**Reverse Mapping**

The most significant and important change to page table management is the introduction of *Reverse Mapping (rmap)*. Referring to it as “rmap” is deliberate as it is the common usage of the “acronym” and should not be confused with the -rmap tree developed by Rik van Riel which has many more alterations to the stock VM than just the reverse mapping.

In a single sentence, rmap grants the ability to locate all PTEs which map a particular page given just the struct page. In 2.4, the only way to find all PTEs which map a shared page, such as a memory mapped shared library, is to linearaly search all page tables belonging to all processes. This is far too expensive and Linux tries to avoid the problem by using the swap cache (see Section [11.4](https://www.kernel.org/doc/gorman/html/understand/understand014.html#sec:%20Swap%20Cache)). This means that with many shared pages, Linux may have to swap out entire processes regardless of the page age and usage patterns. 2.6 instead has a *PTE chain* associated with every struct page which may be traversed to remove a page from all page tables that reference it. This way, pages in the LRU can be swapped out in an intelligent manner without resorting to swapping entire processes.

As might be imagined by the reader, the implementation of this simple concept is a little involved. The first step in understanding the implementation is the union pte that is a field in struct page. This has union has two fields, a pointer to a struct pte\_chain called chain and a pte\_addr\_t called direct. The union is an optisation whereby direct is used to save memory if there is only one PTE mapping the entry, otherwise a chain is used. The type pte\_addr\_t varies between architectures but whatever its type, it can be used to locate a PTE, so we will treat it as a pte\_t for simplicity.

The struct pte\_chain is a little more complex. The struct itself is very simple but it is *compact* with overloaded fields and a lot of development effort has been spent on making it small and efficient. Fortunately, this does not make it indecipherable.

First, it is the responsibility of the slab allocator to allocate and manage struct pte\_chains as it is this type of task the slab allocator is best at. Each struct pte\_chain can hold up to NRPTE pointers to PTE structures. Once that many PTEs have been filled, a struct pte\_chain is allocated and added to the chain.

The struct pte\_chain has two fields. The first is unsigned long next\_and\_idx which has two purposes. When next\_and\_idx is ANDed with NRPTE, it returns the number of PTEs currently in this struct pte\_chain indicating where the next free slot is. When next\_and\_idx is ANDed with the negation of NRPTE (i.e. ∼NRPTE), a pointer to the next struct pte\_chain in the chain is returned[1](https://www.kernel.org/doc/gorman/html/understand/understand006.html" \l "note9). This is basically how a PTE chain is implemented.

To give a taste of the rmap intricacies, we'll give an example of what happens when a new PTE needs to map a page. The basic process is to have the caller allocate a new pte\_chain with pte\_chain\_alloc(). This allocated chain is passed with the struct page and the PTE to page\_add\_rmap(). If the existing PTE chain associated with the page has slots available, it will be used and the pte\_chain allocated by the caller returned. If no slots were available, the allocated pte\_chain will be added to the chain and NULL returned.

There is a quite substantial API associated with rmap, for tasks such as creating chains and adding and removing PTEs to a chain, but a full listing is beyond the scope of this section. Fortunately, the API is confined to mm/rmap.c and the functions are heavily commented so their purpose is clear.

There are two main benefits, both related to pageout, with the introduction of reverse mapping. The first is with the setup and tear-down of pagetables. As will be seen in Section [11.4](https://www.kernel.org/doc/gorman/html/understand/understand014.html#sec:%20Swap%20Cache), pages being paged out are placed in a swap cache and information is written into the PTE necessary to find the page again. This can lead to multiple minor faults as pages are put into the swap cache and then faulted again by a process. With rmap, the setup and removal of PTEs is atomic. The second major benefit is when pages need to paged out, finding all PTEs referencing the pages is a simple operation but impractical with 2.4, hence the swap cache.

Reverse mapping is not without its cost though. The first, and obvious one, is the additional space requirements for the PTE chains. Arguably, the second is a CPU cost associated with reverse mapping but it has not been proved to be significant. What is important to note though is that reverse mapping is only a benefit when pageouts are frequent. If the machines workload does not result in much pageout or memory is ample, reverse mapping is all cost with little or no benefit. At the time of writing, the merits and downsides to rmap is still the subject of a number of discussions.

**Object-Based Reverse Mapping**

The reverse mapping required for each page can have very expensive space requirements. To compound the problem, many of the reverse mapped pages in a VMA will be essentially identical. One way of addressing this is to reverse map based on the VMAs rather than individual pages. That is, instead of having a reverse mapping for each page, all the VMAs which map a particular page would be traversed and unmap the page from each. Note that objects in this case refers to the VMAs, not an object in the object-orientated sense of the word[2](https://www.kernel.org/doc/gorman/html/understand/understand006.html" \l "note10). At the time of writing, this feature has not been merged yet and was last seen in kernel 2.5.68-mm1 but there is a strong incentive to have it available if the problems with it can be resolved. For the very curious, the patch for just file/device backed objrmap at this release is available [3](https://www.kernel.org/doc/gorman/html/understand/understand006.html" \l "note11) but it is only for the very very curious reader.

There are two tasks that require all PTEs that map a page to be traversed. The first task is page\_referenced() which checks all PTEs that map a page to see if the page has been referenced recently. The second task is when a page needs to be unmapped from all processes with try\_to\_unmap(). To complicate matters further, there are two types of mappings that must be reverse mapped, those that are backed by a file or device and those that are anonymous. In both cases, the basic objective is to traverse all VMAs which map a particular page and then walk the page table for that VMA to get the PTE. The only difference is how it is implemented. The case where it is backed by some sort of file is the easiest case and was implemented first so we'll deal with it first. For the purposes of illustrating the implementation, we'll discuss how page\_referenced() is implemented.

page\_referenced() calls page\_referenced\_obj() which is the top level function for finding all PTEs within VMAs that map the page. As the page is mapped for a file or device, page→mapping contains a pointer to a valid address\_space. The address\_space has two linked lists which contain all VMAs which use the mapping with the address\_space→i\_mmap and address\_space→i\_mmap\_shared fields. For every VMA that is on these linked lists, page\_referenced\_obj\_one() is called with the VMA and the page as parameters. The function page\_referenced\_obj\_one() first checks if the page is in an address managed by this VMA and if so, traverses the page tables of the mm\_struct using the VMA (vma→vm\_mm) until it finds the PTE mapping the page for that mm\_struct.

Anonymous page tracking is a lot trickier and was implented in a number of stages. It only made a very brief appearance and was removed again in 2.5.65-mm4 as it conflicted with a number of other changes. The first stage in the implementation was to use page→mapping and page→index fields to track mm\_struct and address pairs. These fields previously had been used to store a pointer to swapper\_space and a pointer to the swp\_entry\_t (See Chapter [11](https://www.kernel.org/doc/gorman/html/understand/understand014.html#chap:%20Swap%20Management)). Exactly how it is addressed is beyond the scope of this section but the summary is that swp\_entry\_t is stored in page→private

try\_to\_unmap\_obj() works in a similar fashion but obviously, all the PTEs that reference a page with this method can do so without needing to reverse map the individual pages. There is a serious search complexity problem that is preventing it being merged. The scenario that describes the problem is as follows;

Take a case where 100 processes have 100 VMAs mapping a single file. To unmap a *single* page in this case with object-based reverse mapping would require 10,000 VMAs to be searched, most of which are totally unnecessary. With page based reverse mapping, only 100 pte\_chain slots need to be examined, one for each process. An optimisation was introduced to order VMAs in the address\_space by virtual address but the search for a single page is still far too expensive for object-based reverse mapping to be merged.