Page migration

Page migration allows the moving of 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.

The main intend of page migration is to reduce the latency of memory access 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 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 ftp://oss.sgi.com/www/projects/libnuma/download/). 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 does only provide 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.
A NUMA profiler may f.e. 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 Documentation/cgroups/cpusets.txt).
Cpusets allows 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.

A. 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 to encounter
   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.

B. 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. Insure that writeback is complete.

3. Prep the new page that we want to move to. It is locked
   and set to not being uptodate so that all accesses to the new
   page immediately lock while the move is in progress.

4. 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.

5. All the page table references to the page are converted
   to migration entries or dropped (nonlinear vmas).
   This decrease 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.

6. The radix tree lock is taken. This will cause all processes trying
   to access the page via the mapping to block on the radix tree spinlock.

7. 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.

8. 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.

9. The radix tree is changed to point to the new page.

10. The reference count of the old page is dropped because the radix tree reference is gone. A reference to the new page is established because the new page is referenced to by the radix tree.

11. The radix tree lock is dropped. With that lookups in the mapping become possible again. Processes will move from spinning on the tree_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 page then replace them with real ptes. Doing so will enable access for user space processes not already waiting for the page lock.

19. 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.

20. The new page is moved to the LRU and can be scanned by the swapper etc again.

Christoph Lameter, May 8, 2006.