

Page stealers pick processes out of the process pool and scan for the best process to steal pages from. To guarantee the existence of the victim mm, a mm_count inc and a mmdrop are done in swap_out(). Page stealers hold kernel_lock to protect against a bunch of races. The vma list of the victim mm is also scanned by the stealer, and the page_table_lock is used to preserve list sanity against the process adding/deleting to the list. This also guarantees existence of the vma. Vma existence is not guaranteed once try_to_swap_out() drops the page_table_lock. To guarantee the existence of the underlying file structure, a get_file is done before the swapout() method is invoked. The page passed into swapout() is guaranteed not to be reused for a different purpose because the page reference count due to being present in the user's pte is not released till after swapout() returns.

Any code that modifies the vmlist, or the vm_start/vm_end/ vm_flags:VM_LOCKED/vm_next of any vma *in the list* must prevent kswapd from looking at the chain.

The rules are:
1. To scan the vmlist (look but don't touch) you must hold the mmap_sem with read bias, i.e. down_read(&mm->mmap_sem)
2. To modify the vmlist you need to hold the mmap_sem with read&write bias, i.e. down_write(&mm->mmap_sem) *AND* you need to take the page_table_lock.
3. The swapper takes _just_ the page_table_lock, this is done because the mmap_sem can be an extremely long lived lock and the swapper just cannot sleep on that.
4. The exception to this rule is expand_stack, which just takes the read lock and the page_table_lock, this is ok because it doesn't really modify fields anybody relies on.
5. You must be able to guarantee that while holding page_table_lock or page_table_lock of mm A, you will not try to get either lock for mm B.

The caveats are:
1. find_vma() makes use of, and updates, the mmap_cache pointer hint. The update of mmap_cache is racy (page stealer can race with other code that invokes find_vma with mmap_sem held), but that is okay, since it is a hint. This can be fixed, if desired, by having find_vma grab the page_table_lock.

Code that add/delete elements from the vmlist chain are
1. callers of insert_vm_struct
2. callers of merge_segments
3. callers of avl_remove

Code that changes vm_start/vm_end/vm_flags:VM_LOCKED of vma's on
the list:
1. expand_stack
2. mprotect
3. mlock
4. mremap

It is advisable that changes to vm_start/vm_end be protected, although
in some cases it is not really needed. Eg, vm_start is modified by
expand_stack(), it is hard to come up with a destructive scenario without
having the vmlist protection in this case.

The page_table_lock nests with the inode i_mmap_lock and the kmem cache
c_spinlock spinlocks. This is okay, since the kmem code asks for pages after
dropping c_spinlock. The page_table_lock also nests with pagecache_lock and
pagemap_lru_lock spinlocks, and no code asks for memory with these locks
held.

The page_table_lock is grabbed while holding the kernel_lock spinning monitor.

The page_table_lock is a spin lock.

Note: PTL can also be used to guarantee that no new clones using the
mm start up ... this is a loose form of stability on mm_users. For
example, it is used in copy_mm to protect against a racing tlb_gather_mmu
single address space optimization, so that the zap_page_range (from
truncate) does not lose sending ipi's to cloned threads that might
be spawned underneath it and go to user mode to drag in pte's into tlbs.

swap_lock

The swap devices are chained in priority order from the "swap_list" header.
The "swap_list" is used for the round-robin swaphandle allocation strategy.
The #free swaphandles is maintained in "nr_swap_pages". These two together
are protected by the swap_lock.

The swap_lock also protects all the device reference counts on the
corresponding swaphandles, maintained in the "swap_map" array, and the
"highest_bit" and "lowest_bit" fields.

The swap_lock is a spinlock, and is never acquired from intr level.

To prevent races between swap space deletion or async readahead swapins
deciding whether a swap handle is being used, ie worthy of being read in
from disk, and an unmap -> swap_free making the handle unused, the swap
delete and readahead code grabs a temp reference on the swaphandle to
prevent warning messages from swap_duplicate <- read_swap_cache_async.

Swap cache locking

Pages are added into the swap cache with kernel_lock held, to make sure
that multiple pages are not being added (and hence lost) by associating
all of them with the same swaphandle.

Pages are guaranteed not to be removed from the scache if the page is
"shared": ie, other processes hold reference on the page or the associated

swap handle. The only code that does not follow this rule is shrink_mmap, which deletes pages from the swap cache if no process has a reference on the page (multiple processes might have references on the corresponding swap handle though). lookup_swap_cache() races with shrink_mmap, when establishing a reference on a scache page, so, it must check whether the page it located is still in the swapcache, or shrink_mmap deleted it. (This race is due to the fact that shrink_mmap looks at the page ref count with pagecache_lock, but then drops pagecache_lock before deleting the page from the scache).

do_wp_page and do_swap_page have MP races in them while trying to figure out whether a page is "shared", by looking at the page_count + swap_count. To preserve the sum of the counts, the page lock _must_ be acquired before calling is_page_shared (else processes might switch their swap_count refs to the page count refs, after the page count ref has been snapshotted).

Swap device deletion code currently breaks all the scache assumptions, since it grabs neither mmap_sem nor page_table_lock.