# Transparent Hugepage Support

This document describes design principles for Transparent Hugepage (THP) support and its interaction with other parts of the memory management system.

## Design principles

- "graceful fallback": mm components which don't have transparent hugepage knowledge fall back to breaking huge pmd mapping into table of ptes and, if necessary, split a transparent hugepage. Therefore these components can continue working on the regular pages or regular pte mappings.
- if a hugepage allocation fails because of memory fragmentation, regular pages should be gracefully allocated instead and mixed in the same vma without any failure or significant delay and without userland noticing
- if some task quits and more hugepages become available (either immediately in the buddy or through the VM), guest physical memory backed by regular pages should be relocated on hugepages automatically (with khugepaged)
- it doesn't require memory reservation and in turn it uses hugepages whenever possible (the only possible reservation here is kernelcore= to avoid unmovable pages to fragment all the memory but such a tweak is not specific to transparent hugepage support and it's a generic feature that applies to all dynamic high order allocations in the kernel)

## get_user_pages and follow_page

get_user_pages and follow_page if run on a hugepage, will return the head or tail pages as usual (exactly as they would do on hugetlbfs). Most GUP users will only care about the actual physical address of the page and its temporary pinning to release after the I/O is complete, so they won't ever notice the fact the page is huge. But if any driver is going to mangle over the page structure of the tail page (like for checking page->mapping or other bits that are relevant for the head page and not the tail page), it should be updated to jump to check head page instead. Taking a reference on any head/tail page would prevent the page from being split by anyone.

> **Note**
>
> these aren't new constraints to the GUP API, and they match the same constraints that apply to hugetlbfs too, so any driver capable of handling GUP on hugetlbfs will also work fine on transparent hugepage backed mappings.

## Graceful fallback

Code walking pagetables but unaware about huge pmds can simply call split_huge_pmd(vma, pmd, addr) where the pmd is the one returned by pmd_offset. It's trivial to make the code transparent hugepage aware by just grepping for "pmd_offset" and adding split_huge_pmd where missing after pmd_offset returns the pmd. Thanks to the graceful fallback design, with a one liner change, you can avoid to write hundreds if not thousands of lines of complex code to make your code hugepage aware.

If you're not walking pagetables but you run into a physical hugepage that you can't handle natively in your code, you can split it by calling split_huge_page(page). This is what the Linux VM does before it tries to swapout the hugepage for example. split_huge_page() can fail if the page is pinned and you must handle this correctly.

Example to make mremap.c transparent hugepage aware with a one liner change:

```
diff --git a/mm/mremap.c b/mm/mremap.c
--- a/mm/mremap.c
+++ b/mm/mremap.c
@@ -41,6 +41,7 @@ static pmd_t *get_old_pmd(struct mm_stru
                return NULL;

        pmd = pmd_offset(pud, addr);
+       split_huge_pmd(vma, pmd, addr);
        if (pmd_none_or_clear_bad(pmd))
                return NULL;
```

## Locking in hugepage aware code

We want as much code as possible hugepage aware, as calling split_huge_page() or split_huge_pmd() has a cost.

To make pagetable walks huge pmd aware, all you need to do is to call pmd_trans_huge() on the pmd returned by pmd_offset. You must hold the mmap_lock in read (or write) mode to be sure a huge pmd cannot be created from under you by khugepaged (khugepaged collapse_huge_page takes the mmap_lock in write mode in addition to the anon_vma lock). If pmd_trans_huge returns false, you just fallback in the old code paths. If instead pmd_trans_huge returns true, you have to take the page table lock (pmd_lock()) and re-run pmd_trans_huge. Taking the page table lock will prevent the huge pmd being converted into a regular pmd from under you (split_huge_pmd can run in parallel to the pagetable walk). If the second pmd_trans_huge returns false, you should just drop the page table lock and fallback to the old code as before. Otherwise, you can proceed to process the huge pmd and the hugepage natively. Once finished, you can drop the page table lock.

## Refcounts and transparent huge pages

Refcounting on THP is mostly consistent with refcounting on other compound pages:

- get_page()/put_page() and GUP operate on head page's ->_refcount.
- ->_refcount in tail pages is always zero: get_page_unless_zero() never succeeds on tail pages.
- map/unmap of the pages with PTE entry increment/decrement ->_mapcount on relevant sub-page of the compound page.
- map/unmap of the whole compound page is accounted for in compound_mapcount (stored in first tail page). For file huge pages, we also increment ->_mapcount of all sub-pages in order to have race-free detection of last unmap of subpages.

PageDoubleMap() indicates that the page is *possibly* mapped with PTEs.

For anonymous pages, PageDoubleMap() also indicates ->_mapcount in all subpages is offset up by one. This additional reference is required to get race-free detection of unmap of subpages when we have them mapped with both PMDs and PTEs.

This optimization is required to lower the overhead of per-subpage mapcount tracking. The alternative is to alter ->_mapcount in all subpages on each map/unmap of the whole compound page.

For anonymous pages, we set PG_double_map when a PMD of the page is split for the first time, but still have a PMD mapping. The additional references go away with the last compound_mapcount.

File pages get PG_double_map set on the first map of the page with PTE and goes away when the page gets evicted from the page cache.

split_huge_page internally has to distribute the refcounts in the head page to the tail pages before clearing all PG_head/tail bits from the page structures. It can be done easily for refcounts taken by page table entries, but we don't have enough information on how to distribute any additional pins (i.e. from get_user_pages). split_huge_page() fails any requests to split pinned huge pages: it expects page count to be equal to the sum of mapcount of all sub-pages plus one (split_huge_page caller must have a reference to the head page).

split_huge_page uses migration entries to stabilize page->_refcount and page->_mapcount of anonymous pages. File pages just get unmapped.

We are safe against physical memory scanners too: the only legitimate way a scanner can get a reference to a page is get_page_unless_zero().

All tail pages have zero ->_refcount until atomic_add(). This prevents the scanner from getting a reference to the tail page up to that point. After the atomic_add() we don't care about the ->_refcount value. We already know how many references should be uncharged from the head page.

For head page get_page_unless_zero() will succeed and we don't mind. It's clear where references should go after split: it will stay on the head page.

Note that split_huge_pmd() doesn't have any limitations on refcounting: pmd can be split at any point and never fails.

## Partial unmap and deferred_split_huge_page()

Unmapping part of THP (with munmap() or other way) is not going to free memory immediately. Instead, we detect that a subpage of THP is not in use in page_remove_rmap() and queue the THP for splitting if memory pressure comes. Splitting will free up unused subpages.

Splitting the page right away is not an option due to locking context in the place where we can detect partial unmap. It also might be counterproductive since in many cases partial unmap happens during exit(2) if a THP crosses a VMA boundary.

The function deferred_split_huge_page() is used to queue a page for splitting. The splitting itself will happen when we get memory pressure via shrinker interface.