
Subject: [RFC] mm-controller

Posted by [Peter Zijlstra](#) on Thu, 21 Jun 2007 09:32:44 GMT

[View Forum Message](#) <> [Reply to Message](#)

Having read the RSS and Pagecache controllers some things bothered me.

- the duplication of much of the reclaim data (not code) and the size increase as a result thereof.
- the clear distinction between mapped (RSS) and unmapped (pagecache) limits. Linux doesn't impose individual limits on these, so I don't see why containers should.
- lack of soft limits (but I understand that that is WIP :-)
- while I appreciate the statistical nature of one container per page, I'm bit sceptical because there is no forced migration cycle.

So, while pondering the problem, I wrote down some ideas....

While I appreciate that people do not like to start over again, I think it would be fruit-full to at least discuss the various design decisions.

(Will be travelling shorty, so replies might take longer than usual)

Mapped pages:

~~~~~

Basic premises:

- accounting per address\_space/anon\_vma

Because, if the data is shared between containers isolation is broken anyway and we might as well charge them equally [1].

Move the full reclaim structures from struct zone to these structures.

```
struct reclaim;
```

```
struct reclaim_zone {  
    spinlock_t lru_lock;
```

```
    struct list_head active;
```

```

struct list_head inactive;

unsigned long nr_active;
unsigned long nr_inactive;

struct reclaim *reclaim;
};

struct reclaim {
    struct reclaim_zone zone_reclaim[MAX_NR_ZONES];

    spinlock_t containers_lock;
    struct list_head containers;
    unsigned long nr_containers;
};

struct address_space {
    ...
    struct reclaim reclaim;
};

struct anon_vma {
    ...
    struct reclaim reclaim;
};

```

Then, on instantiation of either `address_space` or `anon_vma` we string together these reclaim domains with a reclaim scheduler.

```

struct sched_item;

struct reclaim_item {
    struct sched_item sched_item;
    struct reclaim_zone *reclaim_zone;
};

struct container {
    ...
    struct sched_queue reclaim_queue;
};

sched_enqueue(&container->reclaim_queue, &reclaim_item.sched_item);

```

Then, shrink\_zone() would use the appropriate containers' reclaim\_queue to find an reclaim\_item to run isolate\_pages on.

```
struct sched_item *si;
struct reclaim_item *ri;
struct reclaim_zone *rzone;
LIST_HEAD(pages);

si = sched_current(&container->reclaim_queue);
ri = container_of(si, struct reclaim_item, sched_item);
rzone = ri->reclaim_zone;
nr_scanned = isolate_pages(rzone, &pages);

weight = (rzone->nr_active + rzone->nr_inactive) /
(nr_scanned * rzone->reclaim->nr_containers);

sched_account(&container->reclaim_queue,
&rzone->sched_item, weight);
```

We use a scheduler to interleave the various lists instead of a sequence of lists to create the appearance of a single longer list. That is, we want each tail to be of equal age.

[ it would probably make sense to drive the shrinking of the active list from the use of the inactive list. This has the advantage of 'hiding' the active list.

Much like proposed here: <http://lkml.org/lkml/2005/12/7/57>  
and here: <http://programming.kicks-ass.net/kernel-patches/page-replace/2.6.21-pr0/useonce-new-shrinker.patch>  
]

Unmapped pages:

~~~~~

Since unmapped pages lack a 'release' (or dis-associate) event, the fairest thing is to account each container a fraction relative to its use of these unmapped pages.

Use would constitute of the following events:

- pagecache insertion
- pagecache lookup

Each containers proportion will be calculated using the floating proportions introduced in the per BDI dirty limit patches.

```
struct prop_global pagecache_proportion;
struct reclaim_pagecache_reclaim;

enum reclaim_item_flags {
    ri_pagecache,
};

struct reclaim_item {
    ...
    unsigned long flags;
};

struct container {
    ...
    struct prop_local_single pagecache_usage;
};
```

and add it to the vm scheduler.

```
if (ri->flags & ri_pagecache) {
    unsigned long num, denom;
    unsigned long long w;

    prop_fraction(&pagecache_proportion,
                  &container->pagecache_usage,
                  &num, &denom);

    w = (rzone->nr_active + rzone->nr_inactive) * denom;
    do_div(w, num);

    weight = (unsigned long)w * nr_scanned;
} else
    ....
```

Considerations:

~~~~~

Advantages:

- each page is on a single list
- no distinction between container vs global reclaim
- no increase of sizeof(struct page)

- pages are but on a single list
- breaks up `lru_lock` (might get a scheduler lock in return)

Disadvantages:

- major overhaul of the reclaim code
- naive container usage calculation will be  $O(nr \text{ vmas})$   
however a smarter scheme would still be  $O(nr \text{ containers})$

Notes:

~~~~~

[1] if needed one could refine this using floating proportions
charging each container a fraction relative to its usage
