Memory Resource Controller

NOTE:
: This document is hopelessly outdated and it asks for a complete rewrite. It still contains a useful information so we are keeping it here but make sure to check the current code if you need a deeper understanding.

NOTE:
: The Memory Resource Controller has 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 a 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.
- pages are linked to per-memcg LRU exclusively, and there is no global LRU.
- 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
- memory pressure notifier
- oom-killer disable knob and oom-notifier
- Root cgroup has no limit controls.

Kernel memory support is a work in progress, and the current version provides basically functionality. (See Section 2.7)

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() This knob is not available on CONFIG_PREEMPT_RT systems.
memory.usage_in_bytes	show current usage for memory (See 5.5 for details)
memory.memsw.usage_in_bytes	show current usage for memory+Swap (See 5.5 for details)
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.max_usage_in_bytes	show max memory+Swap usage recorded
memory.soft_limit_in_bytes	set/show soft limit of memory usage This knob is not available on CONFIG_PREEMPT_RT systems.
memory.stat	show various statistics
memory.use_hierarchy	set/show hierarchical account enabled This knob is deprecated and shouldn't be used.
memory.force_empty	trigger forced page reclaim
memory.pressure_level	set memory pressure notifications
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.
memory.numa_stat	show the number of memory usage per numa node
memory.kmem.limit_in_bytes	This knob is deprecated and writing to it will return -ENOTSUPP.
memory.kmem.usage_in_bytes	show current kernel memory allocation
memory.kmem.failcnt	show the number of kernel memory usage hits limits
memory.kmem.max_usage_in_bytes	show max kernel memory usage recorded
memory.kmem.tcp.limit_in_bytes	set/show hard limit for tcp buf memory
memory.kmem.tcp.usage_in_bytes	show current tcp buf memory allocation
memory.kmem.tcp.failcnt	show the number of tcp buf memory usage hits limits
memory.kmem.tcp.max_usage_in_bytes	show max tcp buf memory usage recorded

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 page_counter. The page_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        |
        |  (page_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_common() is invoked to set up 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 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.

An 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 are also accounted. A swapped-in page is accounted after adding into swapcache.

Note: The kernel does swapin-readahead and reads multiple swaps at once. Since page's memcg recorded into swap whatever memsw enabled, the page will be accounted after swapin.

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).

But see section 8.2: when moving a task to another cgroup, its pages may be recharged to the new cgroup, if move_charge_at_immigrate has been chosen.

2.4 Swap Extension

Swap usage is always recorded for each of cgroup. Swap Extension allows you to read and limit it.

When CONFIG_SWAP is enabled, 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 the 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 an OS point of view.

What happens when a cgroup hits memory.memsw.limit_in_bytes

When a cgroup hits 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 order is as follows:

```
    Page lock (PG_locked bit of page->flags)
        mm->page_table_lock or split pte_lock
            lock_page_memcg (memcg->move_lock)
                mapping->i_pages lock
                    lruvec->lru_lock.
```

Per-node-per-memcgroup LRU (cgroup's private LRU) is guarded by lruvec->lru_lock; PG_lru bit of page->flags is cleared before isolating a page from its LRU under lruvec->lru_lock.

2.7 Kernel Memory Extension (CONFIG_MEMCG_KMEM)

With the Kernel memory extension, the Memory Controller is able to limit the amount of kernel memory used by the system. Kernel memory is fundamentally different than user memory, since it can't be swapped out, which makes it possible to DoS the system by consuming too much of this precious resource.

Kernel memory accounting is enabled for all memory cgroups by default. But it can be disabled system-wide by passing cgroup.memory=nokmem to the kernel at boot time. In this case, kernel memory will not be accounted at all.

Kernel memory limits are not imposed for the root cgroup. Usage for the root cgroup may or may not be accounted. The memory used is accumulated into memory.kmem.usage_in_bytes, or in a separate counter when it makes sense. (currently only for tcp).

The main "kmem" counter is fed into the main counter, so kmem charges will also be visible from the user counter.

Currently no soft limit is implemented for kernel memory. It is future work to trigger slab reclaim when those limits are reached.

2.7.1 Current Kernel Memory resources accounted

stack pages:
: every process consumes some stack pages. By accounting into kernel memory, we prevent new processes from being created when the kernel memory usage is too high.

slab pages:
: pages allocated by the SLAB or SLUB allocator are tracked. A copy of each kmem_cache is created every time the cache is touched by the first time from inside the memcg. The creation is done lazily, so some objects can still be skipped while the cache is being created. All objects in a slab page should belong to the same memcg. This only fails to hold when a task is migrated to a different memcg during the page allocation by the cache.

sockets memory pressure:
: some sockets protocols have memory pressure thresholds. The Memory Controller allows them to be controlled individually per cgroup, instead of globally.

tcp memory pressure:
: sockets memory pressure for the tcp protocol.

2.7.2 Common use cases

Because the "kmem" counter is fed to the main user counter, kernel memory can never be limited completely independently of user memory. Say "U" is the user limit, and "K" the kernel limit. There are three possible ways limits can be set:

U != 0, K = unlimited:
: This is the standard memcg limitation mechanism already present before kmem accounting. Kernel memory is completely ignored.

U != 0, K < U:
: Kernel memory is a subset of the user memory. This setup is useful in deployments where the total amount of memory per-cgroup is overcommitted. Overcommitting kernel memory limits is definitely not recommended, since the box can still run out of non-reclaimable memory. In this case, the admin could set up K so that the sum of all groups is never greater than the total memory, and freely set U at the cost of his QoS.

WARNING:
: In the current implementation, memory reclaim will NOT be triggered for a cgroup when it hits K while staying below U, which makes this setup impractical.

U != 0, K >= U:
: Since kmem charges will also be fed to the user counter and reclaim will be triggered for the cgroup for both kinds of memory. This setup gives the admin a unified view of memory, and it is also useful for people who just want to track kernel memory usage.

3. User Interface

3.0. Configuration

a. Enable CONFIG_CGROUPS
b. Enable CONFIG_MEMCG
c. Enable CONFIG_MEMCG_SWAP (to use swap extension)
d. Enable CONFIG_MEMCG_KMEM (to use kmem extension)

3.1. Prepare the cgroups (see cgroups.txt, Why are cgroups needed?)

```
# mount -t tmpfs none /sys/fs/cgroup
# mkdir /sys/fs/cgroup/memory
# mount -t cgroup none /sys/fs/cgroup/memory -o memory
```

3.2. Make the new group and move bash into it:

```
# mkdir /sys/fs/cgroup/memory/0
# echo $$ > /sys/fs/cgroup/memory/0/tasks
```

Since now we're in the 0 cgroup, we can alter the memory limit:

```
# echo 4M > /sys/fs/cgroup/memory/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 /sys/fs/cgroup/memory/0/memory.limit_in_bytes
4194304
```

We can check the usage:

```
# cat /sys/fs/cgroup/memory/0/memory.usage_in_bytes
1216512
```

A successful write to this file does not guarantee a successful setting 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 the 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, disabling OOM_Kill as per "10. OOM Control" (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.)

We move the stats to parent, and no change on the charge except uncharging from the child.

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. When writing anything to this:

```
# echo 0 > memory.force_empty
```

the cgroup will be reclaimed and as many pages reclaimed as possible.

The typical use case for this interface is before calling rmdir(). Though rmdir() offlines memcg, but the memcg may still stay there due to charged file caches. Some out-of-use page caches may keep charged until memory pressure happens. 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 (includes transparent hugepages).
rss_huge	# of bytes of anonymous transparent hugepages.
mapped_file	# of bytes of mapped file (includes tmpfs/shmem)
pgpgin	# of charging events to the memory cgroup. The charging event happens each time a page is accounted as either mapped anon page(RSS) or cache page(Page Cache) to the cgroup.
pgpgout	# of uncharging events to the memory cgroup. The uncharging event happens each time a page is unaccounted from the cgroup.
swap	# of bytes of swap usage
dirty	# of bytes that are waiting to get written back to the disk.
writeback	# of bytes of file/anon cache that are queued for syncing to disk.
inactive_anon	# of bytes of anonymous and swap cache memory on inactive LRU list.
active_anon	# of bytes of anonymous and swap cache memory on active 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_<counter>	# hierarchical version of <counter>, which in addition to the cgroup's own value includes the sum of all hierarchical children's values of <counter>, i.e. total_cache

The following additional stats are dependent on CONFIG_DEBUG_VM

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 + mapped_file" will give you resident set size of cgroup.

(Note: file and shmem may be shared among other cgroups. In that case, mapped_file is accounted only when the memory cgroup is owner of page cache.)

5.3 swappiness

Overrides /proc/sys/vm/swappiness for the particular group. The tunable in the root cgroup corresponds to the global swappiness setting.

Please note that unlike during the global reclaim, limit reclaim enforces that 0 swappiness really prevents from any swapping even if there is a swap storage available. This might lead to memcg OOM killer if there are no file pages to reclaim.

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
```

5.5 usage_in_bytes

For efficiency, as other kernel components, memory cgroup uses some optimization to avoid unnecessary cacheline false sharing. usage_in_bytes is affected by the method and doesn't show 'exact' value of memory (and swap) usage, it's a fuzz value for efficient access. (Of course, when necessary, it's synchronized.) If you want to know more exact memory usage, you should use RSS+CACHE(+SWAP) value in memory.stat(see 5.2).

5.6 numa_stat

This is similar to numa_maps but operates on a per-memcg basis. This is useful for providing visibility into the numa locality information within an memcg since the pages are allowed to be allocated from any physical node. One of the use cases is evaluating application performance by combining this information with the application's CPU allocation.

Each memcg's numa_stat file includes "total", "file", "anon" and "unevictable" per-node page counts including "hierarchical_<counter>" which sums up all hierarchical children's values in addition to the memcg's own value.

The output format of memory.numa_stat is:

```
total=<total pages> N0=<node 0 pages> N1=<node 1 pages> ...
file=<total file pages> N0=<node 0 pages> N1=<node 1 pages> ...
anon=<total anon pages> N0=<node 0 pages> N1=<node 1 pages> ...
unevictable=<total anon pages> N0=<node 0 pages> N1=<node 1 pages> ...
hierarchical_<counter>=<counter pages> N0=<node 0 pages> N1=<node 1 pages> ...
```

The "total" count is sum of file + anon + unevictable.

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). 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 Hierarchical accounting and reclaim

Hierarchical accounting is enabled by default. Disabling the hierarchical accounting is deprecated. An attempt to do it will result in a failure and a warning printed to dmesg.

For compatibility reasons writing 1 to memory.use_hierarchy will always pass:

```
# echo 1 > memory.use_hierarchy
```

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 set up 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, in other words, 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 moved

Each bit in move_charge_at_immigrate has its own meaning about what type of charges should be moved. But in any case, 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 ?

bit	what type of charges would be moved ?
0	A charge of an anonymous page (or swap of it) used 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

- 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 the 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, an application must:

- 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 the cgroup notification API (See cgroups.txt). It allows to register multiple OOM notification delivery and gets notification when OOM happens.

To register a notifier, an application must:

- 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

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

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

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

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.)
- oom_kill integer counter The number of processes belonging to this cgroup killed by any kind of OOM killer.

11. Memory Pressure

The pressure level notifications can be used to monitor the memory allocation cost; based on the pressure, applications can implement different strategies of managing their memory resources. The pressure levels are defined as following:

The "low" level means that the system is reclaiming memory for new allocations. Monitoring this reclaiming activity might be useful for maintaining cache level. Upon notification, the program (typically "Activity Manager") might analyze vmstat and act in advance (i.e. prematurely shutdown unimportant services).

The "medium" level means that the system is experiencing medium memory pressure, the system might be making swap, paging out active file caches, etc. Upon this event applications may decide to further analyze vmstat/zoneinfo/memcg or internal memory usage statistics and free any resources that can be easily reconstructed or re-read from a disk.

The "critical" level means that the system is actively thrashing, it is about to out of memory (OOM) or even the in-kernel OOM killer is on its way to trigger. Applications should do whatever they can to help the system. It might be too late to consult with vmstat or any other statistics, so it's advisable to take an immediate action.

By default, events are propagated upward until the event is handled, i.e. the events are not pass-through. For example, you have three cgroups: A->B->C. Now you set up an event listener on cgroups A, B and C, and suppose group C experiences some pressure. In this situation, only group C will receive the notification, i.e. groups A and B will not receive it. This is done to avoid excessive "broadcasting" of messages, which disturbs the system and which is especially bad if we are low on memory or thrashing. Group B, will receive notification only if there are no event listers for group C.

There are three optional modes that specify different propagation behavior:

- "default": this is the default behavior specified above. This mode is the same as omitting the optional mode parameter, preserved by backwards compatibility.
- "hierarchy": events always propagate up to the root, similar to the default behavior, except that propagation continues regardless of whether there are event listeners at each level, with the "hierarchy" mode. In the above example, groups A, B, and C will receive notification of memory pressure.
- "local": events are pass-through, i.e. they only receive notifications when memory pressure is experienced in the memcg for which the notification is registered. In the above example, group C will receive notification if registered for "local" notification and the group experiences memory pressure. However, group B will never receive notification, regardless if there is an event listener for group C or not, if group B is registered for local notification.

The level and event notification mode ("hierarchy" or "local", if necessary) are specified by a comma-delimited string, i.e. "low,hierarchy" specifies hierarchical, pass-through, notification for all ancestor memcgs. Notification that is the default, non pass-through behavior, does not specify a mode. "medium,local" specifies pass-through notification for the medium level.

The file memory.pressure_level is only used to setup an eventfd. To register a notification, an application must:

- create an eventfd using eventfd(2);
- open memory.pressure_level;
- write string as "<event_fd> <fd of memory.pressure_level> <level[,mode]>" to cgroup.event_control.

Application will be notified through eventfd when memory pressure is at the specific level (or higher). Read/write operations to memory.pressure_level are no implemented.

Test:

Here is a small script example that makes a new cgroup, sets up a memory limit, sets up a notification in the cgroup and then makes child cgroup experience a critical pressure:

```
# cd /sys/fs/cgroup/memory/
# mkdir foo
# cd foo
# cgroup_event_listener memory.pressure_level low,hierarchy &
# echo 8000000 > memory.limit_in_bytes
# echo 8000000 > memory.memsw.limit_in_bytes
# echo $$ > tasks
# dd if=/dev/zero | read x
```

(Expect a bunch of notifications, and eventually, the oom-killer will trigger.)

12. TODO

1. Make per-cgroup scanner reclaim not-shared pages first
2. Teach controller to account for shared-pages
3. 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 https://lore.kernel.org/r/45ED7DEC.7010403@sw.ru
4. Emelianov, Pavel. RSS controller based on process cgroups (v2) https://lore.kernel.org/r/461A3010.90403@sw.ru
5. Emelianov, Pavel. RSS controller based on process cgroups (v3) https://lore.kernel.org/r/465D9739.8070209@openvz.org
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), https://lore.kernel.org/r/464C95D4.7070806@linux.vnet.ibm.com
9. Singh, Balbir. RSS controller v2 AIM9 results https://lore.kernel.org/r/464D267A.50107@linux.vnet.ibm.com
10. Singh, Balbir. Memory controller v6 test results, https://lore.kernel.org/r/20070819094658.654.84837.sendpatchset@balbir-laptop
11. Singh, Balbir. Memory controller introduction (v6), https://lore.kernel.org/r/20070817084228.26003.12568.sendpatchset@balbir-laptop
12. Corbet, Jonathan, Controlling memory use in cgroups, http://lwn.net/Articles/243795/