Memory Resource Controller

NOTE: The Memory Resource Controller has been generically been referred
      to as the memory controller in this document. Do not confuse memory
      controller used here with the memory controller that is used in hardware.

(For editors)
In this document:
      When we mention a cgroup (cgroupfs's directory) with memory controller,
      we call it "memory cgroup". When you see git-log and source code, you'll
      see patch's title and function names tend to use "memcg".
      In this document, we avoid using it.

Benefits and Purpose of the memory controller

The memory controller isolates the memory behaviour of a group of tasks
from the rest of the system. The article on LWN [12] mentions some probable
uses of the memory controller. The memory controller can be used to

a. Isolate an application or a group of applications
   Memory hungry applications can be isolated and limited to a smaller
   amount of memory.
b. Create a cgroup with limited amount of memory, this can be used
   as a good alternative to booting with mem=XXXX.
c. Virtualization solutions can control the amount of memory they want
   to assign to a virtual machine instance.
d. A CD/DVD burner could control the amount of memory used by the
   rest of the system to ensure that burning does not fail due to lack
   of available memory.
e. There are several other use cases, find one or use the controller just
   for fun (to learn and hack on the VM subsystem).

Current Status: linux-2.6.34-mmotm(development version of 2010/April)

Features:
 - accounting anonymous pages, file caches, swap caches usage and limiting them.
 - private LRU and reclaim routine. (system's global LRU and private LRU
   work independently from each other)
 - optionally, memory+swap usage can be accounted and limited.
 - hierarchical accounting
 - soft limit
 - moving(recharging) account at moving a task is selectable.
 - usage threshold notifier
 - oom-killer disable knob and oom-notifier
 - Root cgroup has no limit controls.

 Kernel memory and Hugepages are not under control yet. We just manage
 pages on LRU. To add more controls, we have to take care of performance.

Brief summary of control files.

```
 tasks                          # attach a task(thread) and show list of
threads
 cgroup.procs                   # show list of processes
 cgroup.event_control           # an interface for event_fd()
 memory.usage_in_bytes          # show current memory(RSS+Cache) usage.
```

```
 memory.memsw.usage_in_bytes     # show current memory+Swap usage
 memory.limit_in_bytes           # set/show limit of memory usage
 memory.memsw.limit_in_bytes     # set/show limit of memory+Swap usage
 memory.failcnt                  # show the number of memory usage hits limits
 memory.memsw.failcnt            # show the number of memory+Swap hits limits
 memory.max_usage_in_bytes       # show max memory usage recorded
 memory.memsw.usage_in_bytes     # show max memory+Swap usage recorded
 memory.soft_limit_in_bytes      # set/show soft limit of memory usage
 memory.stat                     # show various statistics
 memory.use_hierarchy            # set/show hierarchical account enabled
 memory.force_empty              # trigger forced move charge to parent
 memory.swappiness               # set/show swappiness parameter of vmscan
                                 (See sysctl's vm.swappiness)
 memory.move_charge_at_immigrate # set/show controls of moving charges
 memory.oom_control              # set/show oom controls.
```

1. History

The memory controller has a long history. A request for comments for the memory controller was posted by Balbir Singh [1]. At the time the RFC was posted there were several implementations for memory control. The goal of the RFC was to build consensus and agreement for the minimal features required for memory control. The first RSS controller was posted by Balbir Singh[2] in Feb 2007. Pavel Emelianov [3][4][5] has since posted three versions of the RSS controller. At OLS, at the resource management BoF, everyone suggested that we handle both page cache and RSS together. Another request was raised to allow user space handling of OOM. The current memory controller is at version 6; it combines both mapped (RSS) and unmapped Page Cache Control [11].

2. Memory Control

Memory is a unique resource in the sense that it is present in a limited amount. If a task requires a lot of CPU processing, the task can spread its processing over a period of hours, days, months or years, but with memory, the same physical memory needs to be reused to accomplish the task.

The memory controller implementation has been divided into phases. These are:

1. Memory controller
2. mlock(2) controller
3. Kernel user memory accounting and slab control
4. user mappings length controller

The memory controller is the first controller developed.

2.1. Design

The core of the design is a counter called the res_counter. The res_counter tracks the current memory usage and limit of the group of processes associated with the controller. Each cgroup has a memory controller specific data structure (mem_cgroup) associated with it.

2.2. Accounting

```
        +--------------------+
        |  mem_cgroup     |
        |  (res_counter)     |
        +--------------------+
         /            ^      \
        /             |       \
   +---------------+  |        +---------------+
   | mm_struct     |  |....    | mm_struct     |
   |               |  |        |               |
   +---------------+  |        +---------------+
                      |
                      + --------------+
                                      |
   +---------------+           +------+--------+
   | page          +---------->  page_cgroup|
   |               |           |               |
   +---------------+           +---------------+

     (Figure 1: Hierarchy of Accounting)
```

Figure 1 shows the important aspects of the controller

1. Accounting happens per cgroup
2. Each mm_struct knows about which cgroup it belongs to
3. Each page has a pointer to the page_cgroup, which in turn knows the
   cgroup it belongs to

The accounting is done as follows: mem_cgroup_charge() is invoked to setup the necessary data structures and check if the cgroup that is being charged is over its limit. If it is then reclaim is invoked on the cgroup.
More details can be found in the reclaim section of this document.
If everything goes well, a page meta-data-structure called page_cgroup is updated. page_cgroup has its own LRU on cgroup.
(*) page_cgroup structure is allocated at boot/memory-hotplug time.

2.2.1 Accounting details

All mapped anon pages (RSS) and cache pages (Page Cache) are accounted.
Some pages which are never reclaimable and will not be on the global LRU are not accounted. We just account pages under usual VM management.

RSS pages are accounted at page_fault unless they've already been accounted for earlier. A file page will be accounted for as Page Cache when it's inserted into inode (radix-tree). While it's mapped into the page tables of processes, duplicate accounting is carefully avoided.

A RSS page is unaccounted when it's fully unmapped. A PageCache page is unaccounted when it's removed from radix-tree. Even if RSS pages are fully unmapped (by kswapd), they may exist as SwapCache in the system until they are really freed. Such SwapCaches also also accounted.
A swapped-in page is not accounted until it's mapped.

Note: The kernel does swapin-readahead and read multiple swaps at once. This means swapped-in pages may contain pages for other tasks than a task causing page fault. So, we avoid accounting at swap-in I/O.

At page migration, accounting information is kept.

Note: we just account pages-on-LRU because our purpose is to control amount of used pages; not-on-LRU pages tend to be out-of-control from VM view.

2.3 Shared Page Accounting

Shared pages are accounted on the basis of the first touch approach. The cgroup that first touches a page is accounted for the page. The principle behind this approach is that a cgroup that aggressively uses a shared page will eventually get charged for it (once it is uncharged from the cgroup that brought it in -- this will happen on memory pressure).

Exception: If CONFIG_CGROUP_CGROUP_MEM_RES_CTLR_SWAP is not used..
When you do swapoff and make swapped-out pages of shmem(tmpfs) to be backed into memory in force, charges for pages are accounted against the caller of swapoff rather than the users of shmem.

2.4 Swap Extension (CONFIG_CGROUP_MEM_RES_CTLR_SWAP)

Swap Extension allows you to record charge for swap. A swapped-in page is charged back to original page allocator if possible.

When swap is accounted, following files are added.
 - memory.memsw.usage_in_bytes.
 - memory.memsw.limit_in_bytes.

memsw means memory+swap. Usage of memory+swap is limited by memsw.limit_in_bytes.

Example: Assume a system with 4G of swap. A task which allocates 6G of memory (by mistake) under 2G memory limitation will use all swap.
In this case, setting memsw.limit_in_bytes=3G will prevent bad use of swap.
By using memsw limit, you can avoid system OOM which can be caused by swap shortage.

* why 'memory+swap' rather than swap.
The global LRU(kswapd) can swap out arbitrary pages. Swap-out means to move account from memory to swap...there is no change in usage of memory+swap. In other words, when we want to limit the usage of swap without affecting global LRU, memory+swap limit is better than just limiting swap from OS point of view.

* What happens when a cgroup hits memory.memsw.limit_in_bytes
When a cgroup his memory.memsw.limit_in_bytes, it's useless to do swap-out in this cgroup. Then, swap-out will not be done by cgroup routine and file caches are dropped. But as mentioned above, global LRU can do swapout memory from it for sanity of the system's memory management state. You can't forbid it by cgroup.

2.5 Reclaim

Each cgroup maintains a per cgroup LRU which has the same structure as global VM. When a cgroup goes over its limit, we first try

to reclaim memory from the cgroup so as to make space for the new
pages that the cgroup has touched. If the reclaim is unsuccessful,
an OOM routine is invoked to select and kill the bulkiest task in the
cgroup. (See 10. OOM Control below.)

The reclaim algorithm has not been modified for cgroups, except that
pages that are selected for reclaiming come from the per cgroup LRU
list.

NOTE: Reclaim does not work for the root cgroup, since we cannot set any
limits on the root cgroup.

Note2: When panic_on_oom is set to "2", the whole system will panic.

When oom event notifier is registered, event will be delivered.
(See oom_control section)

2.6 Locking

```
   lock_page_cgroup()/unlock_page_cgroup() should not be called under
   mapping->tree_lock.

   Other lock order is following:
   PG_locked.
   mm->page_table_lock
       zone->lru_lock
          lock_page_cgroup.
  In many cases, just lock_page_cgroup() is called.
  per-zone-per-cgroup LRU (cgroup's private LRU) is just guarded by
  zone->lru_lock, it has no lock of its own.
```

3. User Interface

0. Configuration

a. Enable CONFIG_CGROUPS
b. Enable CONFIG_RESOURCE_COUNTERS
c. Enable CONFIG_CGROUP_MEM_RES_CTLR
d. Enable CONFIG_CGROUP_MEM_RES_CTLR_SWAP (to use swap extension)

1. Prepare the cgroups
```
# mkdir -p /cgroups
# mount -t cgroup none /cgroups -o memory
```

2. Make the new group and move bash into it
```
# mkdir /cgroups/0
# echo $$ > /cgroups/0/tasks
```

Since now we're in the 0 cgroup, we can alter the memory limit:
```
# echo 4M > /cgroups/0/memory.limit_in_bytes
```

NOTE: We can use a suffix (k, K, m, M, g or G) to indicate values in kilo,
mega or gigabytes. (Here, Kilo, Mega, Giga are Kibibytes, Mebibytes, Gibibytes.)

NOTE: We can write "-1" to reset the *.limit_in_bytes(unlimited).
NOTE: We cannot set limits on the root cgroup any more.

```
# cat /cgroups/0/memory.limit_in_bytes
4194304
```

We can check the usage:

```
# cat /cgroups/0/memory.usage_in_bytes
1216512
```

A successful write to this file does not guarantee a successful set of this limit to the value written into the file. This can be due to a number of factors, such as rounding up to page boundaries or the total availability of memory on the system. The user is required to re-read this file after a write to guarantee the value committed by the kernel.

```
# echo 1 > memory.limit_in_bytes
# cat memory.limit_in_bytes
4096
```

The memory.failcnt field gives the number of times that the cgroup limit was exceeded.

The memory.stat file gives accounting information. Now, the number of caches, RSS and Active pages/Inactive pages are shown.

4. Testing

For testing features and implementation, see memcg_test.txt.

Performance test is also important. To see pure memory controller's overhead, testing on tmpfs will give you good numbers of small overheads.
Example: do kernel make on tmpfs.

Page-fault scalability is also important. At measuring parallel page fault test, multi-process test may be better than multi-thread test because it has noise of shared objects/status.

But the above two are testing extreme situations.
Trying usual test under memory controller is always helpful.

4.1 Troubleshooting

Sometimes a user might find that the application under a cgroup is terminated by OOM killer. There are several causes for this:

1. The cgroup limit is too low (just too low to do anything useful)
2. The user is using anonymous memory and swap is turned off or too low

A sync followed by echo 1 > /proc/sys/vm/drop_caches will help get rid of some of the pages cached in the cgroup (page cache pages).

To know what happens, disable OOM_Kill by 10. OOM Control(see below) and seeing what happens will be helpful.

4.2 Task migration

When a task migrates from one cgroup to another, its charge is not carried forward by default. The pages allocated from the original cgroup still remain charged to it, the charge is dropped when the page is freed or reclaimed.

You can move charges of a task along with task migration.
See 8. "Move charges at task migration"

4.3 Removing a cgroup

A cgroup can be removed by rmdir, but as discussed in sections 4.1 and 4.2, a cgroup might have some charge associated with it, even though all tasks have migrated away from it. (because we charge against pages, not against tasks.)

Such charges are freed or moved to their parent. At moving, both of RSS and CACHES are moved to parent.
rmdir() may return -EBUSY if freeing/moving fails. See 5.1 also.

Charges recorded in swap information is not updated at removal of cgroup. Recorded information is discarded and a cgroup which uses swap (swapcache) will be charged as a new owner of it.

5. Misc. interfaces.

5.1 force_empty
  memory.force_empty interface is provided to make cgroup's memory usage empty.
  You can use this interface only when the cgroup has no tasks.
  When writing anything to this

  # echo 0 > memory.force_empty

  Almost all pages tracked by this memory cgroup will be unmapped and freed.
  Some pages cannot be freed because they are locked or in-use. Such pages are
  moved to parent and this cgroup will be empty. This may return -EBUSY if
  VM is too busy to free/move all pages immediately.

  Typical use case of this interface is that calling this before rmdir().
  Because rmdir() moves all pages to parent, some out-of-use page caches can be
  moved to the parent. If you want to avoid that, force_empty will be useful.

5.2 stat file

memory.stat file includes following statistics

```
# per-memory cgroup local status
cache           - # of bytes of page cache memory.
rss             - # of bytes of anonymous and swap cache memory.
mapped_file     - # of bytes of mapped file (includes tmpfs/shmem)
pgpgin          - # of pages paged in (equivalent to # of charging events).
pgpgout         - # of pages paged out (equivalent to # of uncharging events).
swap            - # of bytes of swap usage
inactive_anon   - # of bytes of anonymous memory and swap cache memory on
                LRU list.
active_anon     - # of bytes of anonymous and swap cache memory on active
                inactive LRU list.
```

```
inactive_file   - # of bytes of file-backed memory on inactive LRU list.
active_file     - # of bytes of file-backed memory on active LRU list.
unevictable     - # of bytes of memory that cannot be reclaimed (mlocked etc).

# status considering hierarchy (see memory.use_hierarchy settings)

hierarchical_memory_limit - # of bytes of memory limit with regard to hierarchy
                        under which the memory cgroup is
hierarchical_memsw_limit - # of bytes of memory+swap limit with regard to
                        hierarchy under which memory cgroup is.

total_cache             - sum of all children's "cache"
total_rss               - sum of all children's "rss"
total_mapped_file       - sum of all children's "cache"
total_pgpgin            - sum of all children's "pgpgin"
total_pgpgout           - sum of all children's "pgpgout"
total_swap              - sum of all children's "swap"
total_inactive_anon     - sum of all children's "inactive_anon"
total_active_anon       - sum of all children's "active_anon"
total_inactive_file     - sum of all children's "inactive_file"
total_active_file       - sum of all children's "active_file"
total_unevictable       - sum of all children's "unevictable"

# The following additional stats are dependent on CONFIG_DEBUG_VM.

inactive_ratio          - VM internal parameter. (see mm/page_alloc.c)
recent_rotated_anon     - VM internal parameter. (see mm/vmscan.c)
recent_rotated_file     - VM internal parameter. (see mm/vmscan.c)
recent_scanned_anon     - VM internal parameter. (see mm/vmscan.c)
recent_scanned_file     - VM internal parameter. (see mm/vmscan.c)

Memo:
        recent_rotated means recent frequency of LRU rotation.
        recent_scanned means recent # of scans to LRU.
        showing for better debug please see the code for meanings.

Note:
        Only anonymous and swap cache memory is listed as part of 'rss' stat.
        This should not be confused with the true 'resident set size' or the
        amount of physical memory used by the cgroup.
        'rss + file_mapped" will give you resident set size of cgroup.
        (Note: file and shmem may be shared among other cgroups. In that case,
         file_mapped is accounted only when the memory cgroup is owner of page
         cache.)

5.3 swappiness

Similar to /proc/sys/vm/swappiness, but affecting a hierarchy of groups only.

Following cgroups' swappiness can't be changed.
- root cgroup (uses /proc/sys/vm/swappiness).
- a cgroup which uses hierarchy and it has other cgroup(s) below it.
- a cgroup which uses hierarchy and not the root of hierarchy.

5.4 failcnt
```

A memory cgroup provides memory.failcnt and memory.memsw.failcnt files.
This failcnt(== failure count) shows the number of times that a usage counter hit its limit. When a memory cgroup hits a limit, failcnt increases and memory under it will be reclaimed.

You can reset failcnt by writing 0 to failcnt file.
echo 0 > .../memory.failcnt

6. Hierarchy support

The memory controller supports a deep hierarchy and hierarchical accounting. The hierarchy is created by creating the appropriate cgroups in the cgroup filesystem. Consider for example, the following cgroup filesystem hierarchy

```
                root
              /  |   \
             /   |    \
          a      b       c
                         | \
                         |  \
                         d   e
```

In the diagram above, with hierarchical accounting enabled, all memory usage of e, is accounted to its ancestors up until the root (i.e, c and root), that has memory.use_hierarchy enabled. If one of the ancestors goes over its limit, the reclaim algorithm reclaims from the tasks in the ancestor and the children of the ancestor.

6.1 Enabling hierarchical accounting and reclaim

A memory cgroup by default disables the hierarchy feature. Support can be enabled by writing 1 to memory.use_hierarchy file of the root cgroup

echo 1 > memory.use_hierarchy

The feature can be disabled by

echo 0 > memory.use_hierarchy

NOTE1: Enabling/disabling will fail if the cgroup already has other
       cgroups created below it.

NOTE2: When panic_on_oom is set to "2", the whole system will panic in
       case of an OOM event in any cgroup.

7. Soft limits

Soft limits allow for greater sharing of memory. The idea behind soft limits is to allow control groups to use as much of the memory as needed, provided

a. There is no memory contention
b. They do not exceed their hard limit

When the system detects memory contention or low memory, control groups are pushed back to their soft limits. If the soft limit of each control group is very high, they are pushed back as much as possible to make sure that one control group does not starve the others of memory.

Please note that soft limits is a best effort feature, it comes with no guarantees, but it does its best to make sure that when memory is heavily contended for, memory is allocated based on the soft limit hints/setup. Currently soft limit based reclaim is setup such that it gets invoked from balance_pgdat (kswapd).

7.1 Interface

Soft limits can be setup by using the following commands (in this example we assume a soft limit of 256 MiB)

```
# echo 256M > memory.soft_limit_in_bytes
```

If we want to change this to 1G, we can at any time use

```
# echo 1G > memory.soft_limit_in_bytes
```

```
NOTE1: Soft limits take effect over a long period of time, since they involve
       reclaiming memory for balancing between memory cgroups
NOTE2: It is recommended to set the soft limit always below the hard limit,
       otherwise the hard limit will take precedence.
```

8. Move charges at task migration

Users can move charges associated with a task along with task migration, that is, uncharge task's pages from the old cgroup and charge them to the new cgroup. This feature is not supported in !CONFIG_MMU environments because of lack of page tables.

8.1 Interface

This feature is disabled by default. It can be enabled(and disabled again) by writing to memory.move_charge_at_immigrate of the destination cgroup.

If you want to enable it:

```
# echo (some positive value) > memory.move_charge_at_immigrate
```

```
Note: Each bits of move_charge_at_immigrate has its own meaning about what type
      of charges should be moved. See 8.2 for details.
Note: Charges are moved only when you move mm->owner, IOW, a leader of a thread
      group.
Note: If we cannot find enough space for the task in the destination cgroup, we
      try to make space by reclaiming memory. Task migration may fail if we
      cannot make enough space.
Note: It can take several seconds if you move charges much.
```

And if you want disable it again:

```
# echo 0 > memory.move_charge_at_immigrate
```

8.2 Type of charges which can be move

Each bits of move_charge_at_immigrate has its own meaning about what type of charges should be moved. But in any cases, it must be noted that an account of a page or a swap can be moved only when it is charged to the task's current(old) memory cgroup.

```
  bit | what type of charges would be moved ?
 -----+------------------------------------------------------------------------
   0  | A charge of an anonymous page(or swap of it) used by the target task.
      | Those pages and swaps must be used only by the target task. You must
      | enable Swap Extension(see 2.4) to enable move of swap charges.
 -----+------------------------------------------------------------------------
   1  | A charge of file pages(normal file, tmpfs file(e.g. ipc shared memory)
      | and swaps of tmpfs file) mmapped by the target task. Unlike the case of
      | anonymous pages, file pages(and swaps) in the range mmapped by the task
      | will be moved even if the task hasn't done page fault, i.e. they might
      | not be the task's "RSS", but other task's "RSS" that maps the same file.
      | And mapcount of the page is ignored(the page can be moved even if
      | page_mapcount(page) > 1). You must enable Swap Extension(see 2.4) to
      | enable move of swap charges.
```

8.3 TODO

- Implement madvise(2) to let users decide the vma to be moved or not to be moved.
- All of moving charge operations are done under cgroup_mutex. It's not good behavior to hold the mutex too long, so we may need some trick.

9. Memory thresholds

Memory cgroup implements memory thresholds using cgroups notification API (see cgroups.txt). It allows to register multiple memory and memsw thresholds and gets notifications when it crosses.

To register a threshold application need:
- create an eventfd using eventfd(2);
- open memory.usage_in_bytes or memory.memsw.usage_in_bytes;
- write string like "<event_fd> <fd of memory.usage_in_bytes> <threshold>" to cgroup.event_control.

Application will be notified through eventfd when memory usage crosses threshold in any direction.

It's applicable for root and non-root cgroup.

10. OOM Control

memory.oom_control file is for OOM notification and other controls.

Memory cgroup implements OOM notifier using cgroup notification API (See cgroups.txt). It allows to register multiple OOM notification delivery and gets notification when OOM happens.

To register a notifier, application need:
- create an eventfd using eventfd(2)
- open memory.oom_control file
- write string like "<event_fd> <fd of memory.oom_control>" to

   cgroup.event_control

Application will be notified through eventfd when OOM happens.
OOM notification doesn't work for root cgroup.

You can disable OOM-killer by writing "1" to memory.oom_control file, as:

```
#echo 1 > memory.oom_control
```

This operation is only allowed to the top cgroup of sub-hierarchy.
If OOM-killer is disabled, tasks under cgroup will hang/sleep
in memory cgroup's OOM-waitqueue when they request accountable memory.

For running them, you have to relax the memory cgroup's OOM status by
	* enlarge limit or reduce usage.
To reduce usage,
	* kill some tasks.
	* move some tasks to other group with account migration.
	* remove some files (on tmpfs?)

Then, stopped tasks will work again.

At reading, current status of OOM is shown.
```
	oom_kill_disable 0 or 1 (if 1, oom-killer is disabled)
	under_oom	 0 or 1 (if 1, the memory cgroup is under OOM, tasks may
				 be stopped.)
```

11. TODO

1. Add support for accounting huge pages (as a separate controller)
2. Make per-cgroup scanner reclaim not-shared pages first
3. Teach controller to account for shared-pages
4. Start reclamation in the background when the limit is
   not yet hit but the usage is getting closer

Summary

Overall, the memory controller has been a stable controller and has been
commented and discussed quite extensively in the community.

References

1. Singh, Balbir. RFC: Memory Controller, http://lwn.net/Articles/206697/
2. Singh, Balbir. Memory Controller (RSS Control),
   http://lwn.net/Articles/222762/
3. Emelianov, Pavel. Resource controllers based on process cgroups
   http://lkml.org/lkml/2007/3/6/198
4. Emelianov, Pavel. RSS controller based on process cgroups (v2)
   http://lkml.org/lkml/2007/4/9/78
5. Emelianov, Pavel. RSS controller based on process cgroups (v3)
   http://lkml.org/lkml/2007/5/30/244
6. Menage, Paul. Control Groups v10, http://lwn.net/Articles/236032/
7. Vaidyanathan, Srinivasan, Control Groups: Pagecache accounting and control
   subsystem (v3), http://lwn.net/Articles/235534/
8. Singh, Balbir. RSS controller v2 test results (lmbench),
   http://lkml.org/lkml/2007/5/17/232

9. Singh, Balbir. RSS controller v2 AIM9 results
http://lkml.org/lkml/2007/5/18/1
10. Singh, Balbir. Memory controller v6 test results,
http://lkml.org/lkml/2007/8/19/36
11. Singh, Balbir. Memory controller introduction (v6),
http://lkml.org/lkml/2007/8/17/69
12. Corbet, Jonathan, Controlling memory use in cgroups,
http://lwn.net/Articles/243795/