Subject: [PATCH v5 00/10] per-cgroup tcp memory pressure
Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:25 GMT
View Forum Message <> Reply to Message

Hi all,

This is my new attempt at implementing per-cgroup tcp memory pressure.
I am particularly interested in what the network folks have to comment on
it: my main goal is to achieve the least impact possible in the network code.

Here's a brief description of my approach:

When only the root cgroup is present, the code should behave the same way as
before - with the exception of the inclusion of an extra field in struct sock,
and one in struct proto. All tests are patched out with static branch, and we
still access addresses directly - the same as we did before.

When a cgroup other than root is created, we patch in the branches, and account
resources for that cgroup. The variables in the root cgroup are still updated.
If we were to try to be 100 % coherent with the memcg code, that should depend
on use_hierarchy. However, I feel that this is a good compromise in terms of
leaving the network code untouched, and still having a global vision of its
resources. I also do not compute max_usage for the root cgroup, for a similar
reason.

Please let me know what you think of it.

Glauber Costa (10):
  Basic kernel memory functionality for the Memory Controller
  foundations of per-cgroup memory pressure controlling.
  socket: initial cgroup code.
  per-cgroup tcp buffers control
  per-netns ipv4 sysctl_tcp_mem
  tcp buffer limitation: per-cgroup limit
  Display current tcp memory allocation in kmem cgroup
  Display current tcp memory allocation in kmem cgroup
  Display current tcp memory allocation in kmem cgroup
  Disable task moving when using kernel memory accounting

 Documentation/cgroups/memory.txt | 38 +++-
 include/linux/memcontrol.h | 40 ++++
 include/net/netns/ipv4.h | 1 +
 include/net/sock.h | 231 ++++++++++++++++++++-
 include/net/tcp.h | 22 ++-
 include/net/transp_v6.h | 1 +
 init/Kconfig | 14 ++
 mm/memcontrol.c | 442 ++
 net/core/sock.c | 121 ++++++---

```
 net/ipv4/af_inet.c             |    5 +
 net/ipv4/proc.c                |    7 +-
 net/ipv4/sysctl_net_ipv4.c     |   65 +++++-
 net/ipv4/tcp.c                 |   11 +-
 net/ipv4/tcp_input.c           |   12 +-
 net/ipv4/tcp_ipv4.c            |   17 ++-
 net/ipv4/tcp_output.c          |    2 +-
 net/ipv4/tcp_timer.c           |    2 +-
 net/ipv6/af_inet6.c            |    5 +
 net/ipv6/tcp_ipv6.c            |   13 +-
 19 files changed, 962 insertions(+), 87 deletions(-)

--
1.7.6.4
```

Subject: [PATCH v5 01/10] Basic kernel memory functionality for the Memory Controller

Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:26 GMT

View Forum Message <> Reply to Message

This patch lays down the foundation for the kernel memory component
of the Memory Controller.

As of today, I am only laying down the following files:

 * memory.independent_kmem_limit
 * memory.kmem.limit_in_bytes (currently ignored)
 * memory.kmem.usage_in_bytes (always zero)

Signed-off-by: Glauber Costa <glommer@parallels.com>
Reviewed-by: Kirill A. Shutemov <kirill@shutemov.name>
CC: Paul Menage <paul@paulmenage.org>
CC: Greg Thelen <gthelen@google.com>

```
 Documentation/cgroups/memory.txt |   36 ++++++++++++-
 init/Kconfig                     |   14 +++++
 mm/memcontrol.c                  |  104 ++++++++++++++++++++++++++++++++++++--
 3 files changed, 147 insertions(+), 7 deletions(-)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index 06eb6d9..bf00cd2 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -44,8 +44,9 @@ Features:
  - 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.
+ Hugepages is not under control yet. We just manage pages on LRU. To add more
+ controls, we have to take care of performance. Kernel memory support is work
+ in progress, and the current version provides basically functionality.

 Brief summary of control files.

@@ -56,8 +57,11 @@ Brief summary of control files.
      (See 5.5 for details)
  memory.memsw.usage_in_bytes  # show current res_counter usage for memory+Swap
      (See 5.5 for details)
+ memory.kmem.usage_in_bytes  # show current res_counter usage for kmem only.
+     (See 2.7 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.kmem.limit_in_bytes  # if allowed, set/show limit of kernel memory
  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
@@ -72,6 +76,9 @@ Brief summary of control files.
  memory.oom_control   # set/show oom controls.
  memory.numa_stat   # show the number of memory usage per numa node

+ memory.independent_kmem_limit  # select whether or not kernel memory limits are
+       independent of user limits
+
 1. History

 The memory controller has a long history. A request for comments for the memory
@@ -255,6 +262,31 @@ When oom event notifier is registered, event will be delivered.
   per-zone-per-cgroup LRU (cgroup's private LRU) is just guarded by
   zone->lru_lock, it has no lock of its own.

+2.7 Kernel Memory Extension (CONFIG_CGROUP_MEM_RES_CTLR_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 limits are not imposed for the root cgroup.
+
+Memory limits as specified by the standard Memory Controller may or may not
+take kernel memory into consideration. This is achieved through the file
+memory.independent_kmem_limit. A Value different than 0 will allow for kernel
+memory to be controlled separately.
+
+When kernel memory limits are not independent, the limit values set in
```

```
+memory.kmem files are ignored.
+
+Currently no soft limit is implemented for kernel memory. It is future work
+to trigger slab reclaim when those limits are reached.
+
+CAUTION: As of this writing, the kmem extention may prevent tasks from moving
+among cgroups. If a task has kmem accounting in a cgroup, the task cannot be
+moved until the kmem resource is released. Also, until the resource is fully
+released, the cgroup cannot be destroyed. So, please consider your use cases
+and set kmem extention config option carefully.
+
 3. User Interface

 0. Configuration
diff --git a/init/Kconfig b/init/Kconfig
index 31ba0fd..e4b6246 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -689,6 +689,20 @@ config CGROUP_MEM_RES_CTLR_SWAP_ENABLED
    For those who want to have the feature enabled by default should
    select this option (if, for some reason, they need to disable it
    then swapaccount=0 does the trick).
+config CGROUP_MEM_RES_CTLR_KMEM
+ bool "Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)"
+ depends on CGROUP_MEM_RES_CTLR && EXPERIMENTAL
+ default n
+ help
+   The Kernel Memory extension for Memory Resource Controller can limit
+   the amount of memory used by kernel objects in the system. Those are
+   fundamentally different from the entities handled by the standard
+   Memory Controller, which are page-based, and can be swapped. Users of
+   the kmem extension can use it to guarantee that no group of processes
+   will ever exhaust kernel resources alone.
+
+   WARNING: The current experimental implementation does not allow a
+   task to move among different cgroups with a kmem resource being held.

 config CGROUP_PERF
   bool "Enable perf_event per-cpu per-container group (cgroup) monitoring"
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 2d57555..3389d33 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -226,6 +226,10 @@ struct mem_cgroup {
    */
   struct res_counter memsw;
   /*
+  * the counter to account for kmem usage.
```

```
+  */
+ struct res_counter kmem;
+ /*
   * Per cgroup active and inactive list, similar to the
   * per zone LRU lists.
   */
@@ -276,6 +280,11 @@ struct mem_cgroup {
   */
  unsigned long  move_charge_at_immigrate;
  /*
+  * Should kernel memory limits be stabilished independently
+  * from user memory ?
+  */
+ int  kmem_independent_accounting;
+ /*
   * percpu counter.
   */
  struct mem_cgroup_stat_cpu *stat;
@@ -343,9 +352,14 @@ enum charge_type {
 };

 /* for encoding cft->private value on file */
-#define _MEM   (0)
-#define _MEMSWAP  (1)
-#define _OOM_TYPE  (2)
+
+enum mem_type {
+ _MEM = 0,
+ _MEMSWAP,
+ _OOM_TYPE,
+ _KMEM,
+};
+
 #define MEMFILE_PRIVATE(x, val) (((x) << 16) | (val))
 #define MEMFILE_TYPE(val) (((val) >> 16) & 0xffff)
 #define MEMFILE_ATTR(val) ((val) & 0xffff)
@@ -3838,10 +3852,15 @@ static inline u64 mem_cgroup_usage(struct mem_cgroup *mem,
bool swap)
  u64 val;

  if (!mem_cgroup_is_root(mem)) {
+  val = 0;
+  if (!mem->kmem_independent_accounting)
+   val = res_counter_read_u64(&mem->kmem, RES_USAGE);
   if (!swap)
-   return res_counter_read_u64(&mem->res, RES_USAGE);
+   val += res_counter_read_u64(&mem->res, RES_USAGE);
   else
```

```
-   return res_counter_read_u64(&mem->memsw, RES_USAGE);
+    val += res_counter_read_u64(&mem->memsw, RES_USAGE);
+
+   return val;
   }

   val = mem_cgroup_recursive_stat(mem, MEM_CGROUP_STAT_CACHE);
@@ -3874,6 +3893,10 @@ static u64 mem_cgroup_read(struct cgroup *cont, struct cftype *cft)
    else
     val = res_counter_read_u64(&mem->memsw, name);
    break;
+ case _KMEM:
+   val = res_counter_read_u64(&mem->kmem, name);
+   break;
+
   default:
    BUG();
    break;
@@ -4604,6 +4627,35 @@ static int mem_control_numa_stat_open(struct inode *unused, struct
file *file)
 }
 #endif /* CONFIG_NUMA */

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static u64 kmem_limit_independent_read(struct cgroup *cgroup, struct cftype *cft)
+{
+ return mem_cgroup_from_cont(cgroup)->kmem_independent_accounting;
+}
+
+static int kmem_limit_independent_write(struct cgroup *cgroup, struct cftype *cft,
+     u64 val)
+{
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cgroup);
+ struct mem_cgroup *parent = parent_mem_cgroup(memcg);
+
+ val = !!val;
+
+ if (parent && parent->use_hierarchy &&
+     (val != parent->kmem_independent_accounting))
+   return -EINVAL;
+ /*
+  * TODO: We need to handle the case in which we are doing
+  * independent kmem accounting as authorized by our parent,
+  * but then our parent changes its parameter.
+  */
+ cgroup_lock();
+ memcg->kmem_independent_accounting = val;
+ cgroup_unlock();
```

```
+ return 0;
+}
+#endif
+
 static struct cftype mem_cgroup_files[] = {
  {
   .name = "usage_in_bytes",
@@ -4719,6 +4771,42 @@ static int register_memsw_files(struct cgroup *cont, struct
cgroup_subsys *ss)
 }
 #endif

+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static struct cftype kmem_cgroup_files[] = {
+ {
+  .name = "independent_kmem_limit",
+  .read_u64 = kmem_limit_independent_read,
+  .write_u64 = kmem_limit_independent_write,
+ },
+ {
+  .name = "kmem.usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
+  .read_u64 = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.limit_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
+  .read_u64 = mem_cgroup_read,
+ },
+};
+
+static int register_kmem_files(struct cgroup *cont, struct cgroup_subsys *ss)
+{
+ int ret = 0;
+
+ ret = cgroup_add_files(cont, ss, kmem_cgroup_files,
+         ARRAY_SIZE(kmem_cgroup_files));
+ return ret;
+};
+
+#else
+static int register_kmem_files(struct cgroup *cont, struct cgroup_subsys *ss)
+{
+ return 0;
+}
+#endif
+
```

```
 static int alloc_mem_cgroup_per_zone_info(struct mem_cgroup *mem, int node)
 {
  struct mem_cgroup_per_node *pn;
@@ -4917,6 +5005,7 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
  if (parent && parent->use_hierarchy) {
   res_counter_init(&mem->res, &parent->res);
   res_counter_init(&mem->memsw, &parent->memsw);
+  res_counter_init(&mem->kmem, &parent->kmem);
   /*
    * We increment refcnt of the parent to ensure that we can
    * safely access it on res_counter_charge/uncharge.
@@ -4927,6 +5016,7 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
  } else {
   res_counter_init(&mem->res, NULL);
   res_counter_init(&mem->memsw, NULL);
+  res_counter_init(&mem->kmem, NULL);
  }
  mem->last_scanned_child = 0;
  mem->last_scanned_node = MAX_NUMNODES;
@@ -4970,6 +5060,10 @@ static int mem_cgroup_populate(struct cgroup_subsys *ss,

  if (!ret)
   ret = register_memsw_files(cont, ss);
+
+ if (!ret)
+  ret = register_kmem_files(cont, ss);
+
  return ret;
 }

--
1.7.6.4
```

Subject: [PATCH v5 02/10] foundations of per-cgroup memory pressure controlling.
Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:27 GMT
View Forum Message <> Reply to Message

This patch replaces all uses of struct sock fields' memory_pressure,
memory_allocated, sockets_allocated, and sysctl_mem to acessor
macros. Those macros can either receive a socket argument, or a mem_cgroup
argument, depending on the context they live in.

Since we're only doing a macro wrapping here, no performance impact at all is
expected in the case where we don't have cgroups disabled.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: David S. Miller <davem@davemloft.net>

```
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Eric W. Biederman <ebiederm@xmission.com>
CC: Eric Dumazet <eric.dumazet@gmail.com>
---
 include/linux/memcontrol.h |    4 ++
 include/net/sock.h         |   86 ++++++++++++++++++++++++++++++++++++++-----
 include/net/tcp.h          |    3 +-
 net/core/sock.c            |   55 ++++++++++++++++++-----------
 net/ipv4/proc.c            |    7 ++--
 net/ipv4/tcp_input.c       |   12 +++---
 net/ipv4/tcp_ipv4.c        |    4 +-
 net/ipv4/tcp_output.c      |    2 +-
 net/ipv4/tcp_timer.c       |    2 +-
 net/ipv6/tcp_ipv6.c        |    2 +-
 10 files changed, 130 insertions(+), 47 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index ac797fa..e9ff93a 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -362,6 +362,10 @@ static inline
 void mem_cgroup_count_vm_event(struct mm_struct *mm, enum vm_event_item idx)
 {
 }
+static inline struct mem_cgroup *mem_cgroup_from_task(struct task_struct *p)
+{
+ return NULL;
+}
 #endif /* CONFIG_CGROUP_MEM_CONT */

 #if !defined(CONFIG_CGROUP_MEM_RES_CTLR) || !defined(CONFIG_DEBUG_VM)
diff --git a/include/net/sock.h b/include/net/sock.h
index c6658be..8959dcc 100644
--- a/include/net/sock.h
+++ b/include/net/sock.h
@@ -54,6 +54,7 @@
 #include <linux/security.h>
 #include <linux/slab.h>
 #include <linux/uaccess.h>
+#include <linux/cgroup.h>

 #include <linux/filter.h>
 #include <linux/rculist_nulls.h>
@@ -168,6 +169,8 @@ struct sock_common {
   /* public: */
 };

+struct mem_cgroup;
```

```
+
 /**
   * struct sock - network layer representation of sockets
   * @__sk_common: shared layout with inet_timewait_sock
@@ -793,22 +796,21 @@ struct proto {
  unsigned int  inuse_idx;
 #endif

- /* Memory pressure */
- void   (*enter_memory_pressure)(struct sock *sk);
- atomic_long_t  *memory_allocated; /* Current allocated memory. */
- struct percpu_counter *sockets_allocated; /* Current number of sockets. */
+ void                   (*enter_memory_pressure)(struct sock *sk);
+ atomic_long_t          *memory_allocated;      /* Current allocated memory. */
+ struct percpu_counter   *sockets_allocated;     /* Current number of sockets. */
  /*
   * Pressure flag: try to collapse.
   * Technical note: it is used by multiple contexts non atomically.
   * All the __sk_mem_schedule() is of this nature: accounting
   * is strict, actions are advisory and have some latency.
   */
- int   *memory_pressure;
- long   *sysctl_mem;
- int   *sysctl_wmem;
- int   *sysctl_rmem;
- int   max_header;
- bool   no_autobind;
+ int                    *memory_pressure;
+ long                    *sysctl_mem;
+ int                    *sysctl_wmem;
+ int                    *sysctl_rmem;
+ int                    max_header;
+ bool                    no_autobind;

  struct kmem_cache *slab;
  unsigned int  obj_size;
@@ -863,6 +865,70 @@ static inline void sk_refcnt_debug_release(const struct sock *sk)
 #define sk_refcnt_debug_release(sk) do { } while (0)
 #endif /* SOCK_REFCNT_DEBUG */

+#include <linux/memcontrol.h>
+static inline int *sk_memory_pressure(const struct sock *sk)
+{
+ return sk->sk_prot->memory_pressure;
+}
+
+static inline long sk_prot_mem(const struct sock *sk, int index)
+{
```

```
+ long *prot = sk->sk_prot->sysctl_mem;
+ return prot[index];
+}
+
+static inline long
+sk_memory_allocated(const struct sock *sk)
+{
+ struct proto *prot = sk->sk_prot;
+ return atomic_long_read(prot->memory_allocated);
+}
+
+static inline long
+sk_memory_allocated_add(struct sock *sk, int amt)
+{
+ struct proto *prot = sk->sk_prot;
+ return atomic_long_add_return(amt, prot->memory_allocated);
+}
+
+static inline void
+sk_memory_allocated_sub(struct sock *sk, int amt)
+{
+ struct proto *prot = sk->sk_prot;
+ atomic_long_sub(amt, prot->memory_allocated);
+}
+
+static inline void sk_sockets_allocated_dec(struct sock *sk)
+{
+ struct proto *prot = sk->sk_prot;
+ percpu_counter_dec(prot->sockets_allocated);
+}
+
+static inline void sk_sockets_allocated_inc(struct sock *sk)
+{
+ struct proto *prot = sk->sk_prot;
+ percpu_counter_inc(prot->sockets_allocated);
+}
+
+static inline int
+sk_sockets_allocated_read_positive(struct sock *sk)
+{
+ struct proto *prot = sk->sk_prot;
+
+ return percpu_counter_sum_positive(prot->sockets_allocated);
+}
+
+static inline int
+kcg_sockets_allocated_sum_positive(struct proto *prot, struct mem_cgroup *cg)
+{
```

```
+ return percpu_counter_sum_positive(prot->sockets_allocated);
+}
+
+static inline long
+kcg_memory_allocated(struct proto *prot, struct mem_cgroup *cg)
+{
+ return atomic_long_read(prot->memory_allocated);
+}

 #ifdef CONFIG_PROC_FS
 /* Called with local bh disabled */
diff --git a/include/net/tcp.h b/include/net/tcp.h
index e147f42..ccaa3b6 100644
--- a/include/net/tcp.h
+++ b/include/net/tcp.h
@@ -44,6 +44,7 @@
 #include <net/dst.h>

 #include <linux/seq_file.h>
+#include <linux/memcontrol.h>

 extern struct inet_hashinfo tcp_hashinfo;

@@ -285,7 +286,7 @@ static inline bool tcp_too_many_orphans(struct sock *sk, int shift)
  }

  if (sk->sk_wmem_queued > SOCK_MIN_SNDBUF &&
-     atomic_long_read(&tcp_memory_allocated) > sysctl_tcp_mem[2])
+     sk_memory_allocated(sk) > sk_prot_mem(sk, 2))
   return true;
  return false;
 }
diff --git a/net/core/sock.c b/net/core/sock.c
index 4ed7b1d..26bdb1c 100644
--- a/net/core/sock.c
+++ b/net/core/sock.c
@@ -1288,7 +1288,7 @@ struct sock *sk_clone(const struct sock *sk, const gfp_t priority)
   newsk->sk_wq = NULL;

   if (newsk->sk_prot->sockets_allocated)
-   percpu_counter_inc(newsk->sk_prot->sockets_allocated);
+   sk_sockets_allocated_inc(newsk);

   if (sock_flag(newsk, SOCK_TIMESTAMP) ||
       sock_flag(newsk, SOCK_TIMESTAMPING_RX_SOFTWARE))
@@ -1677,30 +1677,32 @@ int __sk_mem_schedule(struct sock *sk, int size, int kind)
  struct proto *prot = sk->sk_prot;
  int amt = sk_mem_pages(size);
```

```
  long allocated;
+ int *memory_pressure;

  sk->sk_forward_alloc += amt * SK_MEM_QUANTUM;
- allocated = atomic_long_add_return(amt, prot->memory_allocated);
+
+ memory_pressure = sk_memory_pressure(sk);
+ allocated = sk_memory_allocated_add(sk, amt);

  /* Under limit. */
- if (allocated <= prot->sysctl_mem[0]) {
-  if (prot->memory_pressure && *prot->memory_pressure)
-   *prot->memory_pressure = 0;
-  return 1;
- }
+ if (allocated <= sk_prot_mem(sk, 0))
+  if (memory_pressure && *memory_pressure)
+   *memory_pressure = 0;

  /* Under pressure. */
- if (allocated > prot->sysctl_mem[1])
+ if (allocated > sk_prot_mem(sk, 1))
   if (prot->enter_memory_pressure)
    prot->enter_memory_pressure(sk);

  /* Over hard limit. */
- if (allocated > prot->sysctl_mem[2])
+ if (allocated > sk_prot_mem(sk, 2))
   goto suppress_allocation;

  /* guarantee minimum buffer size under pressure */
  if (kind == SK_MEM_RECV) {
   if (atomic_read(&sk->sk_rmem_alloc) < prot->sysctl_rmem[0])
    return 1;
+
  } else { /* SK_MEM_SEND */
   if (sk->sk_type == SOCK_STREAM) {
    if (sk->sk_wmem_queued < prot->sysctl_wmem[0])
@@ -1710,13 +1712,13 @@ int __sk_mem_schedule(struct sock *sk, int size, int kind)
     return 1;
  }

- if (prot->memory_pressure) {
+ if (memory_pressure) {
   int alloc;

-  if (!*prot->memory_pressure)
+  if (!*memory_pressure)
```

```
    return 1;
-  alloc = percpu_counter_read_positive(prot->sockets_allocated);
-  if (prot->sysctl_mem[2] > alloc *
+   alloc = sk_sockets_allocated_read_positive(sk);
+   if (sk_prot_mem(sk, 2) > alloc *
        sk_mem_pages(sk->sk_wmem_queued +
       atomic_read(&sk->sk_rmem_alloc) +
       sk->sk_forward_alloc))
@@ -1739,7 +1741,9 @@ suppress_allocation:

   /* Alas. Undo changes. */
   sk->sk_forward_alloc -= amt * SK_MEM_QUANTUM;
- atomic_long_sub(amt, prot->memory_allocated);
+
+ sk_memory_allocated_sub(sk, amt);
+
   return 0;
  }
  EXPORT_SYMBOL(__sk_mem_schedule);
@@ -1750,15 +1754,15 @@ EXPORT_SYMBOL(__sk_mem_schedule);
   */
  void __sk_mem_reclaim(struct sock *sk)
  {
- struct proto *prot = sk->sk_prot;
+ int *memory_pressure = sk_memory_pressure(sk);

- atomic_long_sub(sk->sk_forward_alloc >> SK_MEM_QUANTUM_SHIFT,
-     prot->memory_allocated);
+ sk_memory_allocated_sub(sk,
+     sk->sk_forward_alloc >> SK_MEM_QUANTUM_SHIFT);
   sk->sk_forward_alloc &= SK_MEM_QUANTUM - 1;

- if (prot->memory_pressure && *prot->memory_pressure &&
-     (atomic_long_read(prot->memory_allocated) < prot->sysctl_mem[0]))
-   *prot->memory_pressure = 0;
+ if (memory_pressure && *memory_pressure &&
+     (sk_memory_allocated(sk) < sk_prot_mem(sk, 0)))
+   *memory_pressure = 0;
  }
  EXPORT_SYMBOL(__sk_mem_reclaim);

@@ -2477,13 +2481,20 @@ static char proto_method_implemented(const void *method)

  static void proto_seq_printf(struct seq_file *seq, struct proto *proto)
  {
+ struct mem_cgroup *cg = mem_cgroup_from_task(current);
+ int *memory_pressure = NULL;
+
```

```
+ if (proto->memory_pressure)
+  memory_pressure = proto->memory_pressure;
+
  seq_printf(seq, "%-9s %4u %6d  %6ld   %-3s %6u   %-3s  %-10s "
    "%2c %2c %2c %2c %2c %2c %2c %2c %2c %2c %2c %2c %2c %2c %2c %2c %2c
%2c\n",
       proto->name,
       proto->obj_size,
       sock_prot_inuse_get(seq_file_net(seq), proto),
-      proto->memory_allocated != NULL ? atomic_long_read(proto->memory_allocated) : -1L,
-      proto->memory_pressure != NULL ? *proto->memory_pressure ? "yes" : "no" : "NI",
+      proto->memory_allocated != NULL ?
+    kcg_memory_allocated(proto, cg) : -1L,
+      memory_pressure != NULL ? *memory_pressure ? "yes" : "no" : "NI",
       proto->max_header,
       proto->slab == NULL ? "no" : "yes",
       module_name(proto->owner),
diff --git a/net/ipv4/proc.c b/net/ipv4/proc.c
index 4bfad5d..535456d 100644
--- a/net/ipv4/proc.c
+++ b/net/ipv4/proc.c
@@ -52,20 +52,21 @@ static int sockstat_seq_show(struct seq_file *seq, void *v)
 {
  struct net *net = seq->private;
  int orphans, sockets;
+ struct mem_cgroup *cg = mem_cgroup_from_task(current);

  local_bh_disable();
  orphans = percpu_counter_sum_positive(&tcp_orphan_count);
- sockets = percpu_counter_sum_positive(&tcp_sockets_allocated);
+ sockets = kcg_sockets_allocated_sum_positive(&tcp_prot, cg);
  local_bh_enable();

  socket_seq_show(seq);
  seq_printf(seq, "TCP: inuse %d orphan %d tw %d alloc %d mem %ld\n",
       sock_prot_inuse_get(net, &tcp_prot), orphans,
       tcp_death_row.tw_count, sockets,
-      atomic_long_read(&tcp_memory_allocated));
+      kcg_memory_allocated(&tcp_prot, cg));
  seq_printf(seq, "UDP: inuse %d mem %ld\n",
       sock_prot_inuse_get(net, &udp_prot),
-      atomic_long_read(&udp_memory_allocated));
+      kcg_memory_allocated(&udp_prot, cg));
  seq_printf(seq, "UDPLITE: inuse %d\n",
       sock_prot_inuse_get(net, &udplite_prot));
  seq_printf(seq, "RAW: inuse %d\n",
diff --git a/net/ipv4/tcp_input.c b/net/ipv4/tcp_input.c
index 52b5c2d..3df862d 100644
```

```
--- a/net/ipv4/tcp_input.c
+++ b/net/ipv4/tcp_input.c
@@ -322,7 +322,7 @@ static void tcp_grow_window(struct sock *sk, const struct sk_buff *skb)
  /* Check #1 */
  if (tp->rcv_ssthresh < tp->window_clamp &&
      (int)tp->rcv_ssthresh < tcp_space(sk) &&
-     !tcp_memory_pressure) {
+     !sk_memory_pressure(sk)) {
   int incr;

   /* Check #2. Increase window, if skb with such overhead
@@ -411,8 +411,8 @@ static void tcp_clamp_window(struct sock *sk)

  if (sk->sk_rcvbuf < sysctl_tcp_rmem[2] &&
      !(sk->sk_userlocks & SOCK_RCVBUF_LOCK) &&
-     !tcp_memory_pressure &&
-     atomic_long_read(&tcp_memory_allocated) < sysctl_tcp_mem[0]) {
+     !sk_memory_pressure(sk) &&
+     sk_memory_allocated(sk) < sk_prot_mem(sk, 0)) {
   sk->sk_rcvbuf = min(atomic_read(&sk->sk_rmem_alloc),
         sysctl_tcp_rmem[2]);
  }
@@ -4864,7 +4864,7 @@ static int tcp_prune_queue(struct sock *sk)

  if (atomic_read(&sk->sk_rmem_alloc) >= sk->sk_rcvbuf)
   tcp_clamp_window(sk);
- else if (tcp_memory_pressure)
+ else if (sk_memory_pressure(sk))
   tp->rcv_ssthresh = min(tp->rcv_ssthresh, 4U * tp->advmss);

  tcp_collapse_ofo_queue(sk);
@@ -4930,11 +4930,11 @@ static int tcp_should_expand_sndbuf(const struct sock *sk)
   return 0;

  /* If we are under global TCP memory pressure, do not expand.  */
- if (tcp_memory_pressure)
+ if (sk_memory_pressure(sk))
   return 0;

  /* If we are under soft global TCP memory pressure, do not expand.  */
- if (atomic_long_read(&tcp_memory_allocated) >= sysctl_tcp_mem[0])
+ if (sk_memory_allocated(sk) >= sk_prot_mem(sk, 0))
   return 0;

  /* If we filled the congestion window, do not expand.  */
diff --git a/net/ipv4/tcp_ipv4.c b/net/ipv4/tcp_ipv4.c
index 0ea10ee..f124a4b 100644
--- a/net/ipv4/tcp_ipv4.c
```

```
+++ b/net/ipv4/tcp_ipv4.c
@@ -1914,7 +1914,7 @@ static int tcp_v4_init_sock(struct sock *sk)
  sk->sk_rcvbuf = sysctl_tcp_rmem[1];

  local_bh_disable();
- percpu_counter_inc(&tcp_sockets_allocated);
+ sk_sockets_allocated_inc(sk);
  local_bh_enable();

  return 0;
@@ -1970,7 +1970,7 @@ void tcp_v4_destroy_sock(struct sock *sk)
   tp->cookie_values = NULL;
  }

- percpu_counter_dec(&tcp_sockets_allocated);
+ sk_sockets_allocated_dec(sk);
 }
 EXPORT_SYMBOL(tcp_v4_destroy_sock);

diff --git a/net/ipv4/tcp_output.c b/net/ipv4/tcp_output.c
index 980b98f..04e229b 100644
--- a/net/ipv4/tcp_output.c
+++ b/net/ipv4/tcp_output.c
@@ -1919,7 +1919,7 @@ u32 __tcp_select_window(struct sock *sk)
  if (free_space < (full_space >> 1)) {
   icsk->icsk_ack.quick = 0;

-  if (tcp_memory_pressure)
+  if (sk_memory_pressure(sk))
    tp->rcv_ssthresh = min(tp->rcv_ssthresh,
           4U * tp->advmss);

diff --git a/net/ipv4/tcp_timer.c b/net/ipv4/tcp_timer.c
index 2e0f0af..c9f830c 100644
--- a/net/ipv4/tcp_timer.c
+++ b/net/ipv4/tcp_timer.c
@@ -261,7 +261,7 @@ static void tcp_delack_timer(unsigned long data)
  }

 out:
- if (tcp_memory_pressure)
+ if (sk_memory_pressure(sk))
   sk_mem_reclaim(sk);
 out_unlock:
  bh_unlock_sock(sk);
diff --git a/net/ipv6/tcp_ipv6.c b/net/ipv6/tcp_ipv6.c
index 10b2b31..3a08fcd 100644
--- a/net/ipv6/tcp_ipv6.c
```

```
+++ b/net/ipv6/tcp_ipv6.c
@@ -1995,7 +1995,7 @@ static int tcp_v6_init_sock(struct sock *sk)
  sk->sk_rcvbuf = sysctl_tcp_rmem[1];

  local_bh_disable();
- percpu_counter_inc(&tcp_sockets_allocated);
+ sk_sockets_allocated_inc(sk);
  local_bh_enable();

  return 0;
--
1.7.6.4
```

Subject: [PATCH v5 03/10] socket: initial cgroup code.
Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:28 GMT
View Forum Message <> Reply to Message

The goal of this work is to move the memory pressure tcp
controls to a cgroup, instead of just relying on global
conditions.

To avoid excessive overhead in the network fast paths,
the code that accounts allocated memory to a cgroup is
hidden inside a static_branch(). This branch is patched out
until the first non-root cgroup is created. So when nobody
is using cgroups, even if it is mounted, no significant performance
penalty should be seen.

This patch handles the generic part of the code, and has nothing
tcp-specific.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Kirill A. Shutemov<kirill@shutemov.name>
Reviewed-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujtsu.com>
CC: David S. Miller <davem@davemloft.net>
CC: Eric W. Biederman <ebiederm@xmission.com>
CC: Eric Dumazet <eric.dumazet@gmail.com>
---
 include/linux/memcontrol.h | 27 ++++++++
 include/net/sock.h | 151 ++-
 mm/memcontrol.c | 85 ++++++++++++++++++++++++--
 net/core/sock.c | 36 ++++++++---
 4 files changed, 281 insertions(+), 18 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index e9ff93a..994a06a 100644
--- a/include/linux/memcontrol.h

```
+++ b/include/linux/memcontrol.h
@@ -381,5 +381,32 @@ mem_cgroup_print_bad_page(struct page *page)
 }
 #endif

+#ifdef CONFIG_INET
+enum {
+ UNDER_LIMIT,
+ SOFT_LIMIT,
+ OVER_LIMIT,
+};
+
+struct sock;
+struct cg_proto;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+void sock_update_memcg(struct sock *sk);
+void memcg_sockets_allocated_dec(struct mem_cgroup *memcg,
+     struct cg_proto *prot);
+void memcg_sockets_allocated_inc(struct mem_cgroup *memcg,
+     struct cg_proto *prot);
+void memcg_memory_allocated_add(struct mem_cgroup *memcg, struct cg_proto *prot,
+    unsigned long amt, int *parent_status);
+void memcg_memory_allocated_sub(struct mem_cgroup *memcg, struct cg_proto *prot,
+    unsigned long amt);
+u64 memcg_memory_allocated_read(struct mem_cgroup *memcg,
+    struct cg_proto *prot);
+#else
+static inline void sock_update_memcg(struct sock *sk)
+{
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+#endif /* CONFIG_INET */
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/include/net/sock.h b/include/net/sock.h
index 8959dcc..02d7cce 100644
--- a/include/net/sock.h
+++ b/include/net/sock.h
@@ -55,6 +55,7 @@
 #include <linux/slab.h>
 #include <linux/uaccess.h>
 #include <linux/cgroup.h>
+#include <linux/res_counter.h>

 #include <linux/filter.h>
 #include <linux/rculist_nulls.h>
@@ -64,6 +65,8 @@
 #include <net/dst.h>
```

```
 #include <net/checksum.h>

+int sockets_populate(struct cgroup *cgrp, struct cgroup_subsys *ss);
+void sockets_destroy(struct cgroup *cgrp, struct cgroup_subsys *ss);
 /*
  * This structure really needs to be cleaned up.
  * Most of it is for TCP, and not used by any of
@@ -231,6 +234,7 @@ struct mem_cgroup;
   * @sk_security: used by security modules
   * @sk_mark: generic packet mark
   * @sk_classid: this socket's cgroup classid
+  * @sk_cgrp: this socket's kernel memory (kmem) cgroup
   * @sk_write_pending: a write to stream socket waits to start
   * @sk_state_change: callback to indicate change in the state of the sock
   * @sk_data_ready: callback to indicate there is data to be processed
@@ -342,6 +346,7 @@ struct sock {
 #endif
  __u32   sk_mark;
  u32   sk_classid;
+ struct mem_cgroup *sk_cgrp;
  void   (*sk_state_change)(struct sock *sk);
  void   (*sk_data_ready)(struct sock *sk, int bytes);
  void   (*sk_write_space)(struct sock *sk);
@@ -733,6 +738,9 @@ struct timewait_sock_ops;
 struct inet_hashinfo;
 struct raw_hashinfo;

+
+struct cg_proto;
+
 /* Networking protocol blocks we attach to sockets.
  * socket layer -> transport layer interface
  * transport -> network interface is defined by struct inet_proto
@@ -835,9 +843,32 @@ struct proto {
 #ifdef SOCK_REFCNT_DEBUG
  atomic_t  socks;
 #endif
+ struct cg_proto  *cg_proto; /* This just makes proto replacement easier */
+};
+
+struct cg_proto {
+ /*
+  * cgroup specific init/deinit functions. Called once for all
+  * protocols that implement it, from cgroups populate function.
+  * This function has to setup any files the protocol want to
+  * appear in the kmem cgroup filesystem.
+  */
+ int   (*init_cgroup)(struct cgroup *cgrp,
```

```
+            struct cgroup_subsys *ss);
+ void   (*destroy_cgroup)(struct cgroup *cgrp,
+         struct cgroup_subsys *ss);
+ struct res_counter *(*memory_allocated)(struct mem_cgroup *memcg);
+ /* Pointer to the current number of sockets in this cgroup. */
+ struct percpu_counter  *(*sockets_allocated)(struct mem_cgroup *memcg);
+
+ int   *(*memory_pressure)(struct mem_cgroup *memcg);
+ long   *(*prot_mem)(struct mem_cgroup *memcg);
+
+ struct list_head node; /* with a bit more effort, we could reuse proto's */
 };

 extern int proto_register(struct proto *prot, int alloc_slab);
+extern void cg_proto_register(struct cg_proto *prot, struct proto *proto);
 extern void proto_unregister(struct proto *prot);

 #ifdef SOCK_REFCNT_DEBUG
@@ -865,15 +896,40 @@ static inline void sk_refcnt_debug_release(const struct sock *sk)
 #define sk_refcnt_debug_release(sk) do { } while (0)
 #endif /* SOCK_REFCNT_DEBUG */

+extern struct jump_label_key cgroup_crap_enabled;
 #include <linux/memcontrol.h>
 static inline int *sk_memory_pressure(const struct sock *sk)
 {
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  int *ret = NULL;
+  struct cg_proto *cg_prot = sk->sk_prot->cg_proto;
+
+  if (!sk->sk_cgrp)
+   goto nocgroup;
+  if (cg_prot->memory_pressure)
+   ret = cg_prot->memory_pressure(sk->sk_cgrp);
+  return ret;
+ } else
+nocgroup:
+#endif
  return sk->sk_prot->memory_pressure;
 }

 static inline long sk_prot_mem(const struct sock *sk, int index)
 {
  long *prot = sk->sk_prot->sysctl_mem;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct mem_cgroup *cg = sk->sk_cgrp;
```

```
+  struct cg_proto *cg_prot = sk->sk_prot->cg_proto;
+  if (!cg) /* this handles the case with existing sockets */
+   goto nocgroup;
+
+  cg_prot->prot_mem(sk->sk_cgrp);
+ }
+nocgroup:
+#endif
  return prot[index];
 }

@@ -881,32 +937,93 @@ static inline long
 sk_memory_allocated(const struct sock *sk)
 {
  struct proto *prot = sk->sk_prot;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct mem_cgroup *cg = sk->sk_cgrp;
+  struct cg_proto *cg_prot = sk->sk_prot->cg_proto;
+  if (!cg) /* this handles the case with existing sockets */
+   goto nocgroup;
+
+  return memcg_memory_allocated_read(cg, cg_prot);
+ }
+nocgroup:
+#endif
  return atomic_long_read(prot->memory_allocated);
 }

 static inline long
-sk_memory_allocated_add(struct sock *sk, int amt)
+sk_memory_allocated_add(struct sock *sk, int amt, int *parent_status)
 {
  struct proto *prot = sk->sk_prot;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct mem_cgroup *cg = sk->sk_cgrp;
+  struct cg_proto *cg_prot = prot->cg_proto;
+
+  if (!cg)
+   goto nocgroup;
+
+  memcg_memory_allocated_add(cg, cg_prot, amt, parent_status);
+ }
+nocgroup:
+#endif
  return atomic_long_add_return(amt, prot->memory_allocated);
 }
```

```
 static inline void
-sk_memory_allocated_sub(struct sock *sk, int amt)
+sk_memory_allocated_sub(struct sock *sk, int amt, int parent_status)
 {
  struct proto *prot = sk->sk_prot;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct mem_cgroup *cg = sk->sk_cgrp;
+  struct cg_proto *cg_prot = prot->cg_proto;
+
+  if (!cg)
+   goto nocgroup;
+
+  /* Otherwise it was uncharged already */
+  if (parent_status != OVER_LIMIT)
+   memcg_memory_allocated_sub(cg, cg_prot, amt);
+ }
+nocgroup:
+#endif
  atomic_long_sub(amt, prot->memory_allocated);
 }

 static inline void sk_sockets_allocated_dec(struct sock *sk)
 {
  struct proto *prot = sk->sk_prot;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct mem_cgroup *cg = sk->sk_cgrp;
+  struct cg_proto *cg_prot = prot->cg_proto;
+
+  if (!cg)
+   goto nocgroup;
+
+  memcg_sockets_allocated_dec(cg, cg_prot);
+ }
+nocgroup:
+#endif
  percpu_counter_dec(prot->sockets_allocated);
 }

 static inline void sk_sockets_allocated_inc(struct sock *sk)
 {
  struct proto *prot = sk->sk_prot;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct mem_cgroup *cg = sk->sk_cgrp;
+  struct cg_proto *cg_prot = prot->cg_proto;
```

```
+
+  if (!cg)
+   goto nocgroup;
+
+  memcg_sockets_allocated_inc(cg, cg_prot);
+ }
+nocgroup:
+#endif
  percpu_counter_inc(prot->sockets_allocated);
 }

@@ -914,19 +1031,47 @@ static inline int
 sk_sockets_allocated_read_positive(struct sock *sk)
 {
  struct proto *prot = sk->sk_prot;
-
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct mem_cgroup *cg = sk->sk_cgrp;
+  struct cg_proto *cg_prot = prot->cg_proto;
+
+  if (!cg)
+   goto nocgroup;
+  return percpu_counter_sum_positive(cg_prot->sockets_allocated(cg));
+ }
+nocgroup:
+#endif
  return percpu_counter_sum_positive(prot->sockets_allocated);
 }

 static inline int
 kcg_sockets_allocated_sum_positive(struct proto *prot, struct mem_cgroup *cg)
 {
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct cg_proto *cg_prot = prot->cg_proto;
+  if (!cg)
+   goto nocgroup;
+  return percpu_counter_sum_positive(cg_prot->sockets_allocated(cg));
+ }
+nocgroup:
+#endif
  return percpu_counter_sum_positive(prot->sockets_allocated);
 }

 static inline long
 kcg_memory_allocated(struct proto *prot, struct mem_cgroup *cg)
 {
```

```
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct cg_proto *cg_prot = prot->cg_proto;
+  if (!cg)
+   goto nocgroup;
+  return memcg_memory_allocated_read(cg, cg_prot);
+ }
+nocgroup:
+#endif
  return atomic_long_read(prot->memory_allocated);
 }

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 3389d33..7d684d0 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -376,6 +376,85 @@ enum mem_type {
 #define MEM_CGROUP_RECLAIM_SOFT_BIT 0x2
 #define MEM_CGROUP_RECLAIM_SOFT  (1 << MEM_CGROUP_RECLAIM_SOFT_BIT)

+static struct mem_cgroup *parent_mem_cgroup(struct mem_cgroup *memcg);
+
+static inline bool mem_cgroup_is_root(struct mem_cgroup *mem)
+{
+ return (mem == root_mem_cgroup);
+}
+
+/* Writing them here to avoid exposing memcg's inner layout */
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+#ifdef CONFIG_INET
+#include <net/sock.h>
+
+void sock_update_memcg(struct sock *sk)
+{
+ /* right now a socket spends its whole life in the same cgroup */
+ if (sk->sk_cgrp) {
+  WARN_ON(1);
+  return;
+ }
+ if (static_branch(&cgroup_crap_enabled)) {
+  struct mem_cgroup *memcg;
+
+  BUG_ON(!sk->sk_prot->cg_proto);
+
+  rcu_read_lock();
+  memcg = mem_cgroup_from_task(current);
+  if (!mem_cgroup_is_root(memcg))
+   sk->sk_cgrp = memcg;
```

```
+  rcu_read_unlock();
+ }
+}
+
+void memcg_sockets_allocated_dec(struct mem_cgroup *memcg,
+     struct cg_proto *prot)
+{
+ for (; memcg; memcg = parent_mem_cgroup(memcg))
+  percpu_counter_dec(prot->sockets_allocated(memcg));
+}
+EXPORT_SYMBOL(memcg_sockets_allocated_dec);
+
+void memcg_sockets_allocated_inc(struct mem_cgroup *memcg,
+     struct cg_proto *prot)
+{
+ for (; memcg; memcg = parent_mem_cgroup(memcg))
+  percpu_counter_inc(prot->sockets_allocated(memcg));
+}
+EXPORT_SYMBOL(memcg_sockets_allocated_inc);
+
+void memcg_memory_allocated_add(struct mem_cgroup *memcg, struct cg_proto *prot,
+    unsigned long amt, int *parent_status)
+{
+ struct res_counter *fail;
+ int ret;
+
+ ret = res_counter_charge(prot->memory_allocated(memcg),
+     amt << PAGE_SHIFT, &fail);
+
+ if (ret < 0)
+  *parent_status = OVER_LIMIT;
+}
+EXPORT_SYMBOL(memcg_memory_allocated_add);
+
+void memcg_memory_allocated_sub(struct mem_cgroup *memcg, struct cg_proto *prot,
+    unsigned long amt)
+{
+ res_counter_uncharge(prot->memory_allocated(memcg), amt << PAGE_SHIFT);
+}
+EXPORT_SYMBOL(memcg_memory_allocated_sub);
+
+u64 memcg_memory_allocated_read(struct mem_cgroup *memcg, struct cg_proto *prot)
+{
+ return res_counter_read_u64(prot->memory_allocated(memcg),
+        RES_USAGE) >> PAGE_SHIFT ;
+}
+EXPORT_SYMBOL(memcg_memory_allocated_read);
+#endif /* CONFIG_INET */
```

```
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
+
 static void mem_cgroup_get(struct mem_cgroup *mem);
 static void mem_cgroup_put(struct mem_cgroup *mem);
 static struct mem_cgroup *parent_mem_cgroup(struct mem_cgroup *mem);
@@ -872,12 +951,6 @@ static struct mem_cgroup *mem_cgroup_get_next(struct mem_cgroup
*iter,
 #define for_each_mem_cgroup_all(iter) \
  for_each_mem_cgroup_tree_cond(iter, NULL, true)

-
-static inline bool mem_cgroup_is_root(struct mem_cgroup *mem)
-{
- return (mem == root_mem_cgroup);
-}
-
 void mem_cgroup_count_vm_event(struct mm_struct *mm, enum vm_event_item idx)
 {
  struct mem_cgroup *mem;
diff --git a/net/core/sock.c b/net/core/sock.c
index 26bdb1c..b64d36a 100644
--- a/net/core/sock.c
+++ b/net/core/sock.c
@@ -111,6 +111,7 @@
 #include <linux/init.h>
 #include <linux/highmem.h>
 #include <linux/user_namespace.h>
+#include <linux/jump_label.h>

 #include <asm/uaccess.h>
 #include <asm/system.h>
@@ -141,6 +142,9 @@
 static struct lock_class_key af_family_keys[AF_MAX];
 static struct lock_class_key af_family_slock_keys[AF_MAX];

+struct jump_label_key cgroup_crap_enabled;
+EXPORT_SYMBOL(cgroup_crap_enabled);
+
 /*
  * Make lock validator output more readable. (we pre-construct these
  * strings build-time, so that runtime initialization of socket
@@ -1678,26 +1682,27 @@ int __sk_mem_schedule(struct sock *sk, int size, int kind)
  int amt = sk_mem_pages(size);
  long allocated;
  int *memory_pressure;
+ int parent_status = UNDER_LIMIT;
```

```
  sk->sk_forward_alloc += amt * SK_MEM_QUANTUM;

  memory_pressure = sk_memory_pressure(sk);
- allocated = sk_memory_allocated_add(sk, amt);
+ allocated = sk_memory_allocated_add(sk, amt, &parent_status);
+
+ /* Over hard limit (we or our parents) */
+ if ((parent_status == OVER_LIMIT) || (allocated > sk_prot_mem(sk, 2)))
+  goto suppress_allocation;

  /* Under limit. */
  if (allocated <= sk_prot_mem(sk, 0))
   if (memory_pressure && *memory_pressure)
    *memory_pressure = 0;

- /* Under pressure. */
- if (allocated > sk_prot_mem(sk, 1))
+ /* Under pressure. (we or our parents) */
+ if ((parent_status == SOFT_LIMIT) || allocated > sk_prot_mem(sk, 1))
   if (prot->enter_memory_pressure)
    prot->enter_memory_pressure(sk);

- /* Over hard limit. */
- if (allocated > sk_prot_mem(sk, 2))
-  goto suppress_allocation;
-
  /* guarantee minimum buffer size under pressure */
  if (kind == SK_MEM_RECV) {
   if (atomic_read(&sk->sk_rmem_alloc) < prot->sysctl_rmem[0])
@@ -1742,7 +1747,7 @@ suppress_allocation:
  /* Alas. Undo changes. */
  sk->sk_forward_alloc -= amt * SK_MEM_QUANTUM;

- sk_memory_allocated_sub(sk, amt);
+ sk_memory_allocated_sub(sk, amt, parent_status);

  return 0;
 }
@@ -1757,7 +1762,7 @@ void __sk_mem_reclaim(struct sock *sk)
  sk_memory_allocated_sub(sk,
-    sk->sk_forward_alloc >> SK_MEM_QUANTUM_SHIFT);
+    sk->sk_forward_alloc >> SK_MEM_QUANTUM_SHIFT, 0);
  sk->sk_forward_alloc &= SK_MEM_QUANTUM - 1;

  if (memory_pressure && *memory_pressure &&
@@ -2257,6 +2262,9 @@ EXPORT_SYMBOL(sk_common_release);
```

```
 static DEFINE_RWLOCK(proto_list_lock);
 static LIST_HEAD(proto_list);

+static DEFINE_RWLOCK(cg_proto_list_lock);
+static LIST_HEAD(cg_proto_list);
+
 #ifdef CONFIG_PROC_FS
 #define PROTO_INUSE_NR 64 /* should be enough for the first time */
 struct prot_inuse {
@@ -2358,6 +2366,16 @@ static inline void release_proto_idx(struct proto *prot)
 }
 #endif

+void cg_proto_register(struct cg_proto *prot, struct proto *parent)
+{
+ write_lock(&cg_proto_list_lock);
+ list_add(&prot->node, &cg_proto_list);
+ write_unlock(&cg_proto_list_lock);
+
+ parent->cg_proto = prot;
+}
+EXPORT_SYMBOL(cg_proto_register);
+
 int proto_register(struct proto *prot, int alloc_slab)
 {
  if (alloc_slab) {
--
1.7.6.4
```

Subject: [PATCH v5 04/10] per-cgroup tcp buffers control
Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:29 GMT
View Forum Message <> Reply to Message

With all the infrastructure in place, this patch implements
per-cgroup control for tcp memory pressure handling.

A resource conter is used to control allocated memory, except
for the root cgroup, that will keep using global counters.

This patch is the one that actually enables/disables the
jump labels controlling cgroup. To this point, they were always
disabled.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujtisu.com>
CC: David S. Miller <davem@davemloft.net>
CC: Eric W. Biederman <ebiederm@xmission.com>

```
CC: Eric Dumazet <eric.dumazet@gmail.com>
---
 include/net/tcp.h       |   18 +++++++
 include/net/transp_v6.h |    1 +
 mm/memcontrol.c         |  125 ++++++++++++++++++++++++++++++++++++++++++++++-
 net/core/sock.c         |   46 ++++++++++++++---
 net/ipv4/af_inet.c      |    3 +
 net/ipv4/tcp_ipv4.c     |   12 +++++
 net/ipv6/af_inet6.c     |    3 +
 net/ipv6/tcp_ipv6.c     |   10 ++++
 8 files changed, 211 insertions(+), 7 deletions(-)

diff --git a/include/net/tcp.h b/include/net/tcp.h
index ccaa3b6..7301ca8 100644
--- a/include/net/tcp.h
+++ b/include/net/tcp.h
@@ -253,6 +253,22 @@ extern int sysctl_tcp_cookie_size;
 extern int sysctl_tcp_thin_linear_timeouts;
 extern int sysctl_tcp_thin_dupack;

+struct tcp_memcontrol {
+ /* per-cgroup tcp memory pressure knobs */
+ struct res_counter tcp_memory_allocated;
+ struct percpu_counter tcp_sockets_allocated;
+ /* those two are read-mostly, leave them at the end */
+ long tcp_prot_mem[3];
+ int tcp_memory_pressure;
+};
+
+long *sysctl_mem_tcp(struct mem_cgroup *memcg);
+struct percpu_counter *sockets_allocated_tcp(struct mem_cgroup *memcg);
+int *memory_pressure_tcp(struct mem_cgroup *memcg);
+struct res_counter *memory_allocated_tcp(struct mem_cgroup *memcg);
+int tcp_init_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss);
+void tcp_destroy_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss);
+
 extern atomic_long_t tcp_memory_allocated;
 extern struct percpu_counter tcp_sockets_allocated;
 extern int tcp_memory_pressure;
@@ -305,6 +321,7 @@ static inline int tcp_synq_no_recent_overflow(const struct sock *sk)
 }

 extern struct proto tcp_prot;
+extern struct cg_proto tcp_cg_prot;

 #define TCP_INC_STATS(net, field) SNMP_INC_STATS((net)->mib.tcp_statistics, field)
 #define TCP_INC_STATS_BH(net, field) SNMP_INC_STATS_BH((net)->mib.tcp_statistics, field)
@@ -1022,6 +1039,7 @@ static inline void tcp_openreq_init(struct request_sock *req,
```

```
  ireq->loc_port = tcp_hdr(skb)->dest;
 }

+extern void tcp_enter_memory_pressure_cg(struct sock *sk);
 extern void tcp_enter_memory_pressure(struct sock *sk);

 static inline int keepalive_intvl_when(const struct tcp_sock *tp)
diff --git a/include/net/transp_v6.h b/include/net/transp_v6.h
index 498433d..1e18849 100644
--- a/include/net/transp_v6.h
+++ b/include/net/transp_v6.h
@@ -11,6 +11,7 @@ extern struct proto rawv6_prot;
 extern struct proto udpv6_prot;
 extern struct proto udplitev6_prot;
 extern struct proto tcpv6_prot;
+extern struct cg_proto tcpv6_cg_prot;

 struct flowi6;

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 7d684d0..f14d7d2 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -49,6 +49,9 @@
 #include <linux/cpu.h>
 #include <linux/oom.h>
 #include "internal.h"
+#ifdef CONFIG_INET
+#include <net/tcp.h>
+#endif

 #include <asm/uaccess.h>

@@ -294,6 +297,10 @@ struct mem_cgroup {
   */
  struct mem_cgroup_stat_cpu nocpu_base;
  spinlock_t pcp_counter_lock;
+
+#ifdef CONFIG_INET
+ struct tcp_memcontrol tcp;
+#endif
 };

 /* Stuffs for move charges at task migration. */
@@ -377,7 +384,7 @@ enum mem_type {
 #define MEM_CGROUP_RECLAIM_SOFT  (1 << MEM_CGROUP_RECLAIM_SOFT_BIT)

 static struct mem_cgroup *parent_mem_cgroup(struct mem_cgroup *memcg);
```

```
-
+static struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont);
 static inline bool mem_cgroup_is_root(struct mem_cgroup *mem)
 {
  return (mem == root_mem_cgroup);
@@ -387,6 +394,7 @@ static inline bool mem_cgroup_is_root(struct mem_cgroup *mem)
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
 #ifdef CONFIG_INET
 #include <net/sock.h>
+#include <net/ip.h>

 void sock_update_memcg(struct sock *sk)
 {
@@ -451,6 +459,93 @@ u64 memcg_memory_allocated_read(struct mem_cgroup *memcg,
struct cg_proto *prot)
        RES_USAGE) >> PAGE_SHIFT ;
 }
 EXPORT_SYMBOL(memcg_memory_allocated_read);
+/*
+ * Pressure flag: try to collapse.
+ * Technical note: it is used by multiple contexts non atomically.
+ * All the __sk_mem_schedule() is of this nature: accounting
+ * is strict, actions are advisory and have some latency.
+ */
+void tcp_enter_memory_pressure_cg(struct sock *sk)
+{
+ struct mem_cgroup *memcg = sk->sk_cgrp;
+ if (!memcg->tcp.tcp_memory_pressure) {
+  NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPMEMORYPRESSURES);
+  memcg->tcp.tcp_memory_pressure = 1;
+ }
+}
+EXPORT_SYMBOL(tcp_enter_memory_pressure_cg);
+
+long *sysctl_mem_tcp(struct mem_cgroup *memcg)
+{
+ return memcg->tcp.tcp_prot_mem;
+}
+EXPORT_SYMBOL(sysctl_mem_tcp);
+
+struct res_counter *memory_allocated_tcp(struct mem_cgroup *memcg)
+{
+ return &memcg->tcp.tcp_memory_allocated;
+}
+EXPORT_SYMBOL(memory_allocated_tcp);
+
+int *memory_pressure_tcp(struct mem_cgroup *memcg)
+{
```

```
+ return &memcg->tcp.tcp_memory_pressure;
+}
+EXPORT_SYMBOL(memory_pressure_tcp);
+
+struct percpu_counter *sockets_allocated_tcp(struct mem_cgroup *memcg)
+{
+ return &memcg->tcp.tcp_sockets_allocated;
+}
+EXPORT_SYMBOL(sockets_allocated_tcp);
+
+static void tcp_create_cgroup(struct mem_cgroup *cg, struct cgroup_subsys *ss)
+{
+ /*
+  * The root cgroup does not use res_counters, but rather,
+  * rely on the data already collected by the network
+  * subsystem
+  */
+ if (!mem_cgroup_is_root(cg)) {
+  struct mem_cgroup *parent = parent_mem_cgroup(cg);
+  struct res_counter *res_parent = NULL;
+  cg->tcp.tcp_memory_pressure = 0;
+  percpu_counter_init(&cg->tcp.tcp_sockets_allocated, 0);
+
+  /*
+   * Because root is not using res_counter, we only need a parent
+   * if we're second in hierarchy.
+   */
+  if (!mem_cgroup_is_root(parent) && parent && parent->use_hierarchy)
+   res_parent = &parent->tcp.tcp_memory_allocated;
+
+  res_counter_init(&cg->tcp.tcp_memory_allocated, res_parent);
+ }
+}
+
+int tcp_init_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss)
+{
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
+ /*
+  * We need to initialize it at populate, not create time.
+  * This is because net sysctl tables are not up until much
+  * later
+  */
+ memcg->tcp.tcp_prot_mem[0] = sysctl_tcp_mem[0];
+ memcg->tcp.tcp_prot_mem[1] = sysctl_tcp_mem[1];
+ memcg->tcp.tcp_prot_mem[2] = sysctl_tcp_mem[2];
+
+ return 0;
+}
```

```
+EXPORT_SYMBOL(tcp_init_cgroup);
+
+void tcp_destroy_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss)
+{
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
+
+ percpu_counter_destroy(&memcg->tcp.tcp_sockets_allocated);
+}
+EXPORT_SYMBOL(tcp_destroy_cgroup);
 #endif /* CONFIG_INET */
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

@@ -4867,17 +4962,39 @@ static struct cftype kmem_cgroup_files[] = {
 static int register_kmem_files(struct cgroup *cont, struct cgroup_subsys *ss)
 {
  int ret = 0;
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);

  ret = cgroup_add_files(cont, ss, kmem_cgroup_files,
          ARRAY_SIZE(kmem_cgroup_files));
+
+ if (!ret)
+  ret = sockets_populate(cont, ss);
+
+ if (!mem_cgroup_is_root(memcg))
+  jump_label_inc(&cgroup_crap_enabled);
+
  return ret;
 };

+static void kmem_cgroup_destroy(struct cgroup_subsys *ss,
+     struct cgroup *cont)
+{
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ sockets_destroy(cont, ss);
+
+ if (!mem_cgroup_is_root(memcg))
+  jump_label_dec(&cgroup_crap_enabled);
+}
 #else
 static int register_kmem_files(struct cgroup *cont, struct cgroup_subsys *ss)
 {
  return 0;
 }
+
+static void kmem_cgroup_destroy(struct cgroup_subsys *ss,
+     struct cgroup *cont)
+{
```

```
+}
 #endif

 static int alloc_mem_cgroup_per_zone_info(struct mem_cgroup *mem, int node)
@@ -5095,6 +5212,10 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
  mem->last_scanned_node = MAX_NUMNODES;
  INIT_LIST_HEAD(&mem->oom_notify);

+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_INET)
+ tcp_create_cgroup(mem, ss);
+#endif
+
  if (parent)
   mem->swappiness = mem_cgroup_swappiness(parent);
  atomic_set(&mem->refcnt, 1);
@@ -5120,6 +5241,8 @@ static void mem_cgroup_destroy(struct cgroup_subsys *ss,
 {
  struct mem_cgroup *mem = mem_cgroup_from_cont(cont);

+ kmem_cgroup_destroy(ss, cont);
+
  mem_cgroup_put(mem);
 }

diff --git a/net/core/sock.c b/net/core/sock.c
index b64d36a..c784173 100644
--- a/net/core/sock.c
+++ b/net/core/sock.c
@@ -135,6 +135,46 @@
 #include <net/tcp.h>
 #endif

+static DEFINE_RWLOCK(proto_list_lock);
+static LIST_HEAD(proto_list);
+
+static DEFINE_RWLOCK(cg_proto_list_lock);
+static LIST_HEAD(cg_proto_list);
+
+int sockets_populate(struct cgroup *cgrp, struct cgroup_subsys *ss)
+{
+ struct cg_proto *proto;
+ int ret = 0;
+
+ read_lock(&cg_proto_list_lock);
+ list_for_each_entry(proto, &cg_proto_list, node) {
+  if (proto->init_cgroup)
+   ret = proto->init_cgroup(cgrp, ss);
+   if (ret)
```

```
+    goto out;
+ }
+
+ read_unlock(&cg_proto_list_lock);
+ return ret;
+out:
+ list_for_each_entry_continue_reverse(proto, &cg_proto_list, node)
+  if (proto->destroy_cgroup)
+   proto->destroy_cgroup(cgrp, ss);
+ read_unlock(&cg_proto_list_lock);
+ return ret;
+}
+
+void sockets_destroy(struct cgroup *cgrp, struct cgroup_subsys *ss)
+{
+ struct cg_proto *proto;
+
+ read_lock(&cg_proto_list_lock);
+ list_for_each_entry_reverse(proto, &cg_proto_list, node)
+  if (proto->destroy_cgroup)
+   proto->destroy_cgroup(cgrp, ss);
+ read_unlock(&cg_proto_list_lock);
+}
+
 /*
  * Each address family might have different locking rules, so we have
  * one slock key per address family:
@@ -2259,12 +2299,6 @@ void sk_common_release(struct sock *sk)
 }
 EXPORT_SYMBOL(sk_common_release);

-static DEFINE_RWLOCK(proto_list_lock);
-static LIST_HEAD(proto_list);
-
-static DEFINE_RWLOCK(cg_proto_list_lock);
-static LIST_HEAD(cg_proto_list);
-
 #ifdef CONFIG_PROC_FS
 #define PROTO_INUSE_NR 64 /* should be enough for the first time */
 struct prot_inuse {
diff --git a/net/ipv4/af_inet.c b/net/ipv4/af_inet.c
index 1b5096a..da19147 100644
--- a/net/ipv4/af_inet.c
+++ b/net/ipv4/af_inet.c
@@ -1661,6 +1661,9 @@ static int __init inet_init(void)
  if (rc)
   goto out_unregister_raw_proto;
```

```
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ cg_proto_register(&tcp_cg_prot, &tcp_prot);
+#endif
  /*
   * Tell SOCKET that we are alive...
   */
diff --git a/net/ipv4/tcp_ipv4.c b/net/ipv4/tcp_ipv4.c
index f124a4b..54f6b96 100644
--- a/net/ipv4/tcp_ipv4.c
+++ b/net/ipv4/tcp_ipv4.c
@@ -1917,6 +1917,7 @@ static int tcp_v4_init_sock(struct sock *sk)
  sk_sockets_allocated_inc(sk);
  local_bh_enable();

+ sock_update_memcg(sk);
  return 0;
 }

@@ -2632,6 +2633,17 @@ struct proto tcp_prot = {
 };
 EXPORT_SYMBOL(tcp_prot);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+struct cg_proto tcp_cg_prot = {
+ .memory_allocated = memory_allocated_tcp,
+ .memory_pressure = memory_pressure_tcp,
+ .sockets_allocated = sockets_allocated_tcp,
+ .prot_mem  = sysctl_mem_tcp,
+ .init_cgroup  = tcp_init_cgroup,
+ .destroy_cgroup  = tcp_destroy_cgroup,
+};
+EXPORT_SYMBOL(tcp_cg_prot);
+#endif

 static int __net_init tcp_sk_init(struct net *net)
 {
diff --git a/net/ipv6/af_inet6.c b/net/ipv6/af_inet6.c
index d27c797..51672f8 100644
--- a/net/ipv6/af_inet6.c
+++ b/net/ipv6/af_inet6.c
@@ -1103,6 +1103,9 @@ static int __init inet6_init(void)
  if (err)
   goto out_unregister_raw_proto;

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ cg_proto_register(&tcpv6_cg_prot, &tcpv6_prot);
+#endif
  /* Register the family here so that the init calls below will
```

```
   * be able to create sockets. (?? is this dangerous ??)
   */
diff --git a/net/ipv6/tcp_ipv6.c b/net/ipv6/tcp_ipv6.c
index 3a08fcd..3c13142 100644
--- a/net/ipv6/tcp_ipv6.c
+++ b/net/ipv6/tcp_ipv6.c
@@ -2224,6 +2224,16 @@ struct proto tcpv6_prot = {
 #endif
 };

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+struct cg_proto tcpv6_cg_prot = {
+ .memory_allocated = memory_allocated_tcp,
+ .memory_pressure = memory_pressure_tcp,
+ .sockets_allocated = sockets_allocated_tcp,
+ .prot_mem  = sysctl_mem_tcp,
+};
+EXPORT_SYMBOL(tcpv6_cg_prot);
+#endif
+
 static const struct inet6_protocol tcpv6_protocol = {
  .handler = tcp_v6_rcv,
  .err_handler = tcp_v6_err,
--
1.7.6.4
```

Subject: [PATCH v5 05/10] per-netns ipv4 sysctl_tcp_mem
Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:30 GMT
View Forum Message <> Reply to Message

This patch allows each namespace to independently set up
its levels for tcp memory pressure thresholds. This patch
alone does not buy much: we need to make this values
per group of process somehow. This is achieved in the
patches that follows in this patchset.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Reviewed-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: David S. Miller <davem@davemloft.net>
CC: Eric W. Biederman <ebiederm@xmission.com>

```
 include/net/netns/ipv4.h   |    1 +
 include/net/tcp.h          |    1 -
 mm/memcontrol.c            |    8 ++++--
 net/ipv4/af_inet.c         |    2 +
 net/ipv4/sysctl_net_ipv4.c |   51 ++++++++++++++++++++++++++++++++++++++++++++
 net/ipv4/tcp.c             |   11 +-------
```

```
 net/ipv4/tcp_ipv4.c        |    1 -
 net/ipv6/af_inet6.c        |    2 +
 net/ipv6/tcp_ipv6.c        |    1 -
 9 files changed, 56 insertions(+), 22 deletions(-)

diff --git a/include/net/netns/ipv4.h b/include/net/netns/ipv4.h
index d786b4f..bbd023a 100644
--- a/include/net/netns/ipv4.h
+++ b/include/net/netns/ipv4.h
@@ -55,6 +55,7 @@ struct netns_ipv4 {
  int current_rt_cache_rebuild_count;

  unsigned int sysctl_ping_group_range[2];
+ long sysctl_tcp_mem[3];

  atomic_t rt_genid;
  atomic_t dev_addr_genid;
diff --git a/include/net/tcp.h b/include/net/tcp.h
index 7301ca8..c34b823 100644
--- a/include/net/tcp.h
+++ b/include/net/tcp.h
@@ -230,7 +230,6 @@ extern int sysctl_tcp_fack;
 extern int sysctl_tcp_reordering;
 extern int sysctl_tcp_ecn;
 extern int sysctl_tcp_dsack;
-extern long sysctl_tcp_mem[3];
 extern int sysctl_tcp_wmem[3];
 extern int sysctl_tcp_rmem[3];
 extern int sysctl_tcp_app_win;
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index f14d7d2..63360f8 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -395,6 +395,7 @@ static inline bool mem_cgroup_is_root(struct mem_cgroup *mem)
 #ifdef CONFIG_INET
 #include <net/sock.h>
 #include <net/ip.h>
+#include <linux/nsproxy.h>

 void sock_update_memcg(struct sock *sk)
 {
@@ -526,14 +527,15 @@ static void tcp_create_cgroup(struct mem_cgroup *cg, struct
cgroup_subsys *ss)
 int tcp_init_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
+ struct net *net = current->nsproxy->net_ns;
  /*
```

```
   * We need to initialize it at populate, not create time.
   * This is because net sysctl tables are not up until much
   * later
   */
- memcg->tcp.tcp_prot_mem[0] = sysctl_tcp_mem[0];
- memcg->tcp.tcp_prot_mem[1] = sysctl_tcp_mem[1];
- memcg->tcp.tcp_prot_mem[2] = sysctl_tcp_mem[2];
+ memcg->tcp.tcp_prot_mem[0] = net->ipv4.sysctl_tcp_mem[0];
+ memcg->tcp.tcp_prot_mem[1] = net->ipv4.sysctl_tcp_mem[1];
+ memcg->tcp.tcp_prot_mem[2] = net->ipv4.sysctl_tcp_mem[2];

  return 0;
 }
diff --git a/net/ipv4/af_inet.c b/net/ipv4/af_inet.c
index da19147..73be7da 100644
--- a/net/ipv4/af_inet.c
+++ b/net/ipv4/af_inet.c
@@ -1674,6 +1674,8 @@ static int __init inet_init(void)
  ip_static_sysctl_init();
 #endif

+ tcp_prot.sysctl_mem = init_net.ipv4.sysctl_tcp_mem;
+
  /*
   * Add all the base protocols.
   */
diff --git a/net/ipv4/sysctl_net_ipv4.c b/net/ipv4/sysctl_net_ipv4.c
index 69fd720..bbd67ab 100644
--- a/net/ipv4/sysctl_net_ipv4.c
+++ b/net/ipv4/sysctl_net_ipv4.c
@@ -14,6 +14,7 @@
 #include <linux/init.h>
 #include <linux/slab.h>
 #include <linux/nsproxy.h>
+#include <linux/swap.h>
 #include <net/snmp.h>
 #include <net/icmp.h>
 #include <net/ip.h>
@@ -174,6 +175,36 @@ static int proc_allowed_congestion_control(ctl_table *ctl,
  return ret;
 }

+static int ipv4_tcp_mem(ctl_table *ctl, int write,
+      void __user *buffer, size_t *lenp,
+      loff_t *ppos)
+{
+ int ret;
+ unsigned long vec[3];
```

```
+ struct net *net = current->nsproxy->net_ns;
+
+ ctl_table tmp = {
+  .data = &vec,
+  .maxlen = sizeof(vec),
+  .mode = ctl->mode,
+ };
+
+ if (!write) {
+  ctl->data = &net->ipv4.sysctl_tcp_mem;
+  return proc_doulongvec_minmax(ctl, write, buffer, lenp, ppos);
+ }
+
+ ret = proc_doulongvec_minmax(&tmp, write, buffer, lenp, ppos);
+ if (ret)
+  return ret;
+
+ net->ipv4.sysctl_tcp_mem[0] = vec[0];
+ net->ipv4.sysctl_tcp_mem[1] = vec[1];
+ net->ipv4.sysctl_tcp_mem[2] = vec[2];
+
+ return 0;
+}
+
 static struct ctl_table ipv4_table[] = {
  {
   .procname = "tcp_timestamps",
@@ -433,13 +464,6 @@ static struct ctl_table ipv4_table[] = {
   .proc_handler = proc_dointvec
  },
  {
-  .procname = "tcp_mem",
-  .data  = &sysctl_tcp_mem,
-  .maxlen  = sizeof(sysctl_tcp_mem),
-  .mode  = 0644,
-  .proc_handler = proc_doulongvec_minmax
- },
- {
   .procname = "tcp_wmem",
   .data  = &sysctl_tcp_wmem,
   .maxlen  = sizeof(sysctl_tcp_wmem),
@@ -721,6 +745,12 @@ static struct ctl_table ipv4_net_table[] = {
   .mode  = 0644,
   .proc_handler = ipv4_ping_group_range,
  },
+ {
+  .procname = "tcp_mem",
+  .maxlen  = sizeof(init_net.ipv4.sysctl_tcp_mem),
```

```
+  .mode  = 0644,
+  .proc_handler = ipv4_tcp_mem,
+ },
  { }
 };

@@ -734,6 +764,7 @@ EXPORT_SYMBOL_GPL(net_ipv4_ctl_path);
 static __net_init int ipv4_sysctl_init_net(struct net *net)
 {
  struct ctl_table *table;
+ unsigned long limit;

  table = ipv4_net_table;
  if (!net_eq(net, &init_net)) {
@@ -769,6 +800,12 @@ static __net_init int ipv4_sysctl_init_net(struct net *net)

  net->ipv4.sysctl_rt_cache_rebuild_count = 4;

+ limit = nr_free_buffer_pages() / 8;
+ limit = max(limit, 128UL);
+ net->ipv4.sysctl_tcp_mem[0] = limit / 4 * 3;
+ net->ipv4.sysctl_tcp_mem[1] = limit;
+ net->ipv4.sysctl_tcp_mem[2] = net->ipv4.sysctl_tcp_mem[0] * 2;
+
  net->ipv4.ipv4_hdr = register_net_sysctl_table(net,
    net_ipv4_ctl_path, table);
  if (net->ipv4.ipv4_hdr == NULL)
diff --git a/net/ipv4/tcp.c b/net/ipv4/tcp.c
index 34f5db1..5f618d1 100644
--- a/net/ipv4/tcp.c
+++ b/net/ipv4/tcp.c
@@ -282,11 +282,9 @@ int sysctl_tcp_fin_timeout __read_mostly = TCP_FIN_TIMEOUT;
 struct percpu_counter tcp_orphan_count;
 EXPORT_SYMBOL_GPL(tcp_orphan_count);

-long sysctl_tcp_mem[3] __read_mostly;
 int sysctl_tcp_wmem[3] __read_mostly;
 int sysctl_tcp_rmem[3] __read_mostly;

-EXPORT_SYMBOL(sysctl_tcp_mem);
 EXPORT_SYMBOL(sysctl_tcp_rmem);
 EXPORT_SYMBOL(sysctl_tcp_wmem);

@@ -3272,14 +3270,9 @@ void __init tcp_init(void)
  sysctl_tcp_max_orphans = cnt / 2;
  sysctl_max_syn_backlog = max(128, cnt / 256);

- limit = nr_free_buffer_pages() / 8;
```

```
- limit = max(limit, 128UL);
- sysctl_tcp_mem[0] = limit / 4 * 3;
- sysctl_tcp_mem[1] = limit;
- sysctl_tcp_mem[2] = sysctl_tcp_mem[0] * 2;
-
  /* Set per-socket limits to no more than 1/128 the pressure threshold */
- limit = ((unsigned long)sysctl_tcp_mem[1]) << (PAGE_SHIFT - 7);
+ limit = ((unsigned long)init_net.ipv4.sysctl_tcp_mem[1])
+  << (PAGE_SHIFT - 7);
  max_share = min(4UL*1024*1024, limit);

  sysctl_tcp_wmem[0] = SK_MEM_QUANTUM;
diff --git a/net/ipv4/tcp_ipv4.c b/net/ipv4/tcp_ipv4.c
index 54f6b96..dd1bab7 100644
--- a/net/ipv4/tcp_ipv4.c
+++ b/net/ipv4/tcp_ipv4.c
@@ -2616,7 +2616,6 @@ struct proto tcp_prot = {
  .orphan_count  = &tcp_orphan_count,
  .memory_allocated = &tcp_memory_allocated,
  .memory_pressure = &tcp_memory_pressure,
- .sysctl_mem  = sysctl_tcp_mem,
  .sysctl_wmem  = sysctl_tcp_wmem,
  .sysctl_rmem  = sysctl_tcp_rmem,
  .max_header  = MAX_TCP_HEADER,
diff --git a/net/ipv6/af_inet6.c b/net/ipv6/af_inet6.c
index 51672f8..69a6da3 100644
--- a/net/ipv6/af_inet6.c
+++ b/net/ipv6/af_inet6.c
@@ -1118,6 +1118,8 @@ static int __init inet6_init(void)
  if (err)
   goto static_sysctl_fail;
 #endif
+ tcpv6_prot.sysctl_mem = init_net.ipv4.sysctl_tcp_mem;
+
  /*
   * ipngwg API draft makes clear that the correct semantics
   * for TCP and UDP is to consider one TCP and UDP instance
diff --git a/net/ipv6/tcp_ipv6.c b/net/ipv6/tcp_ipv6.c
index 3c13142..52f8b64 100644
--- a/net/ipv6/tcp_ipv6.c
+++ b/net/ipv6/tcp_ipv6.c
@@ -2208,7 +2208,6 @@ struct proto tcpv6_prot = {
  .memory_allocated = &tcp_memory_allocated,
  .memory_pressure = &tcp_memory_pressure,
  .orphan_count  = &tcp_orphan_count,
- .sysctl_mem  = sysctl_tcp_mem,
  .sysctl_wmem  = sysctl_tcp_wmem,
  .sysctl_rmem  = sysctl_tcp_rmem,
```

```
  .max_header  = MAX_TCP_HEADER,
--
1.7.6.4
```

Subject: [PATCH v5 06/10] tcp buffer limitation: per-cgroup limit
Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:31 GMT
View Forum Message <> Reply to Message

This patch uses the "tcp.limit_in_bytes" field of the kmem_cgroup to
effectively control the amount of kernel memory pinned by a cgroup.

This value is ignored in the root cgroup, and in all others,
caps the value specified by the admin in the net namespaces'
view of tcp_sysctl_mem.

If namespaces are being used, the admin is allowed to set a
value bigger than cgroup's maximum, the same way it is allowed
to set pretty much unlimited values in a real box.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: David S. Miller <davem@davemloft.net>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Eric W. Biederman <ebiederm@xmission.com>

```
 Documentation/cgroups/memory.txt |    1 +
 include/linux/memcontrol.h       |    9 ++++
 mm/memcontrol.c                  |   76 ++++++++++++++++++++++++++++++++++++++-
 net/ipv4/sysctl_net_ipv4.c       |   14 +++++++
 4 files changed, 99 insertions(+), 1 deletions(-)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index bf00cd2..c1db134 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -78,6 +78,7 @@ Brief summary of control files.

  memory.independent_kmem_limit  # select whether or not kernel memory limits are
       independent of user limits
+ memory.kmem.tcp.limit_in_bytes  # set/show hard limit for tcp buf memory

 1. History

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 994a06a..d025979 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -402,10 +402,19 @@ void memcg_memory_allocated_sub(struct mem_cgroup *memcg,
```

```
struct cg_proto *prot,
     unsigned long amt);
 u64 memcg_memory_allocated_read(struct mem_cgroup *memcg,
     struct cg_proto *prot);
+unsigned long long tcp_max_memory(const struct mem_cgroup *memcg);
+void tcp_prot_mem(struct mem_cgroup *memcg, long val, int idx);
 #else
 static inline void sock_update_memcg(struct sock *sk)
 {
 }
+static inline unsigned long long tcp_max_memory(const struct mem_cgroup *memcg)
+{
+ return 0;
+}
+static inline void tcp_prot_mem(struct mem_cgroup *memcg, long val, int idx)
+{
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
 #endif /* CONFIG_INET */
 #endif /* _LINUX_MEMCONTROL_H */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 63360f8..ee122a6 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -365,6 +365,7 @@ enum mem_type {
  _MEMSWAP,
  _OOM_TYPE,
  _KMEM,
+ _KMEM_TCP,
 };

 #define MEMFILE_PRIVATE(x, val) (((x) << 16) | (val))
@@ -500,6 +501,35 @@ struct percpu_counter *sockets_allocated_tcp(struct mem_cgroup
*memcg)
 }
 EXPORT_SYMBOL(sockets_allocated_tcp);

+static void tcp_update_limit(struct mem_cgroup *memcg, u64 val)
+{
+ struct net *net = current->nsproxy->net_ns;
+ int i;
+
+ val >>= PAGE_SHIFT;
+
+ for (i = 0; i < 3; i++)
+  memcg->tcp.tcp_prot_mem[i]  = min_t(long, val,
+         net->ipv4.sysctl_tcp_mem[i]);
+}
```

```
+
+static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
+       const char *buffer);
+
+static u64 mem_cgroup_read(struct cgroup *cont, struct cftype *cft);
+/*
+ * We need those things internally in pages, so don't reuse
+ * mem_cgroup_{read,write}
+ */
+static struct cftype tcp_files[] = {
+ {
+  .name = "kmem.tcp.limit_in_bytes",
+  .write_string = mem_cgroup_write,
+  .read_u64 = mem_cgroup_read,
+  .private = MEMFILE_PRIVATE(_KMEM_TCP, RES_LIMIT),
+ },
+};
+
 static void tcp_create_cgroup(struct mem_cgroup *cg, struct cgroup_subsys *ss)
 {
  /*
@@ -527,6 +557,7 @@ static void tcp_create_cgroup(struct mem_cgroup *cg, struct
cgroup_subsys *ss)
 int tcp_init_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
+ struct mem_cgroup *parent = parent_mem_cgroup(memcg);
  struct net *net = current->nsproxy->net_ns;
  /*
   * We need to initialize it at populate, not create time.
@@ -537,7 +568,20 @@ int tcp_init_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss)
  memcg->tcp.tcp_prot_mem[1] = net->ipv4.sysctl_tcp_mem[1];
  memcg->tcp.tcp_prot_mem[2] = net->ipv4.sysctl_tcp_mem[2];

- return 0;
+ /* Let root cgroup unlimited. All others, respect parent's if needed */
+ if (parent && !parent->use_hierarchy) {
+  unsigned long limit;
+  int ret;
+  limit = nr_free_buffer_pages() / 8;
+  limit = max(limit, 128UL);
+  ret = res_counter_set_limit(&memcg->tcp.tcp_memory_allocated,
+         limit * 2);
+  if (ret)
+   return ret;
+ }
+
+ return cgroup_add_files(cgrp, ss, tcp_files,
```

```
+    ARRAY_SIZE(tcp_files));
 }
 EXPORT_SYMBOL(tcp_init_cgroup);

@@ -548,6 +592,18 @@ void tcp_destroy_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss)
  percpu_counter_destroy(&memcg->tcp.tcp_sockets_allocated);
 }
 EXPORT_SYMBOL(tcp_destroy_cgroup);
+
+unsigned long long tcp_max_memory(const struct mem_cgroup *memcg)
+{
+ struct mem_cgroup *cmemcg = (struct mem_cgroup *)memcg;
+ return res_counter_read_u64(&cmemcg->tcp.tcp_memory_allocated,
+        RES_LIMIT);
+}
+
+void tcp_prot_mem(struct mem_cgroup *memcg, long val, int idx)
+{
+ memcg->tcp.tcp_prot_mem[idx] = val;
+}
 #endif /* CONFIG_INET */
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

@@ -4067,6 +4123,15 @@ static u64 mem_cgroup_read(struct cgroup *cont, struct cftype *cft)
   val = res_counter_read_u64(&mem->kmem, name);
   break;

+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_INET)
+ case _KMEM_TCP:
+  /* Be explicit: tcp root does not have a res_counter */
+  if (mem_cgroup_is_root(mem))
+   val = RESOURCE_MAX;
+  else
+   val = res_counter_read_u64(&mem->tcp.tcp_memory_allocated, name);
+  break;
+#endif
  default:
   BUG();
   break;
@@ -4099,6 +4164,15 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    break;
   if (type == _MEM)
    ret = mem_cgroup_resize_limit(memcg, val);
+
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_INET)
+  else if (type == _KMEM_TCP) {
+   ret = res_counter_set_limit(&memcg->tcp.tcp_memory_allocated,
+          val);
```

```
+   if (!ret)
+    tcp_update_limit(memcg, val);
+  }
+#endif
   else
    ret = mem_cgroup_resize_memsw_limit(memcg, val);
   break;
diff --git a/net/ipv4/sysctl_net_ipv4.c b/net/ipv4/sysctl_net_ipv4.c
index bbd67ab..915e192 100644
--- a/net/ipv4/sysctl_net_ipv4.c
+++ b/net/ipv4/sysctl_net_ipv4.c
@@ -14,6 +14,7 @@
 #include <linux/init.h>
 #include <linux/slab.h>
 #include <linux/nsproxy.h>
+#include <linux/memcontrol.h>
 #include <linux/swap.h>
 #include <net/snmp.h>
 #include <net/icmp.h>
@@ -182,6 +183,9 @@ static int ipv4_tcp_mem(ctl_table *ctl, int write,
  int ret;
  unsigned long vec[3];
  struct net *net = current->nsproxy->net_ns;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct mem_cgroup *cg;
+#endif

  ctl_table tmp = {
   .data = &vec,
@@ -198,6 +202,16 @@ static int ipv4_tcp_mem(ctl_table *ctl, int write,
  if (ret)
   return ret;

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ rcu_read_lock();
+ cg = mem_cgroup_from_task(current);
+
+ tcp_prot_mem(cg, vec[0], 0);
+ tcp_prot_mem(cg, vec[1], 1);
+ tcp_prot_mem(cg, vec[2], 2);
+ rcu_read_unlock();
+#endif
+
  net->ipv4.sysctl_tcp_mem[0] = vec[0];
  net->ipv4.sysctl_tcp_mem[1] = vec[1];
  net->ipv4.sysctl_tcp_mem[2] = vec[2];
--
1.7.6.4
```

Subject: [PATCH v5 07/10] Display current tcp memory allocation in kmem cgroup
Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:32 GMT
View Forum Message <> Reply to Message

This patch introduces kmem.tcp.usage_in_bytes file, living in the
kmem_cgroup filesystem. It is a simple read-only file that displays the
amount of kernel memory currently consumed by the cgroup.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: David S. Miller <davem@davemloft.net>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Eric W. Biederman <ebiederm@xmission.com>

 Documentation/cgroups/memory.txt | 1 +
 mm/memcontrol.c | 14 ++++++++++---
 2 files changed, 12 insertions(+), 3 deletions(-)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index c1db134..00f1a88 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -79,6 +79,7 @@ Brief summary of control files.
  memory.independent_kmem_limit # select whether or not kernel memory limits are
       independent of user limits
  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

 1. History

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index ee122a6..51b5a55 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -528,6 +528,11 @@ static struct cftype tcp_files[] = {
   .read_u64 = mem_cgroup_read,
   .private = MEMFILE_PRIVATE(_KMEM_TCP, RES_LIMIT),
  },
+ {
+ .name = "kmem.tcp.usage_in_bytes",
+ .read_u64 = mem_cgroup_read,
+ .private = MEMFILE_PRIVATE(_KMEM_TCP, RES_USAGE),
+ },
 };

 static void tcp_create_cgroup(struct mem_cgroup *cg, struct cgroup_subsys *ss)
@@ -4126,9 +4131,12 @@ static u64 mem_cgroup_read(struct cgroup *cont, struct cftype *cft)
 #if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_INET)
  case _KMEM_TCP:
   /* Be explicit: tcp root does not have a res_counter */

```
-  if (mem_cgroup_is_root(mem))
-   val = RESOURCE_MAX;
-  else
+  if (mem_cgroup_is_root(mem)) {
+   if (name == RES_USAGE)
+    val = atomic_long_read(&tcp_memory_allocated) << PAGE_SHIFT;
+   else
+    val = RESOURCE_MAX;
+  } else
    val = res_counter_read_u64(&mem->tcp.tcp_memory_allocated, name);
   break;
 #endif
--
1.7.6.4
```

Subject: [PATCH v5 08/10] Display current tcp memory allocation in kmem cgroup
Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:33 GMT
View Forum Message <> Reply to Message

This patch introduces kmem.tcp.failcnt file, living in the
kmem_cgroup filesystem. Following the pattern in the other
memcg resources, this files keeps a counter of how many times
allocation failed due to limits being hit in this cgroup.
The root cgroup will always show a failcnt of 0.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: David S. Miller <davem@davemloft.net>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Eric W. Biederman <ebiederm@xmission.com>
---
 mm/memcontrol.c | 13 ++++++++++++
 1 files changed, 13 insertions(+), 0 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 51b5a55..9394224 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -517,6 +517,7 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
        const char *buffer);

 static u64 mem_cgroup_read(struct cgroup *cont, struct cftype *cft);
+static int mem_cgroup_reset(struct cgroup *cont, unsigned int event);
 /*
  * We need those things internally in pages, so don't reuse
  * mem_cgroup_{read,write}
@@ -533,6 +534,12 @@ static struct cftype tcp_files[] = {
   .read_u64 = mem_cgroup_read,
```

```
   .private = MEMFILE_PRIVATE(_KMEM_TCP, RES_USAGE),
  },
+ {
+  .name = "kmem.tcp.failcnt",
+  .private = MEMFILE_PRIVATE(_KMEM_TCP, RES_FAILCNT),
+  .trigger = mem_cgroup_reset,
+  .read_u64 = mem_cgroup_read,
+ },
 };

 static void tcp_create_cgroup(struct mem_cgroup *cg, struct cgroup_subsys *ss)
@@ -4134,6 +4141,8 @@ static u64 mem_cgroup_read(struct cgroup *cont, struct cftype *cft)
   if (mem_cgroup_is_root(mem)) {
    if (name == RES_USAGE)
     val = atomic_long_read(&tcp_memory_allocated) << PAGE_SHIFT;
+   else if (name == RES_FAILCNT)
+    val = 0;
    else
     val = RESOURCE_MAX;
   } else
@@ -4251,6 +4260,10 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int event)
  case RES_FAILCNT:
   if (type == _MEM)
    res_counter_reset_failcnt(&mem->res);
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_INET)
+  else if (type == _KMEM_TCP)
+   res_counter_reset_failcnt(&mem->tcp.tcp_memory_allocated);
+#endif
   else
    res_counter_reset_failcnt(&mem->memsw);
   break;
--
1.7.6.4
```

Subject: [PATCH v5 09/10] Display current tcp memory allocation in kmem cgroup
Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:34 GMT
View Forum Message <> Reply to Message

This patch introduces kmem.tcp.max_usage_in_bytes file, living in the
kmem_cgroup filesystem. The root cgroup will display a value equal
to RESOURCE_MAX. This is to avoid introducing any locking schemes in
the network paths when cgroups are not being actively used.

All others, will see the maximum memory ever used by this cgroup.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: David S. Miller <davem@davemloft.net>

```
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Eric W. Biederman <ebiederm@xmission.com>
---
 mm/memcontrol.c |   10 ++++++++++
 1 files changed, 10 insertions(+), 0 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 9394224..b532f91 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -540,6 +540,12 @@ static struct cftype tcp_files[] = {
   .trigger = mem_cgroup_reset,
   .read_u64 = mem_cgroup_read,
  },
+ {
+  .name = "kmem.tcp.max_usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM_TCP, RES_MAX_USAGE),
+  .trigger = mem_cgroup_reset,
+  .read_u64 = mem_cgroup_read,
+ },
 };

 static void tcp_create_cgroup(struct mem_cgroup *cg, struct cgroup_subsys *ss)
@@ -4254,6 +4260,10 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int event)
  case RES_MAX_USAGE:
   if (type == _MEM)
    res_counter_reset_max(&mem->res);
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_INET)
+  else if (type == _KMEM_TCP)
+   res_counter_reset_max(&mem->tcp.tcp_memory_allocated);
+#endif
   else
    res_counter_reset_max(&mem->memsw);
   break;
--
1.7.6.4
```

Subject: [PATCH v5 10/10] Disable task moving when using kernel memory accounting

Posted by Glauber Costa on Mon, 07 Nov 2011 15:26:35 GMT

View Forum Message <> Reply to Message

Since this code is still experimental, we are leaving the exact details of how to move tasks between cgroups when kernel memory accounting is used as future work.

For now, we simply disallow movement if there are any pending

accounted memory.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
---
 mm/memcontrol.c | 23 ++++++++++++++++++++-
 1 files changed, 22 insertions(+), 1 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b532f91..248f92d 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5690,10 +5690,19 @@ static int mem_cgroup_can_attach(struct cgroup_subsys *ss,
 {
  int ret = 0;
  struct mem_cgroup *mem = mem_cgroup_from_cont(cgroup);
+ struct mem_cgroup *from = mem_cgroup_from_task(p);
+
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_INET)
+ if (from != mem && !mem_cgroup_is_root(from) &&
+     res_counter_read_u64(&from->tcp.tcp_memory_allocated, RES_USAGE)) {
+  printk(KERN_WARNING "Can't move tasks between cgroups: "
+   "Kernel memory held. task: %s\n", p->comm);
+  return 1;
+ }
+#endif

  if (mem->move_charge_at_immigrate) {
   struct mm_struct *mm;
-  struct mem_cgroup *from = mem_cgroup_from_task(p);

   VM_BUG_ON(from == mem);

@@ -5861,6 +5870,18 @@ static int mem_cgroup_can_attach(struct cgroup_subsys *ss,
     struct cgroup *cgroup,
     struct task_struct *p)
 {
+ struct mem_cgroup *mem = mem_cgroup_from_cont(cgroup);
+ struct mem_cgroup *from = mem_cgroup_from_task(p);
+
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_INET)
+ if (from != mem && !mem_cgroup_is_root(from) &&
+     res_counter_read_u64(&from->tcp.tcp_memory_allocated, RES_USAGE)) {
+  printk(KERN_WARNING "Can't move tasks between cgroups: "
+   "Kernel memory held. task: %s\n", p->comm);
+  return 1;
+ }
+#endif
```

```
+
  return 0;
 }
 static void mem_cgroup_cancel_attach(struct cgroup_subsys *ss,
--
1.7.6.4
```

Subject: RE: [PATCH v5 04/10] per-cgroup tcp buffers control
Posted by Glauber Costa on Mon, 07 Nov 2011 17:28:50 GMT
View Forum Message <> Reply to Message

Ok, I forgot to change the temporary name I was using for the jump label. Shame on me :)

--- Mensagem Original ---

De: Glauber Costa <glommer@parallels.com>
Enviado: 7 de novembro de 2011 07/11/11
Para: linux-kernel@vger.kernel.org
Cc: paul@paulmenage.org, lizf@cn.fujitsu.com, kamezawa.hiroyu@jp.fujitsu.com, ebiederm@xmission.com, davem@davemloft.net, gthelen@google.com, netdev@vger.kernel.org, linux-mm@kvack.org, kirill@shutemov.name, Andrey Vagin <avagin@parallels.com>, devel@openvz.org, eric.dumazet@gmail.com, Glauber Costa <glommer@parallels.com>, KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujtisu.com>
Assunto: [PATCH v5 04/10] per-cgroup tcp buffers control

With all the infrastructure in place, this patch implements
per-cgroup control for tcp memory pressure handling.

A resource conter is used to control allocated memory, except
for the root cgroup, that will keep using global counters.

This patch is the one that actually enables/disables the
jump labels controlling cgroup. To this point, they were always
disabled.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujtisu.com>
CC: David S. Miller <davem@davemloft.net>
CC: Eric W. Biederman <ebiederm@xmission.com>
CC: Eric Dumazet <eric.dumazet@gmail.com>

```
 include/net/tcp.h       |   18 +++++++
 include/net/transp_v6.h |    1 +
 mm/memcontrol.c         |  125 ++++++++++++++++++++++++++++++++++++++++++++++-
 net/core/sock.c         |   46 ++++++++++++++---
 net/ipv4/af_inet.c      |    3 +
 net/ipv4/tcp_ipv4.c     |   12 +++++
```

```
 net/ipv6/af_inet6.c     |    3 +
 net/ipv6/tcp_ipv6.c     |   10 ++++
 8 files changed, 211 insertions(+), 7 deletions(-)

diff --git a/include/net/tcp.h b/include/net/tcp.h
index ccaa3b6..7301ca8 100644
--- a/include/net/tcp.h
+++ b/include/net/tcp.h
@@ -253,6 +253,22 @@ extern int sysctl_tcp_cookie_size;
 extern int sysctl_tcp_thin_linear_timeouts;
 extern int sysctl_tcp_thin_dupack;

+struct tcp_memcontrol {
+ /* per-cgroup tcp memory pressure knobs */
+ struct res_counter tcp_memory_allocated;
+ struct percpu_counter tcp_sockets_allocated;
+ /* those two are read-mostly, leave them at the end */
+ long tcp_prot_mem[3];
+ int tcp_memory_pressure;
+};
+
+long *sysctl_mem_tcp(struct mem_cgroup *memcg);
+struct percpu_counter *sockets_allocated_tcp(struct mem_cgroup *memcg);
+int *memory_pressure_tcp(struct mem_cgroup *memcg);
+struct res_counter *memory_allocated_tcp(struct mem_cgroup *memcg);
+int tcp_init_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss);
+void tcp_destroy_cgroup(struct cgroup *cgrp, struct cgroup_subsys *ss);
+
 extern atomic_long_t tcp_memory_allocated;
 extern struct percpu_counter tcp_sockets_allocated;
 extern int tcp_memory_pressure;
@@ -305,6 +321,7 @@ static inline int tcp_synq_no_recent_overflow(const struct sock *sk)
 }

 extern struct proto tcp_prot;
+extern struct cg_proto tcp_cg_prot;

 #define TCP_INC_STATS(net, field) SNMP_INC_STATS((net)->mib.tcp_statistics, field)
 #define TCP_INC_STATS_BH(net, field) SNMP_INC_STATS_BH((net)->mib.tcp_statistics, field)
@@ -1022,6 +1039,7 @@ static inline void tcp_openreq_init(struct request_sock *req,
  ireq->loc_port = tcp_hdr(skb)->dest;
 }

+extern void tcp_enter_memory_pressure_cg(struct sock *sk);
 extern void tcp_enter_memory_pressure(struct sock *sk);

 static inline int keepalive_intvl_when(const struct tcp_sock *tp)
diff --git a/include/net/transp_v6.h b/include/net/transp_v6.h
```

```
index 498433d..1e18849 100644
--- a/include/net/transp_v6.h
+++ b/include/net/transp_v6.h
@@ -11,6 +11,7 @@ extern struct proto rawv6_prot;
 extern struct proto udpv6_prot;
 extern struct proto udplitev6_prot;
 extern struct proto tcpv6_prot;
+extern struct cg_proto tcpv6_cg_prot;

 struct flowi6;

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 7d684d0..f14d7d2 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -49,6 +49,9 @@
 #include <linux/cpu.h>
 #include <linux/oom.h>
 #include "internal.h"
+#ifdef CONFIG_INET
+#include <net/tcp.h>
+#endif

 #include <asm/uaccess.h>

@@ -294,6 +297,10 @@ struct mem_cgroup {
   */
  struct mem_cgroup_stat_cpu nocpu_base;
  spinlock_t pcp_counter_lock;
+
+#ifdef CONFIG_INET
+ struct tcp_memcontrol tcp;
+#endif
 };

 /* Stuffs for move charges at task migration. */
@@ -377,7 +384,7 @@ enum mem_type {
 #define MEM_CGROUP_RECLAIM_SOFT  (1 << MEM_CGROUP_RECLAIM_SOFT_BIT)

 static struct mem_cgroup *parent_mem_cgroup(struct mem_cgroup *memcg);
-
+static struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont);
 static inline bool mem_cgroup_is_root(struct mem_cgroup *mem)
 {
  return (mem == root_mem_cgroup);
@@ -387,6 +394,7 @@ static inline bool mem_cgroup_is_root(struct mem_cgroup *mem)
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
 #ifdef CONFIG_INET
```

```
 #include <net/sock.h>
+#include <net/ip.h>

 void sock_update_memcg(struct sock *sk)
 {
@@ -451,6 +459,93 @@ u64 memcg_memory_allocated_read(struct mem_cgroup *memcg,
struct cg_proto *prot)
        RES_USAGE) >> PAGE_SHIFT ;
 }
 EXPORT_SYMBOL(memcg_memory_allocated_read);
+/*
+ * Pressure flag: try to collapse.
+ * Technical note: it is used by multiple contexts non atomically.
+ * All the __sk_mem_schedule() is of this nature: accounting
+ * is strict, actions are advisory and have some latency.
+ */
+void tcp_enter_memory_pressure_cg(struct sock *sk)
+{
+ struct mem_cgroup *memcg = sk->sk_cgrp;
+ if (!memcg->tcp.tcp_memory_pressure) {
+  NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPMEMORYPRESSURES);
+  memcg->tcp.tcp_memory_pressure = 1;
+ }
+}
+EXPORT_SYMBOL(tcp_enter_memory_pressure_cg);
+
+long *sysctl_mem_tcp(struct mem_cgroup *memcg)
+{
+ return memcg-
```

Subject: Re: [PATCH v5 00/10] per-cgroup tcp memory pressure
Posted by Glauber Costa on Wed, 09 Nov 2011 18:02:55 GMT
View Forum Message <> Reply to Message

On 11/07/2011 01:26 PM, Glauber Costa wrote:
> Hi all,
>
> This is my new attempt at implementing per-cgroup tcp memory pressure.
> I am particularly interested in what the network folks have to comment on
> it: my main goal is to achieve the least impact possible in the network code.
>
> Here's a brief description of my approach:
>
> When only the root cgroup is present, the code should behave the same way as
> before - with the exception of the inclusion of an extra field in struct sock,
> and one in struct proto. All tests are patched out with static branch, and we
> still access addresses directly - the same as we did before.

>
> When a cgroup other than root is created, we patch in the branches, and account
> resources for that cgroup. The variables in the root cgroup are still updated.
> If we were to try to be 100 % coherent with the memcg code, that should depend
> on use_hierarchy. However, I feel that this is a good compromise in terms of
> leaving the network code untouched, and still having a global vision of its
> resources. I also do not compute max_usage for the root cgroup, for a similar
> reason.
>
> Please let me know what you think of it.

Dave, Eric,

Can you let me know what you think of the general approach I've followed
in this series? The impact on the common case should be minimal, or at
least as expensive as a static branch (0 in most arches, I believe).

I am mostly interested in knowing if this a valid pursue path. I'll be
happy to address any specific concerns you have once you're ok with the
general approach.

Thanks!

Subject: Re: Re: [PATCH v5 00/10] per-cgroup tcp memory pressure
Posted by James Bottomley on Tue, 15 Nov 2011 18:27:12 GMT
View Forum Message <> Reply to Message

On Wed, 2011-11-09 at 16:02 -0200, Glauber Costa wrote:
> On 11/07/2011 01:26 PM, Glauber Costa wrote:
> > Hi all,
> >
> > This is my new attempt at implementing per-cgroup tcp memory pressure.
> > I am particularly interested in what the network folks have to comment on
> > it: my main goal is to achieve the least impact possible in the network code.
> >
> > Here's a brief description of my approach:
> >
> > When only the root cgroup is present, the code should behave the same way as
> > before - with the exception of the inclusion of an extra field in struct sock,
> > and one in struct proto. All tests are patched out with static branch, and we
> > still access addresses directly - the same as we did before.
> >
> > When a cgroup other than root is created, we patch in the branches, and account
> > resources for that cgroup. The variables in the root cgroup are still updated.
> > If we were to try to be 100 % coherent with the memcg code, that should depend
> > on use_hierarchy. However, I feel that this is a good compromise in terms of
> > leaving the network code untouched, and still having a global vision of its

> > resources. I also do not compute max_usage for the root cgroup, for a similar
> > reason.
> >
> > Please let me know what you think of it.
>
> Dave, Eric,
>
> Can you let me know what you think of the general approach I've followed
> in this series? The impact on the common case should be minimal, or at
> least as expensive as a static branch (0 in most arches, I believe).
>
> I am mostly interested in knowing if this a valid pursue path. I'll be
> happy to address any specific concerns you have once you're ok with the
> general approach.

Ping on this, please. We're blocked on this patch set until we can get
an ack that the approach is acceptable to network people.

Thanks,

James

Subject: Re: Re: [PATCH v5 00/10] per-cgroup tcp memory pressure
Posted by davem on Thu, 17 Nov 2011 21:35:01 GMT
View Forum Message <> Reply to Message

From: James Bottomley <jbottomley@parallels.com>
Date: Tue, 15 Nov 2011 18:27:12 +0000

> Ping on this, please. We're blocked on this patch set until we can get
> an ack that the approach is acceptable to network people.

__sk_mem_schedule is now more expensive, because instead of short-circuiting
the majority of the function's logic when "allocated <= prot->sysctl_mem[0]"
and immediately returning 1, the whole rest of the function is run.

The static branch protecting all of the cgroup code seems to be
enabled if any memory based cgroup'ing is enabled. What if people use
the memory cgroup facility but not for sockets? I am to understand
that, of the very few people who are going to use this stuff in any
capacity, this would be a common usage.

TCP specific stuff in mm/memcontrol.c, at best that's not nice at all.

Otherwise looks mostly good.

Subject: Re: Re: [PATCH v5 00/10] per-cgroup tcp memory pressure
Posted by Glauber Costa on Fri, 18 Nov 2011 19:39:03 GMT
View Forum Message <> Reply to Message

On 11/17/2011 07:35 PM, David Miller wrote:
> From: James Bottomley<jbottomley@parallels.com>
> Date: Tue, 15 Nov 2011 18:27:12 +0000
>
>> Ping on this, please. We're blocked on this patch set until we can get
>> an ack that the approach is acceptable to network people.
>
> __sk_mem_schedule is now more expensive, because instead of short-circuiting
> the majority of the function's logic when "allocated<= prot->sysctl_mem[0]"
> and immediately returning 1, the whole rest of the function is run.

Not the whole rest of the function. Rather, just the other two tests.
But that's the behavior we need since if your parent is on pressure, you
should be as well. How do you feel if we'd also provide two versions for
this:
1) non-cgroup, try to return 1 as fast as we can
2) cgroup, also check your parents.

That could be enclosed in the same static branch we're using right now.

> The static branch protecting all of the cgroup code seems to be
> enabled if any memory based cgroup'ing is enabled. What if people use
> the memory cgroup facility but not for sockets? I am to understand
> that, of the very few people who are going to use this stuff in any
> capacity, this would be a common usage.

How about we make the jump_label only used for sockets (which is basic
what we have now, just need a clear name to indicate that), and then
enable it not when the first non-root cgroup is created, but when the
first one sets the limit to something different than unlimited?

Of course to that point, we'd be accounting only to the root structures,
but I guess this is not a big deal.

> TCP specific stuff in mm/memcontrol.c, at best that's not nice at all.

How crucial is that? Thing is that as far as I am concerned, all the
memcg people really want the inner layout of struct mem_cgroup to be
private to memcontrol.c This means that at some point, we need to have
at least a wrapper in memcontrol.c that is able to calculate the offset
of the tcp structure, and since most functions are actually quite
simple, that would just make us do more function calls.

Well, an alternative to that would be to use a void pointer in the newly
added struct cg_proto to an already parsed memcg-related field

(in this case tcp_memcontrol), that would be passed to the functions
instead of the whole memcg structure. Do you think this would be
preferable ?

> Otherwise looks mostly good.

Thank you for your time.

Subject: Re: Re: [PATCH v5 00/10] per-cgroup tcp memory pressure
Posted by davem on Fri, 18 Nov 2011 19:51:07 GMT
View Forum Message <> Reply to Message

From: Glauber Costa <glommer@parallels.com>
Date: Fri, 18 Nov 2011 17:39:03 -0200

> On 11/17/2011 07:35 PM, David Miller wrote:
>> From: James Bottomley<jbottomley@parallels.com>
>> Date: Tue, 15 Nov 2011 18:27:12 +0000
>>
>>> Ping on this, please. We're blocked on this patch set until we can
>>> get
>>> an ack that the approach is acceptable to network people.
>>
>> __sk_mem_schedule is now more expensive, because instead of
>> short-circuiting
>> the majority of the function's logic when "allocated<=
>> prot->sysctl_mem[0]"
>> and immediately returning 1, the whole rest of the function is run.
>
> Not the whole rest of the function. Rather, just the other two
> tests. But that's the behavior we need since if your parent is on
> pressure, you should be as well. How do you feel if we'd also provide
> two versions for this:
> 1) non-cgroup, try to return 1 as fast as we can
> 2) cgroup, also check your parents.

Fair enough.

> How about we make the jump_label only used for sockets (which is basic
> what we have now, just need a clear name to indicate that), and then
> enable it not when the first non-root cgroup is created, but when the
> first one sets the limit to something different than unlimited?
>
> Of course to that point, we'd be accounting only to the root
> structures,
> but I guess this is not a big deal.

This sounds good for now.

>> TCP specific stuff in mm/memcontrol.c, at best that's not nice at all.
>
> How crucial is that?

It's a big deal. We've been working for years to yank protocol specific
things even out of net/core/*.c, it simply doesn't belong there.

I'd even be happier if you had to create a net/ipv4/tcp_memcg.c and
include/net/tcp_memcg.h

> Thing is that as far as I am concerned, all the
> memcg people
 ...

What the memcg people want is entirely their problem, especially if it
involves crapping up non-networking files with protocol specific junk.

Subject: Re: Re: [PATCH v5 00/10] per-cgroup tcp memory pressure
Posted by KAMEZAWA Hiroyuki on Tue, 22 Nov 2011 02:07:07 GMT
View Forum Message <> Reply to Message

On Fri, 18 Nov 2011 17:39:03 -0200
Glauber Costa <glommer@parallels.com> wrote:

> On 11/17/2011 07:35 PM, David Miller wrote:
> > TCP specific stuff in mm/memcontrol.c, at best that's not nice at all.
>
> How crucial is that? Thing is that as far as I am concerned, all the
> memcg people really want the inner layout of struct mem_cgroup to be
> private to memcontrol.c

This is just because memcg is just related to memory management and I don't
want it be wide spreaded, 'struct mem_cgroup' has been changed often.

But I don't like to have TCP code in memcgroup.c.

New idea is welcome.

> This means that at some point, we need to have
> at least a wrapper in memcontrol.c that is able to calculate the offset
> of the tcp structure, and since most functions are actually quite
> simple, that would just make us do more function calls.
>
> Well, an alternative to that would be to use a void pointer in the newly
> added struct cg_proto to an already parsed memcg-related field

> (in this case tcp_memcontrol), that would be passed to the functions
> instead of the whole memcg structure. Do you think this would be
> preferable ?
>
like this ?

```
struct mem_cgroup_sub_controls {
 struct mem_cgroup *mem;
 union {
  struct tcp_mem_control tcp;
 } data;
};
/* for loosely coupled controls for memcg */
struct memcg_sub_controls_function
{
 struct memcg_sub_controls (*create)(struct mem_cgroup *);
 struct memcg_sub_controls (*destroy)(struct mem_cgroup *);
}

int register_memcg_sub_controls(char *name,
  struct memcg_sub_controls_function *abis);


struct mem_cgroup {
 .....
 .....
 /* Root memcg will have no sub_controls! */
 struct memcg_sub_controls *sub_controls[NR_MEMCG_SUB_CONTROLS];
}
```

Maybe some functions should be exported.

Thanks,
-Kame

Subject: Re: Re: [PATCH v5 00/10] per-cgroup tcp memory pressure
Posted by Glauber Costa on Wed, 23 Nov 2011 10:25:17 GMT
View Forum Message <> Reply to Message

On 11/22/2011 12:07 AM, KAMEZAWA Hiroyuki wrote:
> On Fri, 18 Nov 2011 17:39:03 -0200
> Glauber Costa<glommer@parallels.com> wrote:
>
>> On 11/17/2011 07:35 PM, David Miller wrote:
>>> TCP specific stuff in mm/memcontrol.c, at best that's not nice at all.
>>

>> How crucial is that? Thing is that as far as I am concerned, all the
>> memcg people really want the inner layout of struct mem_cgroup to be
>> private to memcontrol.c
>
> This is just because memcg is just related to memory management and I don't
> want it be wide spreaded, 'struct mem_cgroup' has been changed often.
>
> But I don't like to have TCP code in memcgroup.c.
>
> New idea is welcome.
I don't like it either, but it seemed like an acceptable idea when I
first wrote it, compared to the options.
But I'm happy it was strongly pointed out, I gave this some extra
thought, and managed to come up with a more elegant solution for this.
Unfortunately, right now I still have some very small tcp glue code I am
trying to get rid of - maybe the suggestion you give below can be used

>> This means that at some point, we need to have
>> at least a wrapper in memcontrol.c that is able to calculate the offset
>> of the tcp structure, and since most functions are actually quite
>> simple, that would just make us do more function calls.
>>
>> Well, an alternative to that would be to use a void pointer in the newly
>> added struct cg_proto to an already parsed memcg-related field
>> (in this case tcp_memcontrol), that would be passed to the functions
>> instead of the whole memcg structure. Do you think this would be
>> preferable ?
>>
In the patch I have now, I changed cg_proto's role a bit. It now have
pointers to the variables directly as well, and a parent pointer. It
allows much of the code to be simplified, and the remainder to live
outside memcontrol.c without major problems.

But I still need a tcp control structure and a method to calculate its
address from a mem_cgroup structure.

> like this ?
>
> struct mem_cgroup_sub_controls {
> struct mem_cgroup *mem;
> union {
>   struct tcp_mem_control tcp;
> } data;
> };
> /* for loosely coupled controls for memcg */
> struct memcg_sub_controls_function
> {
> struct memcg_sub_controls (*create)(struct mem_cgroup *);

> struct memcg_sub_controls (*destroy)(struct mem_cgroup *);
> }
>
> int register_memcg_sub_controls(char *name,
> struct memcg_sub_controls_function *abis);
>
>
> struct mem_cgroup {
> .....
> .....
> /* Root memcg will have no sub_controls! */
> struct memcg_sub_controls *sub_controls[NR_MEMCG_SUB_CONTROLS];
> }
>
>
> Maybe some functions should be exported.
>
> Thanks,
> -Kame

This has the disadvantage that we now have to chase one more pointer
(for sub_controls) instead of just accessing tcp_memcontrol directly as in

```
struct mem_cgroup {
 ...
 struct tcp_memcontrol tcp;
};
```

I see how it would be nice to get rid of all tcp references, but I don't
think the above is worse that bundling struct tcp_memcontrol in an union.
We also now have to allocate mem_cgroup_subcontrols separately, creating
another opportunity of failure (that's not the end of the world, but
allocating it all inside memcg is still preferred, IMHO)

This could work if it was more generic than that, with a data pointer,
instead of an union. But then we now have 3 pointers to chase.
My worry here is that we'll end up with a performance a lot worse than
the raw network. Well, of course we'll be worse, but in the end of the
day, we also want to be as fast as we can.

I wonder how bad it is to have just a tiny glue that calculates the
address of the tcp structure in memcontrol.c, and then every other user
is passed that structure?