Multigenerational LRU

Quick start

Build options

Required:	Set `CONFIG_LRU_GEN=y`.
Optional:	Change `CONFIG_NR_LRU_GENS` to a number `X` to support a maximum of `X` generations.
Optional:	Set `CONFIG_LRU_GEN_ENABLED=y` to turn on by default.

Runtime options

Required:	`echo 1 >/sys/kernel/mm/lru_gen/enable` if was not turned on by default.
Optional:	Change `/sys/kernel/mm/lru_gen/spread` to a number `N` to spread pages out into `N+1` generations. Larger values make background aging more aggressive.
Optional:	Read `/sys/kernel/debug/lru_gen` to verify the state of multigenerational LRU. The file has the following format:

```
memcg  memcg_id  memcg_path
  node  node_id
    min_seq  birth_time  anon_size  file_size
    ...
    max_seq  birth_time  anon_size  file_size
```

The minimum generation number a.k.a. `min_seq` is the oldest of all generations; the maximum generation number a.k.a. `max_seq` is the youngest. Birth time is in milliseconds. Anon and file sizes are in pages.

Recipes

Android on ARMv8.1+:	`X=4`, `N=0`
Android on older ARM CPUs:	Not recommended due to the lack of `ARM64_HW_AFDBM`
Linux laptops running Chrome on x86_64:	`X=7`, `N=2`
Working set estimation:	Write `+ memcg_id node_id max_seq [swappiness]` to `/sys/kernel/debug/lru_gen` to scan for accessed pages, update their generation numbers to `max_gen` and create the next generation `max_seq+1`. A swap file is required to enable anon multigenerational LRU. If swap is not desired, set `vm.swappiness` to `0` and overwritten it with the optional parameter `[swappiness]`. Otherwise anon multigenerational LRU will not be scanned even though it is enabled.
Proactive reclaim:	Write `- memcg_id node_id min_seq [swappiness] [nr_to_reclaim]` to `/sys/kernel/debug/lru_gen` to evict generations older than `min_seq`. The optional parameter `nr_to_reclaim` can be used to limit the number of pages to be evicted from the oldest generation.

Workflow

Multigenerational LRU uses generation numbers to partition evictable pages. Raw generation numbers are stored in `struct lru_gen` which is a member of `struct lruvec`. The youngest generation number `max_seq` tracks both anon and file so they are aged on an equal footing. The oldest generation numbers `min_seq[2]` track anon and file separately so clean file pages can be dropped regardless of swappiness. Raw generation numbers are truncated into `ilog2(CONFIG_NR_LRU_GENS)+1` bits to fit into `page->flags`. Sliding window technique is used to prevent truncated generation numbers from overlapping, which could be analogized to a ring buffer, with aging to the writer and eviction to the reader. A set of per-type (anon/file) and per-zone page lists is indexed by each truncated generation number. Pages are added to the lists indexed by `max_seq` when they are faulted in.

Aging

Aging produces new generations. For each aging cycle, all mapped pages that belong to an `lruvec` are scanned. For pages that have been accessed since last scan, their generation numbers are updated to `max_seq`. `max_seq` is incremented at the end of each cycle.

Aging maintains system- or memcg-wide mm list to scan mapped pages at minimum cost.

Eviction

Eviction consumes old generations. Pages on the per-zone lists indexed by `min_seq[2]` are scanned. And pages are either sorted or isolated, depending on whether aging has updated their generation numbers. Either `min_seq` is incremented when there are no pages left on its lists.

Eviction selects a type (anon/file) simply based on generations and swappiness.

Rationale

Characteristics of cloud-era workloads

Warning: though the following observations are made across millions of servers and clients at Google, they may not be universally applicable.

Memory composition

With cloud storage gone mainstream, anonymous memory is now the majority and page cache contains mostly executable pages and negligible unmapped pages. In addition, userspace is smart enough to avoid page cache thrashing by taking advantage of AIO, direct I/O and `io_uring` when streaming large files stored locally.

The profile of `kswapd`

As a result of the aforementioned memory composition, swapping is necessary to achieve substantial memory overcommit. And since almost all pages are mapped, the `rmap` surpasses `zram` and becomes the hottest path in `kswapd`.

For `zram`, a typical `kswapd` profile on v5.11 looks like:

```
31.03%  page_vma_mapped_walk
25.59%  lzo1x_1_do_compress
 4.63%  do_raw_spin_lock
 3.89%  vma_interval_tree_iter_next
 3.33%  vma_interval_tree_subtree_search
```

And for disk swap, it looks like:

```
45.16%  page_vma_mapped_walk
 7.61%  do_raw_spin_lock
 5.69%  vma_interval_tree_iter_next
 4.91%  vma_interval_tree_subtree_search
 3.71%  page_referenced_one
```

Limitations of the current implementation

Granularity of the active/inactive

For large systems that have tens or hundreds gigabytes of memory, the active/inactive sizes become too coarse to be useful for memory overcommit. Pages counted as active can be less recently used than pages counted as inactive because they will not be scanned until most inactive pages have.

For smaller systems, eviction is biased between anon and file because the selection is mainly based on inference not direct comparisons. For example, on Chrome OS, executable pages are frequently evicted despite the fact that there are many less recently used anon pages.

When there are multiple `lruvecs`, the active/inactive notion becomes even less useful because comparisons between pages from different `lruvecs` is impossible based on this notion.

Memory locality of the `rmap`

The `rmap` has poor memory locality because of its complex data structures. On top of it, at least two walks of the `rmap` are required before a page can be evicted after it is faulted in. The first walk clears the accessed bit set upon the fault; the second walk verifies the page has not been accessed since then. Due to the both factors, the cost of memory overcommit becomes significant when almost all pages are mapped but only accessed every few minutes.

How multigenerational LRU solves the problems

Multigenerational LRU introduces a quantitative way to overcommit memory. For an `lruvec`, pages from an older generation are guaranteed to be less recently used than those from a younger generation. For different `lruvecs`, comparisons can be made approximately based on birth times.

Page tables have far better memory locality than the `rmap` when it comes to checking the accessed bit. As counterintuitive as it may seem, for cloud-era workloads, tracking all accessed pages via page tables is a lot less expensive than scanning pages one by one via the `rmap`. The overhead from page table walk is proportional to the number of accessed pages, and the walk is only required when most pages have been accessed. At this point, scanning pages one by one would be very inefficient because of the number of pages to cover.

To-do list

KVM optimization

Support shadow page table walk.

NUMA optimization

Add per-node RSS for should_skip_mm().

Refault tracking optimization

Use generation numbers rather than LRU positions in workingset_eviction() and workingset_refault().