Locking and pinning

Did you know...?

LWN.net is a subscriber-supported publication; we rely on subscribers to keep the entire operation going. Please help out by buying a subscription and keeping LWN on the net.

By **Jonathan Corbet**
June 4, 2014

The kernel has long supported the concept of locking pages into physical memory; the `mlock()` system call is one way to accomplish that. But it turns out that there is more than one way to fix memory in place, and some of those ways have to behave differently than others. The result is confusion with resource accounting and suboptimal memory-management behavior in current kernels. A patch set from Peter Zijlstra may soon straighten things out by formalizing a second type of page locking under the name "pinning."

One of the problems with memory locking is that it doesn't quite meet the needs of all users. A page that has been locked into memory with a call like `mlock()` is required to always be physically present in the system's RAM. At a superficial level, locked pages should thus never cause a page fault when accessed by an application. But there is nothing that requires a locked page to always be present in the same place; the kernel is free to move a locked page if the need arises. Migrating a page will cause a soft page fault (one that is resolved without any I/O) the next time an application tries to access that page. Most of the time, that is not a problem, but developers of hard real-time applications go far out of their way to avoid even the small amount of latency caused by a soft fault. These developers would like a firmer form of locking that is guaranteed to never cause page faults. The kernel does not currently provide that level of memory locking.

Locking also fails to meet the needs of various in-kernel users. In particular, kernel code that uses a range of memory as a DMA buffer needs to know that said memory will not be moved. As a result, the locking mechanism has never been used for these pages; instead, they are fixed in place by incrementing their reference counts or through a call to `get_user_pages()`. Such pages are effectively fixed in place, though there is no way for the kernel to know that they may be nailed down for a long time.

There is an interesting question that arises with these informally locked pages, though: how do they interact with the resource limit mechanism? The kernel allows an administrator to place an upper bound on the number of pages that a user is able to lock into memory. But, in some cases, the creation of a DMA buffer shared with user space is the result of an application's request. So users can, for all practical purposes, lock pages in memory via actions like the creation of remote DMA (RDMA) buffers; those pages are not currently counted against the limit on locked pages. This irritates administrators and developers who want the limit on locked pages to apply to *all* locked pages, not just some of them.

These "back door" locked pages also create another sort of problem. Normally, the memory management subsystem goes out of its way to separate pages that can be moved from those that are fixed in place. But, in this case, the pages are often allocated as normal anonymous memory — movable pages, in other words. Fixing them in place makes them unmovable. At that point, they will be in the way any time the memory management code tries to create contiguous ranges of memory by shifting pages around; they are in a place reserved for movable pages, but, being unmovable, they cannot be moved out of the way to make the creation of larger blocks possible.

Peter's patch set tries to address all of these problems — or, at least, to show how they could be addressed. It creates a formal notion of a "pinned" page, being a page that must remain in memory at its current physical location. Pinned pages are kept in a separate virtual memory area (VMA), which is marked with the `VM_PINNED` flag. Within the kernel, pages can be pinned with the new `mm_mpin()` function:

```
    int mm_mpin(unsigned long start, size_t len);
```

This function will pin the pages in memory, but only if the calling process's resource limits allow it. Kernel code that needs to access the pinned memory directly will still need to call `get_user_pages()`, of course; that call should be done after the call to `mm_mpin()`.

One of the longer-term goals (not part of this patch set) is to make this memory-pinning functionality available to user space. A new `mpin()` system call would function like `mlock()`, but with the additional guarantee that the page would never be moved and, thus, would never generate page faults on access. Adding this functionality would mostly appear to be a matter of setting up the system call plumbing.

Another currently unimplemented feature is the migration of the pages to be pinned prior to nailing them down. The `mm_mpin()` call makes it clear that the pages involved will not be movable in the near future. It would thus make sense for the kernel to shift them out of a movable zone (if that is where they are currently located) and into one of the ranges of memory reserved for non-movable pages. That would prevent pinned pages from interfering with memory compaction and, thus, would facilitate the creation of larger blocks of free memory in those pages' original location.

Finally, putting pinned pages under their own VMA makes it relatively easy to keep track of them. So pinned pages can be counted against the locked-pages limit, eliminating that particular loophole.

Thus far, nobody seems to be overly bothered by this patch set. In previous discussions, there have been concerns that changing the accounting of locked pages could cause regressions on some systems where users are running close to their limits. There are few ways around that problem, though; one could continue to leave pinned pages out of the equation or, perhaps, create a separate limit for them. Neither option has a great deal of appeal, so it may just be that this change will go through as-is.

Index entries for this article

Kernel Memory management/Page locking

(Log in to post comments)

Copyright © 2014, Eklektix, Inc.
This article may be redistributed under the terms of the Creative Commons CC BY-SA 4.0 license
Comments and public postings are copyrighted by their creators.
Linux is a registered trademark of Linus Torvalds