Subject: [PATCH v8 3/9] socket: initial cgroup code.
Posted by Glauber Costa on Mon, 05 Dec 2011 21:34:57 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>
CC: Kirill A. Shutemov <kirill@shutemov.name>
CC: 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>

 Documentation/cgroups/memory.txt | 4 +-
 include/linux/memcontrol.h | 22 ++++++
 include/net/sock.h | 151 ++++++++++++++++++++++++++++++++++++--
 mm/memcontrol.c | 46 ++++++++++-
 net/core/sock.c | 24 ++++--
 5 files changed, 230 insertions(+), 17 deletions(-)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index f245324..23a8dc5 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -289,7 +289,9 @@ to trigger slab reclaim when those limits are reached.

 2.7.1 Current Kernel Memory resources accounted

-None
+* sockets memory pressure: some sockets protocols have memory pressure
+thresholds. The Memory Controller allows them to be controlled individually
+per cgroup, instead of globally.

 3. User Interface

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h

```
index b87068a..f15021b 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -85,6 +85,8 @@ extern struct mem_cgroup *try_get_mem_cgroup_from_page(struct page
*page);
 extern struct mem_cgroup *mem_cgroup_from_task(struct task_struct *p);
 extern struct mem_cgroup *try_get_mem_cgroup_from_mm(struct mm_struct *mm);

+extern struct mem_cgroup *parent_mem_cgroup(struct mem_cgroup *memcg);
+
 static inline
 int mm_match_cgroup(const struct mm_struct *mm, const struct mem_cgroup *cgroup)
 {
@@ -381,5 +383,25 @@ mem_cgroup_print_bad_page(struct page *page)
 }
 #endif

+#ifdef CONFIG_INET
+enum {
+ UNDER_LIMIT,
+ SOFT_LIMIT,
+ OVER_LIMIT,
+};
+
+struct sock;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+void sock_update_memcg(struct sock *sk);
+void sock_release_memcg(struct sock *sk);
+#else
+static inline void sock_update_memcg(struct sock *sk)
+{
+}
+static inline void sock_release_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 5f43fd9..b8a63f8 100644
--- a/include/net/sock.h
+++ b/include/net/sock.h
@@ -54,6 +54,7 @@
 #include <linux/slab.h>
 #include <linux/uaccess.h>
 #include <linux/memcontrol.h>
+#include <linux/res_counter.h>
```

```
 #include <linux/filter.h>
 #include <linux/rculist_nulls.h>
@@ -168,6 +169,7 @@ struct sock_common {
  /* public: */
 };

+struct cg_proto;
 /**
   * struct sock - network layer representation of sockets
   * @__sk_common: shared layout with inet_timewait_sock
@@ -228,6 +230,7 @@ struct sock_common {
   * @sk_security: used by security modules
   * @sk_mark: generic packet mark
   * @sk_classid: this socket's cgroup classid
+  * @sk_cgrp: this socket's cgroup-specific proto data
   * @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
@@ -339,6 +342,7 @@ struct sock {
 #endif
  __u32   sk_mark;
  u32   sk_classid;
+ struct cg_proto  *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);
@@ -834,6 +838,37 @@ struct proto {
 #ifdef SOCK_REFCNT_DEBUG
  atomic_t  socks;
 #endif
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /*
+  * 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 cg_proto  *(*proto_cgroup)(struct mem_cgroup *memcg);
+#endif
+};
+
+struct cg_proto {
+ void   (*enter_memory_pressure)(struct sock *sk);
```

```
+ struct res_counter *memory_allocated; /* Current allocated memory. */
+ struct percpu_counter *sockets_allocated; /* Current number of sockets. */
+ int   *memory_pressure;
+ long   *sysctl_mem;
+ /*
+  * memcg field is used to find which memcg we belong directly
+  * Each memcg struct can hold more than one cg_proto, so container_of
+  * won't really cut.
+  *
+  * The elegant solution would be having an inverse function to
+  * proto_cgroup in struct proto, but that means polluting the structure
+  * for everybody, instead of just for memcg users.
+  */
+ struct mem_cgroup *memcg;
 };

 extern int proto_register(struct proto *prot, int alloc_slab);
@@ -852,7 +887,7 @@ static inline void sk_refcnt_debug_dec(struct sock *sk)
        sk->sk_prot->name, sk, atomic_read(&sk->sk_prot->socks));
 }

-static inline void sk_refcnt_debug_release(const struct sock *sk)
+inline void sk_refcnt_debug_release(const struct sock *sk)
 {
  if (atomic_read(&sk->sk_refcnt) != 1)
   printk(KERN_DEBUG "Destruction of the %s socket %p delayed, refcnt=%d\n",
@@ -864,6 +899,19 @@ static inline void sk_refcnt_debug_release(const struct sock *sk)
 #define sk_refcnt_debug_release(sk) do { } while (0)
 #endif /* SOCK_REFCNT_DEBUG */

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+extern struct jump_label_key memcg_socket_limit_enabled;
+#define mem_cgroup_sockets_enabled static_branch(&memcg_socket_limit_enabled)
+#else
+#define mem_cgroup_sockets_enabled 0
+#endif
+
+static inline struct cg_proto *parent_cg_proto(struct proto *proto,
+           struct cg_proto *cg_proto)
+{
+ return proto->proto_cgroup(parent_mem_cgroup(cg_proto->memcg));
+}
+
 static inline bool sk_has_memory_pressure(const struct sock *sk)
 {
  return sk->sk_prot->memory_pressure != NULL;
@@ -873,6 +921,10 @@ static inline bool sk_under_memory_pressure(const struct sock *sk)
 {
```

```
  if (!sk->sk_prot->memory_pressure)
   return false;
+
+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
+  return !!*sk->sk_cgrp->memory_pressure;
+
  return !!*sk->sk_prot->memory_pressure;
 }

@@ -880,52 +932,136 @@ static inline void sk_leave_memory_pressure(struct sock *sk)
 {
  int *memory_pressure = sk->sk_prot->memory_pressure;

- if (memory_pressure && *memory_pressure)
+ if (!memory_pressure)
+  return;
+
+ if (*memory_pressure)
   *memory_pressure = 0;
+
+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
+  struct cg_proto *cg_proto = sk->sk_cgrp;
+  struct proto *prot = sk->sk_prot;
+
+  for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
+   if (*cg_proto->memory_pressure)
+    *cg_proto->memory_pressure = 0;
+ }
+
 }

 static inline void sk_enter_memory_pressure(struct sock *sk)
 {
- if (sk->sk_prot->enter_memory_pressure)
-  sk->sk_prot->enter_memory_pressure(sk);
+ if (!sk->sk_prot->enter_memory_pressure)
+  return;
+
+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
+  struct cg_proto *cg_proto = sk->sk_cgrp;
+  struct proto *prot = sk->sk_prot;
+
+  for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
+   cg_proto->enter_memory_pressure(sk);
+ }
+
+ sk->sk_prot->enter_memory_pressure(sk);
 }
```

```
 static inline long sk_prot_mem_limits(const struct sock *sk, int index)
 {
  long *prot = sk->sk_prot->sysctl_mem;
+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
+  prot = sk->sk_cgrp->sysctl_mem;
  return prot[index];
 }

+static inline void memcg_memory_allocated_add(struct cg_proto *prot,
+           unsigned long amt,
+           int *parent_status)
+{
+ struct res_counter *fail;
+ int ret;
+
+ ret = res_counter_charge(prot->memory_allocated,
+      amt << PAGE_SHIFT, &fail);
+
+ if (ret < 0)
+  *parent_status = OVER_LIMIT;
+}
+
+static inline void memcg_memory_allocated_sub(struct cg_proto *prot,
+           unsigned long amt)
+{
+ res_counter_uncharge(prot->memory_allocated, amt << PAGE_SHIFT);
+}
+
+static inline u64 memcg_memory_allocated_read(struct cg_proto *prot)
+{
+ u64 ret;
+ ret = res_counter_read_u64(prot->memory_allocated, RES_USAGE);
+ return ret >> PAGE_SHIFT;
+}
+
 static inline long
 sk_memory_allocated(const struct sock *sk)
 {
  struct proto *prot = sk->sk_prot;
+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
+  return memcg_memory_allocated_read(sk->sk_cgrp);
+
  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;
+
+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
+  memcg_memory_allocated_add(sk->sk_cgrp, amt, parent_status);
+  /* update the root cgroup regardless */
+  atomic_long_add_return(amt, prot->memory_allocated);
+  return memcg_memory_allocated_read(sk->sk_cgrp);
+ }
+
  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;
+
+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp &&
+     parent_status != OVER_LIMIT) /* Otherwise was uncharged already */
+  memcg_memory_allocated_sub(sk->sk_cgrp, amt);
+
  atomic_long_sub(amt, prot->memory_allocated);
 }

 static inline void sk_sockets_allocated_dec(struct sock *sk)
 {
  struct proto *prot = sk->sk_prot;
+
+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
+  struct cg_proto *cg_proto = sk->sk_cgrp;
+
+  for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
+   percpu_counter_dec(cg_proto->sockets_allocated);
+ }
+
  percpu_counter_dec(prot->sockets_allocated);
 }

 static inline void sk_sockets_allocated_inc(struct sock *sk)
 {
  struct proto *prot = sk->sk_prot;
+
+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
+  struct cg_proto *cg_proto = sk->sk_cgrp;
+
```

```
+  for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
+   percpu_counter_inc(cg_proto->sockets_allocated);
+ }
+
  percpu_counter_inc(prot->sockets_allocated);
 }

@@ -934,6 +1070,9 @@ sk_sockets_allocated_read_positive(struct sock *sk)
 {
  struct proto *prot = sk->sk_prot;

+ if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
+  return percpu_counter_sum_positive(sk->sk_cgrp->sockets_allocated);
+
  return percpu_counter_sum_positive(prot->sockets_allocated);
 }

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 3becb24..beedff3 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -379,7 +379,48 @@ enum mem_type {

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

```
+   mem_cgroup_get(memcg);
+   sk->sk_cgrp = sk->sk_prot->proto_cgroup(memcg);
+  }
+  rcu_read_unlock();
+ }
+}
+EXPORT_SYMBOL(sock_update_memcg);
+
+void sock_release_memcg(struct sock *sk)
+{
+ if (static_branch(&memcg_socket_limit_enabled) && sk->sk_cgrp) {
+  struct mem_cgroup *memcg;
+  WARN_ON(!sk->sk_cgrp->memcg);
+  memcg = sk->sk_cgrp->memcg;
+  mem_cgroup_put(memcg);
+ }
+}
+#endif /* CONFIG_INET */
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 static void drain_all_stock_async(struct mem_cgroup *memcg);

 static struct mem_cgroup_per_zone *
@@ -4930,12 +4971,13 @@ static void mem_cgroup_put(struct mem_cgroup *memcg)
 /*
  * Returns the parent mem_cgroup in memcgroup hierarchy with hierarchy enabled.
  */
-static struct mem_cgroup *parent_mem_cgroup(struct mem_cgroup *memcg)
+struct mem_cgroup *parent_mem_cgroup(struct mem_cgroup *memcg)
 {
  if (!memcg->res.parent)
   return NULL;
  return mem_cgroup_from_res_counter(memcg->res.parent, res);
 }
+EXPORT_SYMBOL(parent_mem_cgroup);

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_SWAP
 static void __init enable_swap_cgroup(void)
diff --git a/net/core/sock.c b/net/core/sock.c
index 2b86d24..39e5d01 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 memcg_socket_limit_enabled;
+EXPORT_SYMBOL(memcg_socket_limit_enabled);
+
 /*
  * Make lock validator output more readable. (we pre-construct these
  * strings build-time, so that runtime initialization of socket
@@ -1677,21 +1681,25 @@ 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 parent_status = UNDER_LIMIT;

  sk->sk_forward_alloc += amt * SK_MEM_QUANTUM;

- allocated = sk_memory_allocated_add(sk, amt);
+ allocated = sk_memory_allocated_add(sk, amt, &parent_status);

  /* Under limit. */
- if (allocated <= sk_prot_mem_limits(sk, 0))
+ if (parent_status == UNDER_LIMIT &&
+    allocated <= sk_prot_mem_limits(sk, 0))
   sk_leave_memory_pressure(sk);

- /* Under pressure. */
- if (allocated > sk_prot_mem_limits(sk, 1))
+ /* Under pressure. (we or our parents) */
+ if ((parent_status > SOFT_LIMIT) ||
+    allocated > sk_prot_mem_limits(sk, 1))
   sk_enter_memory_pressure(sk);

- /* Over hard limit. */
- if (allocated > sk_prot_mem_limits(sk, 2))
+ /* Over hard limit (we or our parents) */
+ if ((parent_status == OVER_LIMIT) ||
+    (allocated > sk_prot_mem_limits(sk, 2)))
   goto suppress_allocation;

  /* guarantee minimum buffer size under pressure */
@@ -1738,7 +1746,7 @@ 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;
 }
@@ -1751,7 +1759,7 @@ EXPORT_SYMBOL(__sk_mem_schedule);
 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 (sk_under_memory_pressure(sk) &&
--
1.7.6.4
```