Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/net/ipv4/tcp_memcontrol.c

```
 1 #include <net/tcp.h>
 2 #include <net/tcp_memcontrol.h>
 3 #include <net/sock.h>
 4 #include <net/ip.h>
 5 #include <linux/nsproxy.h>
 6 #include <linux/memcontrol.h>
 7 #include <linux/module.h>
 8
 9 int tcp_init_cgroup(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
10 {
11         /*
12          * The root cgroup does not use res_counters, but rather,
13          * rely on the data already collected by the network
14          * subsystem
15          */
16         struct res_counter *res_parent = NULL;
17         struct cg_proto *cg_proto, *parent_cg;
18         struct mem_cgroup *parent = parent_mem_cgroup(memcg);
19
20         cg_proto = tcp_prot.proto_cgroup(memcg);
21         if (!cg_proto)
22                 return 0;
23
24         cg_proto->sysctl_mem[0] = sysctl_tcp_mem[0];
25         cg_proto->sysctl_mem[1] = sysctl_tcp_mem[1];
26         cg_proto->sysctl_mem[2] = sysctl_tcp_mem[2];
27         cg_proto->memory_pressure = 0;
28         cg_proto->memcg = memcg;
29
30         parent_cg = tcp_prot.proto_cgroup(parent);
31         if (parent_cg)
32                 res_parent = &parent_cg->memory_allocated;
33
34         res_counter_init(&cg_proto->memory_allocated, res_parent);
35         percpu_counter_init(&cg_proto->sockets_allocated, 0);
36
37         return 0;
38 }
39 EXPORT_SYMBOL(tcp_init_cgroup);
40
41 void tcp_destroy_cgroup(struct mem_cgroup *memcg)
42 {
43         struct cg_proto *cg_proto;
```

```
 44
 45         cg_proto = tcp_prot.proto_cgroup(memcg);
 46         if (!cg_proto)
 47                 return;
 48
 49         percpu_counter_destroy(&cg_proto->sockets_allocated);
 50 }
 51 EXPORT_SYMBOL(tcp_destroy_cgroup);
 52
 53 static int tcp_update_limit(struct mem_cgroup *memcg, u64 val)
 54 {
 55         struct cg_proto *cg_proto;
 56         int i;
 57         int ret;
 58
 59         cg_proto = tcp_prot.proto_cgroup(memcg);
 60         if (!cg_proto)
 61                 return -EINVAL;
 62
 63         if (val > RES_COUNTER_MAX)
 64                 val = RES_COUNTER_MAX;
 65
 66         ret = res_counter_set_limit(&cg_proto->memory_allocated, val);
 67         if (ret)
 68                 return ret;
 69
 70         for (i = 0; i < 3; i++)
 71                 cg_proto->sysctl_mem[i] = min_t(long, val >> PAGE_SHIFT,
 72                                                 sysctl_tcp_mem[i]);
 73
 74         if (val == RES_COUNTER_MAX)
 75                 clear_bit(MEMCG_SOCK_ACTIVE, &cg_proto->flags);
 76         else if (val != RES_COUNTER_MAX) {
 77                 /*
 78                  * The active bit needs to be written after the static_key
 79                  * update. This is what guarantees that the socket activation
 80                  * function is the last one to run. See sock_update_memcg() for
 81                  * details, and note that we don't mark any socket as belonging
 82                  * to this memcg until that flag is up.
 83                  *
 84                  * We need to do this, because static_keys will span multiple
 85                  * sites, but we can't control their order. If we mark a socket
 86                  * as accounted, but the accounting functions are not patched in
 87                  * yet, we'll lose accounting.
 88                  *
 89                  * We never race with the readers in sock_update_memcg(),
 90                  * because when this value change, the code to process it is not
 91                  * patched in yet.
 92                  *
 93                  * The activated bit is used to guarantee that no two writers
 94                  * will do the update in the same memcg. Without that, we can't
 95                  * properly shutdown the static key.
 96                  */
 97                 if (!test_and_set_bit(MEMCG_SOCK_ACTIVATED, &cg_proto->flags))
 98                         static_key_slow_inc(&memcg_socket_limit_enabled);
 99                 set_bit(MEMCG_SOCK_ACTIVE, &cg_proto->flags);
100         }
101
102         return 0;
103 }
104
105 static ssize_t tcp_cgroup_write(struct kernfs_open_file *of,
106                                 char *buf, size_t nbytes, loff_t off)
107 {
108         struct mem_cgroup *memcg = mem_cgroup_from_css(of_css(of));
```

```
109         unsigned long long val;
110         int ret = 0;
111
112         buf = strstrip(buf);
113
114         switch (of_cft(of)->private) {
115         case RES_LIMIT:
116                 /* see memcontrol.c */
117                 ret = res_counter_memparse_write_strategy(buf, &val);
118                 if (ret)
119                         break;
120                 ret = tcp_update_limit(memcg, val);
121                 break;
122         default:
123                 ret = -EINVAL;
124                 break;
125         }
126         return ret ?: nbytes;
127 }
128
129 static u64 tcp_read_stat(struct mem_cgroup *memcg, int type, u64 default_val)
130 {
131         struct cg_proto *cg_proto;
132
133         cg_proto = tcp_prot.proto_cgroup(memcg);
134         if (!cg_proto)
135                 return default_val;
136
137         return res_counter_read_u64(&cg_proto->memory_allocated, type);
138 }
139
140 static u64 tcp_read_usage(struct mem_cgroup *memcg)
141 {
142         struct cg_proto *cg_proto;
143
144         cg_proto = tcp_prot.proto_cgroup(memcg);
145         if (!cg_proto)
146                 return atomic_long_read(&tcp_memory_allocated) << PAGE_SHIFT;
147
148         return res_counter_read_u64(&cg_proto->memory_allocated, RES_USAGE);
149 }
150
151 static u64 tcp_cgroup_read(struct cgroup_subsys_state *css, struct cftype *cft)
152 {
153         struct mem_cgroup *memcg = mem_cgroup_from_css(css);
154         u64 val;
155
156         switch (cft->private) {
157         case RES_LIMIT:
158                 val = tcp_read_stat(memcg, RES_LIMIT, RES_COUNTER_MAX);
159                 break;
160         case RES_USAGE:
161                 val = tcp_read_usage(memcg);
162                 break;
163         case RES_FAILCNT:
164         case RES_MAX_USAGE:
165                 val = tcp_read_stat(memcg, cft->private, 0);
166                 break;
167         default:
168                 BUG();
169         }
170         return val;
171 }
172
173 static ssize_t tcp_cgroup_reset(struct kernfs_open_file *of,
```

```
174                                 char *buf, size_t nbytes, loff_t off)
175 {
176         struct mem_cgroup *memcg;
177         struct cg_proto *cg_proto;
178
179         memcg = mem_cgroup_from_css(of_css(of));
180         cg_proto = tcp_prot.proto_cgroup(memcg);
181         if (!cg_proto)
182                 return nbytes;
183
184         switch (of_cft(of)->private) {
185         case RES_MAX_USAGE:
186                 res_counter_reset_max(&cg_proto->memory_allocated);
187                 break;
188         case RES_FAILCNT:
189                 res_counter_reset_failcnt(&cg_proto->memory_allocated);
190                 break;
191         }
192
193         return nbytes;
194 }
195
196 static struct cftype tcp_files[] = {
197         {
198                 .name = "kmem.tcp.limit_in_bytes",
199                 .write = tcp_cgroup_write,
200                 .read_u64 = tcp_cgroup_read,
201                 .private = RES_LIMIT,
202         },
203         {
204                 .name = "kmem.tcp.usage_in_bytes",
205                 .read_u64 = tcp_cgroup_read,
206                 .private = RES_USAGE,
207         },
208         {
209                 .name = "kmem.tcp.failcnt",
210                 .private = RES_FAILCNT,
211                 .write = tcp_cgroup_reset,
212                 .read_u64 = tcp_cgroup_read,
213         },
214         {
215                 .name = "kmem.tcp.max_usage_in_bytes",
216                 .private = RES_MAX_USAGE,
217                 .write = tcp_cgroup_reset,
218                 .read_u64 = tcp_cgroup_read,
219         },
220         { }     /* terminate */
221 };
222
223 static int __init tcp_memcontrol_init(void)
224 {
225         WARN_ON(cgroup_add_legacy_cftypes(&memory_cgrp_subsys, tcp_files));
226         return 0;
227 }
228 __initcall(tcp_memcontrol_init);
229
```