Softlimit Reclaim In Memcg

Ying Han (yinghan@google.com)
Last updated: Sep 27, 2012

After going through several round of softlimit discussion, there are some agreements that we reached but still something that unclear. Everyone including myself likes to making forward progress on this (if we still believe the softlimit is a good thing), and also getting rid of the current implementation. Now I end up writing up this doc to capture things on top of my head, and this is the best thing I can think of.

This document is to layout the design and proposed change to rewrite the existing softlimit reclaim in memcg. I really like to get the Acked-by on those Proposed Change from you before going further to implement it this time. All the designs are based on the actual use cases google has in production, so just let you know this is not something I throw out randomly :)

Terminology

- **cold memory**: The memory pages that haven't been accessed by process in last x seconds. The assumption is that those pages will less likely be access again in near future. Those memory should be the first candidate for page reclaim.

- **hot memory**: The memory pages that have been assessed and will be accessed again. We need to preserve the hot memory for cgroup in order to provide predictable throughput/latency. Non reclaimable pages are hot memory.

- **usage**: memory.usage_in_bytes including hot and cold memory

- **hardlimt**: memory.limit_in_bytes. Memory usage of a memcg is capped at hardlimit.

- **softlimit**: memory.soft_limit_in_bytes. Memory usage of a memcg could grow above softlimit if there is a *slack*. The slack could be free pages or cold pages of other cgroups.

- **global reclaim:** triggered through page allocator under global memory pressure

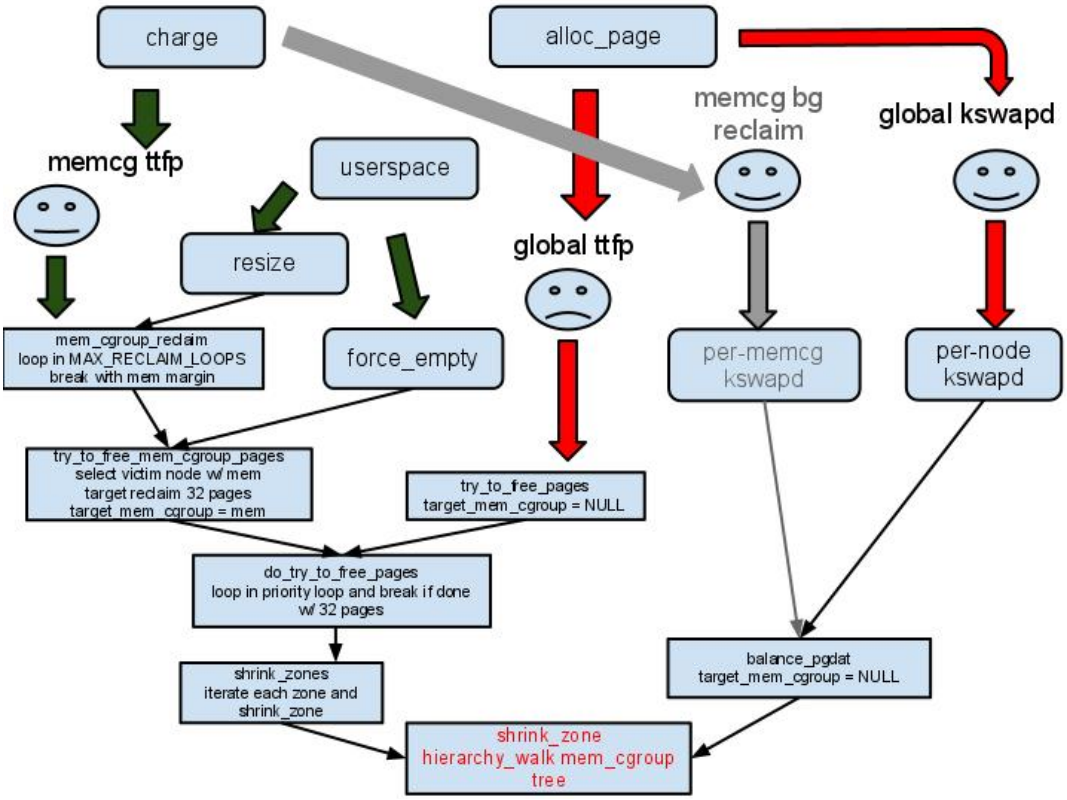
- **target reclaim:** triggered through memcg limit reclaim.

- **Regulated environment:** softlimit of each memcg is set by admin.

- **Non-Regulated environment:** softlimit of each memcg could be modified to arbitrary value by arbitrary user.

Overview:

The graph demonstrates callpath of the different reclaim:



* No worry about the function name since we are using old kernel :(

As we can see, no matter where the reclaim is being triggered from, they are all boiled down to one common function shrink_zone() which is the starting point of all the proposed reclaim changes in this document.

Which memcg to reclaim from?

The reclaim candidate is picked based on usage and softlimit setting. The memcg with usage exceeding its softlimit takes higher priority to be reclaimed.

Problem:
The current implementation is not integrated to the hierarchy walk, and also doesn't fullfill the requirement in reality.

Proposed Change:

Fully integrate the softlimit checking into the hierarchy walk, and a memcg is eligible for reclaim if the following condition hold:

1. Regulated environment:

reclaim if and only if:
(current memcg is eligible to reclaim) AND (all ancestors are eligible to reclaim)

2. Non-regulated environment:
add new per-memcg kernel API memory.usage_soft_limit
*set memory.use_soft_limit equal to true on *root* of the untrusted sub-tree, and reclaim treat all the memcgs under that sub-tree as if they have softlimit = 0*

3. Set root cgroup soft_limit default to 0 and not mutable.

4. Kernel refuse to set soft_limit of a memcg above its hardlimit.

After this, the same algorithm could be applied to both regulated and non-regulated environment. The difference is that no hot memory protection for sub-tree declared to be non-regulated.

Case Study:

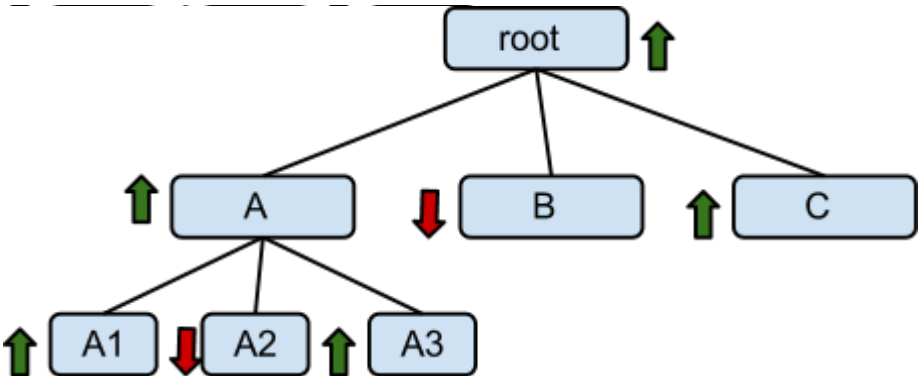
The following examples demonstrate one way to set softlimit by admin (the same for non-regulated) and also the expected output:

- How the softlimit is set?
  * memory.soft_limit_in_bytes(leaf) = hot memory + padding (for spikey app)
  * memory.soft_limit_in_bytes(parent) = sum(soft_limit_in_bytes of all children) + softlimit(parent)

- How the softlimit setting related to usage?
  * memory.usage_in_bytes(A) > memory.soft_limit_in_byte(A)
  when T or its children uses lots of cold pages (green arrow)

  * memory.usage_in_bytes(A) <= memory.soft_limit_in_bytes(A)
  when A or its children have large padding (red arrow)

Case 1:



Expected output:
Reclaim cold pages away from memcgs above their softlimit.

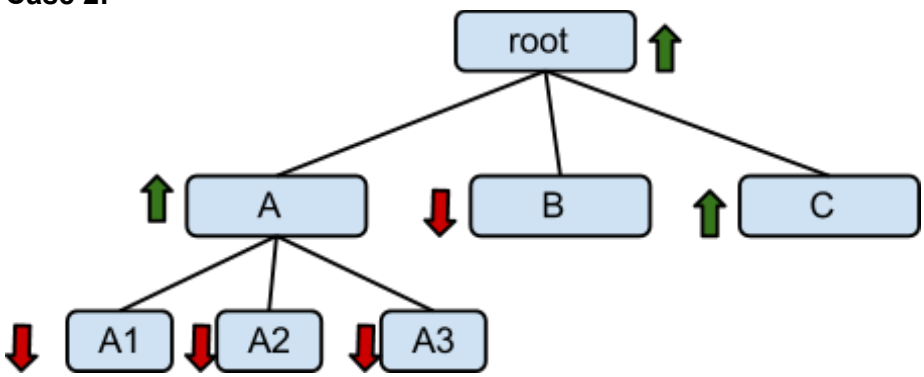
When triggers **global memory pressure**, the following memcgs should bereclaimed
root, A , A1, A3, C

When triggers **target reclaim under A,** the following memcg should be reclaimed
A, A1, A3

When triggers **target reclaim under C,** reclaim happens on C

There will be **no target reclaim under B.**

Case 2:

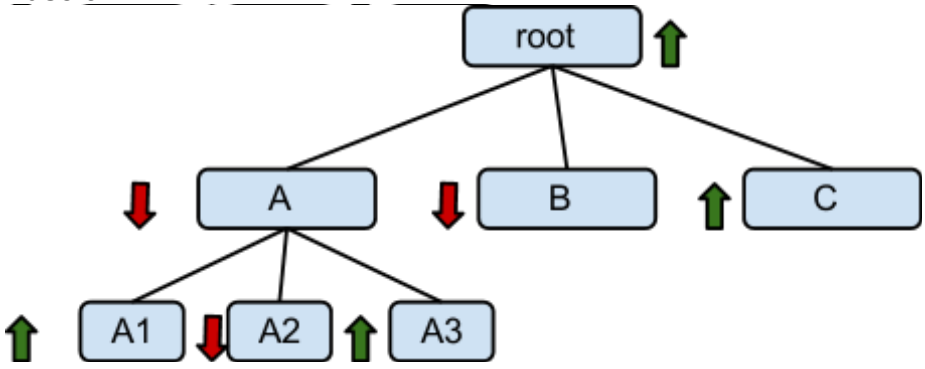


Expected output:
Reclaim exclusive cold pages aways from parent memcg (like reparented).

When triggers **global memory pressure**, the following memcgs should be reclaimed
root, A, C

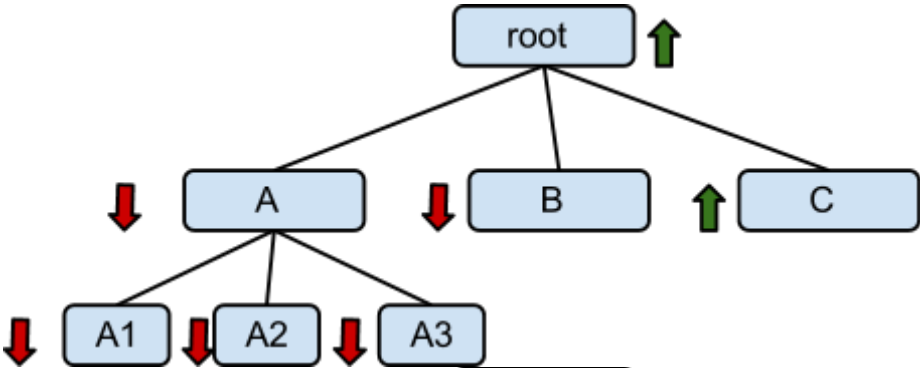
When triggers **target reclaim under A,** the following memcgs should be reclaimed
A

Case 3:



Expected output:
No reclaim on the child memcg on cold pages if the parent memcg is under its softlimit.

When triggers **global memory pressure,** the following memcgs should bereclaimed
root, C
Case 4:



Expected output:
No reclaim on the child/parent memcg if they are all under softlimit.

When triggers **global memory pressure,** the following cgroups should be reclaimed
root, C

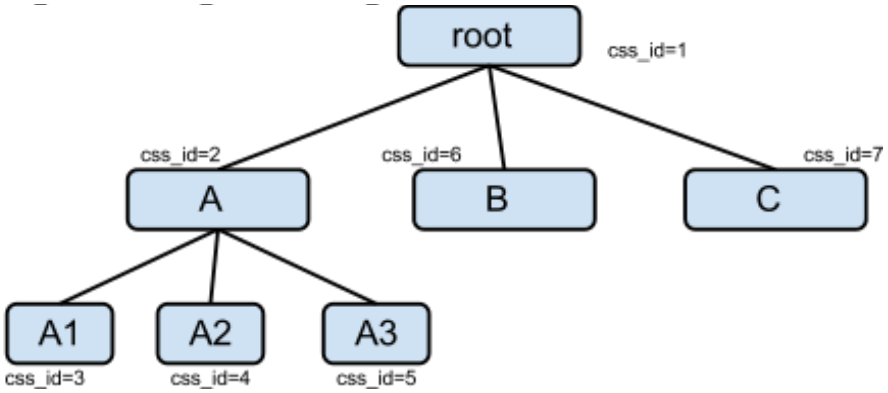
How the hierarchy walk works ?

**mem_cgroup_iter(**): The function takes a memcg as starting point and returns the next one under the same cgroup sub-tree.

Hierarchy walk:

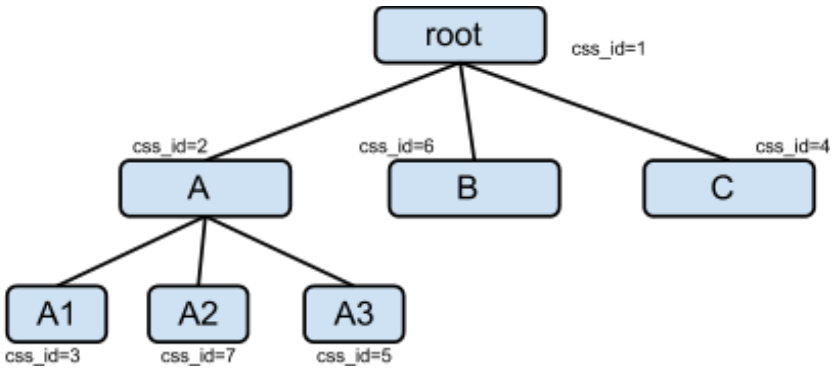
css_get_next(): Each memcg is assigned with css_id at creation time. The function does a linear search based on css_id and returns the next one under the cgroup sub-tree.

For example, the following cgroups are created in order A->A1->A2->A3->B->C:



Global Reclaim: root ->A->A1->A2->A3->B->C
Target Reclaim: A->A1->A2->A3

Problems:
1. The mechanism to traverse the cgroup tree doesn't respect the hierarchy, and it would introduce inconsistent reclaim decision. The same cgroup hierarchy:



Global Reclaim: root ->A->A1->C->A3->B->A2

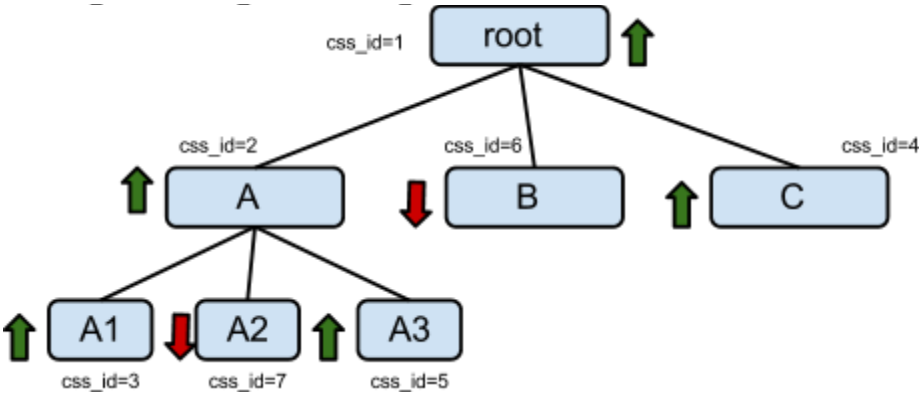
2. Traversing the whole css_id doesn't scale while the number of cgroups increases on the system. On the example above, it makes less sense to scan through all the children if A is not eligible for reclaim at the first place.

Proposed Change:
1. Introduce real hierarchical walk which walks the cgroup tree in a post-order DFS instead of the css_id.

2. When first entering a memcg, only going down the sub-tree if the memcg is eligible for reclaim.

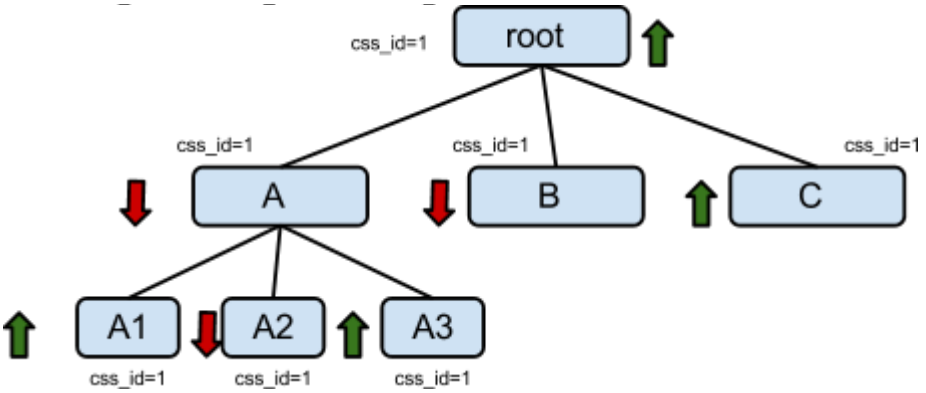
3. When last leaving a memcg, trigger the reclaim on the cgroup if the memcg is still eligible for reclaim. note that reclaiming from child memcg will lower the usage_in_bytes of parent.
After the change, the reclaim ordering becomes:

Case 1:



Walk: root->A->A1->A2->A3->A->B->C->root
Global Reclaim: A1 -> A3 -> A -> C -> root

Case 2:



Walk: root->A->B->C->root
Global Reclaim: C -> root

Global reclaim:

Under global reclaim, it scans all memcgs for the desired zone per-priority and tries to reclaim 32 pages from each. A round-robin mechanism is in-place (iter->position) and a round-trip is detected by having one reclaimer reaches the highest css_id.

Caveat:

* No early break from the loop even enough pages have been reclaimed. In the old days (before memcg naturalisation) where we scan per-zone global lru, the reclaim breaks after getting 32 pages. Today, in worst case we are trying to reclaim 32 pages from each memcg under the system. Well, it is not causing us problems and we might not need to worry about it?

* Global reclaim is different from target reclaim today where scanning each memcg under each loop makes sense. If that is the case, why not letting each reclaimer does a round-trip instead of the way it is today? The current implementation letting the global reclaim does the same round-robin as target reclaim, and one reclaimer could start at very end of the css_id and come out very quickly. My hunch that we did that because to share the same code base for target reclaim?

Potential Change:

1. Allow breaking out the hierarchy walk if 32 pages have been reclaimed. The same "last_scanned_child" mechanism is preserved and each reclaimer picks it up for the rest of the tree walk.

Or

2. Letting each reclaimer scanning the round-trip and no early break. But getting rid of the iter->position.

Target Reclaim:

Under target reclaim, it picks one memcg under the hierarchy and reclaims from it on all the zones with decreasing priority levels.

Problems:
No good reason why target reclaim behaves differently from global reclaim. The later one is just the target reclaim under root.

Proposed Change:
*Apply the same walk mechanism for target reclaim and the *root* could be the memcg reaching its hardlimit.*