```
Memory Resource Controller(Memcg)  Implementation Memo.
Last Updated: 2010/2
Base Kernel Version: based on 2.6.33-rc7-mm(candidate for 34).

Because VM is getting complex (one of reasons is memcg...), memcg's behavior
is complex. This is a document for memcg's internal behavior.
Please note that implementation details can be changed.

(*) Topics on API should be in Documentation/cgroups/memory.txt)

0. How to record usage ?
   2 objects are used.

   page_cgroup ....an object per page.
        Allocated at boot or memory hotplug. Freed at memory hot removal.

   swap_cgroup ... an entry per swp_entry.
        Allocated at swapon(). Freed at swapoff().

   The page_cgroup has USED bit and double count against a page_cgroup never
   occurs. swap_cgroup is used only when a charged page is swapped-out.

1. Charge

   a page/swp_entry may be charged (usage += PAGE_SIZE) at

        mem_cgroup_newpage_charge()
          Called at new page fault and Copy-On-Write.

        mem_cgroup_try_charge_swapin()
          Called at do_swap_page() (page fault on swap entry) and swapoff.
          Followed by charge-commit-cancel protocol. (With swap accounting)
          At commit, a charge recorded in swap_cgroup is removed.

        mem_cgroup_cache_charge()
          Called at add_to_page_cache()

        mem_cgroup_cache_charge_swapin()
          Called at shmem's swapin.

        mem_cgroup_prepare_migration()
          Called before migration. "extra" charge is done and followed by
          charge-commit-cancel protocol.
          At commit, charge against oldpage or newpage will be committed.

2. Uncharge
  a page/swp_entry may be uncharged (usage -= PAGE_SIZE) by

        mem_cgroup_uncharge_page()
          Called when an anonymous page is fully unmapped. I.e., mapcount goes
          to 0. If the page is SwapCache, uncharge is delayed until
          mem_cgroup_uncharge_swapcache().

        mem_cgroup_uncharge_cache_page()
          Called when a page-cache is deleted from radix-tree. If the page is
          SwapCache, uncharge is delayed until mem_cgroup_uncharge_swapcache().
```

```
        mem_cgroup_uncharge_swapcache()
          Called when SwapCache is removed from radix-tree. The charge itself
          is moved to swap_cgroup. (If mem+swap controller is disabled, no
          charge to swap occurs.)

        mem_cgroup_uncharge_swap()
          Called when swp_entry's refcnt goes down to 0. A charge against swap
          disappears.

        mem_cgroup_end_migration(old, new)
        At success of migration old is uncharged (if necessary), a charge
        to new page is committed. At failure, charge to old page is committed.

3. charge-commit-cancel
        In some case, we can't know this "charge" is valid or not at charging
        (because of races).
        To handle such case, there are charge-commit-cancel functions.
                mem_cgroup_try_charge_XXX
                mem_cgroup_commit_charge_XXX
                mem_cgroup_cancel_charge_XXX
        these are used in swap-in and migration.

        At try_charge(), there are no flags to say "this page is charged".
        at this point, usage += PAGE_SIZE.

        At commit(), the function checks the page should be charged or not
        and set flags or avoid charging.(usage -= PAGE_SIZE)

        At cancel(), simply usage -= PAGE_SIZE.

Under below explanation, we assume CONFIG_MEM_RES_CTRL_SWAP=y.

4. Anonymous
        Anonymous page is newly allocated at
                  - page fault into MAP_ANONYMOUS mapping.
                  - Copy-On-Write.
        It is charged right after it's allocated before doing any page table
        related operations. Of course, it's uncharged when another page is used
        for the fault address.

        At freeing anonymous page (by exit() or munmap()), zap_pte() is called
        and pages for ptes are freed one by one.(see mm/memory.c). Uncharges
        are done at page_remove_rmap() when page_mapcount() goes down to 0.

        Another page freeing is by page-reclaim (vmscan.c) and anonymous
        pages are swapped out. In this case, the page is marked as
        PageSwapCache(). uncharge() routine doesn't uncharge the page marked
        as SwapCache(). It's delayed until __delete_from_swap_cache().

        4.1 Swap-in.
        At swap-in, the page is taken from swap-cache. There are 2 cases.

        (a) If the SwapCache is newly allocated and read, it has no charges.
        (b) If the SwapCache has been mapped by processes, it has been
            charged already.
```

This swap-in is one of the most complicated work. In do_swap_page(), following events occur when pte is unchanged.

(1) the page (SwapCache) is looked up.
(2) lock_page()
(3) try_charge_swapin()
(4) reuse_swap_page() (may call delete_swap_cache())
(5) commit_charge_swapin()
(6) swap_free().

Considering following situation for example.

(A) The page has not been charged before (2) and reuse_swap_page() doesn't call delete_from_swap_cache().
(B) The page has not been charged before (2) and reuse_swap_page() calls delete_from_swap_cache().
(C) The page has been charged before (2) and reuse_swap_page() doesn't call delete_from_swap_cache().
(D) The page has been charged before (2) and reuse_swap_page() calls delete_from_swap_cache().

memory.usage/memsw.usage changes to this page/swp_entry will be

Case / Event	(A)	(B)	(C)	(D)
Before (2)	0/ 1	0/ 1	1/ 1	1/ 1
(3)	+1/+1	+1/+1	+1/+1	+1/+1
(4)	-	0/ 0	-	-1/ 0
(5)	0/-1	0/ 0	-1/-1	0/ 0
(6)	-	0/-1	-	0/-1
Result	1/ 1	1/ 1	1/ 1	1/ 1

In any cases, charges to this page should be 1/ 1.

4.2 Swap-out.
At swap-out, typical state transition is below.

(a) add to swap cache. (marked as SwapCache)
    swp_entry's refcnt += 1.
(b) fully unmapped.
    swp_entry's refcnt += # of ptes.
(c) write back to swap.
(d) delete from swap cache. (remove from SwapCache)
    swp_entry's refcnt -= 1.

At (b), the page is marked as SwapCache and not uncharged.
At (d), the page is removed from SwapCache and a charge in page_cgroup is moved to swap_cgroup.

Finally, at task exit,
(e) zap_pte() is called and swp_entry's refcnt -=1 -> 0.
Here, a charge in swap_cgroup disappears.

```
5. Page Cache
	Page Cache is charged at
	- add_to_page_cache_locked().

	uncharged at
	- __remove_from_page_cache().

	The logic is very clear. (About migration, see below)
	Note: __remove_from_page_cache() is called by remove_from_page_cache()
	and __remove_mapping().

6. Shmem(tmpfs) Page Cache
	Memcg's charge/uncharge have special handlers of shmem. The best way
	to understand shmem's page state transition is to read mm/shmem.c.
	But brief explanation of the behavior of memcg around shmem will be
	helpful to understand the logic.

	Shmem's page (just leaf page, not direct/indirect block) can be on
		- radix-tree of shmem's inode.
		- SwapCache.
		- Both on radix-tree and SwapCache. This happens at swap-in
		  and swap-out,

	It's charged when...
	- A new page is added to shmem's radix-tree.
	- A swp page is read. (move a charge from swap_cgroup to page_cgroup)
	It's uncharged when
	- A page is removed from radix-tree and not SwapCache.
	- When SwapCache is removed, a charge is moved to swap_cgroup.
	- When swp_entry's refcnt goes down to 0, a charge in swap_cgroup
	  disappears.

7. Page Migration
	One of the most complicated functions is page-migration-handler.
	Memcg has 2 routines. Assume that we are migrating a page's contents
	from OLDPAGE to NEWPAGE.

	Usual migration logic is..
	(a) remove the page from LRU.
	(b) allocate NEWPAGE (migration target)
	(c) lock by lock_page().
	(d) unmap all mappings.
	(e-1) If necessary, replace entry in radix-tree.
	(e-2) move contents of a page.
	(f) map all mappings again.
	(g) pushback the page to LRU.
	(-) OLDPAGE will be freed.

	Before (g), memcg should complete all necessary charge/uncharge to
	NEWPAGE/OLDPAGE.

	The point is....
	- If OLDPAGE is anonymous, all charges will be dropped at (d) because
	  try_to_unmap() drops all mapcount and the page will not be
	  SwapCache.
```

```
	- If OLDPAGE is SwapCache, charges will be kept at (g) because
	  __delete_from_swap_cache() isn't called at (e-1)

	- If OLDPAGE is page-cache, charges will be kept at (g) because
	  remove_from_swap_cache() isn't called at (e-1)

	memcg provides following hooks.

	- mem_cgroup_prepare_migration(OLDPAGE)
	  Called after (b) to account a charge (usage += PAGE_SIZE) against
	  memcg which OLDPAGE belongs to.

	- mem_cgroup_end_migration(OLDPAGE, NEWPAGE)
	  Called after (f) before (g).
	  If OLDPAGE is used, commit OLDPAGE again. If OLDPAGE is already
	  charged, a charge by prepare_migration() is automatically canceled.
	  If NEWPAGE is used, commit NEWPAGE and uncharge OLDPAGE.

	  But zap_pte() (by exit or munmap) can be called while migration,
	  we have to check if OLDPAGE/NEWPAGE is a valid page after commit().

8. LRU
	Each memcg has its own private LRU. Now, its handling is under global
	VM's control (means that it's handled under global zone->lru_lock).
	Almost all routines around memcg's LRU is called by global LRU's
	list management functions under zone->lru_lock().

	A special function is mem_cgroup_isolate_pages(). This scans
	memcg's private LRU and call __isolate_lru_page() to extract a page
	from LRU.
	(By __isolate_lru_page(), the page is removed from both of global and
	 private LRU.)


9. Typical Tests.

 Tests for racy cases.

 9.1 Small limit to memcg.
	When you do test to do racy case, it's good test to set memcg's limit
	to be very small rather than GB. Many races found in the test under
	xKB or xxMB limits.
	(Memory behavior under GB and Memory behavior under MB shows very
	 different situation.)

 9.2 Shmem
	Historically, memcg's shmem handling was poor and we saw some amount
	of troubles here. This is because shmem is page-cache but can be
	SwapCache. Test with shmem/tmpfs is always good test.

 9.3 Migration
	For NUMA, migration is an another special case. To do easy test, cpuset
	is useful. Following is a sample script to do migration.

	mount -t cgroup -o cpuset none /opt/cpuset
```

```
mkdir /opt/cpuset/01
echo 1 > /opt/cpuset/01/cpuset.cpus
echo 0 > /opt/cpuset/01/cpuset.mems
echo 1 > /opt/cpuset/01/cpuset.memory_migrate
mkdir /opt/cpuset/02
echo 1 > /opt/cpuset/02/cpuset.cpus
echo 1 > /opt/cpuset/02/cpuset.mems
echo 1 > /opt/cpuset/02/cpuset.memory_migrate
```

In above set, when you moves a task from 01 to 02, page migration to node 0 to node 1 will occur. Following is a script to migrate all under cpuset.

```
--
move_task()
{
for pid in $1
do
        /bin/echo $pid >$2/tasks 2>/dev/null
        echo -n $pid
        echo -n " "
done
echo END
}

G1_TASK=`cat ${G1}/tasks`
G2_TASK=`cat ${G2}/tasks`
move_task "${G1_TASK}" ${G2} &
--
```

9.4 Memory hotplug.

memory hotplug test is one of good test.
to offline memory, do following.

```
# echo offline > /sys/devices/system/memory/memoryXXX/state
```

(XXX is the place of memory)
This is an easy way to test page migration, too.

9.5 mkdir/rmdir

When using hierarchy, mkdir/rmdir test should be done.
Use tests like the following.

```
echo 1 >/opt/cgroup/01/memory/use_hierarchy
mkdir /opt/cgroup/01/child_a
mkdir /opt/cgroup/01/child_b
```

set limit to 01.
add limit to 01/child_b
run jobs under child_a and child_b

create/delete following groups at random while jobs are running.

```
/opt/cgroup/01/child_a/child_aa
/opt/cgroup/01/child_b/child_bb
/opt/cgroup/01/child_c
```

running new jobs in new group is also good.

9.6 Mount with other subsystems.

Mounting with other subsystems is a good test because there is a

```
        race and lock dependency with other cgroup subsystems.

        example)
        # mount -t cgroup none /cgroup -o cpuset,memory,cpu,devices

        and do task move, mkdir, rmdir etc...under this.

9.7 swapoff.
        Besides management of swap is one of complicated parts of memcg,
        call path of swap-in at swapoff is not same as usual swap-in path..
        It's worth to be tested explicitly.

        For example, test like following is good.
        (Shell-A)
        # mount -t cgroup none /cgroup -o memory
        # mkdir /cgroup/test
        # echo 40M > /cgroup/test/memory.limit_in_bytes
        # echo 0 > /cgroup/test/tasks
        Run malloc(100M) program under this. You'll see 60M of swaps.
        (Shell-B)
        # move all tasks in /cgroup/test to /cgroup
        # /sbin/swapoff -a
        # rmdir /cgroup/test
        # kill malloc task.

        Of course, tmpfs v.s. swapoff test should be tested, too.

9.8 OOM-Killer
        Out-of-memory caused by memcg's limit will kill tasks under
        the memcg. When hierarchy is used, a task under hierarchy
        will be killed by the kernel.
        In this case, panic_on_oom shouldn't be invoked and tasks
        in other groups shouldn't be killed.

        It's not difficult to cause OOM under memcg as following.
        Case A) when you can swapoff
        #swapoff -a
        #echo 50M > /memory.limit_in_bytes
        run 51M of malloc

        Case B) when you use mem+swap limitation.
        #echo 50M > memory.limit_in_bytes
        #echo 50M > memory.memsw.limit_in_bytes
        run 51M of malloc

9.9 Move charges at task migration
        Charges associated with a task can be moved along with task migration.

        (Shell-A)
        #mkdir /cgroup/A
        #echo $$ >/cgroup/A/tasks
        run some programs which uses some amount of memory in /cgroup/A.

        (Shell-B)
        #mkdir /cgroup/B
        #echo 1 >/cgroup/B/memory.move_charge_at_immigrate
```

```
#echo "pid of the program running in group A" >/cgroup/B/tasks
```

You can see charges have been moved by reading *.usage_in_bytes or memory.stat of both A and B.
See 8.2 of Documentation/cgroups/memory.txt to see what value should be written to move_charge_at_immigrate.

9.10 Memory thresholds

Memory controler implements memory thresholds using cgroups notification API. You can use Documentation/cgroups/cgroup_event_listener.c to test it.

(Shell-A) Create cgroup and run event listener

```
# mkdir /cgroup/A
# ./cgroup_event_listener /cgroup/A/memory.usage_in_bytes 5M
```

(Shell-B) Add task to cgroup and try to allocate and free memory

```
# echo $$ >/cgroup/A/tasks
# a="$(dd if=/dev/zero bs=1M count=10)"
# a=
```

You will see message from cgroup_event_listener every time you cross the thresholds.

Use /cgroup/A/memory.memsw.usage_in_bytes to test memsw thresholds.

It's good idea to test root cgroup as well.