
Subject: Re: [RFC][PATCH][3/4] Add reclaim support
Posted by [Andrew Morton](#) on Mon, 19 Feb 2007 08:59:12 GMT
[View Forum Message](#) <> [Reply to Message](#)

On Mon, 19 Feb 2007 12:20:42 +0530 Balbir Singh <balbir@in.ibm.com> wrote:

```
>
> This patch reclaims pages from a container when the container limit is hit.
> The executable is oom'ed only when the container it is running in, is overlimit
> and we could not reclaim any pages belonging to the container
>
> A parameter called pushback, controls how much memory is reclaimed when the
> limit is hit. It should be easy to expose this knob to user space, but
> currently it is hard coded to 20% of the total limit of the container.
>
> isolate_lru_pages() has been modified to isolate pages belonging to a
> particular container, so that reclaim code will reclaim only container
> pages. For shared pages, reclaim does not unmap all mappings of the page,
> it only unmaps those mappings that are over their limit. This ensures
> that other containers are not penalized while reclaiming shared pages.
>
> Parallel reclaim per container is not allowed. Each controller has a wait
> queue that ensures that only one task per control is running reclaim on
> that container.
>
>
> ...
>
> --- linux-2.6.20/include/linux/rmap.h~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000
+0530
> +++ linux-2.6.20-balbir/include/linux/rmap.h 2007-02-18 23:29:14.000000000 +0530
> @@ -90,7 +90,15 @@ static inline void page_dup_rmap(struct
>  * Called from mm/vmscan.c to handle paging out
>  */
> int page_referenced(struct page *, int is_locked);
> -int try_to_unmap(struct page *, int ignore_refs);
> +int try_to_unmap(struct page *, int ignore_refs, void *container);
> +#ifdef CONFIG_CONTAINER_MEMCTLR
> +bool page_in_container(struct page *page, struct zone *zone, void *container);
> +#else
> +static inline bool page_in_container(struct page *page, struct zone *zone, void *container)
> +{
> + return true;
> +}
> +#endif /* CONFIG_CONTAINER_MEMCTLR */
>
> /*
>  * Called from mm/filemap_xip.c to unmap empty zero page
```

```

> @@ -118,7 +126,8 @@ int page_mkclean(struct page *);
> #define anon_vma_link(vma) do {} while (0)
>
> #define page_referenced(page,l) TestClearPageReferenced(page)
> -#define try_to_unmap(page, refs) SWAP_FAIL
> +#define try_to_unmap(page, refs, container) SWAP_FAIL
> +#define page_in_container(page, zone, container) true

```

I spy a compile error.

The static-inline version looks nicer.

```

> static inline int page_mkclean(struct page *page)
> {
> diff -puN include/linux/swap.h~memctlr-reclaim-on-limit include/linux/swap.h
> --- linux-2.6.20/include/linux/swap.h~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000
+0530
> +++ linux-2.6.20-balbir/include/linux/swap.h 2007-02-18 23:29:14.000000000 +0530
> @@ -188,6 +188,10 @@ extern void swap_setup(void);
> /* linux/mm/vmscan.c */
> extern unsigned long try_to_free_pages(struct zone **, gfp_t);
> extern unsigned long shrink_all_memory(unsigned long nr_pages);
> +#ifdef CONFIG_CONTAINER_MEMCTL
> +extern unsigned long memctlr_shrink_mapped_memory(unsigned long nr_pages,
> + void *container);
> +#endif

```

Usually one doesn't need to put ifdefs around the declaration like this.
 If the function doesn't exist and nobody calls it, we're fine. If someone
does call it, we'll find out the error at link-time.

```

>
> +/*
> + * checks if the mm's container and scan control passed container match, if
> + * so, is the container over it's limit. Returns 1 if the container is above
> + * its limit.
> + */
> +int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont)
> +{
> + struct container *cont;
> + struct memctlr *mem;
> + long usage, limit;
> + int ret = 1;
> +
> + if (!sc_cont)
> + goto out;
> +
> + read_lock(&mm->container_lock);

```

```

> + cont = mm->container;
> +
> + /*
> +  * Regular reclaim, let it proceed as usual
> +  */
> + if (!sc_cont)
> + goto out;
> +
> + ret = 0;
> + if (cont != sc_cont)
> + goto out;
> +
> + mem = memctlr_from_cont(cont);
> + usage = atomic_long_read(&mem->counter.usage);
> + limit = atomic_long_read(&mem->counter.limit);
> + if (limit && (usage > limit))
> + ret = 1;
> +out:
> + read_unlock(&mm->container_lock);
> + return ret;
> +}

```

hm, I wonder how much additional lock traffic all this adds.

```

> int memctlr_mm_init(struct mm_struct *mm)
> {
> mm->counter = kcalloc(sizeof(struct res_counter), GFP_KERNEL);
> @@ -77,6 +125,46 @@ void memctlr_mm_assign_container(struct
> write_unlock(&mm->container_lock);
> }
>
> +static int memctlr_check_and_reclaim(struct container *cont, long usage,
> + long limit)
> +{
> + unsigned long nr_pages = 0;
> + unsigned long nr_reclaimed = 0;
> + int retries = nr_retries;
> + int ret = 1;
> + struct memctlr *mem;
> +
> + mem = memctlr_from_cont(cont);
> + spin_lock(&mem->lock);
> + while ((retries-- > 0) && limit && (usage > limit)) {
> + if (mem->reclaim_in_progress) {
> + spin_unlock(&mem->lock);
> + wait_event(mem->wq, !mem->reclaim_in_progress);
> + spin_lock(&mem->lock);
> + } else {

```

```

> + if (!nr_pages)
> +   nr_pages = (pushback * limit) / 100;
> +   mem->reclaim_in_progress = true;
> +   spin_unlock(&mem->lock);
> +   nr_reclaimed += memctlr_shrink_mapped_memory(nr_pages,
> +     cont);
> +   spin_lock(&mem->lock);
> +   mem->reclaim_in_progress = false;
> +   wake_up_all(&mem->wq);
> + }
> + /*
> +  * Resample usage and limit after reclaim
> +  */
> + usage = atomic_long_read(&mem->counter.usage);
> + limit = atomic_long_read(&mem->counter.limit);
> + }
> + spin_unlock(&mem->lock);
> +
> + if (limit && (usage > limit))
> +   ret = 0;
> + return ret;
> +}

```

This all looks a bit racy. And that's common in memory reclaim. We just have to ensure that when the race happens, we do reasonable things.

I suspect the locking in here could simply be removed.

```

> @@ -66,6 +67,9 @@ struct scan_control {
>   int swappiness;
>
>   int all_unreclaimable;
> +
> + void *container; /* Used by containers for reclaiming */
> +   /* pages when the limit is exceeded */
> };

```

eww. Why void*?

```

> +#ifdef CONFIG_CONTAINER_MEMCTL
> +/*
> + * Try to free `nr_pages' of memory, system-wide, and return the number of
> + * freed pages.
> + * Modelled after shrink_all_memory()
> + */
> +unsigned long memctlr_shrink_mapped_memory(unsigned long nr_pages, void *container)

```

80-columns, please.

```

> +{
> + unsigned long ret = 0;
> + int pass;
> + unsigned long nr_total_scanned = 0;
> +
> + struct scan_control sc = {
> + .gfp_mask = GFP_KERNEL,
> + .may_swap = 0,
> + .swap_cluster_max = nr_pages,
> + .may_writepage = 1,
> + .swappiness = vm_swappiness,
> + .container = container,
> + .may_swap = 1,
> + .swappiness = 100,
> + };

```

swappiness got initialised twice.

```

> + /*
> + * We try to shrink LRUs in 3 passes:
> + * 0 = Reclaim from inactive_list only
> + * 1 = Reclaim mapped (normal reclaim)
> + * 2 = 2nd pass of type 1
> + */
> + for (pass = 0; pass < 3; pass++) {
> + int prio;
> +
> + for (prio = DEF_PRIORITY; prio >= 0; prio--) {
> + unsigned long nr_to_scan = nr_pages - ret;
> +
> + sc.nr_scanned = 0;
> + ret += shrink_all_zones(nr_to_scan, prio,
> + pass, 1, &sc);
> + if (ret >= nr_pages)
> + goto out;
> +
> + nr_total_scanned += sc.nr_scanned;
> + if (sc.nr_scanned && prio < DEF_PRIORITY - 2)
> + congestion_wait(WRITE, HZ / 10);
> + }
> + }
> +out:
> + return ret;
> +}
> +#endif

```
