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 :ref:`Heterogeneous Memory Management (HMM) <hmm>` for migrating pages to or from device private memory.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\(linux-master) (Documentation) (vm)page_migration.rst`, line 12); *backlink*
>
> Unknown interpreted text role "ref".

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 :ref:`CPUSETS <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.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\(linux-master) (Documentation) (vm)page_migration.rst`, line 44); *backlink*
>
> Unknown interpreted text role "ref".

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.

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_page_t that can be passed to migrate_pages(). This function should figure out how to allocate the correct new page given the old page.

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

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.

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.

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

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

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

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

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

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

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

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

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

   Driver shouldn't touch the page.lru field while in the migratepage() function.

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

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

Non-LRU movable page flags

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

- PG_movable

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

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

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

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

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

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

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

  Drivers using __SetPageMovable() should clear the flag via __ClearMovablePage() under page_lock() before the releasing the page.

- PG_isolated

  To prevent concurrent isolation among several CPUs, VM marks isolated page as PG_isolated under lock_page(). So if a CPU encounters PG_isolated non-LRU movable page, it can skip it. Driver doesn't need to manipulate the flag because VM will set/clear it automatically. Keep in mind that if the driver sees a PG_isolated page, it means the page has been isolated by the VM so it shouldn't touch the page.lru field. The PG_isolated flag is aliased with the PG_reclaim flag so drivers shouldn't use PG_isolated for its own purposes.

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 it's 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.