Memory Management with Huge Pages

Vlastimil Babka
SUSE Labs
vbabka@suse.cz
Agenda

• Why huge pages?
  - Benefits (and disadvantages)

• Implementation in Linux kernel
  - HugeTLBfs and THP

• Allocating huge pages (from the buddy allocator)
  - Memory compaction
  - Page grouping by mobility

• Allocations beyond buddy allocator limits
  - Gigantic hugepages, ZONE_MOVABLE, CMA
Why Huge Pages?

- Buddy allocator page granularity is few KB (i.e. 4KB)
  - Reflects CPU virtual address translation granularity
- Translation via 4 (soon 5) level page tables too slow (by CPU, or kernel in a exception handler, e.g. MIPS)
  - CPU caches the translations in TLB, but capacity is limited
    - AMD Ryzen / Intel Skylake: 1600 entries will cover 6400 KB range
- Idea: larger pages will cover wider range with same number of TLB entries
  - Ryzen/Skylake: 1600 2MB entries covers 3200 MB range
  - Skipping a level of address translation via page tables
  - Variable TLB entry size (MIPS)
Example: x86_64 page tables

Virtual Address

<table>
<thead>
<tr>
<th>Sign Extend</th>
<th>Page-Map Level-4 Offset (PML4)</th>
<th>Page-Directory-Pointer Offset</th>
<th>Page-Directory Offset</th>
<th>Page-Table Offset</th>
<th>Physical-Page Offset</th>
</tr>
</thead>
</table>

Page-Map Level-4 Table

pgd

Page-Directory-Pointer Table

pud

Page-Directory Table

pmd

Page Table

PTE

4 Kbyte Physical Page

Physical Address

*This is an architectural limit. A given processor implementation may support fewer bits.

Image from AMD64 Architecture Programmer’s Manual Volume 2: System Programming
Less Known/Advertised Benefits

• Smaller memory usage by page tables
  - Also less (L2, L3) cache pressure by walking these tables

• More uniform cache set utilization in physically indexed set-associative caches
  - Better average performance
  - More predictable performance
  - Achievable also without huge pages, but more complicated
Less Known/Advertised Benefits

• More uniform cache set utilization in physically indexed set-associative caches
  - Better average performance
  - More predictable performance
  - Achievable also without huge pages, but more complicated [1]

• Example: 32-bit Intel 512KB 8-way L2 cache, 64B line

![Diagram of cache address and offset]

TLB and cache miss costs?

• Intel Xeon E5345 (Clovertown) [1]
  - 2 cycles DTLB0, +7 DTLB1, +4 pmd cache, +8 pud cache
  - 11 cycles L1 data, 256 L2 unified

• AMD Opteron 2356 (Barcelona) [1]
  - 5 cycles L1 DTLB, +35 L2 DTLB, +21 for each cache level
  - 12-40 L1 data, 16-63 L2 unified, 159-211 L3 unified

• Real workload hugepage speedups on PPC970MP [2]
  - STREAM synthetic benchmark – 11-16%
  - SysBench OLTP benchmark – 1-3.5%
  - SpecCPU 2006 – 13% integer, 7-8% floating point
  - SpecJVM 2008 – 4.4%

Disadvantages of Using Huge Pages

• Potentially wasted memory when sparsely used
  - Or sparse modifications in COW scenario
• Huge page allocation overhead (more on that later)
• I/O amplification when swapping
  - Or no swapping support at all
• False sharing amplification on NUMA machines

• Basically same arguments as against simply increasing base page size for everyone
Implementation in Linux

- Kernel mapping of physical memory
  - Uses 1GB or 2MB huge pages when possible
  - Direct mapping, ioremap() for device memory ranges

- Older, explicit hugepage usage – HugeTLBfs
  - Pre-allocated in pools, accessible by several interfaces
  - Private or shared, no splitting, no swapping
  - Multiple sizes supported; page table sharing support

- Newer, transparent hugepage usage – THP
  - Allocated implicitly, possible to prefer or disallow by hints
  - Anonymous, private (except fork+COW), can be split back to base pages and then swapped out

- Windows: large pages (explicit), FreeBSD: superpages (like THP)
HugeTLBfs
HugeTLBfs Usage

• SysV shared memory segment
  - `shmid = shmget(key, SIZE, SHM_HUGETLB | ...);`
  - `addr = shmat(shmid, NULL, 0);`
  - Since 3.8: alternative flags SHM_HUGE_2MB, SHM_HUGE_1GB, and SHM_HUGE_SHIFT

• Anonymous `mmap()`
  - `addr = mmap(NULL, SIZE, PROT_*, MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);`
  - Since 3.8: same alternative flags as `shmget()`
**HugeTLBfs Usage**

- Mount a special virtual filesystem
  - `mount -t hugetlbfs none /dev/hugepages -o <pagesize=2M>`

- Create and map files within the filesystem
  - `fd = open("/dev/hugepages/1", O_CREAT | O_RDWR, 0755);`  
    `addr = mmap(NULL, SIZE, PROT_*, MAP_SHARED, fd, 0);`

- Use libhugetlbfs library – man `libhugetlbfs(7)`
  - `get_huge_pages()`, `get_hugepage_region()`...
  - `LD_PRELOAD` for legacy applications
    - Text, data, malloc(), shared memory backed by hugepages
    - Controlled by environment variables
    - Relinking might be needed for proper alignment
  - Useful tools: hugeadm, hugectl
HugeTLBfs reservations and COW

• Hugetlb pages used to be prefaulted (unlike other pages), now they are just reserved on mmap()
  - Cheaper mmap(), potentially better NUMA placement

• Private mappings can fork() + COW fault at any time
  - Potential copies not reserved – fork() won’t fail
  - COW will try to allocate without reserve, but that can fail
    - Child COW alloc fails → SIGBUS
    - Parent COW alloc fails → child’s mapping removed, fault → SIGBUS

• Reservations don’t guarantee NUMA placement
  - Mempolicy/cpuset restrictions? HugeTLB fault can fail when reserved pages are placed on the wrong nodes.
**HugeTLB Shared Page Tables**

- Scenario: many processes mapping the same region of 2MB hugepages
  - Each 1GB large region (fully populated or not) would have 4KB pmd-level page table for each process
  - This page table will be shared when mappings are properly aligned, reducing the memory usage

- Sharing 1GB hugepages has the same effect
  - Only when they are fully used

- In theory not limited to HugeTLB, but not implemented

- Example: Memory usage of (system running Oracle) by page tables 150GB without vs 1GB with HugeTLB
Transparent Hugepages
Transparent Hugepages (THP)

- Using HugeTLB pages is not straightforward
  - Admin action needed to setup pools
  - Application has to map them explicitly
  - Huge pages not split/swapped out on memory pressure

- Since 2.6.38 (2011) – Transparent hugepages
  - No admin setup necessary (tuning possible)
  - No app support necessary (hints possible)
  - Created when possible, split to base pages when needed, swapping of base pages possible
  - Single supported size (2MB), 1GB only for NVM devices (DAX)
THP instantiation

• In memory areas (vma) created by `mmap(MAP_ANONYMOUS|MAP_PRIVATE)`
  - First page fault in each huge-page aligned part of vma (last-level page table does not yet exist)
    - Read fault → map a shared “THP zero page” first
  - During `mmap()` with `MAP_POPULATE`
    - By a kernel thread `khugepaged`
• Also allocate page table for eventual pmd split
• If allocating huge page fails, fallback to mapping a page table with a single PTE entry for a base page
• COW – alloc+copy whole huge page, fallback to alloc+copy many base pages mapped by PTEs
THP teardown

- **Split of pmd mapping to (partial) mapping via pte’s (using the preallocated “deposited” page table)**
  - COW from huge zero page fallback
  - mremap()/munmap()/mprotect() not on huge page boundary
  - Page itself remains a single huge page

- **Split of whole huge page to base pages, some freed**
  - Mempolicy change by mbind()
  - Partial madvise(MADV_FREE)

- **Deferred splitting from a shrinker**
  - Page no longer mapped as huge page anywhere, but parts mapped as base pages via pte’s
The khugepaged Kernel Thread

• Not all THP’s can be created at page fault time
  - VMA too small at initial fault, grows later
  - Could not allocate huge page, fallback to base pages
  - Pages have been swapped out and in

• Khugepaged slowly scans all processes’ page tables
  - Replaces eligible groups of base pages with THP’s
  - By default, 8 hugepage candidates each 10s
THP tuning options

- Virtual files in `/sys/kernel/mm/transparent_hugepage`
  - enabled: [always] madvise never
    - Where THP's can be created. madvise means MADV_HUGEPAGE
  - defrag: always defer defer+madvise [madvise] never
    - Tunes compaction effort, affecting page fault latency. defer means only wake up kswapd+kcompactd, can help the next fault, not current one.
  - `use_zero_page`: 1
  - `hpage_pmd_size`: 2097152 (read-only)
  - `shmem_enabled`: always within_size advise [never] deny force
    - like huge=X tmpfs mount opt. tmpfs, for anonymous shared mmap etc.
    - advise is for madvise/fadvise areas
    - within_size – no THP for smaller files to avoid memory waste
    - deny/force override all tmpfs mount options
khugepaged tuning options

- /sys/kernel/mm/transparent_hugepage/khugepaged
  - scan_sleep_millisecs: 10000
    - Sleep between page table scans
  - pages_to_scan: 4096 (8 huge page candidates)
    - How many page table entries to scan before sleep
  - alloc_sleep_millisecs: 60000
    - Sleep after failure to allocate huge page
  - max_ptes_none: 511
    - How much memory potentially wasted without being accessed
  - max_ptes_swap: 64
    - Potentially unused pages that will need I/O to swap in and occupy memory
  - defrag: 1
    - khugepaged can perform compaction to allocate huge pages
THP related statistics

• /proc/meminfo
  - AnonHugePages: 1929216 kB
  - ShmemHugePages, ShmemPmdMapped

• /proc/vmstat

thp_fault_alloc 174171  thp_split_page 5542
thp_faultfallback 61457 thp_split_page_failed 4
thp_collapse_alloc 35893 thp_deferred_split_page 199
thp_collapse_alloc_failed 703 thp_split_pmd 26504
thp_file_alloc 0 thp_split_pud 0
thp_file_mapped 0 thp_zero_page_alloc 1
thp_zero_page_alloc_failed 0
THP related statistics

- /sys/kernel/mm/transparent_hugepage/khugepaged
  - full_scans: 751
  - pages_collapsed: 26272
THP Implementation Issues

• A rather intrusive addition to the existing MM
  - (Page walk) code that’s not THP ready has to split it first
    - Huge page splits cannot ever be blocked → lots of tricky code
  - Many subtle races with e.g. MADV_DONTNEED still being fixed years later

• More intrusive changes that need stabilizing
  - 2015: refcounting rework → huge pmd split without page split
  - 2016: tmpfs support (of two competing implementations)
  - 2017: 1GB support for DAX, ext4 support?

“Sigh. A pox on whoever thought up huge pages. Words cannot express how much of a godawful mess they have made of Linux MM. And it hasn't ended yet :(

– Andrew Morton, September 2013
Allocating Huge Pages
(and other high-order pages)
Allocating Huge Pages?

• Recall: Physical memory divided into several zones
  - 1+ zone per NUMA node

• Binary buddy allocator for pages in each zone
  - Free base page (e.g. 4KB) coalesced to groups of power-of-2 pages, naturally aligned
  - Exponent = page order; 0 for 4KB -> 10 for 4MB pages
  - Good performance, finds page of requested order quickly
Allocating Huge Pages?

• Recall: Physical memory divided into several zones
  - 1+ zone per NUMA node

• Binary buddy allocator for pages in each zone
  - Free base page (e.g. 4KB) coalesced to groups of power-of-2 pages, naturally aligned
  - Exponent = page order; 0 for 4KB -> 10 for 4MB pages
  - Good performance, finds page of requested order quickly
Allocating Huge Pages?

• Recall: Physical memory divided into several zones
  - 1+ zone per NUMA node

• Binary buddy allocator for pages in each zone
  - Free base page (e.g. 4KB) coalesced to groups of power-of-2 pages, naturally aligned
  - Exponent = page order; 0 for 4KB -> 10 for 4MB pages
  - Good performance, finds page of requested order quickly
Allocating Huge Pages?

• Recall: Physical memory divided into several zones
  - 1+ zone per NUMA node

• Binary buddy allocator for pages in each zone
  - Free base page (e.g. 4KB) coalesced to groups of power-of-2 pages, naturally aligned
  - Exponent = page order; 0 for 4KB -> 10 for 4MB pages
  - Good performance, finds page of requested order quickly

• Problem: allocations of order > 0 may fail due to external memory fragmentation
  - There is enough free memory, but not contiguous
Apropos, Why Buddy Allocator?

• Why not simply merge all adjacent blocks?
• $2^n$ possible sizes limits number of free lists
• Simple/fast `__find_buddy_pfn()` during merge
  - `return page_pfn ^ (1 << order);`
  - Struct page for that pfn has buddy page flag + same order
  - Merge just removes flag + order from a single struct page

• It’s true that splits are more complicated, though
• For huge pages, non-aligned pages would not be usable anyway
Why We Need High-order Allocations?

• Huge pages, obviously (both hugetlb and THP)
  - 2MB is order-9; 1GB is order-18, but max order is 10...

• Other physically contiguous area of memory
  - Buffers for hardware that requires it (no scatter/gather)
  - Potentially page cache (64KB?)

• Virtually contiguous area of memory
  - Kernel stacks until recently (order-2 on x86), now vmalloc
  - SLUB caches (max 32KB by default) for performance reasons
    - Fallback to smaller sizes when possible
  - vmalloc is a generic alternative, but not for free
    - Limited area (on 32bit), need to allocate and setup page tables…
    - Somewhat discouraged, but now a kvmalloc() helper exists
Example: Failed High-order Allocation

chrome: page allocation failure: order:4, mode:0xc0d0
CPU: 4 PID: 18907 Comm: chrome Not tainted 3.16.1-gentoo #1
Hardware name: Dell Inc. OptiPlex 980 /0D441T, BIOS A15 01/09/2014
Node 0 DMA free:15888kB min:84kB low:104kB high:124kB active_anon:0kB inactive_anon:0kB
active_file:0kB inactive_file:0kB unevictable:0kB isolated(anonymous):0kB isolated(file):0kB present:15988kB managed:15904kB
mlocked:0kB dirty:0kB writeback:0kB mapped:0kB shmem:0kB slab_reclaimable:0kB slab_unreclaimable:16kB
kernel_stack:0kB pagetables:0kB unstable:0kB bounce:0kB free_cma:0kB writeback_tmp:0kB pages_scanned:0
all_unreclaimable? Yes
Node 0 DMA32 free:157036kB min:19340kB low:24172kB high:29008kB active_anon:1444992kB
inactive_anon:480776kB active_file:538856kB inactive_file:513452kB unevictable:0kB isolated(anonymous):0kB isolated(file):0kB
present:3578684kB managed:3504680kB mlocked:0kB dirty:1304kB writeback:0kB mapped:157908kB shmem:85752kB
slab_reclaimable:278324kB slab_unreclaimable:20852kB kernel_stack:4688kB pagetables:28472kB unstable:0kB bounce:0kB
free_cma:0kB writeback_tmp:0kB pages_scanned:0 all_unreclaimable? no
Node 0 Normal free:100168kB min:48152kB low:60188kB high:72228kB active_anon:4518020kB
inactive_anon:746232kB active_file:1271196kB inactive_file:1261912kB unevictable:96kB isolated(anonymous):0kB isolated(file):0kB
present:8912896kB managed:8714728kB mlocked:96kB dirty:5224kB writeback:0kB mapped:327904kB shmem:143496kB
slab_reclaimable:502940kB slab_unreclaimable:52156kB kernel_stack:11264kB pagetables:70644kB unstable:0kB
bounce:0kB free_cma:0kB writeback_tmp:0kB all_unreclaimable? no
Node 0 DMA: 0*4kB 0*8kB 1*16kB (U) 2*32kB (U) 1*64kB (U) 1*128kB (U) 1*256kB (U) 0*512kB 1*1024kB
(U) 1*2048kB (R) 3*4096kB (M) = 15888kB
Node 0 DMA32: 31890*4kB (UEM) 3571*8kB (UEM) 31*16kB (UEM) 16*32kB (UMR) 6*64kB (UEMR)
1*128kB (R) 0*256kB 0*512kB 1*1024kB (R) 0*2048kB 0*4096kB = 158672kB
Node 0 Normal: 22272*4kB (UEM) 726*8kB (UEM) 75*16kB (UEM) 24*32kB (UEM) 1*64kB (M) 0*128kB
0*256kB 0*512kB 0*1024kB 0*2048kB 1*4096kB (R) = 101024kB
[drm:radeon_cs_ioctl] *ERROR* Failed to parse relocation -12!
Enabling High-Order Allocations

• Prevent memory fragmentation?
  - Buddy allocator design helps by splitting the smallest page
  - Works only until memory becomes full (as it always should)

• Reclaim contiguous areas?
  - LRU based reclaim → pages of same age not guaranteed to be near each other physically
  - “Lumpy reclaim” did exist, but it violated the LRU aging

• Defragment memory by moving pages around?
  - Memory compaction can do that within each zone
  - Relies on page migration functionality
Memory Compaction Overview

• Execution alternates between two page (pfn) scanners

  • Migration scanner looks for migration source pages
    - Starts at beginning (first page) of zone, moves towards end
    - Isolates movable pages from LRU list

  • Free scanner looks for migration target pages
    - Starts at the end of zone, moves towards beginning
    - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

- Execution alternates between two page (pfn) scanners
  - Migration scanner looks for migration source pages
    - Starts at beginning (first page) of zone, moves towards end
    - Isolates movable pages from LRU list
  - Free scanner looks for migration target pages
    - Starts at the end of zone, moves towards beginning
    - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

• Execution alternates between two page (PFN) scanners
  • Migration scanner looks for migration source pages
    - Starts at beginning (first page) of zone, moves towards end
    - Isolates movable pages from LRU list
  • Free scanner looks for migration target pages
    - Starts at the end of zone, moves towards beginning
    - Isolates free pages from buddy allocator (splits as needed)

![Diagram showing memory compaction process with migratePFN, freePFN, and free pages](image-url)
Memory Compaction Overview

• Execution alternates between two page (pfn) scanners
  • Migration scanner looks for migration source pages
    - Starts at beginning (first page) of zone, moves towards end
    - Isolates movable pages from LRU list
  • Free scanner looks for migration target pages
    - Starts at the end of zone, moves towards beginning
    - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

• Execution alternates between two page (pfn) scanners
  • Migration scanner looks for migration source pages
    - Starts at beginning (first page) of zone, moves towards end
    - Isolates movable pages from LRU list
  • Free scanner looks for migration target pages
    - Starts at the end of zone, moves towards beginning
    - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

• Execution alternates between two page (pfn) scanners
  • Migration scanner looks for migration source pages
    - Starts at beginning (first page) of zone, moves towards end
    - Isolates movable pages from LRU list
  • Free scanner looks for migration target pages
    - Starts at the end of zone, moves towards beginning
    - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

- Execution alternates between two page (pfn) scanners
  - Migration scanner looks for migration source pages
    - Starts at beginning (first page) of zone, moves towards end
    - Isolates movable pages from LRU list
  - Free scanner looks for migration target pages
    - Starts at the end of zone, moves towards beginning
    - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

• Execution alternates between two page (pfn) scanners

• Migration scanner looks for migration source pages
  - Starts at beginning (first page) of zone, moves towards end
  - Isolates movable pages from LRU list

• Free scanner looks for migration target pages
  - Starts at the end of zone, moves towards beginning
  - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

- Execution alternates between two page (pfn) scanners
- Migration scanner looks for migration source pages
  - Starts at beginning (first page) of zone, moves towards end
  - Isolates movable pages from LRU list
- Free scanner looks for migration target pages
  - Starts at the end of zone, moves towards beginning
  - Isolates free pages from buddy allocator (splits as needed)

We have enough, time to migrate
Memory Compaction Overview

• Execution alternates between two page (pfn) scanners
  • Migration scanner looks for migration source pages
    - Starts at beginning (first page) of zone, moves towards end
    - Isolates movable pages from LRU list
  • Free scanner looks for migration target pages
    - Starts at the end of zone, moves towards beginning
    - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

- Execution alternates between two page (pfn) scanners
- Migration scanner looks for migration source pages
  - Starts at beginning (first page) of zone, moves towards end
  - Isolates movable pages from LRU list
- Free scanner looks for migration target pages
  - Starts at the end of zone, moves towards beginning
  - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

• Execution alternates between two page (pfn) scanners

• Migration scanner looks for migration source pages
  - Starts at beginning (first page) of zone, moves towards end
  - Isolates movable pages from LRU list

• Free scanner looks for migration target pages
  - Starts at the end of zone, moves towards beginning
  - Isolates free pages from buddy allocator (splits as needed)
Memory Compaction Overview

- Execution alternates between two page (pfn) scanners
  - Migration scanner looks for migration source pages
    - Starts at beginning (first page) of zone, moves towards end
    - Isolates movable pages from LRU list
  - Free scanner looks for migration target pages
    - Starts at the end of zone, moves towards beginning
    - Isolates free pages from buddy allocator (splits as needed)

Scanners have met, end the compaction
Memory Compaction Overview

- Execution alternates between two page (PFN) scanners
- Migration scanner looks for migration source pages
  - Starts at beginning (first page) of zone, moves towards end
  - Isolates movable pages from LRU list
- Free scanner looks for migration target pages
  - Starts at the end of zone, moves towards beginning
  - Isolates free pages from buddy allocator (splits as needed)
- Stops when scanner positions cross each other
  - Or, when free page of desired order has been created
  - Or due to lock contention, exhausted timeslice, fatal signal...
Memory Compaction Limitations

• Only a subset of pages can be isolated and migrated
  - Pages on LRU lists (user-space mapped, either anonymous or page cache)
  - Pages marked with PageMovable “flag”
    - Currently just zswap/zmalloc compressed in-memory “swap”
    - Candidates: vmalloc, page tables, some SLAB caches?
  - No other pins except mappings, only clean pages on some filesystems…

• A single non-migratable page in an order-9 block can prevent allocating a whole huge page there – permanent fragmentation

• We should keep such pages close together
  - Page grouping by mobility
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE

- Separate buddy free lists for each migratetype

- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE
    migratetype (there are few more for other purposes)

- Separate buddy free lists for each migratetype

- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

![Diagram of movable and unmovable pageblocks]
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE
    *migratetype* (there are few more for other purposes)
- Separate buddy free lists for each migratetype
- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

![Diagram showing movable and unmovable pageblocks](image-url)
Grouping by Mobility Overview

• Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE
    migratetype (there are few more for other purposes)

• Separate buddy free lists for each migratetype

• Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

![Diagram of pageblocks]

Movable pageblock | Unmovable pageblock

UNMOVABLE allocation has to fall back, finds block with largest free page
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE migratetype (there are few more for other purposes)
- Separate buddy free lists for each migratetype
- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

Movable pageblock

Unmovable pageblock

UNMOVABLE allocation steals all free pages from the pageblock (too few to also steal the pageblock itself) and uses the smallest
Grouping by Mobility Overview

• Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE
    migratetype (there are few more for other purposes)

• Separate buddy free lists for each migratetype

• Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

- UNMOVABLE allocation steals all free pages from the pageblock
  (too few to also steal the pageblock itself) and uses the smallest
Grouping by Mobility Overview

• Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE
    

• Separate buddy free lists for each migratetype

• Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

![Diagram of pageblocks]

Some pages are freed within UNMOVABLE pageblock, so they go on UNMOVABLE freelist
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE
    *migratetype* (there are few more for other purposes)
- Separate buddy free lists for each migratetype
- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

Some pages are freed within UNMOVABLE pageblock, so they go on UNMOVABLE freelist
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE migratetype (there are few more for other purposes)
- Separate buddy free lists for each migratetype
- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

The next MOVABLE allocation has to fall back, finds largest UNMOVABLE freepage
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE migratetype (there are few more for other purposes)

- Separate buddy free lists for each migratetype

- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

The next MOVABLE allocation has to fall back, finds largest UNMOVABLE freepage
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE
    *migratetype* (there are few more for other purposes)

- Separate buddy free lists for each migratetype

- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

The next MOVABLE allocation has to fall back, finds largest UNMOVABLE freepage
Grouping by Mobility Overview

• Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE
  
• Separate buddy free lists for each migratetype

• Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

Movable pageblock  Unmovable pageblock

Temporary allocation immediately freed
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE migratetype (there are few more for other purposes)
- Separate buddy free lists for each migratetype
- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

![Diagram showing movable and unmovable pageblocks]

Free page goes to UNMOVABLE free list as the pageblock is UNMOVABLE
Grouping by Mobility Overview

• Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE 
migratetype (there are few more for other purposes)

• Separate buddy free lists for each migratetype

• Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

Movable pageblock

Unmovable pageblock

Merging works across migratetypes, the type that initiated the merge “wins”
Grouping by Mobility Overview

- Zones divided to pageblocks (order-9 = 2MB on x86)
  - Each marked as MOVABLE, UNMOVABLE or RECLAIMABLE migratetype (there are few more for other purposes)
- Separate buddy free lists for each migratetype
- Allocations declare (via GFP flags) intended type
  - Tries to be satisfied first from matching pageblock type
  - Fallback to other type when matching pageblocks full

This page would fit in UNMOVABLE pageblock but we could not have predicted the pattern
Mobility Grouping Fallback Heuristics

• Perfection generally impossible without knowing future
  - Also the effort has to be reasonable wrt allocation latency

• Find+steal the largest free page of any migratetype
  - Approximates finding a pageblock with the most free pages
  - Multiple types available? Preferences given by alloc. type

• Can we steal all pages from the pageblock?
  - UNMOVABLE and RECLAIMABLE allocations always can.
  - MOVABLE: the found page has to be order $\geq 4$

• Steal $X$ free pages, count $Y$ pages of compatible type
  - If $X + Y \geq 256$ (half of pageblock), change pageblock type

• Allocate from the stolen pages, splitting the smallest
Allocating Gigantic Pages
Beyond Buddy Allocator
Allocating beyond buddy orders

• Buddy allocator supports up to 4MB pages

• What if we need more? E.g. 1GB hugepages
  - Initially only via kernel boot parameter early enough, when all memory is free

• Later, alloc_contig_range() added
  - Converts the range to MIGRATE_ISOLATE migratetype
    - Pages still in buddy lists, but effectively unreachable
  - Scans the range, migrating all movable pages away
    - Similar to compaction migrate scanner, but direct free page allocation

• Unmovable or pinned pages can prevent success
  - Still better chances to allocate 1GB pages early at boot
ZONE_MOVABLE

• A special zone allowing only GFP_HIGHUSER_MOVABLE allocations
  - Hard restriction compared to migratetypes grouping
  - size controlled by kernelcore=X and movablecore=Y params

• Guarantees compaction/alloc_contig_range success
  - Except when pages get pinned for a long time
  - (Re-) Introduces reclaim issues (of 32bit HIGHMEM)

• Later reused for memory hot-remove support
  - Hot-removable nodes have all memory in ZONE_MOVABLE
  - movable_node boot param
CMA – Contiguous Memory Allocation

• Some cheap hardware requires large physically contiguous area to do DMA (no scatter/gather)
  - Cell phone cameras – used relatively rarely
  - Cell phone memory limited – permanent reservation no-go
• Use alloc_contig_range() to allocate the memory
• Areas “reserved” (any zone(s)) on boot, hardcoded list
  - Change pageblock migratetype to MIGRATE_CMA
    - Like MIGRATE_MOVABLE, but cannot be stolen by other types’ fallback
    - Memory usable by user-space until camera app started
  - Requires extra counter for watermarks, various special cases
• Ongoing effort to replace this by ZONE_CMA
  - Why not reuse ZONE_MOVABLE? Good question...
Questions?

Thank you.