



Transparent Huge Pages



Table of contents

- Why we need contiguous memory and huge pages?
- Page sizes
- Hugetlbfs, Transparent Huge Pages and DAX
- Transparent Huge Pages
 - Advantages and disadvantages
 - Original implementation
 - Improvements
 - Huge zero page
 - Controlling
 - Support for tmpfs/shmem
 - THP for filesystems
 - Reducing data structures
- Compound pages and folios



How to capture 100G Ethernet traffic at wire speed to local disk”

Christoph Lameter (LCA 2020)

<https://www.youtube.com/watch?v=uBBaVtHkiOI>

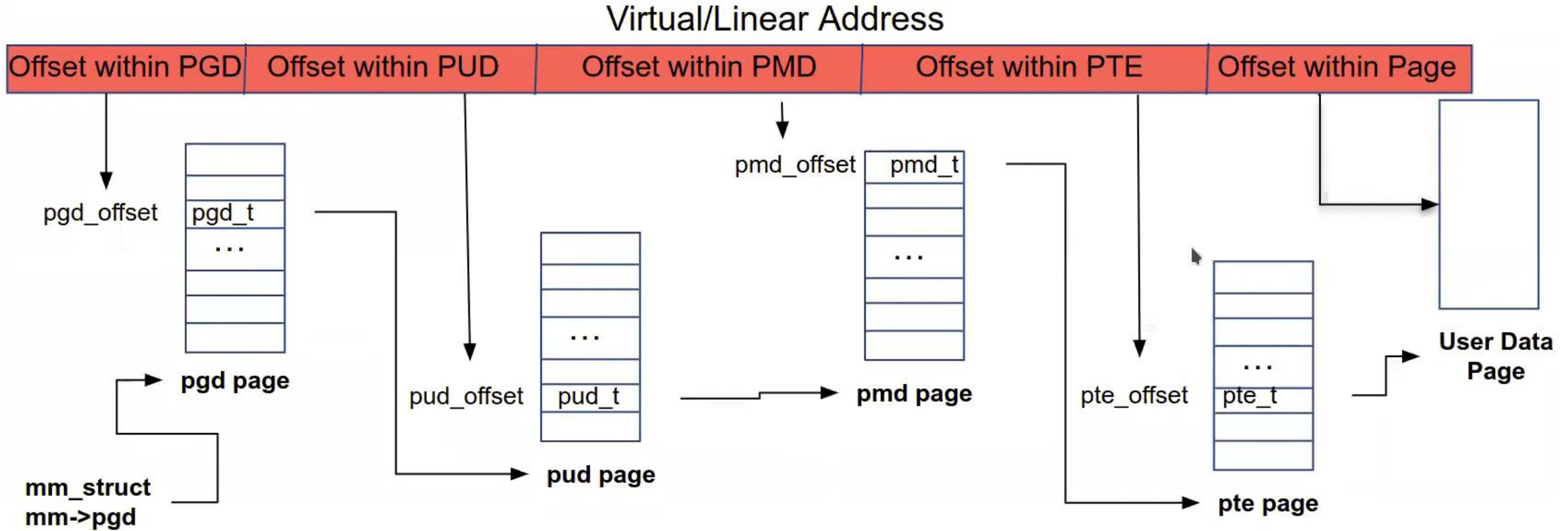
The best solution would be to have the OS do what is necessary for high performance. We need

- **Contiguous memory.**
- **Larger chunks of memory than 4K** managed by the OS.
- If we want to support **multiple page sizes** then we need the ability to defragment memory. A fundamental change how we manage objects in the kernel. They would need to be **movable** in order to recover contiguous memory areas to be able to consistently provide larger page sizes than the basic page.

Significant work is required: it will take at least **another 2 years** to implement these things upstream. This means **another 5 years** are required until the Linux distributions will have these features.



Page size 4KB

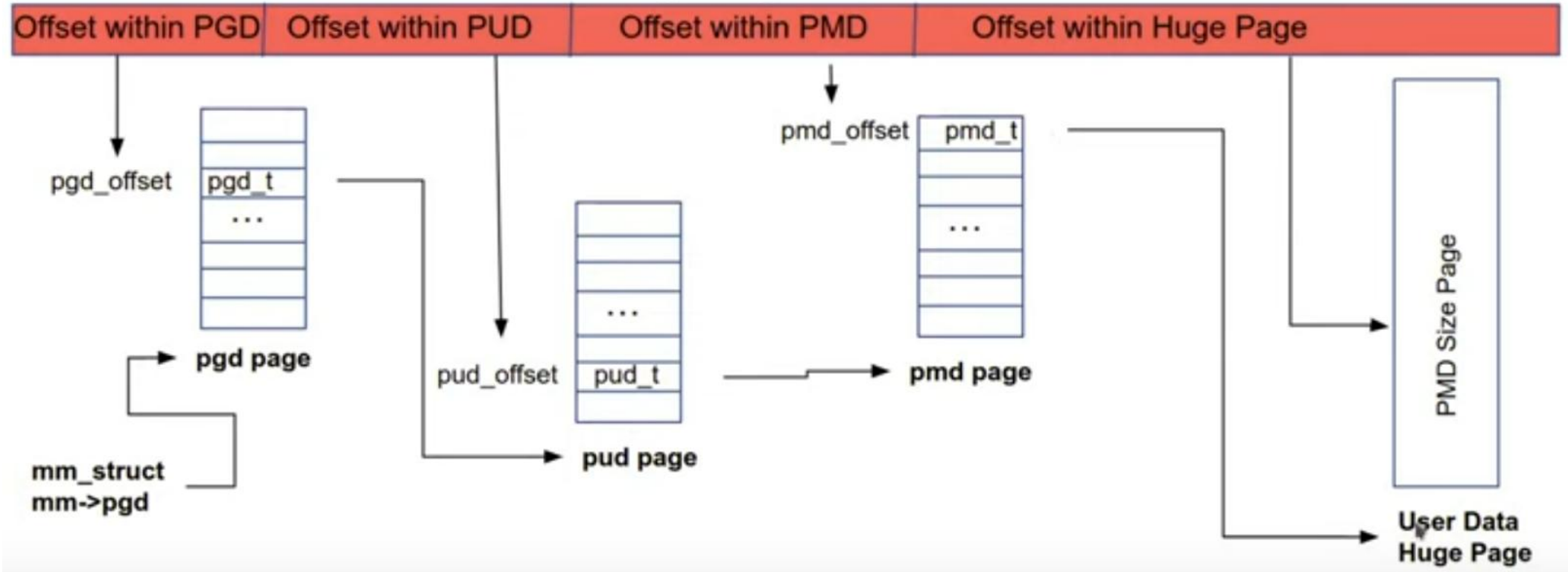


Mike Kravetz, Huge page concepts in Linux,
<https://www.youtube.com/watch?v=n67gCNiKVcw>



Page size 2MB

Huge Page at PMD Level



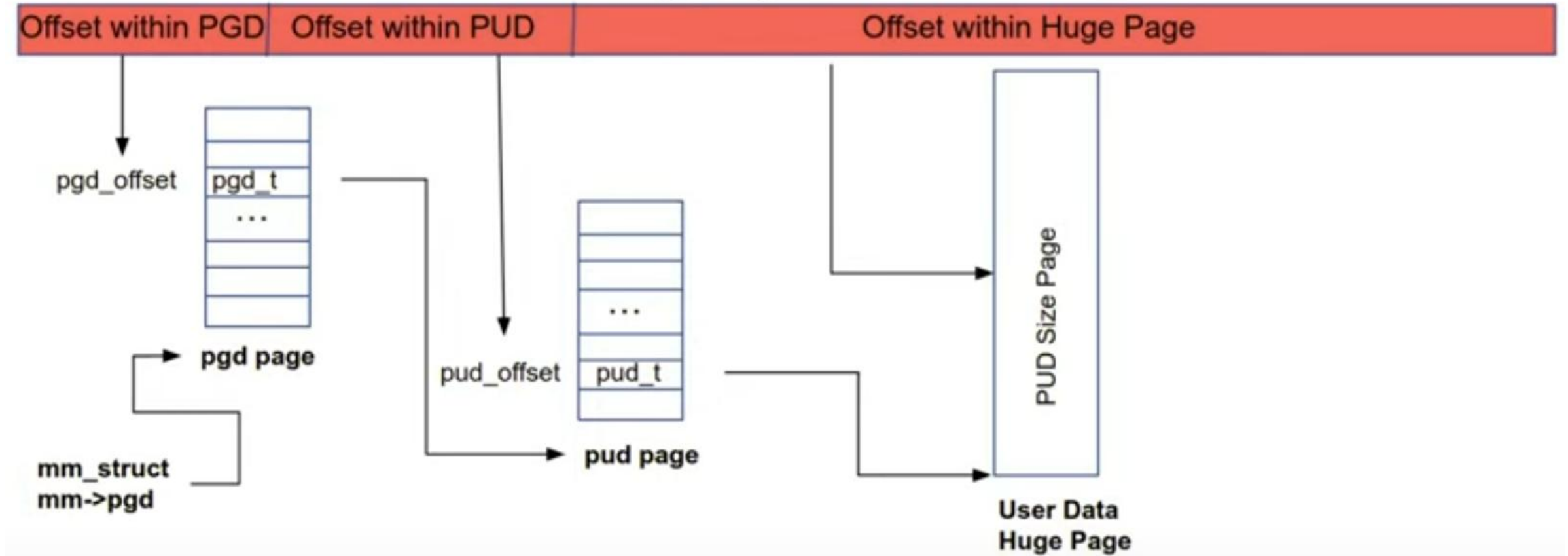
Page_PSE – Page Size Extension
Huge page size = $2^{(12+9)} = 2^{21} = 2\text{MB}$

Mike Kravetz, Huge page concepts in Linux,
<https://www.youtube.com/watch?v=n67gCNiKVcw>



Page size 1GB

Huge Page at PUD Level



Huge page size = $2^{(12+9+9)} = 2^{30} = 1\text{GB}$

Mike Kravetz, Huge page concepts in Linux,
<https://www.youtube.com/watch?v=n67gCNiKVcw>



Possible approaches – overview (2018)

- **Hugetlbfs** (RAM-based filesystem)
 - Original/oldest method.
 - Preallocation at boot or early system init time.
 - Memory ONLY available for hugetlbfs.
 - Applications must change (open/mmap).
 - Multiple huge page sizes (architecture dependent).
 - Good for ‘single purpose’ use cases.
- **Transparent Huge Pages (THP)**
 - Enabled by default on most distributions.
 - No system configuration or application changes required (although desirable for optimal usage).
 - Single huge page size (PMD_SIZE).
- **DAX (Persistent Memory)**
 - Mechanism that is used to bypass the page cache and map files stored in persistent memory directly into user space.
 - Uses 2M and 1G mappings by default.

Promoting huge page usage

Christopher Lameter, Mike Kravetz

<https://www.linuxplumbersconf.org/event/2/contributions/157/>

Linux Plumbers Conference 2018,
Vancouver



Huge pages

```
jmd@duch:~$ tree /sys/kernel/mm/hugepages/  
/sys/kernel/mm/hugepages/  
|-- hugepages-1048576kB  
|   |-- demote  
|   |-- demote_size  
|   |-- free_hugepages  
|   |-- nr_hugepages  
|   |-- nr_hugepages_mempolicy  
|   |-- nr_overcommit_hugepages  
|   |-- resv_hugepages  
|   `-- surplus_hugepages  
`-- hugepages-2048kB  
    |-- free_hugepages  
    |-- nr_hugepages  
    |-- nr_hugepages_mempolicy  
    |-- nr_overcommit_hugepages  
    |-- resv_hugepages  
    `-- surplus_hugepages  
  
3 directories, 14 files
```

Hugetlbfs multiple page size pools

$1048576\text{KB} = 1024 * 1024\text{KB} = 2^{20} * 2^{10}\text{B} = 2^{30}\text{B} = 1\text{GB}$
 $2048\text{KB} = 2 * 2^{10} * 2^{10}\text{B} = 2\text{MB}$

```
jmd@duch:~$ grep Huge /proc/meminfo  
AnonHugePages:      4939776 kB  
ShmemHugePages:        0 kB  
FileHugePages:       0 kB  
HugePages_Total:      0  
HugePages_Free:       0  
HugePages_Rsvd:       0  
HugePages_Surp:       0  
Hugepagesize:        2048 kB  
Hugetlb:              0 kB
```

Default hugetlb page size: **2MB**



THP – Advantages and disadvantages



- THP works by **quietly substituting huge pages** into a process's address space when
 - those page are available and
 - it appears that the process would benefit from the change.
- Introduced to Linux in 2.6.38 (**2011**) by **Andrea Arcangeli**.
- **Advantages**
 - The size of page tables decreases, as does the number of page faults required to get an application into RAM.
 - A single TLB entry will be mapping a much larger amount of virtual memory in turn reducing the number of TLB misses.
 - The TLB miss will run faster.
- **Disadvantages**
 - The amount of wasted memory will increase as a result of internal fragmentation.
 - Larger pages take longer to transfer from secondary storage, increasing **page fault latency** (while decreasing **page fault counts**).
 - The time required to simply clear very large pages can create significant kernel latencies.



THP – Original implementation

- **Patchset by Andrea Arcangeli (2011)**
 - When a fault happens, the kernel will **attempt to allocate** a huge page to satisfy it. Should the allocation succeed, the huge page will be filled, any existing small pages in the new page's address range will be released, and the huge page will be inserted into the VMA. If no huge pages are available, the kernel **falls back** to small pages and the application never knows the difference.
 - Huge pages must be **swappable**, lest the system run out of memory. Rather than complicate the swapping code with an understanding of huge pages, **a huge page is split back into its component small pages** if that page needs to be reclaimed.
 - **Khugepaged** kernel thread will occasionally attempt to allocate a huge page; if it succeeds, it will scan through memory looking for a place where that huge page can be substituted for a bunch of smaller pages. Thus, available huge pages should be quickly placed into service, maximizing the use of huge pages in the system as a whole.
 - The current patch only works with **anonymous pages**.
 - Up to 10% improvements for some benchmarks.



THP – Enhancements

- **Patchset by Kirill Shutemov – Zero pages** (2012)
 - Adds a special, zero-filled huge page to function as the huge zero page. Only one such page is needed, since the transparent huge pages feature only uses one size of huge page.
- **Patchset by Andi Kleen – Supporting variable-sized huge pages** (2013)
 - Some modern architectures permit multiple huge page sizes, and where the system admin has configured the system to provide huge page pools of different sizes, applications may want to choose the page size used for their allocation.
 - Extends the `shmget()` and `mmap()` system calls to allow the caller to select the size used for huge page allocations.
- **Patchset by Alex Thorlton – Controlling transparent huge pages** (2013)
 - The feature can be **turned off globally**, but what about situations where some applications benefit while others do not?
 - Provides an option to **disable transparent huge pages on a per-process basis**. This operation sets a flag in the `task_struct` structure; setting that flag causes the memory management system to avoid using huge pages for the associated process. And that allows the creation of mixed workloads, where some processes use transparent huge pages and others do not.



THP for filesystems (page cache)

- [Patchset by Kirill Shutemov](#) – **THP in the page cache** (October 2016, ver. 4.8)
 - Two implementations competed (Hugh Dickins from Google lost).
 - Adds support for transparent huge pages in the page cache in **tmpfs/shmem** (other filesystems may be added in the future).
 - One of the primary goals was the ability for applications to access individual 4KB subpages of a huge page without the need to split the huge page itself.
 - Using **compound pages** (used also for anonymous THP and for files in the hugetlbfs). The first of the range of (small) pages that makes up a huge page is the **head page**, while the rest are **tail pages**. Most of the important metadata is stored in the head page. Using compound pages allows the entire huge page to be represented by a single entry in the LRU lists, and all buffer-head structures, if any, are tied to the head page.
 - Unlike DAX, transparent huge pages do not force any constraints on a file's on-disk layout.
- [Huge pages in the ext4fs](#) (2017)

With tmpfs the creation of a huge page causes the addition of 512 (single-page) entries to the radix tree; this cannot work in ext4. It is also necessary to add DAX support and to make it work consistently. There are a few other problems; for example, readahead doesn't currently work with huge pages. The maximum size of the readahead window is 128 KB, far less than the size of a huge page. Huge pages also cause any [shadow entries](#) in the page cache to be ignored, which could worsen the system's page-reclaim decisions.



Kirill A. Shutemov



THP for filesystems, performance

- [Transparent huge pages for filesystems](#) (2019)
 - Facebook is trying to reduce misses on the TLB for instructions by putting hot functions into huge pages. Those functions are collected up into an 8MB region in the generated executable.
 - At run time, the application creates an 8MB temporary buffer and the hot section of the executable memory is copied to it. The 8MB region in the executable memory is then converted to a huge page.
 - This results in a 5-10% performance boost without requiring any kernel changes to support it.
 - There is no support for writing to the THP, thus no **writeback** is required. That would prove to be a sticking point.
- [Transparent huge pages, NUMA locality and performance regressions, Dueling memory-management performance regressions](#) (2019)
 - Some patches were made to tackle with performance regressions, but then have to be reverted.
 - If the system is configured to always enable memory compaction and a huge page allocation is requested, the page allocator will refuse to allocate pages on remote nodes. It behaves as if the program had been explicitly bound to the current node, which was never the intended result. The reasoning that led to this behavior is that it is better to allocate local 4KB pages than remote huge pages. But the kernel goes beyond that in this situation, refusing to allocate *any* pages on remote nodes and potentially forcing the local node deeply into swap.



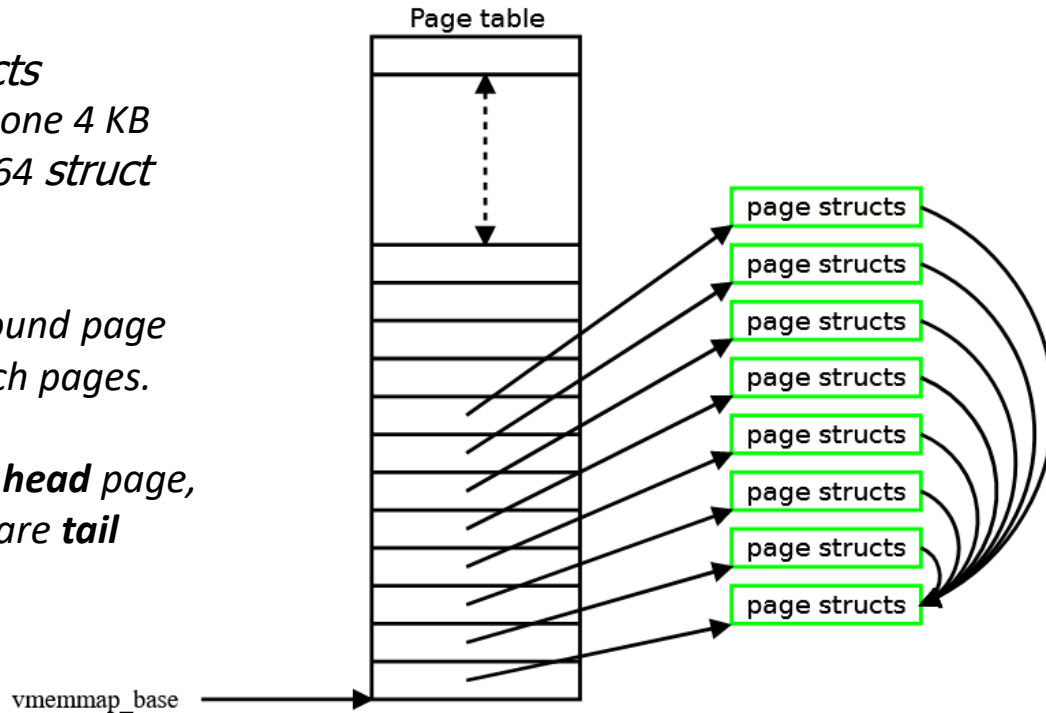
THP – Data structures

- [Minimizing the use of tail pages](#) (2019), [Reducing page structures for huge pages](#) (December 2020) – not merged in May 2021
 - Struct page structures (64 bytes) typically take up 1,5% of the available memory.
 - For 1GB huge pages the patchset can eliminate 4094 of the 4096 pages representing tail pages.
 - When huge pages are split up, the full set of page structures must be restored.

Page structs represents one 4 KB page with 64 struct pages.

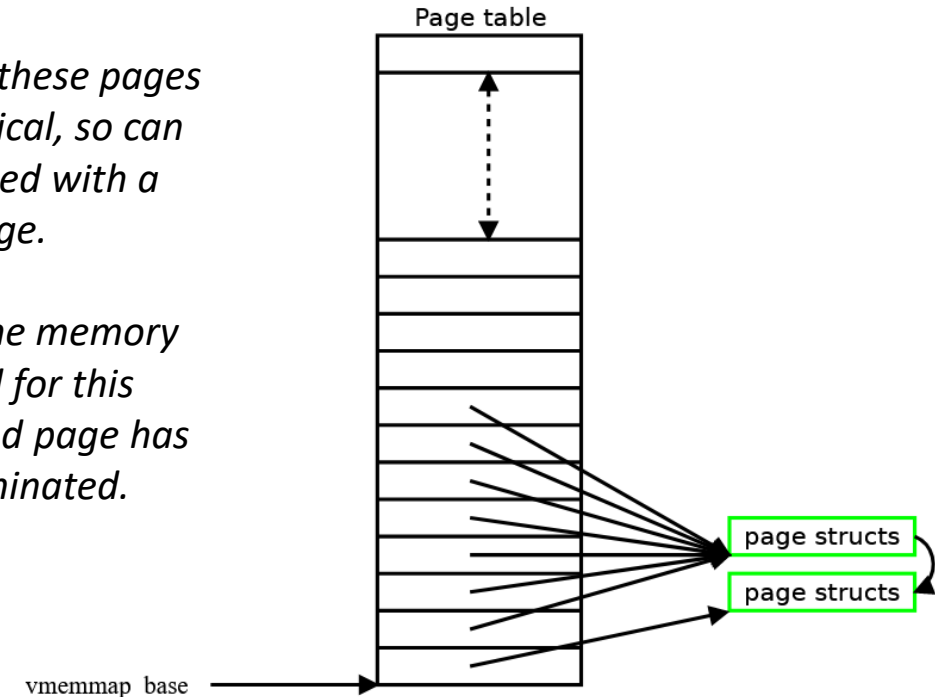
One compound page needs 8 such pages.

*First one is **head** page, the others are **tail** pages.*



Seven of these pages are identical, so can be replaced with a single page.

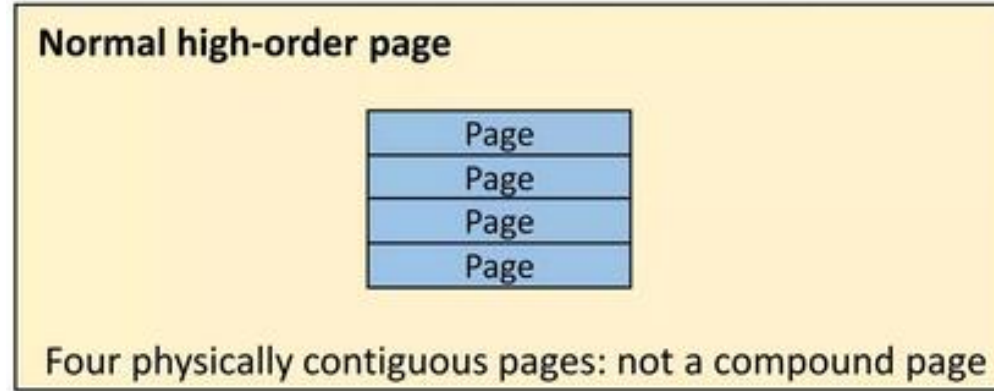
75% of the memory overhead for this compound page has been eliminated.



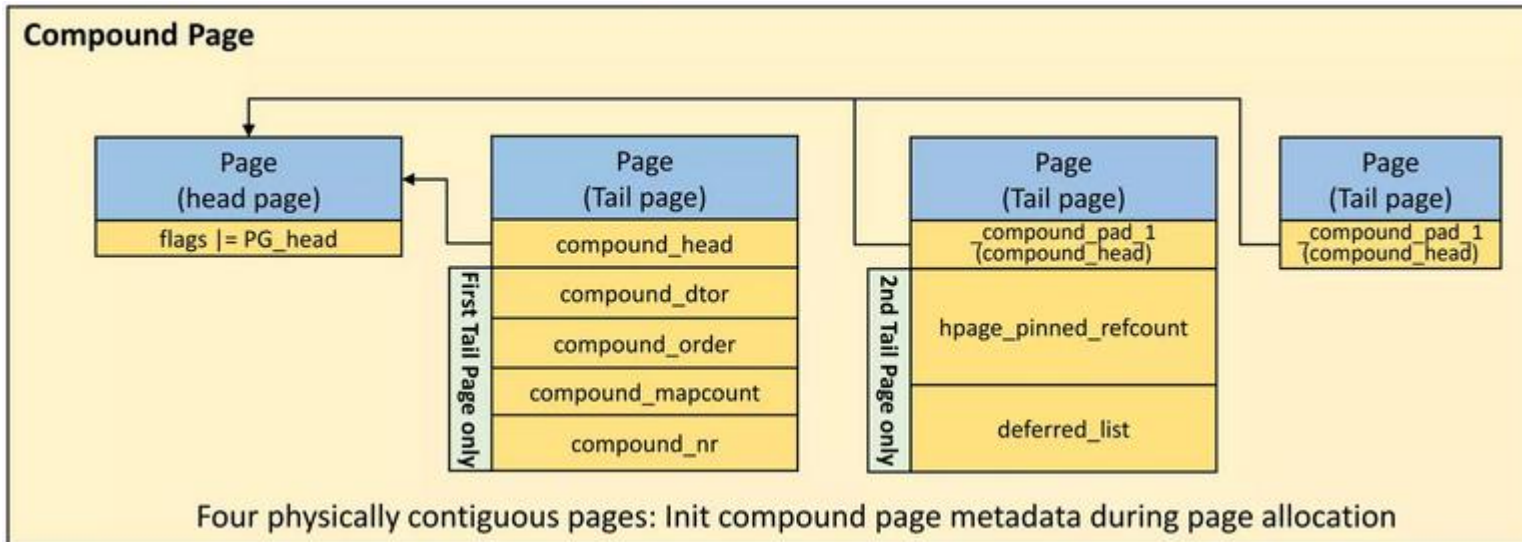


Normal high-order page and compound page

```
pages = alloc_pages(GFP_KERNEL, 2);
```



```
pages = alloc_pages(GFP_KERNEL | __GFP_COMP, 2);
```



A page of a size 2MB contains 512 pages of a size 4KB.

It needs 512 page table entries for page structs of a size 64KB. These page structs use 8 pages of size 4KB, but only the first of them contains useful data. The other page struct entries (all but the first two tail pages) just point to the first page struct (head page).

(source: Adrian Huang, [Memory Management with Page Folios](#), 2023)



Page folios

[Clarifying memory management with page folios](#), Jonathan Corbet, March 2021

- Many interfaces within the kernel make use of that feature, but it creates a fundamental ambiguity: if a function is passed a pointer to a page structure for a tail page, is it expected to act on that tail page or on the compound page as a whole?
- Wilcox has come up with the concept of a "**page folio**", which is really just a page structure that is guaranteed not to be a tail page. Any function accepting a folio will operate on the **full compound page** (if, indeed, it is a compound page) with no ambiguity.
- The result is **greater clarity** in the kernel's memory-management subsystem; as functions are converted to take folios as arguments, it will become clear that they are not meant to operate on tail pages.
- Memory folio's goal
 - Unified interface (all accesses via folio struct)
 - Shorter LRU list
 - Original: one struct page per 4KB to be added to the LRU list
 - Folio: one struct page (page head) per 8KB, 16KB, 64KB, 128KB ... and so on (include) THP to be added to the LRU list.
 - [Anonymous page] THP support



page struct vs folio struct

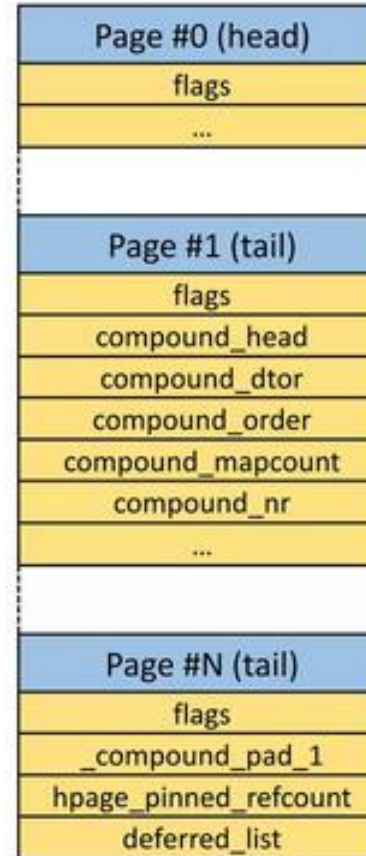
folio's benefit

- [Example] 512KB compound page
 - ✓ page struct: Need to maintain 128 page structs (1 head page and 127 tail pages)
 - ✓ folio struct: Maintain 3 page structs regardless of the size of compound pages.

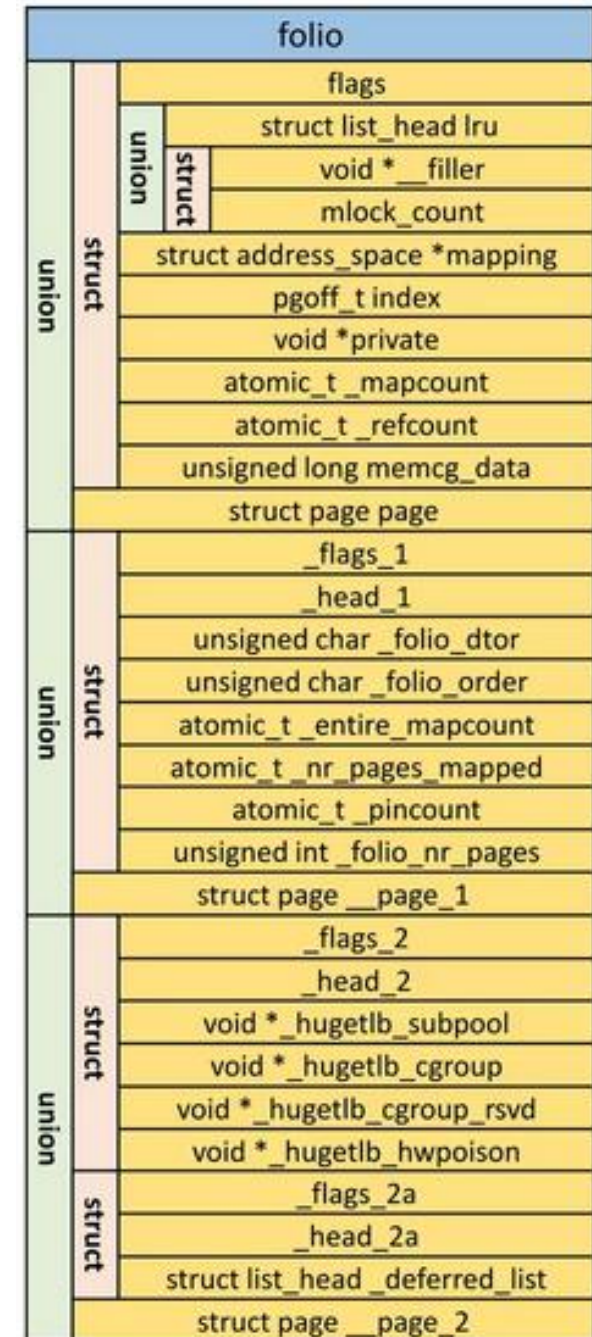
folio struct's members

- `_entire_mapcount`
 - ✓ The compound page is mapped via a single PMD (huge page).
- `_nr_pages_mapped`
 - ✓ Number of individual subpages (PTE: 4KB pages) are mapped.
 - ✓ Scenario: Two processes map the same memory range
 - ✓ One process maps the entire 2MB compound page (Transparent Huge Page - THP): mapped via a single PMD
 - ✓ The other process maps some 4KB pages within this 2MB memory area: mapped via PTEs
 - ✓ Benefit about THP: No need to split the huge page if other processes map 4KB pages within the same memory area.
- `_folio_nr_pages`
 - ✓ Number of pages in this folio.
 - ✓ `_folio_nr_pages = 1 << order`, where `order > 0`.

Compound pages



•
•
•



(source: Adrian Huang, [Memory Management with Page Folios](#), 2023)



Additional reading

- https://lwn.net/Kernel/Index/#Huge_pages
- <https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt>
- [/Documentation/mm/transhuge.rst](#)
- [Proactive compaction](#), Jonathan Corbet, 2017.
- [Proactive compaction for the kernel](#), Nitin Gupta, April 2020.
- [Memory: the flat, the discontinuous, and the sparse](#), Mike Rapoport, May 2019.
- [The end of the DAX experiment](#), Jonathan Corbet, May 2019.
- [Sidestepping kernel memory management with DMEMFS](#), Jonathan Corbet, December 2020.
- [Large Pages in Linux](#), Matthew Wilcox, LCA 2020.
- [Large Pages in the Linux kernel](#), Matthew Wilcox, February 2021.
- [LWN – A memory-folio update](#), Jonathan Corbet, May 2022