Page Reclaiming

Advanced Operating Systems

Overview

- Memory pressure and page reclaiming
- Cache shrinking
- OOM killing
- Swapping
  - LRU
- Compression
  - zram
  - zswap
  - zcache
- Deduplication
  - KSM

Page Reclaiming: When?

- **Memory pressure**: shortage of memory
- Need to reclaim memory pages
  - Hibernation reclaiming
  - Direct reclaiming (sync)
  - Periodic reclaiming (async)
- Direct reclaiming
  - Triggered when the kernel fails to allocate memory
  - Note: there are exceptions (e.g., THPs)
- Periodic reclaiming (e.g., `kswapd`)
  - Triggered when memory pressure approaching
  - E.g., free pages below the low watermark in a zone

Page Reclaiming: How?

- Cache shrinking
  - Shrink (or drop) OS-maintained caches
- Out-Of-Memory (OOM) killing
  - Kill a likely unimportant process to reclaim its pages
- Swapping
  - Swap likely unused pages to disk
- Compression
  - Compress likely unused pages in memory
- Deduplication
  - Merge & COW pages with identical content

Cache Shrinking

- **Idea**: eliminate “harmless” pages first
- Page cache
    - Reclaim pages not referenced by any process
- Slab caches
    - Reclaim pages from slabs with no allocated objects
- Dentry cache
    - Reclaim dentries not referenced by any process
- Inode cache
    - Reclaim inode objects with no controlling dentry

OOM Killing

- Crude page reclaiming solution, however:
  - Acceptable with a number of unimportant processes
  - Efficient way to reclaim page frames
  - Necessary as a last resort when overcommitting
- Direct reclaiming
  - Linux' OOM killer
  - Assigns badness scores and kills task with highest
  - Can we use it for kernel allocs? ('*Too small to fail*')
- Periodic reclaiming
  - Android' low-memory killer
  - Similar to OOM killer, targets background apps

Linux’ OOM Killer: Example

```
INFO: memcached invoked oom-killer
CPU: 1 PID: 2859
Call Trace:
 [<c10e1c15>] dump_header.isra.7+0x85/0xc0
 [<c10e1e6c>] oom_kill_process+0x5c/0x80
 [<c10e225f>] out_of_memory+0xbf/0x1d0
 [<c10fec2c>] handle_pte_fault+0xec/0x220
 [<c10fee68>] handle_mm_fault+0x108/0x210
 [<c152fb5b>] do_page_fault+0x15b/0x4a0
 [<c152cfcf>] error_code+0x67/0x6c
Out of memory: Kill process 2603 score 761 or sacrifice child
Killed: process 2603 vm:1498MB, anon-rss:721MB, file-rss:4MB
```

Linux' OOM Killer: Badness Score Heuristic

mm/oom_kill.c:

```
/*
 * The baseline for the badness score is the proportion of RAM that each
 * task's rss, pagetable and swap space use.
 */
points = get_mm_rss(p->mm) + get_mm_counter(p->mm,MM_SWAPENTS) +
   mm_pgtables_bytes(p->mm) / PAGE_SIZE;

/* Normalize to oom_score_adj units */
adj = totalpages / 1000;
points += adj;
```

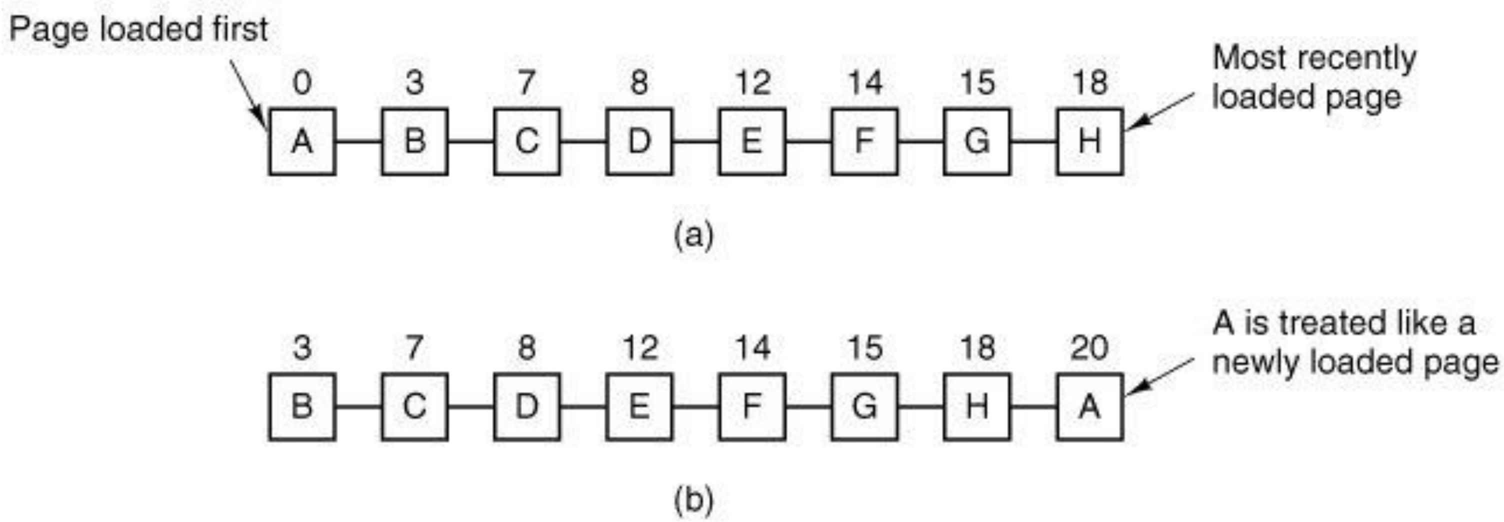
Page Swapping

- Swap space on disk creates illusion of huge virtual memory (page backing store)
- Traditional solution for page reclaiming
- Nowadays less prominent than before:
  - Alternative solutions exist
  - RAM is increasingly cheap
  - I/O operations are slow (disk) or disruptive (SSD)
- However:
  - Still important on consumer platforms (hibernation)
  - Still important on some server platforms (databases)

Page Replacement

- **Problem:** Which page(s) should we swap under memory pressure?
  - Need for page replacement algorithm
- **Ideal:**
  - Replace the page that will be referenced as far away in the future as possible
  - As an approximation, we use LRU-like (Least Recently Used) strategies in practice
- **FIFO**:
  - Use a queue of faulting pages and replace the head
  - What happens to pages accessed all the time?

Page Replacement: Second Chance / CLOCK



- Improved FIFO to preserve important pages
- For each visited page:
  - If **R**=0 then replace page
  - If **R**=1 then set **R**=0 and move page to tail
- CLOCK uses a circular list and moves head

Linux’ Page Reclaiming: Active and Inactive LRUs

- Split LRU list into active and inactive LRUs
- LRUs internally managed CLOCK-like
- Active pages considered in use (working set)
- Inactive pages unmapped (ready for reclaim)
- Many per-zone LRUs nowadays:

```
enum lru_list {
        LRU_INACTIVE_ANON = LRU_BASE,
        LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,
        LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,
        LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,
        LRU_UNEVICTABLE,
        NR_LRU_LISTS
};
```

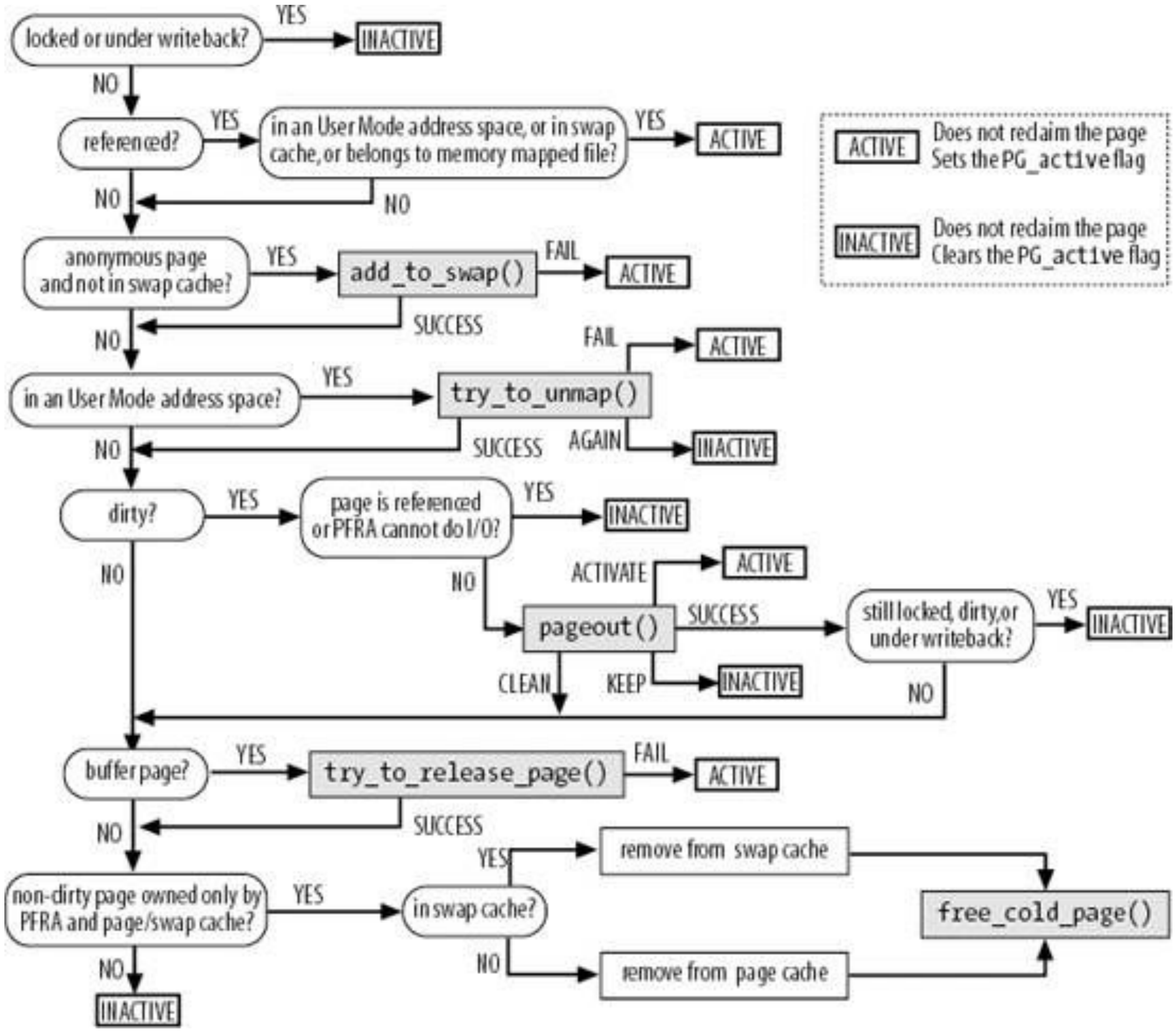
Linux’ Page Reclaiming: Active / Inactive Updates

- New anon pages added to active LRU
- New file pages added to inactive LRU
- Inactive pages become active on PF
- LRU active pages periodically made inactive by `kswapd` to keep LRUs balanced
- Problem: What is the right ratio?
  - Originally 1:1, proved inadequate over time
- What we want:
  - Large enough active list to avoid PFs in working set
  - Large enough inactive list to give pages a 2nd chance

Linux' Page Reclaiming: Active / Inactive Balancing

- Active / inactive LRU ratio is determined adaptively using global **refault distance**:
  - Per-page distance: How much larger should the inactive list be to avoid evicting this page?
  - When a page is evicted from inactive list, a distance counter tracks #evictions until page faults in again

```
$ grep active /proc/vmstat
nr_inactive_anon 30484
nr_active_anon 155005
nr_inactive_file 65502
nr_active_file 149127
```



Linux' Page Reclaiming Logic (Simplified!)

Swapping: Mechanics

- **Swap-out**: when reclaiming a given page frame, store content in a swap entry on disk
  - Need to unmap it from all the owning PTEs
- **Swap-in**: when page faulting, store content of swap entry on new page frame to map in
  - Need to locate swap entry on the disk
- Requirements:
  - Reverse mapping page → PTEs (swap-out)
  - Direct mapping PTE → swap entry (swap-in)
  - Handle races with concurrent swap-ins/outs
    - Scratch area in the page cache (**swap cache**)

Page Cache

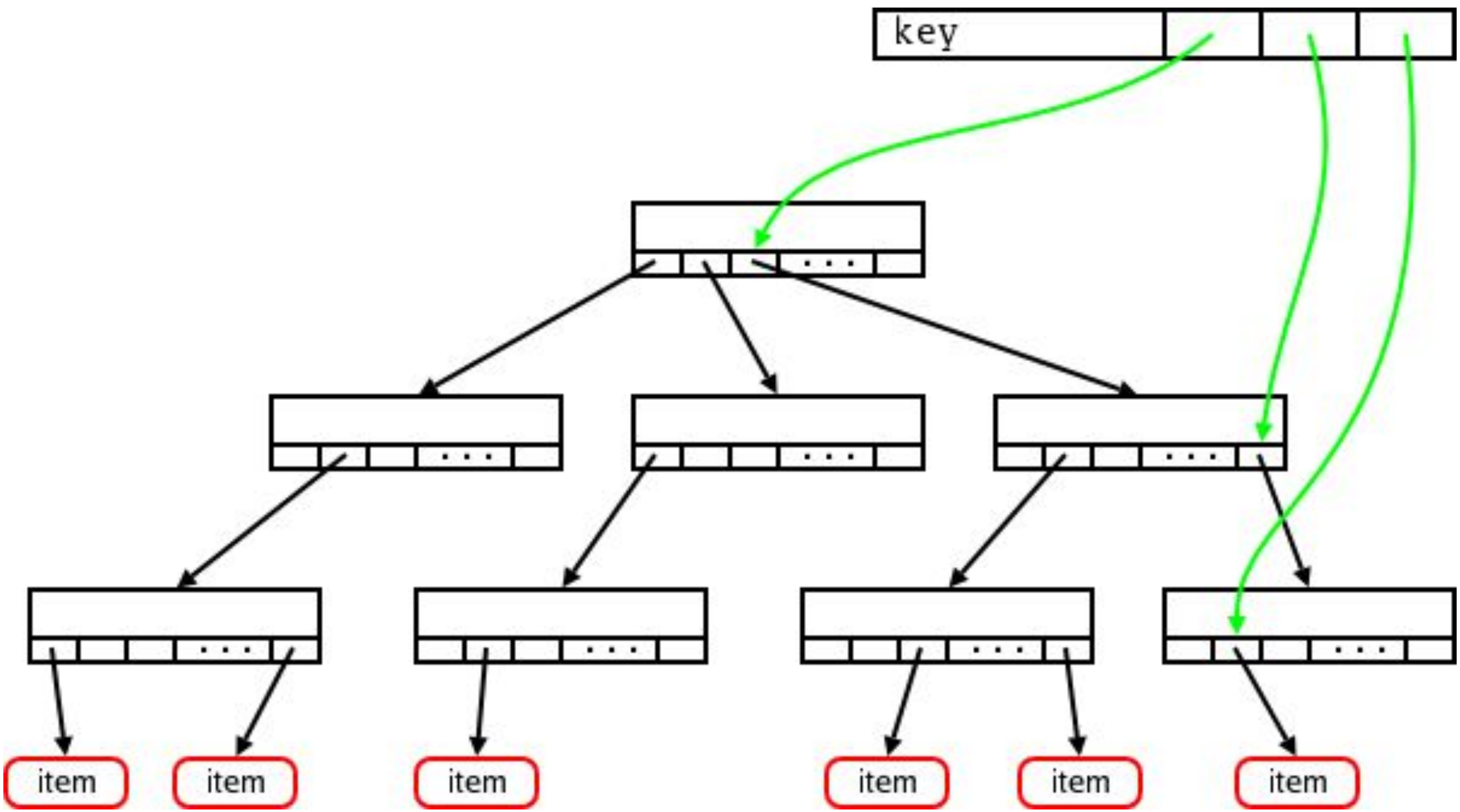
- Stores pages of a given owner (i.e., file/dev)
- The swap cache is a subset (owner: swap)
- Unified page cache for many purposes:
  - Hold in-transit swap pages (swap-in / swap-out)
  - Efficient demand paging and sharing for file pages
  - Serve file `read()/write()` efficiently (write-back)
  - Reverse file mappings (see later)
- Fundamental operations (by owner, offset):
  - Lookup: used by read/PF handler (disk read on miss)
  - Add: used when fetching page from disk
  - Delete: used by page reclaiming (using LRU)

Page Cache

- N `address_space` objects, 1 for each `owner`
- Lookup/add/delete ops: `i_pages` xarray
- E.g., lookup operation:
  - `struct page *find_get_page(struct address_space *mapping, pgoff_t offset)`

```
struct address_space {
  struct inode              *host; /* owner: inode,device */
  struct xarray             i_pages; /* cached pages */
  struct rb_root_cached     i_mmap; /* VMA tree */
  struct rw_semaphore       i_mmap_rwsem; /* protect tree */
  unsigned long             nrpages; /* # of total pages */
  const struct address_space_operations *a_ops; /* methods */
  void                      *private_data;
} __attribute__((aligned(sizeof(long))));
```

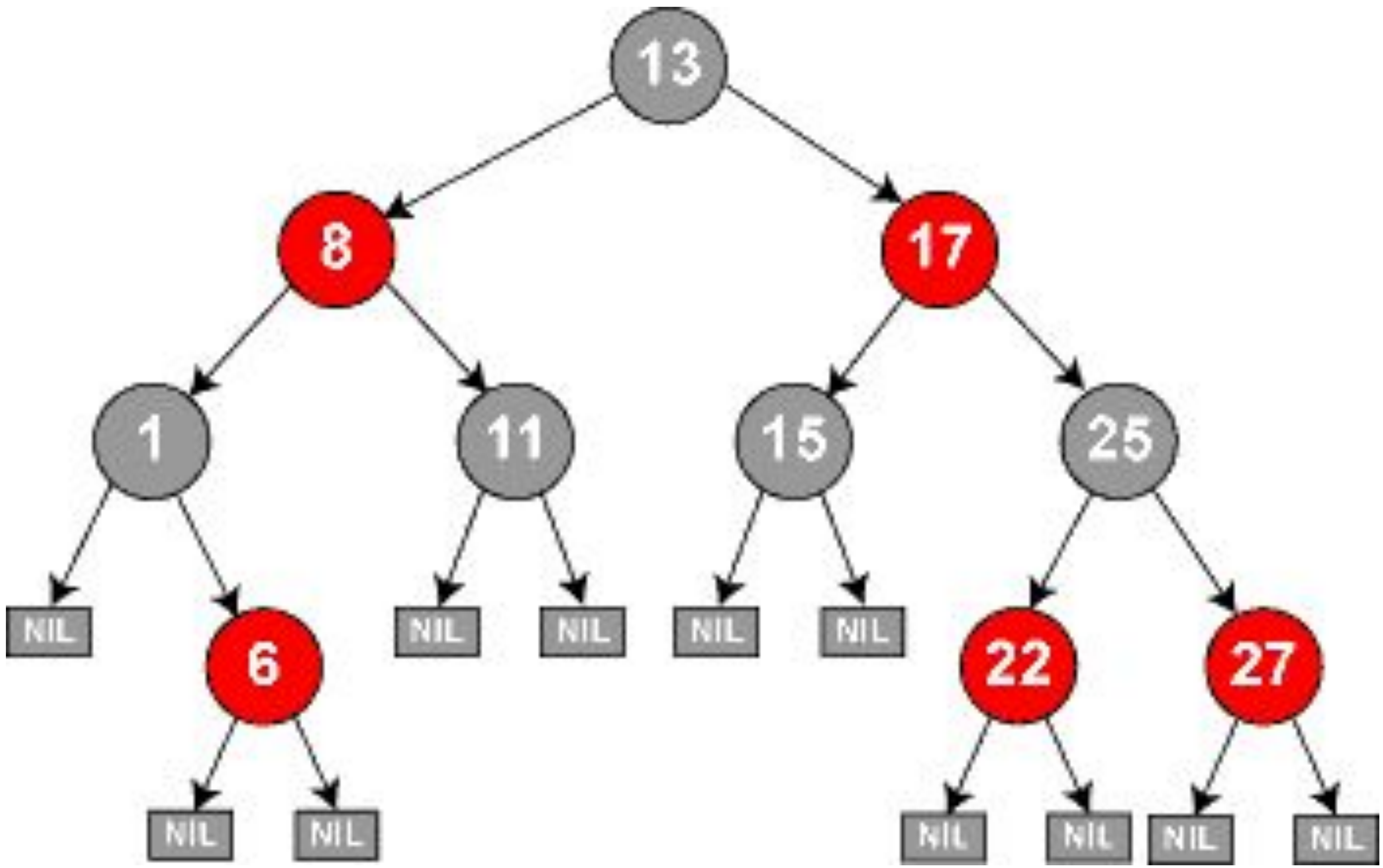
Xarray: A Better Radix Tree

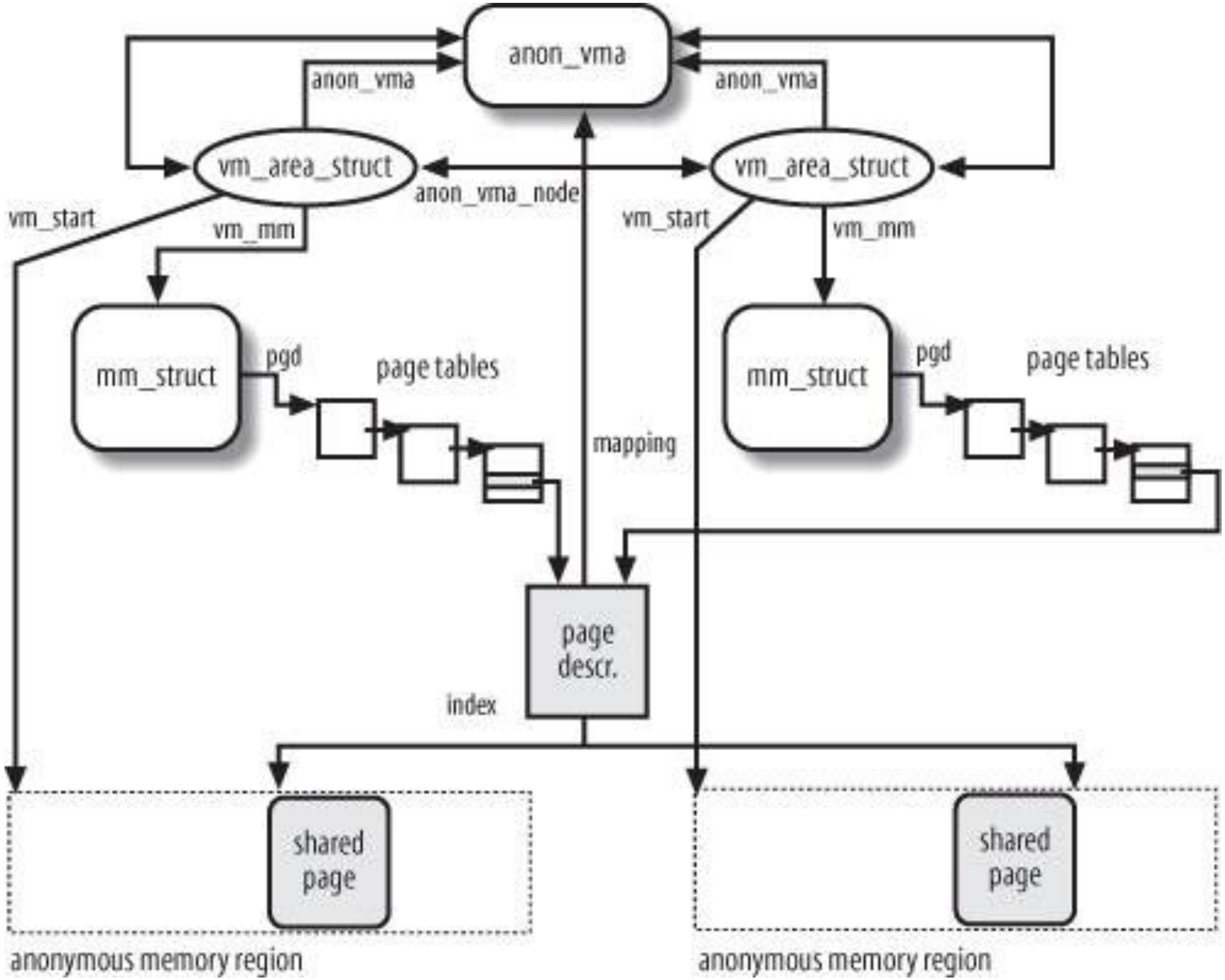


Reverse mapping: Page Desc to PTEs

- Each page descriptor stores pointer (`mapping`, `index`) to `rmap` data structure, indexing all VMAs which *may* map the page
- Given a VMA, we can PT walk via `mm->pgd`
- File pages
  - `mapping` points to `address_space` data structure
  - Uses `i_mmap` red-black tree to index the possibly many VMAs mapping the target page by `index`
- (Private) anonymous pages
  - `mapping` points to `anon_vma` structure (VMA list)
  - Unless page is in swap cache (`swapper_space`)

Red-black Tree





Reverse mapping: anon_vma

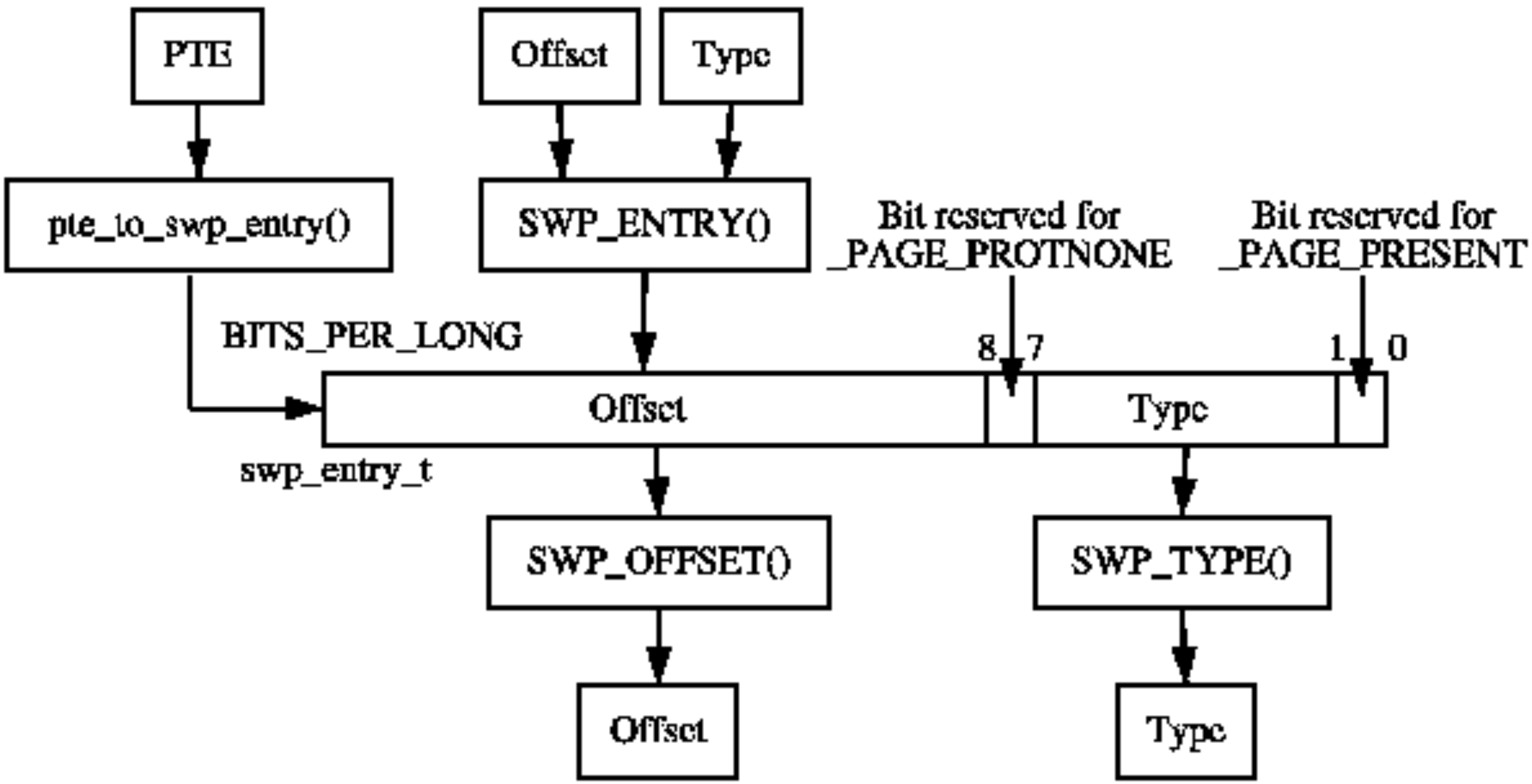
Reverse mapping: Better anon_vma Scalability

- **Problem 1**: `anon_vma` could link 1M VMA clones together when forking 1M times
  - Horizontal scalability problem
- **Solution 1**: per-process `anon_vma` linked together by `anon_vma_chain`
- **Problem 2**: `anon_vma` could link 1M splitted VMAs together when mprotecting 1M times
  - Vertical scalability problem
- **Solution 2**: `anon_vma` uses a `i_mmap`-like RB tree of per-process VMAs, not a list
  - Index is virtual memory address in this case

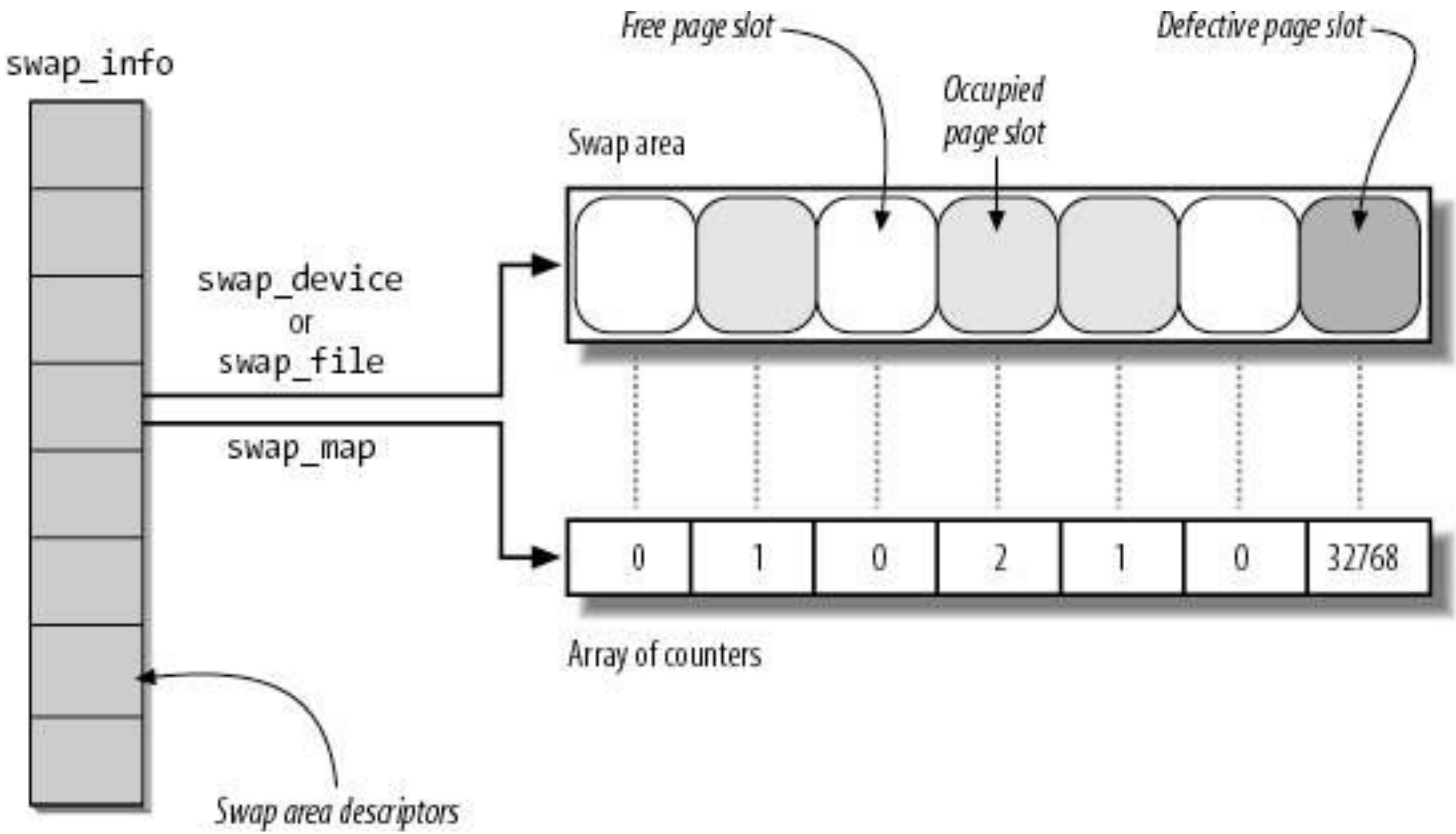
Direct Mapping: PTE to Swap Entry

- The system maintains a number of swap areas, each with a priority and an array of available swap entries
- Each swap entry maps to a block on the disk at a linear offset
- When a page is swapped out, all the owning PTEs are simply filled with:
  - Area number (or type)
  - Swap entry offset
- On PF, entry lookup on swap cache or disk

Direct Mapping: PTE to Swap Entry



Direct Mapping: PTE to Swap Entry



Direct Mapping: PTE to Swap Entry

```
struct swap_info_struct {
  unsigned long   flags;      /* SWP_USED etc: see above */
  signed short    prio;       /* swap priority of this type */
  signed char     type;       /* strange name for an index */
  unsigned int    max;        /* extent of the swap_map */
  unsigned char *swap_map;    /* usage counts */
  unsigned int lowest_bit;    /* index of first free */
  unsigned int highest_bit;   /* index of last free */
  unsigned int pages;         /* total pages of swap */
  unsigned int inuse_pages;   /* number of those in use */
  unsigned int cluster_next; /* index for next allocation */
  struct block_device *bdev; /* swap device */
  struct file *swap_file;     /* swap file */
  spinlock_t lock;
};
```

Page Reclaiming in OpenLSD

Lab 7: Under pressure
(page reclaiming)

Core:

- Implement LRU-like page reclaiming with swapping support

Bonuses:

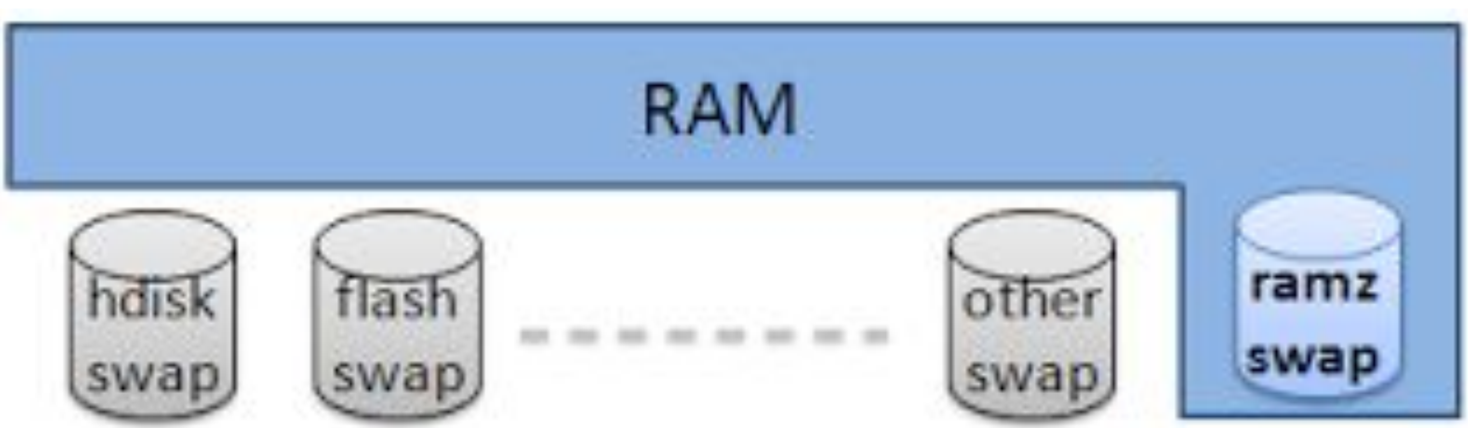
- Page compression, page deduplication

Page Compression

- **Idea**: Use CPU for “swapping” rather than much slower I/O subsystem
- Page-out to a compressed memory cache
- Can replace or complement swapping
- Different compressions algorithms
  - LZO by default, slower LZ4 for better compression
- Different mechanisms
  - ZRAM: Swap replacement
  - ZSWAP: Swap complement
  - ZCACHE: Swap complement

Compression: ZRAM

- Uses a compressed RAM disk swap device
- Pages are compressed when swapped out
- Pages are decompressed when swapped in
- No changes to swap management required



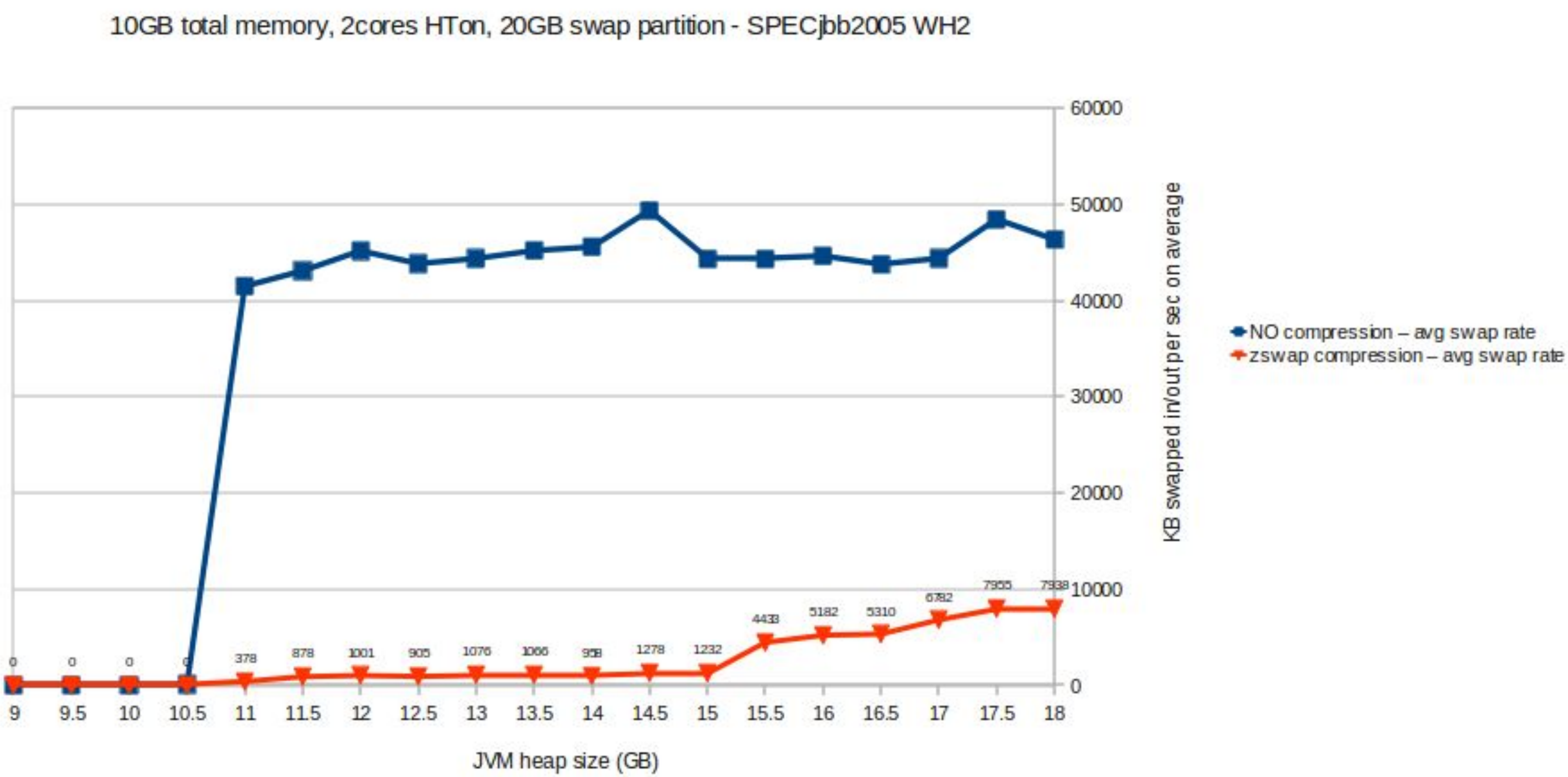
Compression: ZRAM

- Can work in absence of pure swapping or next to other swap devices
- zram device is typically configured as the highest-priority swap device
- Other devices (swap areas) are picked when the cache (of predetermined size) is full
- Disadvantages:
    - Requires manual swap device configuration
    - When cache full, pages sent to disk uncompressed
    - LRU inversion: MR swapped pages go to slow disk

Compression: ZSWAP

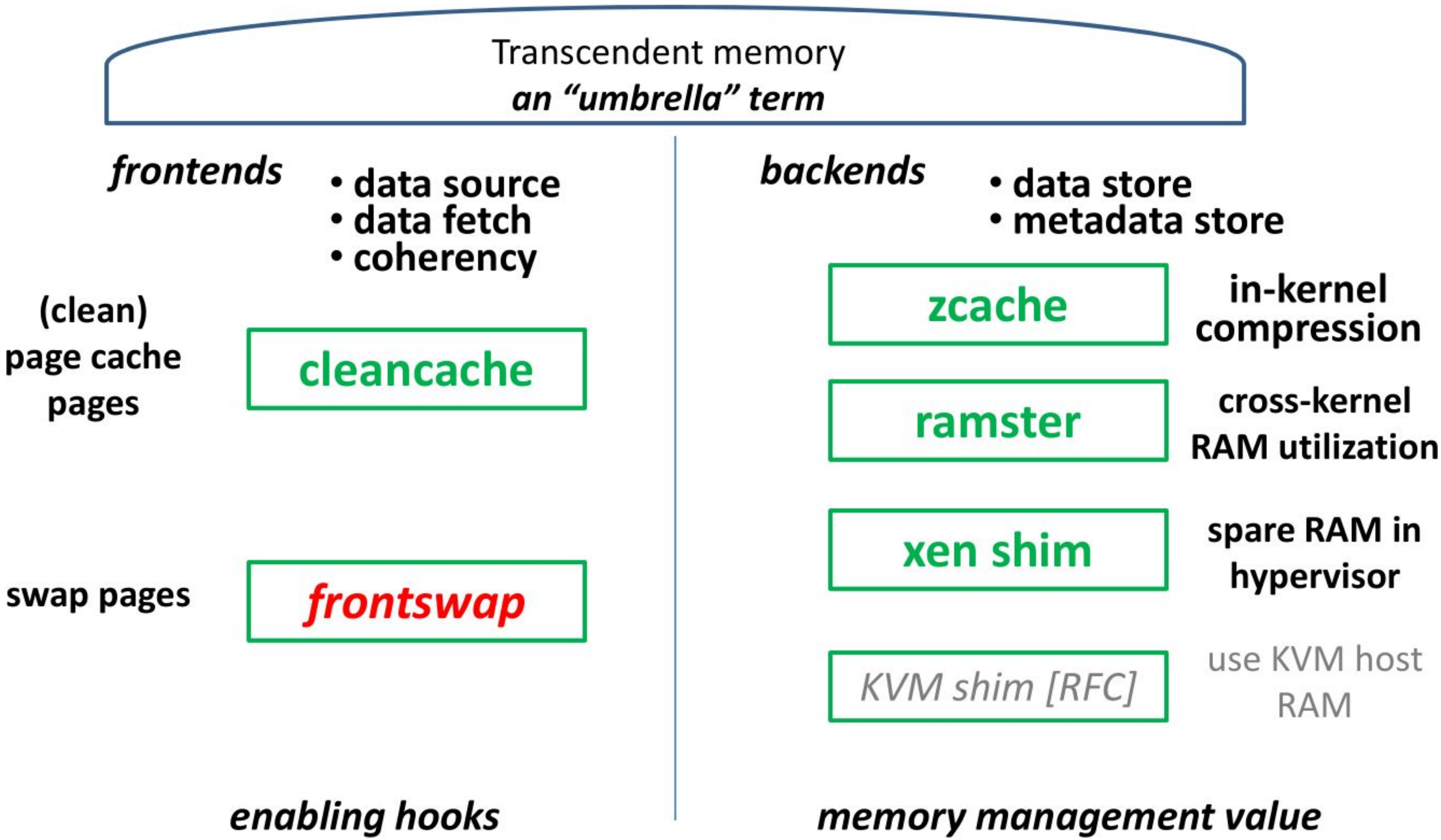
- Write-back cache for regular swap devices
- Stores compressed pages in zswap pool
- Uses zbud allocator by default to store 2 pages in a compressed page
- Swap-out noncompressible pages directly
- Compressed pages swapped to disk in LRU fashion when pool reaches static size limit
- Greatly reduces disk I/O due to swapping
- Disadvantage:
  - Assumes a dedicated disk swap device

Compression: ZSWAP



Compression: ZCACHE

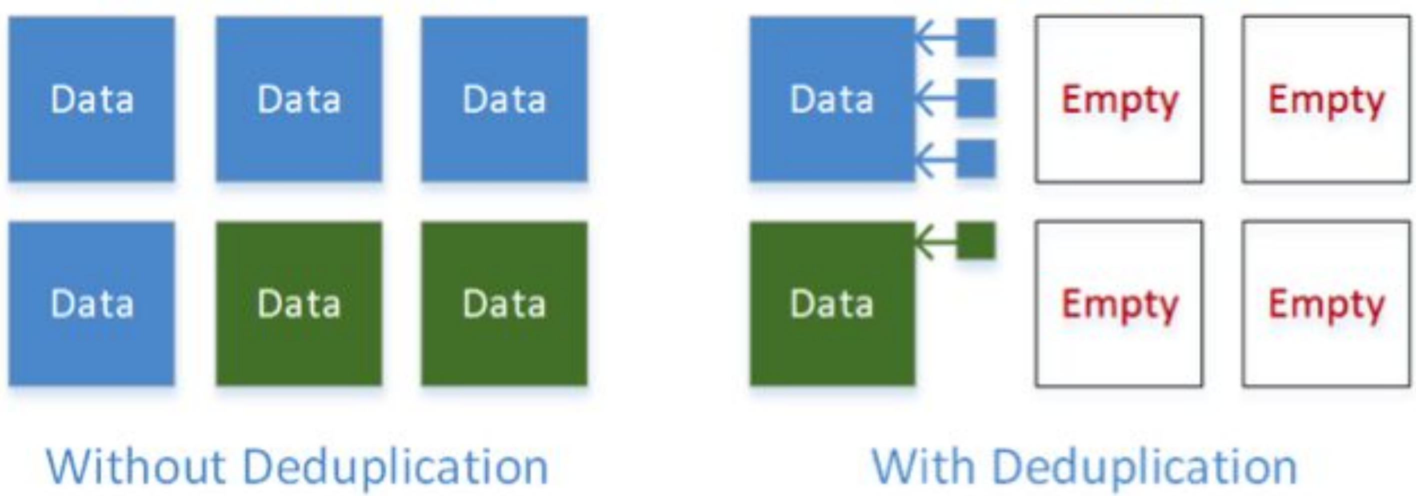
- Write-back cache design similar to zswap
- More ambitious goal:
  - Compressed cache for swapped pages
  - Compressed cache for clean file pages
- More complicated, never made it mainline
- Based on **transcendent memory** interfaces
- Models a remote page-granular RAM cache
- Remote pages are **ephemeral** or **persistent**
- **Frontend** gets/puts page to remote **backend**
- Many frontends/backends not just zcache



Transcendent Memory (`tmem`)

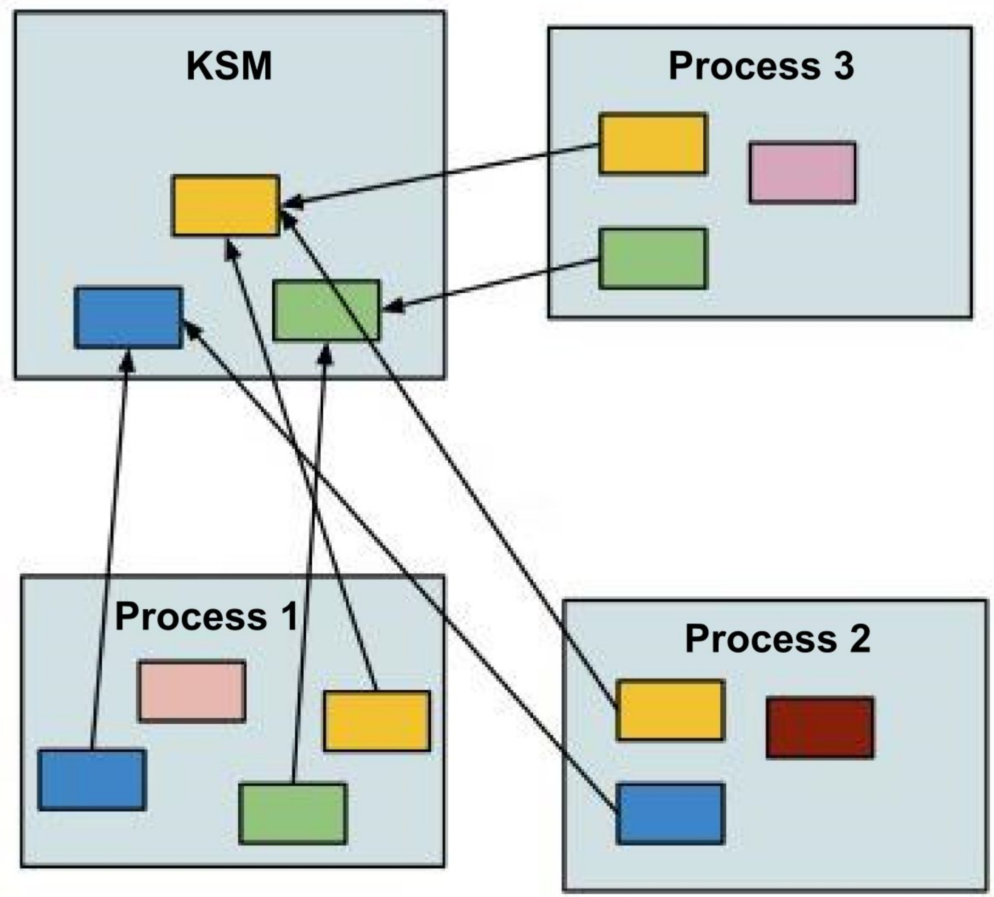
Page Deduplication

- Proactively find pages with identical content
- Merge them and use COW for safe sharing
- With COW-enabled `fork` / `exec` why bother?
  - Virtualization: duplicated anon pages across guests
  - Particular (e.g., scientific) workloads duplicating data



KSM: Kernel Samepage Merging

- In-kernel dedup system for priv anon memory
- Designed for virtualization but works on regular processes as well
- Opt-in: explicitly mark VMAs as `MADV_MERGEABLE`



KSM: Mechanics

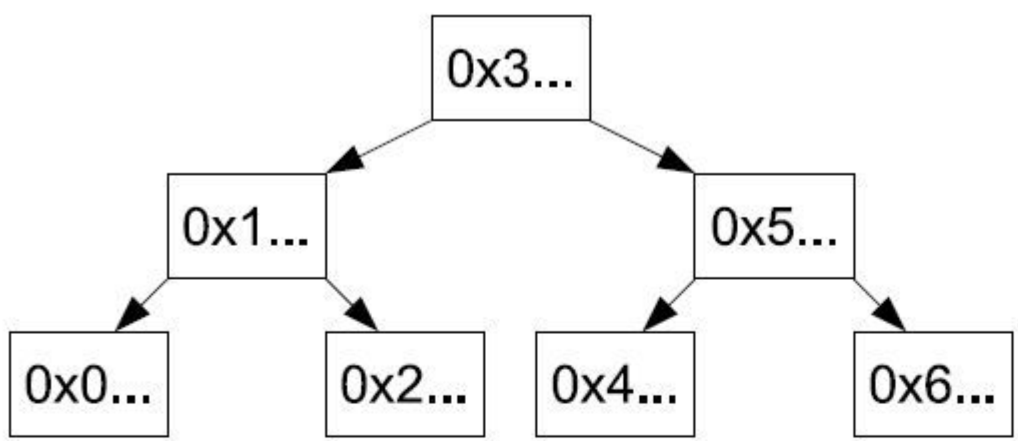
- Kernel thread (`ksmd`) scans P mergeable pages every S milliseconds
- Maintains red-black tree(s) of merged / nonmerged pages indexed by page content
- At every scan, `ksmd` walks the tree(s) P times, looking for an exact match each time
- For each exact match, `ksmd` merges the page frames and write-protects owning PTEs
- Merged page frames can be COWed or even swapped later as necessary

KSM:
Tunables and Stats

```
$ grep “” /sys/kernel/mm/ksm/*
/sys/kernel/mm/ksm/full_scans:0
/sys/kernel/mm/ksm/merge_across_nodes:1
/sys/kernel/mm/ksm/pages_shared:0
/sys/kernel/mm/ksm/pages_sharing:0
/sys/kernel/mm/ksm/pages_to_scan:100
/sys/kernel/mm/ksm/pages_unshared:0
/sys/kernel/mm/ksm/pages_volatile:0
/sys/kernel/mm/ksm/run:0
/sys/kernel/mm/ksm/sleep_millisecs:20
```

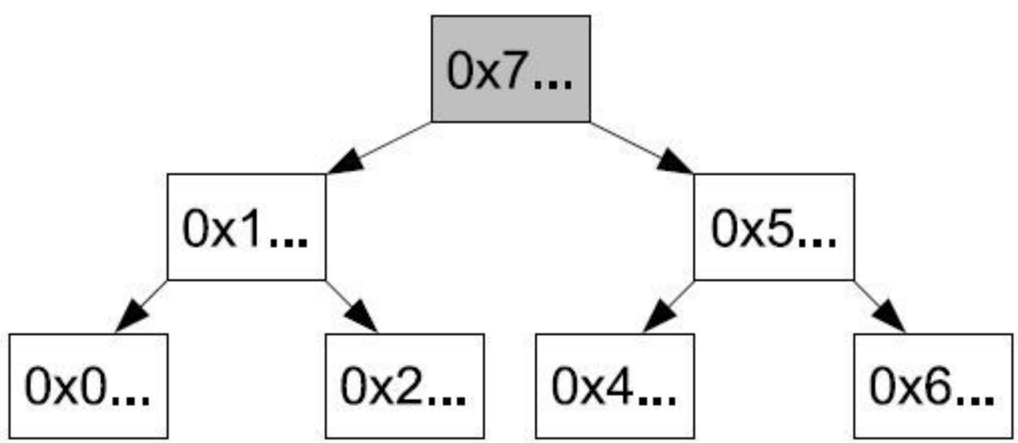
KSM: Stable Tree

- Contains only already merged pages
- Their content cannot change (or COW PF)
- Red-black tree is consistent by design
- `ksmd` can safely walk it for each page looking for exact matches with `memcmp()`



KSM: Unstable Tree

- Contains nonmerged pages, content changes
- Can be inconsistent, lookups can lead to FNs
- To mitigate needless walks:
  - Unstable red-black tree rebuilt at each `ksmd` scan
  - No lookups for (checksummed) pages in the WWS



Page Deduplication: Problems

- **Performance**:
  - Proactive reclaiming may eliminate memory pressure or just waste much CPU time for little gain
  - Awkward behavior in degenerate cases (COW storm)
- **Security**:
  - **Cache attacks**: enables flush+reload for otherwise unrelated pages in different security domains (VMs)
  - **Dedup Est Machina**: yields a “does this page exist on the system?” oracle to leak data via COW timing
  - **Flip Feng Shui**: provides a massaging primitive for an attacker to map a targeted victim page into a physical page vulnerable to hardware bit flips

References

[1] Bovet, Daniel P., and Marco Cesati. Understanding the Linux Kernel, 2005.

[2] Gorman, Mel. Understanding the Linux Virtual Memory Manager, 2004.

[3] “Linux Cross Reference,” n.d. https://elixir.bootlin.com/linux/latest/source.

[4] “Better Active/inactive List Balancing,” n.d. https://lwn.net/Articles/495543.

[5] “Transcendent Memory in a Nutshell,” n.d. https://lwn.net/Articles/454795.

[6] “Frontswap,” n.d. https://www.kernel.org/doc/Documentation/vm/frontswap.txt.

[7] “Cleancache,” n.d. https://www.kernel.org/doc/Documentation/vm/cleancache.txt.

[8] “Overcommit,” n.d. https://www.kernel.org/doc/Documentation/vm/overcommit-accounting.

[9] “Zswap,” n.d. https://www.kernel.org/doc/Documentation/vm/zswap.txt.

[10] “Zram,” n.d. https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[11] “KSM,” n.d. https://www.kernel.org/doc/Documentation/vm/ksm.txt.