Subject: [PATCH 6/7] Split Cpusets into Cpusets and Memsets
Posted by Paul Menage on Thu, 23 Nov 2006 12:08:54 GMT
View Forum Message <> Reply to Message

This patch splits the Cpusets container subsystem into two independent
subsystems; currently CPUsets are the cpu and memory node control
functionality in Cpusets are pretty much disjoint and unrelated; now
that the common process container abstraction has been moved out,
there's no particular reason to keep them together in the same
subsystem.

Signed-off-by: Paul Menage <menage@google.com>

```
 fs/proc/array.c          |    2
 include/linux/cpuset.h   |   75 --
 include/linux/mempolicy.h |    2
 include/linux/memset.h   |  125 ++++
 include/linux/sched.h    |   10
 init/Kconfig             |   14
 init/main.c              |    3
 kernel/Makefile          |    1
 kernel/cpuset.c          |  994 +-------------------------------
 kernel/memset.c          | 1352 ++++++++++++++++++++++++++++++++++++++++++++++++
 mm/filemap.c             |    6
 mm/hugetlb.c             |    4
 mm/memory_hotplug.c      |    4
 mm/mempolicy.c           |   36 -
 mm/migrate.c             |    4
 mm/oom_kill.c            |   14
 mm/page_alloc.c          |   26
 mm/slab.c                |   12
 mm/vmscan.c              |   10
 19 files changed, 1593 insertions(+), 1101 deletions(-)

Index: container-2.6.19-rc6/include/linux/cpuset.h
===================================================================
--- container-2.6.19-rc6.orig/include/linux/cpuset.h
+++ container-2.6.19-rc6/include/linux/cpuset.h
@@ -10,58 +10,22 @@

 #include <linux/sched.h>
 #include <linux/cpumask.h>
-#include <linux/nodemask.h>
 #include <linux/container.h>

 #ifdef CONFIG_CPUSETS
```

```
-extern int number_of_cpusets;  /* How many cpusets are defined in system? */
-
 extern int cpuset_init_early(void);
 extern int cpuset_init(void);
 extern void cpuset_init_smp(void);
 extern cpumask_t cpuset_cpus_allowed(struct task_struct *p);
-extern nodemask_t cpuset_mems_allowed(struct task_struct *p);
-void cpuset_init_current_mems_allowed(void);
-void cpuset_update_task_memory_state(void);
-#define cpuset_nodes_subset_current_mems_allowed(nodes) \
-  nodes_subset((nodes), current->mems_allowed)
-int cpuset_zonelist_valid_mems_allowed(struct zonelist *zl);
-
-extern int __cpuset_zone_allowed(struct zone *z, gfp_t gfp_mask);
-static int inline cpuset_zone_allowed(struct zone *z, gfp_t gfp_mask)
-{
- return number_of_cpusets <= 1 || __cpuset_zone_allowed(z, gfp_mask);
-}

 extern int cpuset_excl_nodes_overlap(const struct task_struct *p);

-#define cpuset_memory_pressure_bump()     \
- do {       \
-  if (cpuset_memory_pressure_enabled)  \
-   __cpuset_memory_pressure_bump(); \
- } while (0)
-extern int cpuset_memory_pressure_enabled;
-extern void __cpuset_memory_pressure_bump(void);
-
 extern struct file_operations proc_cpuset_operations;
 extern char *cpuset_task_status_allowed(struct task_struct *task, char *buffer);
-extern int cpuset_mem_spread_node(void);
-
-static inline int cpuset_do_page_mem_spread(void)
-{
- return current->flags & PF_SPREAD_PAGE;
-}
-
-static inline int cpuset_do_slab_mem_spread(void)
-{
- return current->flags & PF_SPREAD_SLAB;
-}

 extern void cpuset_track_online_nodes(void);

-extern int current_cpuset_is_being_rebound(void);
-
```

```
 #else /* !CONFIG_CPUSETS */

 static inline int cpuset_init_early(void) { return 0; }
@@ -73,60 +37,21 @@ static inline cpumask_t cpuset_cpus_allo
  return cpu_possible_map;
 }

-static inline nodemask_t cpuset_mems_allowed(struct task_struct *p)
-{
- return node_possible_map;
-}
-
-static inline void cpuset_init_current_mems_allowed(void) {}
-static inline void cpuset_update_task_memory_state(void) {}
-#define cpuset_nodes_subset_current_mems_allowed(nodes) (1)
-
-static inline int cpuset_zonelist_valid_mems_allowed(struct zonelist *zl)
-{
- return 1;
-}
-
-static inline int cpuset_zone_allowed(struct zone *z, gfp_t gfp_mask)
-{
- return 1;
-}

 static inline int cpuset_excl_nodes_overlap(const struct task_struct *p)
 {
  return 1;
 }

-static inline void cpuset_memory_pressure_bump(void) {}
-
 static inline char *cpuset_task_status_allowed(struct task_struct *task,
        char *buffer)
 {
  return buffer;
 }

-static inline int cpuset_mem_spread_node(void)
-{
- return 0;
-}
-
-static inline int cpuset_do_page_mem_spread(void)
-{
- return 0;
-}
```

```
-
-static inline int cpuset_do_slab_mem_spread(void)
-{
- return 0;
-}

 static inline void cpuset_track_online_nodes(void) {}

-static inline int current_cpuset_is_being_rebound(void)
-{
- return 0;
-}
-
 #endif /* !CONFIG_CPUSETS */

 #endif /* _LINUX_CPUSET_H */
Index: container-2.6.19-rc6/include/linux/memset.h
===================================================================
--- /dev/null
+++ container-2.6.19-rc6/include/linux/memset.h
@@ -0,0 +1,125 @@
+#ifndef _LINUX_MEMSET_H
+#define _LINUX_MEMSET_H
+/*
+ *  memset interface
+ *
+ *  Copyright (C) 2003 BULL SA
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *
+ */
+
+#include <linux/sched.h>
+#include <linux/nodemask.h>
+#include <linux/container.h>
+
+#ifdef CONFIG_MEMSETS
+
+extern int number_of_memsets;  /* How many memsets are defined in system? */
+
+extern int memset_init_early(void);
+extern int memset_init(void);
+extern void memset_init_smp(void);
+extern nodemask_t memset_mems_allowed(struct task_struct *p);
+void memset_init_current_mems_allowed(void);
+void memset_update_task_memory_state(void);
+#define memset_nodes_subset_current_mems_allowed(nodes) \
+  nodes_subset((nodes), current->mems_allowed)
+int memset_zonelist_valid_mems_allowed(struct zonelist *zl);
```

```
+
+extern int __memset_zone_allowed(struct zone *z, gfp_t gfp_mask);
+static int inline memset_zone_allowed(struct zone *z, gfp_t gfp_mask)
+{
+ return number_of_memsets <= 1 || __memset_zone_allowed(z, gfp_mask);
+}
+
+extern int memset_excl_nodes_overlap(const struct task_struct *p);
+
+#define memset_memory_pressure_bump()     \
+ do {       \
+  if (memset_memory_pressure_enabled)  \
+   __memset_memory_pressure_bump(); \
+ } while (0)
+extern int memset_memory_pressure_enabled;
+extern void __memset_memory_pressure_bump(void);
+
+extern struct file_operations proc_memset_operations;
+extern char *memset_task_status_allowed(struct task_struct *task, char *buffer);
+extern int memset_mem_spread_node(void);
+
+static inline int memset_do_page_mem_spread(void)
+{
+ return current->flags & PF_SPREAD_PAGE;
+}
+
+static inline int memset_do_slab_mem_spread(void)
+{
+ return current->flags & PF_SPREAD_SLAB;
+}
+
+extern void memset_track_online_nodes(void);
+
+extern int current_memset_is_being_rebound(void);
+
+#else /* !CONFIG_MEMSETS */
+
+static inline int memset_init_early(void) { return 0; }
+static inline int memset_init(void) { return 0; }
+static inline void memset_init_smp(void) {}
+
+static inline nodemask_t memset_mems_allowed(struct task_struct *p)
+{
+ return node_possible_map;
+}
+
+static inline void memset_init_current_mems_allowed(void) {}
+static inline void memset_update_task_memory_state(void) {}
```

```
+#define memset_nodes_subset_current_mems_allowed(nodes) (1)
+
+static inline int memset_zonelist_valid_mems_allowed(struct zonelist *zl)
+{
+ return 1;
+}
+
+static inline int memset_zone_allowed(struct zone *z, gfp_t gfp_mask)
+{
+ return 1;
+}
+
+static inline int memset_excl_nodes_overlap(const struct task_struct *p)
+{
+ return 1;
+}
+
+static inline void memset_memory_pressure_bump(void) {}
+
+static inline char *memset_task_status_allowed(struct task_struct *task,
+       char *buffer)
+{
+ return buffer;
+}
+
+static inline int memset_mem_spread_node(void)
+{
+ return 0;
+}
+
+static inline int memset_do_page_mem_spread(void)
+{
+ return 0;
+}
+
+static inline int memset_do_slab_mem_spread(void)
+{
+ return 0;
+}
+
+static inline void memset_track_online_nodes(void) {}
+
+static inline int current_memset_is_being_rebound(void)
+{
+ return 0;
+}
+
+#endif /* !CONFIG_MEMSETS */
```

```
+
+#endif /* _LINUX_MEMSET_H */
Index: container-2.6.19-rc6/kernel/cpuset.c
===================================================================
--- container-2.6.19-rc6.orig/kernel/cpuset.c
+++ container-2.6.19-rc6/kernel/cpuset.c
@@ -32,7 +32,6 @@
 #include <linux/kernel.h>
 #include <linux/kmod.h>
 #include <linux/list.h>
-#include <linux/mempolicy.h>
 #include <linux/mm.h>
 #include <linux/module.h>
 #include <linux/mount.h>
@@ -56,42 +55,18 @@
 #include <asm/atomic.h>
 #include <linux/mutex.h>

-/*
- * Tracks how many cpusets are currently defined in system.
- * When there is only one cpuset (the root cpuset) we can
- * short circuit some hooks.
- */
-int number_of_cpusets __read_mostly;
-
 /* Retrieve the cpuset from a container */
 static struct container_subsys cpuset_subsys;
 struct cpuset;

-/* See "Frequency meter" comments, below. */
-
-struct fmeter {
- int cnt;  /* unprocessed events count */
- int val;  /* most recent output value */
- time_t time;  /* clock (secs) when val computed */
- spinlock_t lock; /* guards read or write of above */
-};
-
 struct cpuset {
  struct container_subsys_state css;

  unsigned long flags;  /* "unsigned long" so bitops work */
  cpumask_t cpus_allowed;  /* CPUs allowed to tasks in cpuset */
- nodemask_t mems_allowed; /* Memory Nodes allowed to tasks */

  struct cpuset *parent;  /* my parent */

- /*
```

```
-  * Copy of global cpuset_mems_generation as of the most
-  * recent time this cpuset changed its mems_allowed.
-  */
- int mems_generation;
-
- struct fmeter fmeter;  /* memory_pressure filter */
 };

 /* Update the cpuset for a container */
@@ -117,10 +92,6 @@ static inline struct cpuset *task_cs(str
 /* bits in struct cpuset flags field */
 typedef enum {
   CS_CPU_EXCLUSIVE,
- CS_MEM_EXCLUSIVE,
- CS_MEMORY_MIGRATE,
- CS_SPREAD_PAGE,
- CS_SPREAD_SLAB,
 } cpuset_flagbits_t;

 /* convenient tests for these bits */
@@ -129,51 +100,10 @@ static inline int is_cpu_exclusive(const
   return test_bit(CS_CPU_EXCLUSIVE, &cs->flags);
 }

-static inline int is_mem_exclusive(const struct cpuset *cs)
-{
- return test_bit(CS_MEM_EXCLUSIVE, &cs->flags);
-}
-
-static inline int is_memory_migrate(const struct cpuset *cs)
-{
- return test_bit(CS_MEMORY_MIGRATE, &cs->flags);
-}
-
-static inline int is_spread_page(const struct cpuset *cs)
-{
- return test_bit(CS_SPREAD_PAGE, &cs->flags);
-}
-
-static inline int is_spread_slab(const struct cpuset *cs)
-{
- return test_bit(CS_SPREAD_SLAB, &cs->flags);
-}
-
-/*
- * Increment this integer everytime any cpuset changes its
- * mems_allowed value.  Users of cpusets can track this generation
- * number, and avoid having to lock and reload mems_allowed unless
```

```
- * the cpuset they're using changes generation.
- *
- * A single, global generation is needed because attach_task() could
- * reattach a task to a different cpuset, which must not have its
- * generation numbers aliased with those of that tasks previous cpuset.
- *
- * Generations are needed for mems_allowed because one task cannot
- * modify anothers memory placement.  So we must enable every task,
- * on every visit to __alloc_pages(), to efficiently check whether
- * its current->cpuset->mems_allowed has changed, requiring an update
- * of its current->mems_allowed.
- *
- * Since cpuset_mems_generation is guarded by manage_mutex,
- * there is no need to mark it atomic.
- */
-static int cpuset_mems_generation;

 static struct cpuset top_cpuset = {
- .flags = ((1 << CS_CPU_EXCLUSIVE) | (1 << CS_MEM_EXCLUSIVE)),
+ .flags = 1 << CS_CPU_EXCLUSIVE,
  .cpus_allowed = CPU_MASK_ALL,
- .mems_allowed = NODE_MASK_ALL,
 };

 /* This is ugly, but preserves the userspace API for existing cpuset
@@ -189,7 +119,7 @@ static int cpuset_get_sb(struct file_sys
  if (container_fs) {
   ret = container_fs->get_sb(container_fs, flags,
         unused_dev_name,
-        "cpuset", mnt);
+        "cpuset,memset", mnt);
   put_filesystem(container_fs);
  }
  return ret;
@@ -226,120 +156,17 @@ static void guarantee_online_cpus(const
 }

 /*
- * Return in *pmask the portion of a cpusets's mems_allowed that
- * are online.  If none are online, walk up the cpuset hierarchy
- * until we find one that does have some online mems.  If we get
- * all the way to the top and still haven't found any online mems,
- * return node_online_map.
- *
- * One way or another, we guarantee to return some non-empty subset
- * of node_online_map.
- *
- * Call with callback_mutex held.
```

```
- */
-
-static void guarantee_online_mems(const struct cpuset *cs, nodemask_t *pmask)
-{
- while (cs && !nodes_intersects(cs->mems_allowed, node_online_map))
-  cs = cs->parent;
- if (cs)
-  nodes_and(*pmask, cs->mems_allowed, node_online_map);
- else
-  *pmask = node_online_map;
- BUG_ON(!nodes_intersects(*pmask, node_online_map));
-}
-
-/**
- * cpuset_update_task_memory_state - update task memory placement
- *
- * If the current tasks cpusets mems_allowed changed behind our
- * backs, update current->mems_allowed, mems_generation and task NUMA
- * mempolicy to the new value.
- *
- * Task mempolicy is updated by rebinding it relative to the
- * current->cpuset if a task has its memory placement changed.
- * Do not call this routine if in_interrupt().
- *
- * Call without callback_mutex or task_lock() held.  May be
- * called with or without manage_mutex held.  Thanks in part to
- * 'the_top_cpuset_hack', the tasks cpuset pointer will never
- * be NULL.  This routine also might acquire callback_mutex and
- * current->mm->mmap_sem during call.
- *
- * Reading current->cpuset->mems_generation doesn't need task_lock
- * to guard the current->cpuset derefence, because it is guarded
- * from concurrent freeing of current->cpuset by attach_task(),
- * using RCU.
- *
- * The rcu_dereference() is technically probably not needed,
- * as I don't actually mind if I see a new cpuset pointer but
- * an old value of mems_generation.  However this really only
- * matters on alpha systems using cpusets heavily.  If I dropped
- * that rcu_dereference(), it would save them a memory barrier.
- * For all other arch's, rcu_dereference is a no-op anyway, and for
- * alpha systems not using cpusets, another planned optimization,
- * avoiding the rcu critical section for tasks in the root cpuset
- * which is statically allocated, so can't vanish, will make this
- * irrelevant.  Better to use RCU as intended, than to engage in
- * some cute trick to save a memory barrier that is impossible to
- * test, for alpha systems using cpusets heavily, which might not
- * even exist.
```

```
- *
- * This routine is needed to update the per-task mems_allowed data,
- * within the tasks context, when it is trying to allocate memory
- * (in various mm/mempolicy.c routines) and notices that some other
- * task has been modifying its cpuset.
- */
-
-void cpuset_update_task_memory_state(void)
-{
- int my_cpusets_mem_gen;
- struct task_struct *tsk = current;
- struct cpuset *cs;
-
- if (task_cs(tsk) == &top_cpuset) {
-  /* Don't need rcu for top_cpuset.  It's never freed. */
-  my_cpusets_mem_gen = top_cpuset.mems_generation;
- } else {
-  rcu_read_lock();
-  my_cpusets_mem_gen = task_cs(current)->mems_generation;
-  rcu_read_unlock();
- }
-
- if (my_cpusets_mem_gen != tsk->cpuset_mems_generation) {
-  container_lock();
-  task_lock(tsk);
-  cs = task_cs(tsk); /* Maybe changed when task not locked */
-  guarantee_online_mems(cs, &tsk->mems_allowed);
-  tsk->cpuset_mems_generation = cs->mems_generation;
-  if (is_spread_page(cs))
-   tsk->flags |= PF_SPREAD_PAGE;
-  else
-   tsk->flags &= ~PF_SPREAD_PAGE;
-  if (is_spread_slab(cs))
-   tsk->flags |= PF_SPREAD_SLAB;
-  else
-   tsk->flags &= ~PF_SPREAD_SLAB;
-  task_unlock(tsk);
-  container_unlock();
-  mpol_rebind_task(tsk, &tsk->mems_allowed);
- }
-}
-
-/*
  * is_cpuset_subset(p, q) - Is cpuset p a subset of cpuset q?
  *
- * One cpuset is a subset of another if all its allowed CPUs and
- * Memory Nodes are a subset of the other, and its exclusive flags
- * are only set if the other's are set.  Call holding manage_mutex.
```

```
+ * One cpuset is a subset of another if all its allowed CPUs are a
+ * subset of the other, and its exclusive flags are only set if the
+ * other's are set.  Call holding manage_mutex.
  */

 static int is_cpuset_subset(const struct cpuset *p, const struct cpuset *q)
 {
  return cpus_subset(p->cpus_allowed, q->cpus_allowed) &&
-  nodes_subset(p->mems_allowed, q->mems_allowed) &&
-  is_cpu_exclusive(p) <= is_cpu_exclusive(q) &&
-  is_mem_exclusive(p) <= is_mem_exclusive(q);
+  is_cpu_exclusive(p) <= is_cpu_exclusive(q);
 }

 /*
@@ -356,7 +183,7 @@ static int is_cpuset_subset(const struct
  * cpuset in the list must use cur below, not trial.
  *
  * 'trial' is the address of bulk structure copy of cur, with
- * perhaps one or more of the fields cpus_allowed, mems_allowed,
+ * perhaps one or more of the fields cpus_allowed,
  * or flags changed to new, trial values.
  *
  * Return 0 if valid, -errno if not.
@@ -388,10 +215,6 @@ static int validate_change(const struct
       c != cur &&
       cpus_intersects(trial->cpus_allowed, c->cpus_allowed))
    return -EINVAL;
-  if ((is_mem_exclusive(trial) || is_mem_exclusive(c)) &&
-      c != cur &&
-      nodes_intersects(trial->mems_allowed, c->mems_allowed))
-   return -EINVAL;
  }

  return 0;
@@ -487,211 +310,10 @@ static int update_cpumask(struct cpuset
  return 0;
 }

-/*
- * cpuset_migrate_mm
- *
- *    Migrate memory region from one set of nodes to another.
- *
- *    Temporarilly set tasks mems_allowed to target nodes of migration,
- *    so that the migration code can allocate pages on these nodes.
- *
- *    Call holding manage_mutex, so our current->cpuset won't change
```

```
- *    during this call, as manage_mutex holds off any attach_task()
- *    calls.  Therefore we don't need to take task_lock around the
- *    call to guarantee_online_mems(), as we know no one is changing
- *    our tasks cpuset.
- *
- *    Hold callback_mutex around the two modifications of our tasks
- *    mems_allowed to synchronize with cpuset_mems_allowed().
- *
- *    While the mm_struct we are migrating is typically from some
- *    other task, the task_struct mems_allowed that we are hacking
- *    is for our current task, which must allocate new pages for that
- *    migrating memory region.
- *
- *    We call cpuset_update_task_memory_state() before hacking
- *    our tasks mems_allowed, so that we are assured of being in
- *    sync with our tasks cpuset, and in particular, callbacks to
- *    cpuset_update_task_memory_state() from nested page allocations
- *    won't see any mismatch of our cpuset and task mems_generation
- *    values, so won't overwrite our hacked tasks mems_allowed
- *    nodemask.
- */
-
-static void cpuset_migrate_mm(struct mm_struct *mm, const nodemask_t *from,
-       const nodemask_t *to)
-{
- struct task_struct *tsk = current;
-
- cpuset_update_task_memory_state();
-
- container_lock();
- tsk->mems_allowed = *to;
- container_unlock();
-
- do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);
-
- container_lock();
- guarantee_online_mems(task_cs(tsk),&tsk->mems_allowed);
- container_unlock();
-}
-
-/*
- * Handle user request to change the 'mems' memory placement
- * of a cpuset.  Needs to validate the request, update the
- * cpusets mems_allowed and mems_generation, and for each
- * task in the cpuset, rebind any vma mempolicies and if
- * the cpuset is marked 'memory_migrate', migrate the tasks
- * pages to the new memory.
- *
```

```
- * Call with manage_mutex held.  May take callback_mutex during call.
- * Will take tasklist_lock, scan tasklist for tasks in cpuset cs,
- * lock each such tasks mm->mmap_sem, scan its vma's and rebind
- * their mempolicies to the cpusets new mems_allowed.
- */
-
-static void *cpuset_being_rebound;
-
-static int update_nodemask(struct cpuset *cs, char *buf)
-{
- struct cpuset trialcs;
- nodemask_t oldmem;
- struct task_struct *g, *p;
- struct mm_struct **mmarray;
- int i, n, ntasks;
- int migrate;
- int fudge;
- int retval;
- struct container *cont;
-
- /* top_cpuset.mems_allowed tracks node_online_map; it's read-only */
- if (cs == &top_cpuset)
-  return -EACCES;
-
- trialcs = *cs;
- cont = cs->css.container;
- retval = nodelist_parse(buf, trialcs.mems_allowed);
- if (retval < 0)
-  goto done;
- nodes_and(trialcs.mems_allowed, trialcs.mems_allowed, node_online_map);
- oldmem = cs->mems_allowed;
- if (nodes_equal(oldmem, trialcs.mems_allowed)) {
-  retval = 0;  /* Too easy - nothing to do */
-  goto done;
- }
- if (nodes_empty(trialcs.mems_allowed)) {
-  retval = -ENOSPC;
-  goto done;
- }
- retval = validate_change(cs, &trialcs);
- if (retval < 0)
-  goto done;
-
- container_lock();
- cs->mems_allowed = trialcs.mems_allowed;
- cs->mems_generation = cpuset_mems_generation++;
- container_unlock();
-
```

```
- cpuset_being_rebound = cs;  /* causes mpol_copy() rebind */
-
- fudge = 10;    /* spare mmarray[] slots */
- fudge += cpus_weight(cs->cpus_allowed); /* imagine one fork-bomb/cpu */
- retval = -ENOMEM;
-
- /*
-  * Allocate mmarray[] to hold mm reference for each task
-  * in cpuset cs.  Can't kmalloc GFP_KERNEL while holding
-  * tasklist_lock.  We could use GFP_ATOMIC, but with a
-  * few more lines of code, we can retry until we get a big
-  * enough mmarray[] w/o using GFP_ATOMIC.
-  */
- while (1) {
-  ntasks = atomic_read(&cs->css.container->count);  /* guess */
-  ntasks += fudge;
-  mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
-  if (!mmarray)
-   goto done;
-  write_lock_irq(&tasklist_lock);  /* block fork */
-  if (atomic_read(&cs->css.container->count) <= ntasks)
-   break;    /* got enough */
-  write_unlock_irq(&tasklist_lock); /* try again */
-  kfree(mmarray);
- }
-
- n = 0;
-
- /* Load up mmarray[] with mm reference for each task in cpuset. */
- do_each_thread(g, p) {
-  struct mm_struct *mm;
-
-  if (n >= ntasks) {
-   printk(KERN_WARNING
-    "Cpuset mempolicy rebind incomplete.\n");
-   continue;
-  }
-  if (task_cs(p) != cs)
-   continue;
-  mm = get_task_mm(p);
-  if (!mm)
-   continue;
-  mmarray[n++] = mm;
- } while_each_thread(g, p);
- write_unlock_irq(&tasklist_lock);
-
- /*
-  * Now that we've dropped the tasklist spinlock, we can
```

```
-  * rebind the vma mempolicies of each mm in mmarray[] to their
-  * new cpuset, and release that mm.  The mpol_rebind_mm()
-  * call takes mmap_sem, which we couldn't take while holding
-  * tasklist_lock.  Forks can happen again now - the mpol_copy()
-  * cpuset_being_rebound check will catch such forks, and rebind
-  * their vma mempolicies too.  Because we still hold the global
-  * cpuset manage_mutex, we know that no other rebind effort will
-  * be contending for the global variable cpuset_being_rebound.
-  * It's ok if we rebind the same mm twice; mpol_rebind_mm()
-  * is idempotent.  Also migrate pages in each mm to new nodes.
-  */
- migrate = is_memory_migrate(cs);
- for (i = 0; i < n; i++) {
-  struct mm_struct *mm = mmarray[i];
-
-  mpol_rebind_mm(mm, &cs->mems_allowed);
-  if (migrate)
-   cpuset_migrate_mm(mm, &oldmem, &cs->mems_allowed);
-  mmput(mm);
- }
-
- /* We're done rebinding vma's to this cpusets new mems_allowed. */
- kfree(mmarray);
- cpuset_being_rebound = NULL;
- retval = 0;
-done:
- return retval;
-}
-
-int current_cpuset_is_being_rebound(void)
-{
- return task_cs(current) == cpuset_being_rebound;
-}
-
-/*
- * Call with manage_mutex held.
- */
-
-static int update_memory_pressure_enabled(struct cpuset *cs, char *buf)
-{
- if (simple_strtoul(buf, NULL, 10) != 0)
-  cpuset_memory_pressure_enabled = 1;
- else
-  cpuset_memory_pressure_enabled = 0;
- return 0;
-}

 /*
```

```
  * update_flag - read a 0 or a 1 in a file and update associated flag
- * bit: the bit to update (CS_CPU_EXCLUSIVE, CS_MEM_EXCLUSIVE,
- *    CS_NOTIFY_ON_RELEASE, CS_MEMORY_MIGRATE,
- *    CS_SPREAD_PAGE, CS_SPREAD_SLAB)
+ * bit: the bit to update (CS_CPU_EXCLUSIVE)
  * cs: the cpuset to update
  * buf: the buffer where we read the 0 or 1
  *
@@ -729,110 +351,12 @@ static int update_flag(cpuset_flagbits_t
  return 0;
 }

-/*
- * Frequency meter - How fast is some event occurring?
- *
- * These routines manage a digitally filtered, constant time based,
- * event frequency meter.  There are four routines:
- *   fmeter_init() - initialize a frequency meter.
- *   fmeter_markevent() - called each time the event happens.
- *   fmeter_getrate() - returns the recent rate of such events.
- *   fmeter_update() - internal routine used to update fmeter.
- *
- * A common data structure is passed to each of these routines,
- * which is used to keep track of the state required to manage the
- * frequency meter and its digital filter.
- *
- * The filter works on the number of events marked per unit time.
- * The filter is single-pole low-pass recursive (IIR).  The time unit
- * is 1 second.  Arithmetic is done using 32-bit integers scaled to
- * simulate 3 decimal digits of precision (multiplied by 1000).
- *
- * With an FM_COEF of 933, and a time base of 1 second, the filter
- * has a half-life of 10 seconds, meaning that if the events quit
- * happening, then the rate returned from the fmeter_getrate()
- * will be cut in half each 10 seconds, until it converges to zero.
- *
- * It is not worth doing a real infinitely recursive filter.  If more
- * than FM_MAXTICKS ticks have elapsed since the last filter event,
- * just compute FM_MAXTICKS ticks worth, by which point the level
- * will be stable.
- *
- * Limit the count of unprocessed events to FM_MAXCNT, so as to avoid
- * arithmetic overflow in the fmeter_update() routine.
- *
- * Given the simple 32 bit integer arithmetic used, this meter works
- * best for reporting rates between one per millisecond (msec) and
- * one per 32 (approx) seconds.  At constant rates faster than one
- * per msec it maxes out at values just under 1,000,000.  At constant
```

```
- * rates between one per msec, and one per second it will stabilize
- * to a value N*1000, where N is the rate of events per second.
- * At constant rates between one per second and one per 32 seconds,
- * it will be choppy, moving up on the seconds that have an event,
- * and then decaying until the next event.  At rates slower than
- * about one in 32 seconds, it decays all the way back to zero between
- * each event.
- */
-
-#define FM_COEF 933  /* coefficient for half-life of 10 secs */
-#define FM_MAXTICKS ((time_t)99) /* useless computing more ticks than this */
-#define FM_MAXCNT 1000000 /* limit cnt to avoid overflow */
-#define FM_SCALE 1000  /* faux fixed point scale */
-
-/* Initialize a frequency meter */
-static void fmeter_init(struct fmeter *fmp)
-{
- fmp->cnt = 0;
- fmp->val = 0;
- fmp->time = 0;
- spin_lock_init(&fmp->lock);
-}
-
-/* Internal meter update - process cnt events and update value */
-static void fmeter_update(struct fmeter *fmp)
-{
- time_t now = get_seconds();
- time_t ticks = now - fmp->time;
-
- if (ticks == 0)
-  return;
-
- ticks = min(FM_MAXTICKS, ticks);
- while (ticks-- > 0)
-  fmp->val = (FM_COEF * fmp->val) / FM_SCALE;
- fmp->time = now;
-
- fmp->val += ((FM_SCALE - FM_COEF) * fmp->cnt) / FM_SCALE;
- fmp->cnt = 0;
-}
-
-/* Process any previous ticks, then bump cnt by one (times scale). */
-static void fmeter_markevent(struct fmeter *fmp)
-{
- spin_lock(&fmp->lock);
- fmeter_update(fmp);
- fmp->cnt = min(FM_MAXCNT, fmp->cnt + FM_SCALE);
- spin_unlock(&fmp->lock);
```

```
-}
-
-/* Process any previous ticks, then return current value. */
-static int fmeter_getrate(struct fmeter *fmp)
-{
- int val;
-
- spin_lock(&fmp->lock);
- fmeter_update(fmp);
- val = fmp->val;
- spin_unlock(&fmp->lock);
- return val;
-}
-
 int cpuset_can_attach(struct container_subsys *ss,
         struct container *cont, struct task_struct *tsk)
 {
  struct cpuset *cs = container_cs(cont);

- if (cpus_empty(cs->cpus_allowed) || nodes_empty(cs->mems_allowed))
+ if (cpus_empty(cs->cpus_allowed))
    return -ENOSPC;

  return security_task_setscheduler(tsk, 0, NULL);
@@ -846,40 +370,11 @@ void cpuset_attach(struct container_subs
  set_cpus_allowed(tsk, cpus);
 }

-void cpuset_post_attach(struct container_subsys *ss,
-    struct container *cont,
-    struct container *oldcont,
-    struct task_struct *tsk)
-{
- nodemask_t from, to;
- struct mm_struct *mm;
- struct cpuset *cs = container_cs(cont);
- struct cpuset *oldcs = container_cs(oldcont);
-
- from = oldcs->mems_allowed;
- to = cs->mems_allowed;
- mm = get_task_mm(tsk);
- if (mm) {
-  mpol_rebind_mm(mm, &to);
-  if (is_memory_migrate(cs))
-    cpuset_migrate_mm(mm, &from, &to);
-  mmput(mm);
- }
-
```

```
-}
-
 /* The various types of files and directories in a cpuset file system */

 typedef enum {
- FILE_MEMORY_MIGRATE,
  FILE_CPULIST,
- FILE_MEMLIST,
  FILE_CPU_EXCLUSIVE,
- FILE_MEM_EXCLUSIVE,
- FILE_MEMORY_PRESSURE_ENABLED,
- FILE_MEMORY_PRESSURE,
- FILE_SPREAD_PAGE,
- FILE_SPREAD_SLAB,
 } cpuset_filetype_t;

 static ssize_t cpuset_common_file_write(struct container *cont,
@@ -894,7 +389,7 @@ static ssize_t cpuset_common_file_write(
  int retval = 0;

  /* Crude upper limit on largest legitimate list user might write. */
- if (nbytes > 100 + 6 * max(NR_CPUS, MAX_NUMNODES))
+ if (nbytes > 100 + 6 * NR_CPUS)
   return -E2BIG;

  /* +1 for nul-terminator */
@@ -918,32 +413,9 @@ static ssize_t cpuset_common_file_write(
  case FILE_CPULIST:
   retval = update_cpumask(cs, buffer);
   break;
- case FILE_MEMLIST:
-  retval = update_nodemask(cs, buffer);
-  break;
  case FILE_CPU_EXCLUSIVE:
   retval = update_flag(CS_CPU_EXCLUSIVE, cs, buffer);
   break;
- case FILE_MEM_EXCLUSIVE:
-  retval = update_flag(CS_MEM_EXCLUSIVE, cs, buffer);
-  break;
- case FILE_MEMORY_MIGRATE:
-  retval = update_flag(CS_MEMORY_MIGRATE, cs, buffer);
-  break;
- case FILE_MEMORY_PRESSURE_ENABLED:
-  retval = update_memory_pressure_enabled(cs, buffer);
-  break;
- case FILE_MEMORY_PRESSURE:
-  retval = -EACCES;
-  break;
```

```
- case FILE_SPREAD_PAGE:
-  retval = update_flag(CS_SPREAD_PAGE, cs, buffer);
-  cs->mems_generation = cpuset_mems_generation++;
-  break;
- case FILE_SPREAD_SLAB:
-  retval = update_flag(CS_SPREAD_SLAB, cs, buffer);
-  cs->mems_generation = cpuset_mems_generation++;
-  break;
  default:
   retval = -EINVAL;
   goto out2;
@@ -981,17 +453,6 @@ static int cpuset_sprintf_cpulist(char *
  return cpulist_scnprintf(page, PAGE_SIZE, mask);
 }

-static int cpuset_sprintf_memlist(char *page, struct cpuset *cs)
-{
- nodemask_t mask;
-
- container_lock();
- mask = cs->mems_allowed;
- container_unlock();
-
- return nodelist_scnprintf(page, PAGE_SIZE, mask);
-}
-
 static ssize_t cpuset_common_file_read(struct container *cont,
            struct cftype *cft,
            struct file *file,
@@ -1013,30 +474,9 @@ static ssize_t cpuset_common_file_read(s
  case FILE_CPULIST:
   s += cpuset_sprintf_cpulist(s, cs);
   break;
- case FILE_MEMLIST:
-  s += cpuset_sprintf_memlist(s, cs);
-  break;
  case FILE_CPU_EXCLUSIVE:
   *s++ = is_cpu_exclusive(cs) ? '1' : '0';
   break;
- case FILE_MEM_EXCLUSIVE:
-  *s++ = is_mem_exclusive(cs) ? '1' : '0';
-  break;
- case FILE_MEMORY_MIGRATE:
-  *s++ = is_memory_migrate(cs) ? '1' : '0';
-  break;
- case FILE_MEMORY_PRESSURE_ENABLED:
-  *s++ = cpuset_memory_pressure_enabled ? '1' : '0';
-  break;
```

```
- case FILE_MEMORY_PRESSURE:
-  s += sprintf(s, "%d", fmeter_getrate(&cs->fmeter));
-  break;
- case FILE_SPREAD_PAGE:
-  *s++ = is_spread_page(cs) ? '1' : '0';
-  break;
- case FILE_SPREAD_SLAB:
-  *s++ = is_spread_slab(cs) ? '1' : '0';
-  break;
  default:
   retval = -EINVAL;
   goto out;
@@ -1061,13 +501,6 @@ static struct cftype cft_cpus = {
  .private = FILE_CPULIST,
 };

-static struct cftype cft_mems = {
- .name = "mems",
- .read = cpuset_common_file_read,
- .write = cpuset_common_file_write,
- .private = FILE_MEMLIST,
-};
-
 static struct cftype cft_cpu_exclusive = {
  .name = "cpu_exclusive",
  .read = cpuset_common_file_read,
@@ -1075,71 +508,14 @@ static struct cftype cft_cpu_exclusive =
  .private = FILE_CPU_EXCLUSIVE,
 };

-static struct cftype cft_mem_exclusive = {
- .name = "mem_exclusive",
- .read = cpuset_common_file_read,
- .write = cpuset_common_file_write,
- .private = FILE_MEM_EXCLUSIVE,
-};
-
-static struct cftype cft_memory_migrate = {
- .name = "memory_migrate",
- .read = cpuset_common_file_read,
- .write = cpuset_common_file_write,
- .private = FILE_MEMORY_MIGRATE,
-};
-
-static struct cftype cft_memory_pressure_enabled = {
- .name = "memory_pressure_enabled",
- .read = cpuset_common_file_read,
- .write = cpuset_common_file_write,
```

```
- .private = FILE_MEMORY_PRESSURE_ENABLED,
-};
-
-static struct cftype cft_memory_pressure = {
- .name = "memory_pressure",
- .read = cpuset_common_file_read,
- .write = cpuset_common_file_write,
- .private = FILE_MEMORY_PRESSURE,
-};
-
-static struct cftype cft_spread_page = {
- .name = "memory_spread_page",
- .read = cpuset_common_file_read,
- .write = cpuset_common_file_write,
- .private = FILE_SPREAD_PAGE,
-};
-
-static struct cftype cft_spread_slab = {
- .name = "memory_spread_slab",
- .read = cpuset_common_file_read,
- .write = cpuset_common_file_write,
- .private = FILE_SPREAD_SLAB,
-};
-
 int cpuset_populate(struct container_subsys *ss, struct container *cont)
 {
  int err;

  if ((err = container_add_file(cont, &cft_cpus)) < 0)
   return err;
- if ((err = container_add_file(cont, &cft_mems)) < 0)
-  return err;
  if ((err = container_add_file(cont, &cft_cpu_exclusive)) < 0)
   return err;
- if ((err = container_add_file(cont, &cft_mem_exclusive)) < 0)
-  return err;
- if ((err = container_add_file(cont, &cft_memory_migrate)) < 0)
-  return err;
- if ((err = container_add_file(cont, &cft_memory_pressure)) < 0)
-  return err;
- if ((err = container_add_file(cont, &cft_spread_page)) < 0)
-  return err;
- if ((err = container_add_file(cont, &cft_spread_slab)) < 0)
-  return err;
- /* memory_pressure_enabled is in root cpuset only */
- if (err == 0 && !cont->parent)
-  err = container_add_file(cont, &cft_memory_pressure_enabled);
  return 0;
```

```
 }

@@ -1161,7 +537,6 @@ int cpuset_create(struct container_subsy
   /* This is early initialization for the top container */
   set_container_cs(cont, &top_cpuset);
   top_cpuset.css.container = cont;
-  top_cpuset.mems_generation = cpuset_mems_generation++;
   return 0;
  }
  parent = container_cs(cont->parent);
@@ -1169,21 +544,12 @@ int cpuset_create(struct container_subsy
  if (!cs)
   return -ENOMEM;

- cpuset_update_task_memory_state();
  cs->flags = 0;
- if (is_spread_page(parent))
-  set_bit(CS_SPREAD_PAGE, &cs->flags);
- if (is_spread_slab(parent))
-  set_bit(CS_SPREAD_SLAB, &cs->flags);
  cs->cpus_allowed = CPU_MASK_NONE;
- cs->mems_allowed = NODE_MASK_NONE;
- cs->mems_generation = cpuset_mems_generation++;
- fmeter_init(&cs->fmeter);

  cs->parent = parent;
  set_container_cs(cont, cs);
  cs->css.container = cont;
- number_of_cpusets++;
  return 0;
 }

@@ -1202,12 +568,10 @@ void cpuset_destroy(struct container_sub
 {
  struct cpuset *cs = container_cs(cont);

- cpuset_update_task_memory_state();
  if (is_cpu_exclusive(cs)) {
   int retval = update_flag(CS_CPU_EXCLUSIVE, cs, "0");
   BUG_ON(retval);
  }
- number_of_cpusets--;
  kfree(cs);
 }

@@ -1217,7 +581,6 @@ static struct container_subsys cpuset_su
  .destroy  = cpuset_destroy,
  .can_attach = cpuset_can_attach,
```

```
  .attach = cpuset_attach,
- .post_attach = cpuset_post_attach,
  .populate = cpuset_populate,
  .subsys_id = -1,
 };
@@ -1232,7 +595,6 @@ int __init cpuset_init_early(void)
 {
  if (container_register_subsys(&cpuset_subsys) < 0)
   panic("Couldn't register cpuset subsystem");
- top_cpuset.mems_generation = cpuset_mems_generation++;
  return 0;
 }

@@ -1248,119 +610,76 @@ int __init cpuset_init(void)
  int err = 0;

  top_cpuset.cpus_allowed = CPU_MASK_ALL;
- top_cpuset.mems_allowed = NODE_MASK_ALL;
-
- fmeter_init(&top_cpuset.fmeter);
- top_cpuset.mems_generation = cpuset_mems_generation++;

  err = register_filesystem(&cpuset_fs_type);
  if (err < 0)
   return err;

- number_of_cpusets = 1;
  return 0;
 }

-#if defined(CONFIG_HOTPLUG_CPU) || defined(CONFIG_MEMORY_HOTPLUG)
+#if defined(CONFIG_HOTPLUG_CPU)
 /*
- * If common_cpu_mem_hotplug_unplug(), below, unplugs any CPUs
- * or memory nodes, we need to walk over the cpuset hierarchy,
- * removing that CPU or node from all cpusets.  If this removes the
- * last CPU or node from a cpuset, then the guarantee_online_cpus()
- * or guarantee_online_mems() code will use that emptied cpusets
- * parent online CPUs or nodes.  Cpusets that were already empty of
- * CPUs or nodes are left empty.
+ * If we unplug any CPUs or memory nodes, we need to walk over the
+ * cpuset hierarchy, removing that CPU from all cpusets.  If this
+ * removes the last CPU from a cpuset, then the
+ * guarantee_online_cpus() code will use that emptied cpusets parent
+ * online CPUs.  Cpusets that were already empty of CPUs are left
+ * empty.
  *
- * This routine is intentionally inefficient in a couple of regards.
```

```
- * It will check all cpusets in a subtree even if the top cpuset of
- * the subtree has no offline CPUs or nodes.  It checks both CPUs and
- * nodes, even though the caller could have been coded to know that
- * only one of CPUs or nodes needed to be checked on a given call.
- * This was done to minimize text size rather than cpu cycles.
+ * This routine will check all cpusets in a subtree even if the top
+ * cpuset of the subtree has no offline CPUs.
  *
  * Call with both manage_mutex and callback_mutex held.
  *
  * Recursive, on depth of cpuset subtree.
  */

-static void guarantee_online_cpus_mems_in_subtree(const struct cpuset *cur)
+static void guarantee_online_cpus_in_subtree(const struct cpuset *cur)
 {
  struct container *cont;
  struct cpuset *c;

- /* Each of our child cpusets mems must be online */
+ /* Each of our child cpusets cpus must be online */
  list_for_each_entry(cont, &cur->css.container->children, sibling) {
   c = container_cs(cont);
-  guarantee_online_cpus_mems_in_subtree(c);
+  guarantee_online_cpus_in_subtree(c);
   if (!cpus_empty(c->cpus_allowed))
    guarantee_online_cpus(c, &c->cpus_allowed);
-  if (!nodes_empty(c->mems_allowed))
-   guarantee_online_mems(c, &c->mems_allowed);
  }
 }

 /*
- * The cpus_allowed and mems_allowed nodemasks in the top_cpuset track
- * cpu_online_map and node_online_map.  Force the top cpuset to track
- * whats online after any CPU or memory node hotplug or unplug event.
+ * The top_cpuset tracks what CPUs are online,
+ * period.  This is necessary in order to make cpusets transparent
+ * (of no affect) on systems that are actively using CPU hotplug
+ * but making no active use of cpusets.
  *
- * To ensure that we don't remove a CPU or node from the top cpuset
+ * To ensure that we don't remove a CPU from the top cpuset
  * that is currently in use by a child cpuset (which would violate
  * the rule that cpusets must be subsets of their parent), we first
- * call the recursive routine guarantee_online_cpus_mems_in_subtree().
+ * call the recursive routine guarantee_online_cpus_in_subtree().
  *
```

```
- * Since there are two callers of this routine, one for CPU hotplug
- * events and one for memory node hotplug events, we could have coded
- * two separate routines here.  We code it as a single common routine
- * in order to minimize text size.
+ * This routine ensures that top_cpuset.cpus_allowed tracks
+ * cpu_online_map on each CPU hotplug (cpuhp) event.
  */

-static void common_cpu_mem_hotplug_unplug(void)
+static int cpuset_handle_cpuhp(struct notifier_block *nb,
+    unsigned long phase, void *cpu)
 {
  container_manage_lock();
  container_lock();

- guarantee_online_cpus_mems_in_subtree(&top_cpuset);
+ guarantee_online_cpus_in_subtree(&top_cpuset);
  top_cpuset.cpus_allowed = cpu_online_map;
- top_cpuset.mems_allowed = node_online_map;

  container_unlock();
  container_manage_unlock();
-}
-#endif

-#ifdef CONFIG_HOTPLUG_CPU
-/*
- * The top_cpuset tracks what CPUs and Memory Nodes are online,
- * period.  This is necessary in order to make cpusets transparent
- * (of no affect) on systems that are actively using CPU hotplug
- * but making no active use of cpusets.
- *
- * This routine ensures that top_cpuset.cpus_allowed tracks
- * cpu_online_map on each CPU hotplug (cpuhp) event.
- */
-
-static int cpuset_handle_cpuhp(struct notifier_block *nb,
-    unsigned long phase, void *cpu)
-{
- common_cpu_mem_hotplug_unplug();
  return 0;
 }
 #endif

-#ifdef CONFIG_MEMORY_HOTPLUG
-/*
- * Keep top_cpuset.mems_allowed tracking node_online_map.
- * Call this routine anytime after you change node_online_map.
```

```
- * See also the previous routine cpuset_handle_cpuhp().
- */
-
-void cpuset_track_online_nodes(void)
-{
- common_cpu_mem_hotplug_unplug();
-}
-#endif
-
 /**
  * cpuset_init_smp - initialize cpus_allowed
  *
@@ -1370,7 +689,6 @@ void cpuset_track_online_nodes(void)
 void __init cpuset_init_smp(void)
 {
  top_cpuset.cpus_allowed = cpu_online_map;
- top_cpuset.mems_allowed = node_online_map;

  hotcpu_notifier(cpuset_handle_cpuhp, 0);
 }
@@ -1398,249 +716,6 @@ cpumask_t cpuset_cpus_allowed(struct tas
  return mask;
 }

-void cpuset_init_current_mems_allowed(void)
-{
- current->mems_allowed = NODE_MASK_ALL;
-}
-
-/**
- * cpuset_mems_allowed - return mems_allowed mask from a tasks cpuset.
- * @tsk: pointer to task_struct from which to obtain cpuset->mems_allowed.
- *
- * Description: Returns the nodemask_t mems_allowed of the cpuset
- * attached to the specified @tsk.  Guaranteed to return some non-empty
- * subset of node_online_map, even if this means going outside the
- * tasks cpuset.
- **/
-
-nodemask_t cpuset_mems_allowed(struct task_struct *tsk)
-{
- nodemask_t mask;
-
- container_lock();
- task_lock(tsk);
- guarantee_online_mems(task_cs(tsk), &mask);
- task_unlock(tsk);
- container_unlock();
```

```
-
- return mask;
-}
-
-/**
- * cpuset_zonelist_valid_mems_allowed - check zonelist vs. curremt mems_allowed
- * @zl: the zonelist to be checked
- *
- * Are any of the nodes on zonelist zl allowed in current->mems_allowed?
- */
-int cpuset_zonelist_valid_mems_allowed(struct zonelist *zl)
-{
- int i;
-
- for (i = 0; zl->zones[i]; i++) {
-  int nid = zone_to_nid(zl->zones[i]);
-
-  if (node_isset(nid, current->mems_allowed))
-   return 1;
- }
- return 0;
-}
-
-/*
- * nearest_exclusive_ancestor() - Returns the nearest mem_exclusive
- * ancestor to the specified cpuset.  Call holding callback_mutex.
- * If no ancestor is mem_exclusive (an unusual configuration), then
- * returns the root cpuset.
- */
-static const struct cpuset *nearest_exclusive_ancestor(const struct cpuset *cs)
-{
- while (!is_mem_exclusive(cs) && cs->parent)
-  cs = cs->parent;
- return cs;
-}
-
-/**
- * cpuset_zone_allowed - Can we allocate memory on zone z's memory node?
- * @z: is this zone on an allowed node?
- * @gfp_mask: memory allocation flags (we use __GFP_HARDWALL)
- *
- * If we're in interrupt, yes, we can always allocate.  If zone
- * z's node is in our tasks mems_allowed, yes.  If it's not a
- * __GFP_HARDWALL request and this zone's nodes is in the nearest
- * mem_exclusive cpuset ancestor to this tasks cpuset, yes.
- * Otherwise, no.
- *
- * GFP_USER allocations are marked with the __GFP_HARDWALL bit,
```

```
- * and do not allow allocations outside the current tasks cpuset.
- * GFP_KERNEL allocations are not so marked, so can escape to the
- * nearest mem_exclusive ancestor cpuset.
- *
- * Scanning up parent cpusets requires callback_mutex.  The __alloc_pages()
- * routine only calls here with __GFP_HARDWALL bit _not_ set if
- * it's a GFP_KERNEL allocation, and all nodes in the current tasks
- * mems_allowed came up empty on the first pass over the zonelist.
- * So only GFP_KERNEL allocations, if all nodes in the cpuset are
- * short of memory, might require taking the callback_mutex mutex.
- *
- * The first call here from mm/page_alloc:get_page_from_freelist()
- * has __GFP_HARDWALL set in gfp_mask, enforcing hardwall cpusets, so
- * no allocation on a node outside the cpuset is allowed (unless in
- * interrupt, of course).
- *
- * The second pass through get_page_from_freelist() doesn't even call
- * here for GFP_ATOMIC calls.  For those calls, the __alloc_pages()
- * variable 'wait' is not set, and the bit ALLOC_CPUSET is not set
- * in alloc_flags.  That logic and the checks below have the combined
- * affect that:
- * in_interrupt - any node ok (current task context irrelevant)
- * GFP_ATOMIC   - any node ok
- * GFP_KERNEL   - any node in enclosing mem_exclusive cpuset ok
- * GFP_USER     - only nodes in current tasks mems allowed ok.
- *
- * Rule:
- *    Don't call cpuset_zone_allowed() if you can't sleep, unless you
- *    pass in the __GFP_HARDWALL flag set in gfp_flag, which disables
- *    the code that might scan up ancestor cpusets and sleep.
- **/
-
-int __cpuset_zone_allowed(struct zone *z, gfp_t gfp_mask)
-{
- int node;   /* node that zone z is on */
- const struct cpuset *cs; /* current cpuset ancestors */
- int allowed;   /* is allocation in zone z allowed? */
-
- if (in_interrupt() || (gfp_mask & __GFP_THISNODE))
-  return 1;
- node = zone_to_nid(z);
- might_sleep_if(!(gfp_mask & __GFP_HARDWALL));
- if (node_isset(node, current->mems_allowed))
-  return 1;
- if (gfp_mask & __GFP_HARDWALL) /* If hardwall request, stop here */
-  return 0;
-
- if (current->flags & PF_EXITING) /* Let dying task have memory */
```

```
-  return 1;
-
- /* Not hardwall and node outside mems_allowed: scan up cpusets */
- container_lock();
-
- task_lock(current);
- cs = nearest_exclusive_ancestor(task_cs(current));
- task_unlock(current);
-
- allowed = node_isset(node, cs->mems_allowed);
- container_unlock();
- return allowed;
-}
-
-/**
- * cpuset_mem_spread_node() - On which node to begin search for a page
- *
- * If a task is marked PF_SPREAD_PAGE or PF_SPREAD_SLAB (as for
- * tasks in a cpuset with is_spread_page or is_spread_slab set),
- * and if the memory allocation used cpuset_mem_spread_node()
- * to determine on which node to start looking, as it will for
- * certain page cache or slab cache pages such as used for file
- * system buffers and inode caches, then instead of starting on the
- * local node to look for a free page, rather spread the starting
- * node around the tasks mems_allowed nodes.
- *
- * We don't have to worry about the returned node being offline
- * because "it can't happen", and even if it did, it would be ok.
- *
- * The routines calling guarantee_online_mems() are careful to
- * only set nodes in task->mems_allowed that are online.  So it
- * should not be possible for the following code to return an
- * offline node.  But if it did, that would be ok, as this routine
- * is not returning the node where the allocation must be, only
- * the node where the search should start.  The zonelist passed to
- * __alloc_pages() will include all nodes.  If the slab allocator
- * is passed an offline node, it will fall back to the local node.
- * See kmem_cache_alloc_node().
- */
-
-int cpuset_mem_spread_node(void)
-{
- int node;
-
- node = next_node(current->cpuset_mem_spread_rotor, current->mems_allowed);
- if (node == MAX_NUMNODES)
-  node = first_node(current->mems_allowed);
- current->cpuset_mem_spread_rotor = node;
```

```
- return node;
-}
-EXPORT_SYMBOL_GPL(cpuset_mem_spread_node);
-
-/**
- * cpuset_excl_nodes_overlap - Do we overlap @p's mem_exclusive ancestors?
- * @p: pointer to task_struct of some other task.
- *
- * Description: Return true if the nearest mem_exclusive ancestor
- * cpusets of tasks @p and current overlap.  Used by oom killer to
- * determine if task @p's memory usage might impact the memory
- * available to the current task.
- *
- * Call while holding callback_mutex.
- **/
-
-int cpuset_excl_nodes_overlap(const struct task_struct *p)
-{
- const struct cpuset *cs1, *cs2; /* my and p's cpuset ancestors */
- int overlap = 1;  /* do cpusets overlap? */
-
- task_lock(current);
- if (current->flags & PF_EXITING) {
-  task_unlock(current);
-  goto done;
- }
- cs1 = nearest_exclusive_ancestor(task_cs(current));
- task_unlock(current);
-
- task_lock((struct task_struct *)p);
- if (p->flags & PF_EXITING) {
-  task_unlock((struct task_struct *)p);
-  goto done;
- }
- cs2 = nearest_exclusive_ancestor(task_cs((struct task_struct *)p));
- task_unlock((struct task_struct *)p);
-
- overlap = nodes_intersects(cs1->mems_allowed, cs2->mems_allowed);
-done:
- return overlap;
-}
-
-/*
- * Collection of memory_pressure is suppressed unless
- * this flag is enabled by writing "1" to the special
- * cpuset file 'memory_pressure_enabled' in the root cpuset.
- */
-
```

```
-int cpuset_memory_pressure_enabled __read_mostly;
-
-/**
- * cpuset_memory_pressure_bump - keep stats of per-cpuset reclaims.
- *
- * Keep a running average of the rate of synchronous (direct)
- * page reclaim efforts initiated by tasks in each cpuset.
- *
- * This represents the rate at which some task in the cpuset
- * ran low on memory on all nodes it was allowed to use, and
- * had to enter the kernels page reclaim code in an effort to
- * create more free memory by tossing clean pages or swapping
- * or writing dirty pages.
- *
- * Display to user space in the per-cpuset read-only file
- * "memory_pressure".  Value displayed is an integer
- * representing the recent rate of entry into the synchronous
- * (direct) page reclaim by any task attached to the cpuset.
- **/
-
-void __cpuset_memory_pressure_bump(void)
-{
- task_lock(current);
- fmeter_markevent(&task_cs(current)->fmeter);
- task_unlock(current);
-}
-
 #ifdef CONFIG_PROC_PID_CPUSET
 /*
  * proc_cpuset_show()
@@ -1709,8 +784,5 @@ char *cpuset_task_status_allowed(struct
  buffer += sprintf(buffer, "Cpus_allowed:\t");
  buffer += cpumask_scnprintf(buffer, PAGE_SIZE, task->cpus_allowed);
  buffer += sprintf(buffer, "\n");
- buffer += sprintf(buffer, "Mems_allowed:\t");
- buffer += nodemask_scnprintf(buffer, PAGE_SIZE, task->mems_allowed);
- buffer += sprintf(buffer, "\n");
  return buffer;
 }
Index: container-2.6.19-rc6/kernel/memset.c
===================================================================
--- /dev/null
+++ container-2.6.19-rc6/kernel/memset.c
@@ -0,0 +1,1352 @@
+/*
+ *  kernel/memset.c
+ *
+ *  Processor and Memory placement constraints for sets of tasks.
```

```
+ *
+ *  Copyright (C) 2003 BULL SA.
+ *  Copyright (C) 2004-2006 Silicon Graphics, Inc.
+ *  Copyright (C) 2006 Google, Inc
+ *
+ *  Portions derived from Patrick Mochel's sysfs code.
+ *  sysfs is Copyright (c) 2001-3 Patrick Mochel
+ *
+ *  2003-10-10 Written by Simon Derr.
+ *  2003-10-22 Updates by Stephen Hemminger.
+ *  2004 May-July Rework by Paul Jackson.
+ *  2006 Rework by Paul Menage to use generic containers
+ *
+ *  This file is subject to the terms and conditions of the GNU General Public
+ *  License.  See the file COPYING in the main directory of the Linux
+ *  distribution for more details.
+ */
+
+#include <linux/cpu.h>
+#include <linux/cpumask.h>
+#include <linux/memset.h>
+#include <linux/err.h>
+#include <linux/errno.h>
+#include <linux/file.h>
+#include <linux/fs.h>
+#include <linux/init.h>
+#include <linux/interrupt.h>
+#include <linux/kernel.h>
+#include <linux/kmod.h>
+#include <linux/list.h>
+#include <linux/mempolicy.h>
+#include <linux/mm.h>
+#include <linux/module.h>
+#include <linux/mount.h>
+#include <linux/namei.h>
+#include <linux/pagemap.h>
+#include <linux/proc_fs.h>
+#include <linux/rcupdate.h>
+#include <linux/sched.h>
+#include <linux/seq_file.h>
+#include <linux/security.h>
+#include <linux/slab.h>
+#include <linux/smp_lock.h>
+#include <linux/spinlock.h>
+#include <linux/stat.h>
+#include <linux/string.h>
+#include <linux/time.h>
+#include <linux/backing-dev.h>
```

```
+#include <linux/sort.h>
+
+#include <asm/uaccess.h>
+#include <asm/atomic.h>
+#include <linux/mutex.h>
+
+/*
+ * Tracks how many memsets are currently defined in system.
+ * When there is only one memset (the root memset) we can
+ * short circuit some hooks.
+ */
+int number_of_memsets __read_mostly;
+
+/* Retrieve the memset from a container */
+static struct container_subsys memset_subsys;
+struct memset;
+
+/* See "Frequency meter" comments, below. */
+
+struct fmeter {
+ int cnt;  /* unprocessed events count */
+ int val;  /* most recent output value */
+ time_t time;  /* clock (secs) when val computed */
+ spinlock_t lock; /* guards read or write of above */
+};
+
+struct memset {
+ struct container_subsys_state css;
+
+ unsigned long flags;  /* "unsigned long" so bitops work */
+ nodemask_t mems_allowed; /* Memory Nodes allowed to tasks */
+
+ struct memset *parent;  /* my parent */
+
+ /*
+  * Copy of global memset_mems_generation as of the most
+  * recent time this memset changed its mems_allowed.
+  */
+ int mems_generation;
+
+ struct fmeter fmeter;  /* memory_pressure filter */
+};
+
+/* Update the memset for a container */
+static inline void set_container_ms(struct container *cont, struct memset *ms)
+{
+ cont->subsys[memset_subsys.subsys_id] = &ms->css;
+}
```

```
+
+/* Retrieve the memset for a container */
+static inline struct memset *container_ms(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, &memset_subsys),
+       struct memset, css);
+}
+
+/* Retrieve the memset for a task */
+static inline struct memset *task_ms(struct task_struct *task)
+{
+ return container_ms(task_container(task, &memset_subsys));
+}
+
+
+/* bits in struct memset flags field */
+typedef enum {
+ MS_CPU_EXCLUSIVE,
+ MS_MEM_EXCLUSIVE,
+ MS_MEMORY_MIGRATE,
+ MS_SPREAD_PAGE,
+ MS_SPREAD_SLAB,
+} memset_flagbits_t;
+
+/* convenient tests for these bits */
+static inline int is_cpu_exclusive(const struct memset *ms)
+{
+ return test_bit(MS_CPU_EXCLUSIVE, &ms->flags);
+}
+
+static inline int is_mem_exclusive(const struct memset *ms)
+{
+ return test_bit(MS_MEM_EXCLUSIVE, &ms->flags);
+}
+
+static inline int is_memory_migrate(const struct memset *ms)
+{
+ return test_bit(MS_MEMORY_MIGRATE, &ms->flags);
+}
+
+static inline int is_spread_page(const struct memset *ms)
+{
+ return test_bit(MS_SPREAD_PAGE, &ms->flags);
+}
+
+static inline int is_spread_slab(const struct memset *ms)
+{
+ return test_bit(MS_SPREAD_SLAB, &ms->flags);
```

```
+}
+
+/*
+ * Increment this integer everytime any memset changes its
+ * mems_allowed value.  Users of memsets can track this generation
+ * number, and avoid having to lock and reload mems_allowed unless
+ * the memset they're using changes generation.
+ *
+ * A single, global generation is needed because attach_task() could
+ * reattach a task to a different memset, which must not have its
+ * generation numbers aliased with those of that tasks previous memset.
+ *
+ * Generations are needed for mems_allowed because one task cannot
+ * modify anothers memory placement.  So we must enable every task,
+ * on every visit to __alloc_pages(), to efficiently check whether
+ * its current->memset->mems_allowed has changed, requiring an update
+ * of its current->mems_allowed.
+ *
+ * Since memset_mems_generation is guarded by manage_mutex,
+ * there is no need to mark it atomic.
+ */
+static int memset_mems_generation;
+
+static struct memset top_memset = {
+ .flags = 1 << MS_MEM_EXCLUSIVE,
+ .mems_allowed = NODE_MASK_ALL,
+};
+
+/*
+ * Return in *pmask the portion of a memsets's mems_allowed that
+ * are online.  If none are online, walk up the memset hierarchy
+ * until we find one that does have some online mems.  If we get
+ * all the way to the top and still haven't found any online mems,
+ * return node_online_map.
+ *
+ * One way or another, we guarantee to return some non-empty subset
+ * of node_online_map.
+ *
+ * Call with callback_mutex held.
+ */
+
+static void guarantee_online_mems(const struct memset *ms, nodemask_t *pmask)
+{
+ while (ms && !nodes_intersects(ms->mems_allowed, node_online_map))
+  ms = ms->parent;
+ if (ms)
+  nodes_and(*pmask, ms->mems_allowed, node_online_map);
+ else
```

```
+  *pmask = node_online_map;
+ BUG_ON(!nodes_intersects(*pmask, node_online_map));
+}
+
+/**
+ * memset_update_task_memory_state - update task memory placement
+ *
+ * If the current tasks memsets mems_allowed changed behind our
+ * backs, update current->mems_allowed, mems_generation and task NUMA
+ * mempolicy to the new value.
+ *
+ * Task mempolicy is updated by rebinding it relative to the
+ * current->memset if a task has its memory placement changed.
+ * Do not call this routine if in_interrupt().
+ *
+ * Call without callback_mutex or task_lock() held.  May be
+ * called with or without manage_mutex held.  Thanks in part to
+ * 'the_top_memset_hack', the tasks memset pointer will never
+ * be NULL.  This routine also might acquire callback_mutex and
+ * current->mm->mmap_sem during call.
+ *
+ * Reading current->memset->mems_generation doesn't need task_lock
+ * to guard the current->memset derefence, because it is guarded
+ * from concurrent freeing of current->memset by attach_task(),
+ * using RCU.
+ *
+ * The rcu_dereference() is technically probably not needed,
+ * as I don't actually mind if I see a new memset pointer but
+ * an old value of mems_generation.  However this really only
+ * matters on alpha systems using memsets heavily.  If I dropped
+ * that rcu_dereference(), it would save them a memory barrier.
+ * For all other arch's, rcu_dereference is a no-op anyway, and for
+ * alpha systems not using memsets, another planned optimization,
+ * avoiding the rcu critical section for tasks in the root memset
+ * which is statically allocated, so can't vanish, will make this
+ * irrelevant.  Better to use RCU as intended, than to engage in
+ * some cute trick to save a memory barrier that is impossible to
+ * test, for alpha systems using memsets heavily, which might not
+ * even exist.
+ *
+ * This routine is needed to update the per-task mems_allowed data,
+ * within the tasks context, when it is trying to allocate memory
+ * (in various mm/mempolicy.c routines) and notices that some other
+ * task has been modifying its memset.
+ */
+
+void memset_update_task_memory_state(void)
+{
```

```
+ int my_memsets_mem_gen;
+ struct task_struct *tsk = current;
+ struct memset *ms;
+
+ if (task_ms(tsk) == &top_memset) {
+  /* Don't need rcu for top_memset.  It's never freed. */
+  my_memsets_mem_gen = top_memset.mems_generation;
+ } else {
+  rcu_read_lock();
+  my_memsets_mem_gen = task_ms(current)->mems_generation;
+  rcu_read_unlock();
+ }
+
+ if (my_memsets_mem_gen != tsk->memset_mems_generation) {
+  container_lock();
+  task_lock(tsk);
+  ms = task_ms(tsk); /* Maybe changed when task not locked */
+  guarantee_online_mems(ms, &tsk->mems_allowed);
+  tsk->memset_mems_generation = ms->mems_generation;
+  if (is_spread_page(ms))
+   tsk->flags |= PF_SPREAD_PAGE;
+  else
+   tsk->flags &= ~PF_SPREAD_PAGE;
+  if (is_spread_slab(ms))
+   tsk->flags |= PF_SPREAD_SLAB;
+  else
+   tsk->flags &= ~PF_SPREAD_SLAB;
+  task_unlock(tsk);
+  container_unlock();
+  mpol_rebind_task(tsk, &tsk->mems_allowed);
+ }
+}
+
+/*
+ * is_memset_subset(p, q) - Is memset p a subset of memset q?
+ *
+ * One memset is a subset of another if all its allowed
+ * Memory Nodes are a subset of the other, and its exclusive flags
+ * are only set if the other's are set.  Call holding manage_mutex.
+ */
+
+static int is_memset_subset(const struct memset *p, const struct memset *q)
+{
+ return nodes_subset(p->mems_allowed, q->mems_allowed) &&
+  is_mem_exclusive(p) <= is_mem_exclusive(q);
+}
+
+/*
```

```
+ * validate_change() - Used to validate that any proposed memset change
+ *        follows the structural rules for memsets.
+ *
+ * If we replaced the flag and mask values of the current memset
+ * (cur) with those values in the trial memset (trial), would
+ * our various subset and exclusive rules still be valid?  Presumes
+ * manage_mutex held.
+ *
+ * 'cur' is the address of an actual, in-use memset.  Operations
+ * such as list traversal that depend on the actual address of the
+ * memset in the list must use cur below, not trial.
+ *
+ * 'trial' is the address of bulk structure copy of cur, with
+ * perhaps one or more of the fields mems_allowed,
+ * or flags changed to new, trial values.
+ *
+ * Return 0 if valid, -errno if not.
+ */
+
+static int validate_change(const struct memset *cur, const struct memset *trial)
+{
+ struct container *cont;
+ struct memset *c, *par;
+
+ /* Each of our child memsets must be a subset of us */
+ list_for_each_entry(cont, &cur->css.container->children, sibling) {
+  if (!is_memset_subset(container_ms(cont), trial))
+   return -EBUSY;
+ }
+
+ /* Remaining checks don't apply to root memset */
+ if ((par = cur->parent) == NULL)
+  return 0;
+
+ /* We must be a subset of our parent memset */
+ if (!is_memset_subset(trial, par))
+  return -EACCES;
+
+ /* If either I or some sibling (!= me) is exclusive, we can't overlap */
+ list_for_each_entry(cont, &par->css.container->children, sibling) {
+  c = container_ms(cont);
+  if ((is_mem_exclusive(trial) || is_mem_exclusive(c)) &&
+      c != cur &&
+      nodes_intersects(trial->mems_allowed, c->mems_allowed))
+   return -EINVAL;
+ }
+
+ return 0;
```

```
+}
+
+/*
+ * memset_migrate_mm
+ *
+ *    Migrate memory region from one set of nodes to another.
+ *
+ *    Temporarilly set tasks mems_allowed to target nodes of migration,
+ *    so that the migration code can allocate pages on these nodes.
+ *
+ *    Call holding manage_mutex, so our current->memset won't change
+ *    during this call, as manage_mutex holds off any attach_task()
+ *    calls.  Therefore we don't need to take task_lock around the
+ *    call to guarantee_online_mems(), as we know no one is changing
+ *    our tasks memset.
+ *
+ *    Hold callback_mutex around the two modifications of our tasks
+ *    mems_allowed to synchronize with memset_mems_allowed().
+ *
+ *    While the mm_struct we are migrating is typically from some
+ *    other task, the task_struct mems_allowed that we are hacking
+ *    is for our current task, which must allocate new pages for that
+ *    migrating memory region.
+ *
+ *    We call memset_update_task_memory_state() before hacking
+ *    our tasks mems_allowed, so that we are assured of being in
+ *    sync with our tasks memset, and in particular, callbacks to
+ *    memset_update_task_memory_state() from nested page allocations
+ *    won't see any mismatch of our memset and task mems_generation
+ *    values, so won't overwrite our hacked tasks mems_allowed
+ *    nodemask.
+ */
+
+static void memset_migrate_mm(struct mm_struct *mm, const nodemask_t *from,
+       const nodemask_t *to)
+{
+ struct task_struct *tsk = current;
+
+ memset_update_task_memory_state();
+
+ container_lock();
+ tsk->mems_allowed = *to;
+ container_unlock();
+
+ do_migrate_pages(mm, from, to, MPOL_MF_MOVE_ALL);
+
+ container_lock();
+ guarantee_online_mems(task_ms(tsk),&tsk->mems_allowed);
```

```
+ container_unlock();
+}
+
+/*
+ * Handle user request to change the 'mems' memory placement
+ * of a memset.  Needs to validate the request, update the
+ * memsets mems_allowed and mems_generation, and for each
+ * task in the memset, rebind any vma mempolicies and if
+ * the memset is marked 'memory_migrate', migrate the tasks
+ * pages to the new memory.
+ *
+ * Call with manage_mutex held.  May take callback_mutex during call.
+ * Will take tasklist_lock, scan tasklist for tasks in memset ms,
+ * lock each such tasks mm->mmap_sem, scan its vma's and rebind
+ * their mempolicies to the memsets new mems_allowed.
+ */
+
+static void *memset_being_rebound;
+
+static int update_nodemask(struct memset *ms, char *buf)
+{
+ struct memset trialms;
+ nodemask_t oldmem;
+ struct task_struct *g, *p;
+ struct mm_struct **mmarray;
+ int i, n, ntasks;
+ int migrate;
+ int fudge;
+ int retval;
+ struct container *cont;
+
+ /* top_memset.mems_allowed tracks node_online_map; it's read-only */
+ if (ms == &top_memset)
+  return -EACCES;
+
+ trialms = *ms;
+ cont = ms->css.container;
+ retval = nodelist_parse(buf, trialms.mems_allowed);
+ if (retval < 0)
+  goto done;
+ nodes_and(trialms.mems_allowed, trialms.mems_allowed, node_online_map);
+ oldmem = ms->mems_allowed;
+ if (nodes_equal(oldmem, trialms.mems_allowed)) {
+  retval = 0;  /* Too easy - nothing to do */
+  goto done;
+ }
+ if (nodes_empty(trialms.mems_allowed)) {
+  retval = -ENOSPC;
```

```
+  goto done;
+ }
+ retval = validate_change(ms, &trialms);
+ if (retval < 0)
+  goto done;
+
+ container_lock();
+ ms->mems_allowed = trialms.mems_allowed;
+ ms->mems_generation = memset_mems_generation++;
+ container_unlock();
+
+ memset_being_rebound = ms;  /* causes mpol_copy() rebind */
+
+ fudge = 10;    /* spare mmarray[] slots */
+ fudge += cpus_weight(cpu_online_map); /* imagine one fork-bomb/cpu */
+ retval = -ENOMEM;
+
+ /*
+  * Allocate mmarray[] to hold mm reference for each task
+  * in memset ms.  Can't kmalloc GFP_KERNEL while holding
+  * tasklist_lock.  We could use GFP_ATOMIC, but with a
+  * few more lines of code, we can retry until we get a big
+  * enough mmarray[] w/o using GFP_ATOMIC.
+  */
+ while (1) {
+  ntasks = atomic_read(&ms->css.container->count);  /* guess */
+  ntasks += fudge;
+  mmarray = kmalloc(ntasks * sizeof(*mmarray), GFP_KERNEL);
+  if (!mmarray)
+   goto done;
+  write_lock_irq(&tasklist_lock);  /* block fork */
+  if (atomic_read(&ms->css.container->count) <= ntasks)
+   break;    /* got enough */
+  write_unlock_irq(&tasklist_lock); /* try again */
+  kfree(mmarray);
+ }
+
+ n = 0;
+
+ /* Load up mmarray[] with mm reference for each task in memset. */
+ do_each_thread(g, p) {
+  struct mm_struct *mm;
+
+  if (n >= ntasks) {
+   printk(KERN_WARNING
+    "Memset mempolicy rebind incomplete.\n");
+   continue;
+  }
```

```
+  if (task_ms(p) != ms)
+   continue;
+  mm = get_task_mm(p);
+  if (!mm)
+   continue;
+  mmarray[n++] = mm;
+ } while_each_thread(g, p);
+ write_unlock_irq(&tasklist_lock);
+
+ /*
+  * Now that we've dropped the tasklist spinlock, we can
+  * rebind the vma mempolicies of each mm in mmarray[] to their
+  * new memset, and release that mm.  The mpol_rebind_mm()
+  * call takes mmap_sem, which we couldn't take while holding
+  * tasklist_lock.  Forks can happen again now - the mpol_copy()
+  * memset_being_rebound check will catch such forks, and rebind
+  * their vma mempolicies too.  Because we still hold the global
+  * memset manage_mutex, we know that no other rebind effort will
+  * be contending for the global variable memset_being_rebound.
+  * It's ok if we rebind the same mm twice; mpol_rebind_mm()
+  * is idempotent.  Also migrate pages in each mm to new nodes.
+  */
+ migrate = is_memory_migrate(ms);
+ for (i = 0; i < n; i++) {
+  struct mm_struct *mm = mmarray[i];
+
+  mpol_rebind_mm(mm, &ms->mems_allowed);
+  if (migrate)
+   memset_migrate_mm(mm, &oldmem, &ms->mems_allowed);
+  mmput(mm);
+ }
+
+ /* We're done rebinding vma's to this memsets new mems_allowed. */
+ kfree(mmarray);
+ memset_being_rebound = NULL;
+ retval = 0;
+done:
+ return retval;
+}
+
+int current_memset_is_being_rebound(void)
+{
+ return task_ms(current) == memset_being_rebound;
+}
+
+/*
+ * Call with manage_mutex held.
+ */
```

```
+
+static int update_memory_pressure_enabled(struct memset *ms, char *buf)
+{
+ if (simple_strtoul(buf, NULL, 10) != 0)
+  memset_memory_pressure_enabled = 1;
+ else
+  memset_memory_pressure_enabled = 0;
+ return 0;
+}
+
+/*
+ * update_flag - read a 0 or a 1 in a file and update associated flag
+ * bit: the bit to update (MS_MEM_EXCLUSIVE,
+ *    MS_NOTIFY_ON_RELEASE, MS_MEMORY_MIGRATE,
+ *    MS_SPREAD_PAGE, MS_SPREAD_SLAB)
+ * ms: the memset to update
+ * buf: the buffer where we read the 0 or 1
+ *
+ * Call with manage_mutex held.
+ */
+
+static int update_flag(memset_flagbits_t bit, struct memset *ms, char *buf)
+{
+ int turning_on;
+ struct memset trialms;
+ int err;
+
+ turning_on = (simple_strtoul(buf, NULL, 10) != 0);
+
+ trialms = *ms;
+ if (turning_on)
+  set_bit(bit, &trialms.flags);
+ else
+  clear_bit(bit, &trialms.flags);
+
+ err = validate_change(ms, &trialms);
+ if (err < 0)
+  return err;
+ container_lock();
+ if (turning_on)
+  set_bit(bit, &ms->flags);
+ else
+  clear_bit(bit, &ms->flags);
+ container_unlock();
+
+ return 0;
+}
+
```

```
+/*
+ * Frequency meter - How fast is some event occurring?
+ *
+ * These routines manage a digitally filtered, constant time based,
+ * event frequency meter.  There are four routines:
+ *   fmeter_init() - initialize a frequency meter.
+ *   fmeter_markevent() - called each time the event happens.
+ *   fmeter_getrate() - returns the recent rate of such events.
+ *   fmeter_update() - internal routine used to update fmeter.
+ *
+ * A common data structure is passed to each of these routines,
+ * which is used to keep track of the state required to manage the
+ * frequency meter and its digital filter.
+ *
+ * The filter works on the number of events marked per unit time.
+ * The filter is single-pole low-pass recursive (IIR).  The time unit
+ * is 1 second.  Arithmetic is done using 32-bit integers scaled to
+ * simulate 3 decimal digits of precision (multiplied by 1000).
+ *
+ * With an FM_COEF of 933, and a time base of 1 second, the filter
+ * has a half-life of 10 seconds, meaning that if the events quit
+ * happening, then the rate returned from the fmeter_getrate()
+ * will be cut in half each 10 seconds, until it converges to zero.
+ *
+ * It is not worth doing a real infinitely recursive filter.  If more
+ * than FM_MAXTICKS ticks have elapsed since the last filter event,
+ * just compute FM_MAXTICKS ticks worth, by which point the level
+ * will be stable.
+ *
+ * Limit the count of unprocessed events to FM_MAXCNT, so as to avoid
+ * arithmetic overflow in the fmeter_update() routine.
+ *
+ * Given the simple 32 bit integer arithmetic used, this meter works
+ * best for reporting rates between one per millisecond (msec) and
+ * one per 32 (approx) seconds.  At constant rates faster than one
+ * per msec it maxes out at values just under 1,000,000.  At constant
+ * rates between one per msec, and one per second it will stabilize
+ * to a value N*1000, where N is the rate of events per second.
+ * At constant rates between one per second and one per 32 seconds,
+ * it will be choppy, moving up on the seconds that have an event,
+ * and then decaying until the next event.  At rates slower than
+ * about one in 32 seconds, it decays all the way back to zero between
+ * each event.
+ */
+
+#define FM_COEF 933  /* coefficient for half-life of 10 secs */
+#define FM_MAXTICKS ((time_t)99) /* useless computing more ticks than this */
+#define FM_MAXCNT 1000000 /* limit cnt to avoid overflow */
```

```
+#define FM_SCALE 1000  /* faux fixed point scale */
+
+/* Initialize a frequency meter */
+static void fmeter_init(struct fmeter *fmp)
+{
+ fmp->cnt = 0;
+ fmp->val = 0;
+ fmp->time = 0;
+ spin_lock_init(&fmp->lock);
+}
+
+/* Internal meter update - process cnt events and update value */
+static void fmeter_update(struct fmeter *fmp)
+{
+ time_t now = get_seconds();
+ time_t ticks = now - fmp->time;
+
+ if (ticks == 0)
+  return;
+
+ ticks = min(FM_MAXTICKS, ticks);
+ while (ticks-- > 0)
+  fmp->val = (FM_COEF * fmp->val) / FM_SCALE;
+ fmp->time = now;
+
+ fmp->val += ((FM_SCALE - FM_COEF) * fmp->cnt) / FM_SCALE;
+ fmp->cnt = 0;
+}
+
+/* Process any previous ticks, then bump cnt by one (times scale). */
+static void fmeter_markevent(struct fmeter *fmp)
+{
+ spin_lock(&fmp->lock);
+ fmeter_update(fmp);
+ fmp->cnt = min(FM_MAXCNT, fmp->cnt + FM_SCALE);
+ spin_unlock(&fmp->lock);
+}
+
+/* Process any previous ticks, then return current value. */
+static int fmeter_getrate(struct fmeter *fmp)
+{
+ int val;
+
+ spin_lock(&fmp->lock);
+ fmeter_update(fmp);
+ val = fmp->val;
+ spin_unlock(&fmp->lock);
+ return val;
```

```
+}
+
+int memset_can_attach(struct container_subsys *ss,
+        struct container *cont, struct task_struct *tsk)
+{
+ struct memset *ms = container_ms(cont);
+
+ if (nodes_empty(ms->mems_allowed))
+  return -ENOSPC;
+ return 0;
+}
+
+void memset_post_attach(struct container_subsys *ss,
+   struct container *cont,
+   struct container *oldcont,
+   struct task_struct *tsk)
+{
+ nodemask_t from, to;
+ struct mm_struct *mm;
+ struct memset *ms = container_ms(cont);
+ struct memset *oldms = container_ms(oldcont);
+
+ from = oldms->mems_allowed;
+ to = ms->mems_allowed;
+ mm = get_task_mm(tsk);
+ if (mm) {
+  mpol_rebind_mm(mm, &to);
+  if (is_memory_migrate(ms))
+   memset_migrate_mm(mm, &from, &to);
+  mmput(mm);
+ }
+
+}
+
+/* The various types of files and directories in a memset file system */
+
+typedef enum {
+ FILE_MEMORY_MIGRATE,
+ FILE_MEMLIST,
+ FILE_MEM_EXCLUSIVE,
+ FILE_MEMORY_PRESSURE_ENABLED,
+ FILE_MEMORY_PRESSURE,
+ FILE_SPREAD_PAGE,
+ FILE_SPREAD_SLAB,
+} memset_filetype_t;
+
+static ssize_t memset_common_file_write(struct container *cont,
+     struct cftype *cft,
```

```
+     struct file *file,
+     const char __user *userbuf,
+     size_t nbytes, loff_t *unused_ppos)
+{
+ struct memset *ms = container_ms(cont);
+ memset_filetype_t type = cft->private;
+ char *buffer;
+ int retval = 0;
+
+ /* Crude upper limit on largest legitimate list user might write. */
+ if (nbytes > 100 + 6 * MAX_NUMNODES)
+  return -E2BIG;
+
+ /* +1 for nul-terminator */
+ if ((buffer = kmalloc(nbytes + 1, GFP_KERNEL)) == 0)
+  return -ENOMEM;
+
+ if (copy_from_user(buffer, userbuf, nbytes)) {
+  retval = -EFAULT;
+  goto out1;
+ }
+ buffer[nbytes] = 0; /* nul-terminate */
+
+ container_manage_lock();
+
+ if (container_is_removed(cont)) {
+  retval = -ENODEV;
+  goto out2;
+ }
+
+ switch (type) {
+ case FILE_MEMLIST:
+  retval = update_nodemask(ms, buffer);
+  break;
+ case FILE_MEM_EXCLUSIVE:
+  retval = update_flag(MS_MEM_EXCLUSIVE, ms, buffer);
+  break;
+ case FILE_MEMORY_MIGRATE:
+  retval = update_flag(MS_MEMORY_MIGRATE, ms, buffer);
+  break;
+ case FILE_MEMORY_PRESSURE_ENABLED:
+  retval = update_memory_pressure_enabled(ms, buffer);
+  break;
+ case FILE_MEMORY_PRESSURE:
+  retval = -EACCES;
+  break;
+ case FILE_SPREAD_PAGE:
+  retval = update_flag(MS_SPREAD_PAGE, ms, buffer);
```

```
+  ms->mems_generation = memset_mems_generation++;
+  break;
+ case FILE_SPREAD_SLAB:
+  retval = update_flag(MS_SPREAD_SLAB, ms, buffer);
+  ms->mems_generation = memset_mems_generation++;
+  break;
+ default:
+  retval = -EINVAL;
+  goto out2;
+ }
+
+ if (retval == 0)
+  retval = nbytes;
+out2:
+ container_manage_unlock();
+out1:
+ kfree(buffer);
+ return retval;
+}
+
+/*
+ * These ascii lists should be read in a single call, by using a user
+ * buffer large enough to hold the entire map.  If read in smaller
+ * chunks, there is no guarantee of atomicity.  Since the display format
+ * used, list of ranges of sequential numbers, is variable length,
+ * and since these maps can change value dynamically, one could read
+ * gibberish by doing partial reads while a list was changing.
+ * A single large read to a buffer that crosses a page boundary is
+ * ok, because the result being copied to user land is not recomputed
+ * across a page fault.
+ */
+
+static int memset_sprintf_memlist(char *page, struct memset *ms)
+{
+ nodemask_t mask;
+
+ container_lock();
+ mask = ms->mems_allowed;
+ container_unlock();
+
+ return nodelist_scnprintf(page, PAGE_SIZE, mask);
+}
+
+static ssize_t memset_common_file_read(struct container *cont,
+           struct cftype *cft,
+           struct file *file,
+           char __user *buf,
+           size_t nbytes, loff_t *ppos)
```

```
+{
+ struct memset *ms = container_ms(cont);
+ memset_filetype_t type = cft->private;
+ char *page;
+ ssize_t retval = 0;
+ char *s;
+
+ if (!(page = (char *)__get_free_page(GFP_KERNEL)))
+  return -ENOMEM;
+
+ s = page;
+
+ switch (type) {
+ case FILE_MEMLIST:
+  s += memset_sprintf_memlist(s, ms);
+  break;
+ case FILE_MEM_EXCLUSIVE:
+  *s++ = is_mem_exclusive(ms) ? '1' : '0';
+  break;
+ case FILE_MEMORY_MIGRATE:
+  *s++ = is_memory_migrate(ms) ? '1' : '0';
+  break;
+ case FILE_MEMORY_PRESSURE_ENABLED:
+  *s++ = memset_memory_pressure_enabled ? '1' : '0';
+  break;
+ case FILE_MEMORY_PRESSURE:
+  s += sprintf(s, "%d", fmeter_getrate(&ms->fmeter));
+  break;
+ case FILE_SPREAD_PAGE:
+  *s++ = is_spread_page(ms) ? '1' : '0';
+  break;
+ case FILE_SPREAD_SLAB:
+  *s++ = is_spread_slab(ms) ? '1' : '0';
+  break;
+ default:
+  retval = -EINVAL;
+  goto out;
+ }
+ *s++ = '\n';
+
+ retval = simple_read_from_buffer(buf, nbytes, ppos, page, s - page);
+out:
+ free_page((unsigned long)page);
+ return retval;
+}
+
+
+/*
```

```
+ * for the common functions, 'private' gives the type of file
+ */
+
+static struct cftype cft_mems = {
+ .name = "mems",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEMLIST,
+};
+
+static struct cftype cft_mem_exclusive = {
+ .name = "mem_exclusive",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEM_EXCLUSIVE,
+};
+
+static struct cftype cft_memory_migrate = {
+ .name = "memory_migrate",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEMORY_MIGRATE,
+};
+
+static struct cftype cft_memory_pressure_enabled = {
+ .name = "memory_pressure_enabled",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEMORY_PRESSURE_ENABLED,
+};
+
+static struct cftype cft_memory_pressure = {
+ .name = "memory_pressure",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_MEMORY_PRESSURE,
+};
+
+static struct cftype cft_spread_page = {
+ .name = "memory_spread_page",
+ .read = memset_common_file_read,
+ .write = memset_common_file_write,
+ .private = FILE_SPREAD_PAGE,
+};
+
+static struct cftype cft_spread_slab = {
+ .name = "memory_spread_slab",
+ .read = memset_common_file_read,
```

```
+ .write = memset_common_file_write,
+ .private = FILE_SPREAD_SLAB,
+};
+
+int memset_populate(struct container_subsys *ss, struct container *cont)
+{
+ int err;
+
+ if ((err = container_add_file(cont, &cft_mems)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_mem_exclusive)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_memory_migrate)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_memory_pressure)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_spread_page)) < 0)
+  return err;
+ if ((err = container_add_file(cont, &cft_spread_slab)) < 0)
+  return err;
+ /* memory_pressure_enabled is in root memset only */
+ if (err == 0 && !cont->parent)
+  err = container_add_file(cont, &cft_memory_pressure_enabled);
+ return 0;
+}
+
+/*
+ * memset_create - create a memset
+ * parent: memset that will be parent of the new memset.
+ * name:  name of the new memset. Will be strcpy'ed.
+ * mode:  mode to set on new inode
+ *
+ * Must be called with the mutex on the parent inode held
+ */
+
+int memset_create(struct container_subsys *ss, struct container *cont)
+{
+ struct memset *ms;
+ struct memset *parent;
+
+ if (!cont->parent) {
+  /* This is early initialization for the top container */
+  set_container_ms(cont, &top_memset);
+  top_memset.css.container = cont;
+  top_memset.mems_generation = memset_mems_generation++;
+  return 0;
+ }
+ parent = container_ms(cont->parent);
```

```
+ ms = kmalloc(sizeof(*ms), GFP_KERNEL);
+ if (!ms)
+  return -ENOMEM;
+
+ memset_update_task_memory_state();
+ ms->flags = 0;
+ if (is_spread_page(parent))
+  set_bit(MS_SPREAD_PAGE, &ms->flags);
+ if (is_spread_slab(parent))
+  set_bit(MS_SPREAD_SLAB, &ms->flags);
+ ms->mems_allowed = NODE_MASK_NONE;
+ ms->mems_generation = memset_mems_generation++;
+ fmeter_init(&ms->fmeter);
+
+ ms->parent = parent;
+ set_container_ms(cont, ms);
+ ms->css.container = cont;
+ number_of_memsets++;
+ return 0;
+}
+
+void memset_reparent(struct container_subsys *ss, struct container *cont,
+       void *state)
+{
+ struct memset *ms = state;
+ ms->css.container = cont;
+}
+
+void memset_destroy(struct container_subsys *ss, struct container *cont)
+{
+ struct memset *ms = container_ms(cont);
+ memset_update_task_memory_state();
+ number_of_memsets--;
+ kfree(ms);
+}
+
+static struct container_subsys memset_subsys = {
+ .name = "memset",
+ .create = memset_create,
+ .destroy  = memset_destroy,
+ .can_attach = memset_can_attach,
+ .post_attach = memset_post_attach,
+ .populate = memset_populate,
+ .subsys_id = -1,
+};
+
+/*
+ * memset_init_early - just enough so that the calls to
```

```
+ * memset_update_task_memory_state() in early init code
+ * are harmless.
+ */
+
+int __init memset_init_early(void)
+{
+ if (container_register_subsys(&memset_subsys) < 0)
+  panic("Couldn't register memset subsystem");
+ top_memset.mems_generation = memset_mems_generation++;
+ return 0;
+}
+
+
+/**
+ * memset_init - initialize memsets at system boot
+ *
+ * Description: Initialize top_memset and the memset internal file system,
+ **/
+
+int __init memset_init(void)
+{
+ top_memset.mems_allowed = NODE_MASK_ALL;
+
+ fmeter_init(&top_memset.fmeter);
+ top_memset.mems_generation = memset_mems_generation++;
+
+ number_of_memsets = 1;
+ return 0;
+}
+
+#if defined(CONFIG_MEMORY_HOTPLUG)
+/*
+ * If memset_track_online_nodes(), below, unplugs any memory nodes, we
+ * need to walk over the memset hierarchy, removing that node from all
+ * memsets.  If this removes the last node from a memset, then
+ * guarantee_online_mems() will use that emptied memsets parent online
+ * nodes.  Memsets that were already empty of nodes are left empty.
+ *
+ * This routine will check all memsets in a subtree even if the top
+ * memset of the subtree has no offline nodes.
+ *
+ * Call with both manage_mutex and callback_mutex held.
+ *
+ * Recursive, on depth of memset subtree.
+ */
+
+static void guarantee_online_mems_in_subtree(const struct memset *cur)
+{
```

```
+ struct container *cont;
+ struct memset *c;
+
+ /* Each of our child memsets mems must be online */
+ list_for_each_entry(cont, &cur->css.container->children, sibling) {
+  c = container_ms(cont);
+  guarantee_online_mems_in_subtree(c);
+  if (!nodes_empty(c->mems_allowed))
+   guarantee_online_mems(c, &c->mems_allowed);
+ }
+}
+
+/*
+ * Keep top_memset.mems_allowed tracking node_online_map.
+ * Call this routine anytime after you change node_online_map.
+ */
+
+void memset_track_online_nodes(void)
+{
+ container_manage_lock();
+ container_lock();
+
+ guarantee_online_mems_in_subtree(&top_memset);
+ top_memset.mems_allowed = node_online_map;
+
+ container_unlock();
+ container_manage_unlock();
+}
+#endif
+
+/**
+ * memset_init_smp - initialize mems_allowed
+ *
+ * Description: Finish top memset after cpu, node maps are initialized
+ **/
+
+void __init memset_init_smp(void)
+{
+ top_memset.mems_allowed = node_online_map;
+}
+
+void memset_init_current_mems_allowed(void)
+{
+ current->mems_allowed = NODE_MASK_ALL;
+}
+
+/**
+ * memset_mems_allowed - return mems_allowed mask from a tasks memset.
```

```
+ * @tsk: pointer to task_struct from which to obtain memset->mems_allowed.
+ *
+ * Description: Returns the nodemask_t mems_allowed of the memset
+ * attached to the specified @tsk.  Guaranteed to return some non-empty
+ * subset of node_online_map, even if this means going outside the
+ * tasks memset.
+ **/
+
+nodemask_t memset_mems_allowed(struct task_struct *tsk)
+{
+ nodemask_t mask;
+
+ container_lock();
+ task_lock(tsk);
+ guarantee_online_mems(task_ms(tsk), &mask);
+ task_unlock(tsk);
+ container_unlock();
+
+ return mask;
+}
+
+/**
+ * memset_zonelist_valid_mems_allowed - check zonelist vs. curremt mems_allowed
+ * @zl: the zonelist to be checked
+ *
+ * Are any of the nodes on zonelist zl allowed in current->mems_allowed?
+ */
+int memset_zonelist_valid_mems_allowed(struct zonelist *zl)
+{
+ int i;
+
+ for (i = 0; zl->zones[i]; i++) {
+  int nid = zone_to_nid(zl->zones[i]);
+
+  if (node_isset(nid, current->mems_allowed))
+   return 1;
+ }
+ return 0;
+}
+
+/*
+ * nearest_exclusive_ancestor() - Returns the nearest mem_exclusive
+ * ancestor to the specified memset.  Call holding callback_mutex.
+ * If no ancestor is mem_exclusive (an unusual configuration), then
+ * returns the root memset.
+ */
+static const struct memset *nearest_exclusive_ancestor(const struct memset *ms)
+{
```

```
+ while (!is_mem_exclusive(ms) && ms->parent)
+  ms = ms->parent;
+ return ms;
+}
+
+/**
+ * memset_zone_allowed - Can we allocate memory on zone z's memory node?
+ * @z: is this zone on an allowed node?
+ * @gfp_mask: memory allocation flags (we use __GFP_HARDWALL)
+ *
+ * If we're in interrupt, yes, we can always allocate.  If zone
+ * z's node is in our tasks mems_allowed, yes.  If it's not a
+ * __GFP_HARDWALL request and this zone's nodes is in the nearest
+ * mem_exclusive memset ancestor to this tasks memset, yes.
+ * Otherwise, no.
+ *
+ * GFP_USER allocations are marked with the __GFP_HARDWALL bit,
+ * and do not allow allocations outside the current tasks memset.
+ * GFP_KERNEL allocations are not so marked, so can escape to the
+ * nearest mem_exclusive ancestor memset.
+ *
+ * Scanning up parent memsets requires callback_mutex.  The __alloc_pages()
+ * routine only calls here with __GFP_HARDWALL bit _not_ set if
+ * it's a GFP_KERNEL allocation, and all nodes in the current tasks
+ * mems_allowed came up empty on the first pass over the zonelist.
+ * So only GFP_KERNEL allocations, if all nodes in the memset are
+ * short of memory, might require taking the callback_mutex mutex.
+ *
+ * The first call here from mm/page_alloc:get_page_from_freelist()
+ * has __GFP_HARDWALL set in gfp_mask, enforcing hardwall memsets, so
+ * no allocation on a node outside the memset is allowed (unless in
+ * interrupt, of course).
+ *
+ * The second pass through get_page_from_freelist() doesn't even call
+ * here for GFP_ATOMIC calls.  For those calls, the __alloc_pages()
+ * variable 'wait' is not set, and the bit ALLOC_MEMSET is not set
+ * in alloc_flags.  That logic and the checks below have the combined
+ * affect that:
+ * in_interrupt - any node ok (current task context irrelevant)
+ * GFP_ATOMIC   - any node ok
+ * GFP_KERNEL   - any node in enclosing mem_exclusive memset ok
+ * GFP_USER     - only nodes in current tasks mems allowed ok.
+ *
+ * Rule:
+ *    Don't call memset_zone_allowed() if you can't sleep, unless you
+ *    pass in the __GFP_HARDWALL flag set in gfp_flag, which disables
+ *    the code that might scan up ancestor memsets and sleep.
+ **/
```

```
+
+int __memset_zone_allowed(struct zone *z, gfp_t gfp_mask)
+{
+ int node;   /* node that zone z is on */
+ const struct memset *ms; /* current memset ancestors */
+ int allowed;   /* is allocation in zone z allowed? */
+
+ if (in_interrupt() || (gfp_mask & __GFP_THISNODE))
+  return 1;
+ node = zone_to_nid(z);
+ might_sleep_if(!(gfp_mask & __GFP_HARDWALL));
+ if (node_isset(node, current->mems_allowed))
+  return 1;
+ if (gfp_mask & __GFP_HARDWALL) /* If hardwall request, stop here */
+  return 0;
+
+ if (current->flags & PF_EXITING) /* Let dying task have memory */
+  return 1;
+
+ /* Not hardwall and node outside mems_allowed: scan up memsets */
+ container_lock();
+
+ task_lock(current);
+ ms = nearest_exclusive_ancestor(task_ms(current));
+ task_unlock(current);
+
+ allowed = node_isset(node, ms->mems_allowed);
+ container_unlock();
+ return allowed;
+}
+
+/**
+ * memset_mem_spread_node() - On which node to begin search for a page
+ *
+ * If a task is marked PF_SPREAD_PAGE or PF_SPREAD_SLAB (as for
+ * tasks in a memset with is_spread_page or is_spread_slab set),
+ * and if the memory allocation used memset_mem_spread_node()
+ * to determine on which node to start looking, as it will for
+ * certain page cache or slab cache pages such as used for file
+ * system buffers and inode caches, then instead of starting on the
+ * local node to look for a free page, rather spread the starting
+ * node around the tasks mems_allowed nodes.
+ *
+ * We don't have to worry about the returned node being offline
+ * because "it can't happen", and even if it did, it would be ok.
+ *
+ * The routines calling guarantee_online_mems() are careful to
+ * only set nodes in task->mems_allowed that are online.  So it
```

```
+ * should not be possible for the following code to return an
+ * offline node.  But if it did, that would be ok, as this routine
+ * is not returning the node where the allocation must be, only
+ * the node where the search should start.  The zonelist passed to
+ * __alloc_pages() will include all nodes.  If the slab allocator
+ * is passed an offline node, it will fall back to the local node.
+ * See kmem_cache_alloc_node().
+ */
+
+int memset_mem_spread_node(void)
+{
+ int node;
+
+ node = next_node(current->memset_mem_spread_rotor, current->mems_allowed);
+ if (node == MAX_NUMNODES)
+  node = first_node(current->mems_allowed);
+ current->memset_mem_spread_rotor = node;
+ return node;
+}
+EXPORT_SYMBOL_GPL(memset_mem_spread_node);
+
+/**
+ * memset_excl_nodes_overlap - Do we overlap @p's mem_exclusive ancestors?
+ * @p: pointer to task_struct of some other task.
+ *
+ * Description: Return true if the nearest mem_exclusive ancestor
+ * memsets of tasks @p and current overlap.  Used by oom killer to
+ * determine if task @p's memory usage might impact the memory
+ * available to the current task.
+ *
+ * Call while holding callback_mutex.
+ **/
+
+int memset_excl_nodes_overlap(const struct task_struct *p)
+{
+ const struct memset *ms1, *ms2; /* my and p's memset ancestors */
+ int overlap = 1;  /* do memsets overlap? */
+
+ task_lock(current);
+ if (current->flags & PF_EXITING) {
+  task_unlock(current);
+  goto done;
+ }
+ ms1 = nearest_exclusive_ancestor(task_ms(current));
+ task_unlock(current);
+
+ task_lock((struct task_struct *)p);
+ if (p->flags & PF_EXITING) {
```

```
+  task_unlock((struct task_struct *)p);
+  goto done;
+ }
+ ms2 = nearest_exclusive_ancestor(task_ms((struct task_struct *)p));
+ task_unlock((struct task_struct *)p);
+
+ overlap = nodes_intersects(ms1->mems_allowed, ms2->mems_allowed);
+done:
+ return overlap;
+}
+
+/*
+ * Collection of memory_pressure is suppressed unless
+ * this flag is enabled by writing "1" to the special
+ * memset file 'memory_pressure_enabled' in the root memset.
+ */
+
+int memset_memory_pressure_enabled __read_mostly;
+
+/**
+ * memset_memory_pressure_bump - keep stats of per-memset reclaims.
+ *
+ * Keep a running average of the rate of synchronous (direct)
+ * page reclaim efforts initiated by tasks in each memset.
+ *
+ * This represents the rate at which some task in the memset
+ * ran low on memory on all nodes it was allowed to use, and
+ * had to enter the kernels page reclaim code in an effort to
+ * create more free memory by tossing clean pages or swapping
+ * or writing dirty pages.
+ *
+ * Display to user space in the per-memset read-only file
+ * "memory_pressure".  Value displayed is an integer
+ * representing the recent rate of entry into the synchronous
+ * (direct) page reclaim by any task attached to the memset.
+ **/
+
+void __memset_memory_pressure_bump(void)
+{
+ task_lock(current);
+ fmeter_markevent(&task_ms(current)->fmeter);
+ task_unlock(current);
+}
+
+/* Display task mems_allowed in /proc/<pid>/status file. */
+char *memset_task_status_allowed(struct task_struct *task, char *buffer)
+{
+ buffer += sprintf(buffer, "Mems_allowed:\t");
```

```
+ buffer += nodemask_scnprintf(buffer, PAGE_SIZE, task->mems_allowed);
+ buffer += sprintf(buffer, "\n");
+ return buffer;
+}
Index: container-2.6.19-rc6/include/linux/mempolicy.h
===================================================================
--- container-2.6.19-rc6.orig/include/linux/mempolicy.h
+++ container-2.6.19-rc6/include/linux/mempolicy.h
@@ -68,7 +68,7 @@ struct mempolicy {
   nodemask_t  nodes;  /* interleave */
   /* undefined for default */
  } v;
- nodemask_t cpuset_mems_allowed; /* mempolicy relative to these nodes */
+ nodemask_t memset_mems_allowed; /* mempolicy relative to these nodes */
 };

 /*
Index: container-2.6.19-rc6/include/linux/sched.h
===================================================================
--- container-2.6.19-rc6.orig/include/linux/sched.h
+++ container-2.6.19-rc6/include/linux/sched.h
@@ -999,10 +999,10 @@ struct task_struct {
    struct mempolicy *mempolicy;
  short il_next;
 #endif
-#ifdef CONFIG_CPUSETS
+#ifdef CONFIG_MEMSETS
  nodemask_t mems_allowed;
- int cpuset_mems_generation;
- int cpuset_mem_spread_rotor;
+ int memset_mems_generation;
+ int memset_mem_spread_rotor;
 #endif
 #ifdef CONFIG_CONTAINERS
  struct container *container[CONFIG_MAX_CONTAINER_HIERARCHIES];
@@ -1112,8 +1112,8 @@ static inline void put_task_struct(struc
 #define PF_BORROWED_MM 0x00200000 /* I am a kthread doing use_mm */
 #define PF_RANDOMIZE 0x00400000 /* randomize virtual address space */
 #define PF_SWAPWRITE 0x00800000 /* Allowed to write to swap */
-#define PF_SPREAD_PAGE 0x01000000 /* Spread page cache over cpuset */
-#define PF_SPREAD_SLAB 0x02000000 /* Spread some slab caches over cpuset */
+#define PF_SPREAD_PAGE 0x01000000 /* Spread page cache over memset */
+#define PF_SPREAD_SLAB 0x02000000 /* Spread some slab caches over memset */
 #define PF_MEMPOLICY 0x10000000 /* Non-default NUMA mempolicy */
 #define PF_MUTEX_TESTER 0x20000000 /* Thread belongs to the rt mutex tester */

Index: container-2.6.19-rc6/kernel/Makefile
===================================================================
```

```
--- container-2.6.19-rc6.orig/kernel/Makefile
+++ container-2.6.19-rc6/kernel/Makefile
@@ -38,6 +38,7 @@ obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CONTAINERS) += container.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
+obj-$(CONFIG_MEMSETS) += memset.o
 obj-$(CONFIG_CONTAINER_CPUACCT) += cpu_acct.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
Index: container-2.6.19-rc6/mm/mempolicy.c
===================================================================
--- container-2.6.19-rc6.orig/mm/mempolicy.c
+++ container-2.6.19-rc6/mm/mempolicy.c
@@ -74,7 +74,7 @@
 #include <linux/sched.h>
 #include <linux/mm.h>
 #include <linux/nodemask.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/gfp.h>
 #include <linux/slab.h>
 #include <linux/string.h>
@@ -198,7 +198,7 @@ static struct mempolicy *mpol_new(int mo
   break;
  }
  policy->policy = mode;
- policy->cpuset_mems_allowed = cpuset_mems_allowed(current);
+ policy->memset_mems_allowed = memset_mems_allowed(current);
  return policy;
 }

@@ -423,8 +423,8 @@ static int contextualize_policy(int mode
  if (!nodes)
   return 0;

- cpuset_update_task_memory_state();
- if (!cpuset_nodes_subset_current_mems_allowed(*nodes))
+ memset_update_task_memory_state();
+ if (!memset_nodes_subset_current_mems_allowed(*nodes))
   return -EINVAL;
  return mpol_check_policy(mode, nodes);
 }
@@ -529,7 +529,7 @@ long do_get_mempolicy(int *policy, nodem
  struct vm_area_struct *vma = NULL;
  struct mempolicy *pol = current->mempolicy;

- cpuset_update_task_memory_state();
```

```
+ memset_update_task_memory_state();
  if (flags & ~(unsigned long)(MPOL_F_NODE|MPOL_F_ADDR))
   return -EINVAL;
  if (flags & MPOL_F_ADDR) {
@@ -945,7 +945,7 @@ asmlinkage long sys_migrate_pages(pid_t
   goto out;
  }

- task_nodes = cpuset_mems_allowed(task);
+ task_nodes = memset_mems_allowed(task);
  /* Is the user allowed to access the target nodes? */
  if (!nodes_subset(new, task_nodes) && !capable(CAP_SYS_NICE)) {
   err = -EPERM;
@@ -1102,7 +1102,7 @@ static struct zonelist *zonelist_policy(
   /* Lower zones don't get a policy applied */
   /* Careful: current->mems_allowed might have moved */
   if (gfp_zone(gfp) >= policy_zone)
-   if (cpuset_zonelist_valid_mems_allowed(policy->v.zonelist))
+   if (memset_zonelist_valid_mems_allowed(policy->v.zonelist))
     return policy->v.zonelist;
   /*FALL THROUGH*/
  case MPOL_INTERLEAVE: /* should not happen */
@@ -1256,7 +1256,7 @@ alloc_page_vma(gfp_t gfp, struct vm_area
 {
  struct mempolicy *pol = get_vma_policy(current, vma, addr);

- cpuset_update_task_memory_state();
+ memset_update_task_memory_state();

  if (unlikely(pol->policy == MPOL_INTERLEAVE)) {
   unsigned nid;
@@ -1282,8 +1282,8 @@ alloc_page_vma(gfp_t gfp, struct vm_area
  * interrupt context and apply the current process NUMA policy.
  * Returns NULL when no page can be allocated.
  *
- * Don't call cpuset_update_task_memory_state() unless
- * 1) it's ok to take cpuset_sem (can WAIT), and
+ * Don't call memset_update_task_memory_state() unless
+ * 1) it's ok to take memset_sem (can WAIT), and
  * 2) allocating for current task (not interrupt).
  */
 struct page *alloc_pages_current(gfp_t gfp, unsigned order)
@@ -1291,7 +1291,7 @@ struct page *alloc_pages_current(gfp_t g
  struct mempolicy *pol = current->mempolicy;

  if ((gfp & __GFP_WAIT) && !in_interrupt())
-  cpuset_update_task_memory_state();
+  memset_update_task_memory_state();
```

```
  if (!pol || in_interrupt() || (gfp & __GFP_THISNODE))
   pol = &default_policy;
  if (pol->policy == MPOL_INTERLEAVE)
@@ -1301,11 +1301,11 @@ struct page *alloc_pages_current(gfp_t g
 EXPORT_SYMBOL(alloc_pages_current);

 /*
- * If mpol_copy() sees current->cpuset == cpuset_being_rebound, then it
+ * If mpol_copy() sees current->memset == memset_being_rebound, then it
   * rebinds the mempolicy its copying by calling mpol_rebind_policy()
- * with the mems_allowed returned by cpuset_mems_allowed().  This
- * keeps mempolicies cpuset relative after its cpuset moves.  See
- * further kernel/cpuset.c update_nodemask().
+ * with the mems_allowed returned by memset_mems_allowed().  This
+ * keeps mempolicies memset relative after its memset moves.  See
+ * further kernel/memset.c update_nodemask().
   */

 /* Slow path of a mempolicy copy */
@@ -1315,8 +1315,8 @@ struct mempolicy *__mpol_copy(struct mem

  if (!new)
   return ERR_PTR(-ENOMEM);
- if (current_cpuset_is_being_rebound()) {
-  nodemask_t mems = cpuset_mems_allowed(current);
+ if (current_memset_is_being_rebound()) {
+  nodemask_t mems = memset_mems_allowed(current);
   mpol_rebind_policy(old, &mems);
  }
  *new = *old;
@@ -1623,7 +1623,7 @@ void mpol_rebind_policy(struct mempolicy

  if (!pol)
   return;
- mpolmask = &pol->cpuset_mems_allowed;
+ mpolmask = &pol->memset_mems_allowed;
  if (nodes_equal(*mpolmask, *newmask))
   return;

Index: container-2.6.19-rc6/mm/migrate.c
===================================================================
--- container-2.6.19-rc6.orig/mm/migrate.c
+++ container-2.6.19-rc6/mm/migrate.c
@@ -23,7 +23,7 @@
 #include <linux/rmap.h>
 #include <linux/topology.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
```

```
+#include <linux/memset.h>
 #include <linux/writeback.h>
 #include <linux/mempolicy.h>
 #include <linux/vmalloc.h>
@@ -911,7 +911,7 @@ asmlinkage long sys_move_pages(pid_t pid
    goto out2;


- task_nodes = cpuset_mems_allowed(task);
+ task_nodes = memset_mems_allowed(task);

  /* Limit nr_pages so that the multiplication may not overflow */
  if (nr_pages >= ULONG_MAX / sizeof(struct page_to_node) - 1) {
Index: container-2.6.19-rc6/mm/page_alloc.c
===================================================================
--- container-2.6.19-rc6.orig/mm/page_alloc.c
+++ container-2.6.19-rc6/mm/page_alloc.c
@@ -31,7 +31,7 @@
 #include <linux/topology.h>
 #include <linux/sysctl.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/memory_hotplug.h>
 #include <linux/nodemask.h>
 #include <linux/vmalloc.h>
@@ -891,7 +891,7 @@ failed:
 #define ALLOC_WMARK_HIGH 0x08 /* use pages_high watermark */
 #define ALLOC_HARDER  0x10 /* try to alloc harder */
 #define ALLOC_HIGH  0x20 /* __GFP_HIGH set */
-#define ALLOC_CPUSET  0x40 /* check for correct cpuset */
+#define ALLOC_MEMSET  0x40 /* check for correct memset */

 /*
  * Return 1 if free pages are above 'mark'. This takes into account the order
@@ -940,15 +940,15 @@ get_page_from_freelist(gfp_t gfp_mask, u

  /*
   * Go through the zonelist once, looking for a zone with enough free.
-  * See also cpuset_zone_allowed() comment in kernel/cpuset.c.
+  * See also memset_zone_allowed() comment in kernel/memset.c.
   */
  do {
   zone = *z;
   if (unlikely(NUMA_BUILD && (gfp_mask & __GFP_THISNODE) &&
    zone->zone_pgdat != zonelist->zones[0]->zone_pgdat))
     break;
-  if ((alloc_flags & ALLOC_CPUSET) &&
```

```
-    !cpuset_zone_allowed(zone, gfp_mask))
+  if ((alloc_flags & ALLOC_MEMSET) &&
+    !memset_zone_allowed(zone, gfp_mask))
     continue;

   if (!(alloc_flags & ALLOC_NO_WATERMARKS)) {
@@ -1001,7 +1001,7 @@ restart:
  }

  page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, order,
-    zonelist, ALLOC_WMARK_LOW|ALLOC_CPUSET);
+    zonelist, ALLOC_WMARK_LOW|ALLOC_MEMSET);
  if (page)
   goto got_pg;

@@ -1025,15 +1025,15 @@ restart:
  if (gfp_mask & __GFP_HIGH)
   alloc_flags |= ALLOC_HIGH;
  if (wait)
-  alloc_flags |= ALLOC_CPUSET;
+  alloc_flags |= ALLOC_MEMSET;

  /*
   * Go through the zonelist again. Let __GFP_HIGH and allocations
   * coming from realtime tasks go deeper into reserves.
   *
   * This is the last chance, in general, before the goto nopage.
-  * Ignore cpuset if GFP_ATOMIC (!wait) rather than fail alloc.
-  * See also cpuset_zone_allowed() comment in kernel/cpuset.c.
+  * Ignore memset if GFP_ATOMIC (!wait) rather than fail alloc.
+  * See also memset_zone_allowed() comment in kernel/memset.c.
   */
  page = get_page_from_freelist(gfp_mask, order, zonelist, alloc_flags);
  if (page)
@@ -1066,7 +1066,7 @@ rebalance:
  cond_resched();

  /* We now go into synchronous reclaim */
- cpuset_memory_pressure_bump();
+ memset_memory_pressure_bump();
  p->flags |= PF_MEMALLOC;
  reclaim_state.reclaimed_slab = 0;
  p->reclaim_state = &reclaim_state;
@@ -1091,7 +1091,7 @@ rebalance:
    * under heavy pressure.
    */
   page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, order,
-    zonelist, ALLOC_WMARK_HIGH|ALLOC_CPUSET);
```

```
+    zonelist, ALLOC_WMARK_HIGH|ALLOC_MEMSET);
   if (page)
    goto got_pg;

@@ -1594,12 +1594,12 @@ void __meminit build_all_zonelists(void)
 {
  if (system_state == SYSTEM_BOOTING) {
   __build_all_zonelists(NULL);
-  cpuset_init_current_mems_allowed();
+  memset_init_current_mems_allowed();
  } else {
   /* we have to stop all cpus to guaranntee there is no user
      of zonelist */
   stop_machine_run(__build_all_zonelists, NULL, NR_CPUS);
-  /* cpuset refresh routine should be here */
+  /* memset refresh routine should be here */
  }
  vm_total_pages = nr_free_pagecache_pages();
  printk("Built %i zonelists.  Total pages: %ld\n",
Index: container-2.6.19-rc6/mm/filemap.c
===================================================================
--- container-2.6.19-rc6.orig/mm/filemap.c
+++ container-2.6.19-rc6/mm/filemap.c
@@ -29,7 +29,7 @@
 #include <linux/blkdev.h>
 #include <linux/security.h>
 #include <linux/syscalls.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include "filemap.h"
 #include "internal.h"

@@ -469,8 +469,8 @@ int add_to_page_cache_lru(struct page *p
 #ifdef CONFIG_NUMA
 struct page *__page_cache_alloc(gfp_t gfp)
 {
- if (cpuset_do_page_mem_spread()) {
-  int n = cpuset_mem_spread_node();
+ if (memset_do_page_mem_spread()) {
+  int n = memset_mem_spread_node();
   return alloc_pages_node(n, gfp, 0);
  }
  return alloc_pages(gfp, 0);
Index: container-2.6.19-rc6/mm/hugetlb.c
===================================================================
--- container-2.6.19-rc6.orig/mm/hugetlb.c
+++ container-2.6.19-rc6/mm/hugetlb.c
@@ -12,7 +12,7 @@
```

```
 #include <linux/nodemask.h>
 #include <linux/pagemap.h>
 #include <linux/mempolicy.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/mutex.h>

 #include <asm/page.h>
@@ -73,7 +73,7 @@ static struct page *dequeue_huge_page(st

  for (z = zonelist->zones; *z; z++) {
   nid = zone_to_nid(*z);
-  if (cpuset_zone_allowed(*z, GFP_HIGHUSER) &&
+  if (memset_zone_allowed(*z, GFP_HIGHUSER) &&
       !list_empty(&hugepage_freelists[nid]))
    break;
  }
Index: container-2.6.19-rc6/mm/memory_hotplug.c
===================================================================
--- container-2.6.19-rc6.orig/mm/memory_hotplug.c
+++ container-2.6.19-rc6/mm/memory_hotplug.c
@@ -22,7 +22,7 @@
 #include <linux/highmem.h>
 #include <linux/vmalloc.h>
 #include <linux/ioport.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>

 #include <asm/tlbflush.h>

@@ -284,7 +284,7 @@ int add_memory(int nid, u64 start, u64 s
  /* we online node here. we can't roll back from here. */
  node_set_online(nid);

- cpuset_track_online_nodes();
+ memset_track_online_nodes();

  if (new_pgdat) {
   ret = register_one_node(nid);
Index: container-2.6.19-rc6/mm/oom_kill.c
===================================================================
--- container-2.6.19-rc6.orig/mm/oom_kill.c
+++ container-2.6.19-rc6/mm/oom_kill.c
@@ -21,7 +21,7 @@
 #include <linux/swap.h>
 #include <linux/timex.h>
 #include <linux/jiffies.h>
-#include <linux/cpuset.h>
```

```
+#include <linux/memset.h>
 #include <linux/module.h>
 #include <linux/notifier.h>

@@ -140,7 +140,7 @@ unsigned long badness(struct task_struct
   * because p may have allocated or otherwise mapped memory on
   * this node before. However it will be less likely.
   */
- if (!cpuset_excl_nodes_overlap(p))
+ if (!memset_excl_nodes_overlap(p))
   points /= 8;

  /*
@@ -165,7 +165,7 @@ unsigned long badness(struct task_struct
  */
 #define CONSTRAINT_NONE 1
 #define CONSTRAINT_MEMORY_POLICY 2
-#define CONSTRAINT_CPUSET 3
+#define CONSTRAINT_MEMSET 3

 /*
  * Determine the type of allocation constraint.
@@ -177,10 +177,10 @@ static inline int constrained_alloc(stru
  nodemask_t nodes = node_online_map;

  for (z = zonelist->zones; *z; z++)
-  if (cpuset_zone_allowed(*z, gfp_mask))
+  if (memset_zone_allowed(*z, gfp_mask))
    node_clear(zone_to_nid(*z), nodes);
   else
-   return CONSTRAINT_CPUSET;
+   return CONSTRAINT_MEMSET;

  if (!nodes_empty(nodes))
   return CONSTRAINT_MEMORY_POLICY;
@@ -408,9 +408,9 @@ void out_of_memory(struct zonelist *zone
     "No available memory (MPOL_BIND)");
   break;

- case CONSTRAINT_CPUSET:
+ case CONSTRAINT_MEMSET:
   oom_kill_process(current, points,
-    "No available memory in cpuset");
+    "No available memory in memset");
   break;

  case CONSTRAINT_NONE:
Index: container-2.6.19-rc6/mm/slab.c
```

```
===================================================================
--- container-2.6.19-rc6.orig/mm/slab.c
+++ container-2.6.19-rc6/mm/slab.c
@@ -94,7 +94,7 @@
 #include <linux/interrupt.h>
 #include <linux/init.h>
 #include <linux/compiler.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/seq_file.h>
 #include <linux/notifier.h>
 #include <linux/kallsyms.h>
@@ -3120,7 +3120,7 @@ static __always_inline void *__cache_all
 /*
  * Try allocating on another node if PF_SPREAD_SLAB|PF_MEMPOLICY.
  *
- * If we are in_interrupt, then process context, including cpusets and
+ * If we are in_interrupt, then process context, including memsets and
  * mempolicy, may not apply and should not be used for allocation policy.
  */
 static void *alternate_node_alloc(struct kmem_cache *cachep, gfp_t flags)
@@ -3130,8 +3130,8 @@ static void *alternate_node_alloc(struct
  if (in_interrupt() || (flags & __GFP_THISNODE))
   return NULL;
  nid_alloc = nid_here = numa_node_id();
- if (cpuset_do_slab_mem_spread() && (cachep->flags & SLAB_MEM_SPREAD))
-  nid_alloc = cpuset_mem_spread_node();
+ if (memset_do_slab_mem_spread() && (cachep->flags & SLAB_MEM_SPREAD))
+  nid_alloc = memset_mem_spread_node();
  else if (current->mempolicy)
   nid_alloc = slab_node(current->mempolicy);
  if (nid_alloc != nid_here)
@@ -3143,7 +3143,7 @@ static void *alternate_node_alloc(struct
  * Fallback function if there was no memory available and no objects on a
  * certain node and we are allowed to fall back. We mimick the behavior of
  * the page allocator. We fall back according to a zonelist determined by
- * the policy layer while obeying cpuset constraints.
+ * the policy layer while obeying memset constraints.
  */
 void *fallback_alloc(struct kmem_cache *cache, gfp_t flags)
 {
@@ -3156,7 +3156,7 @@ void *fallback_alloc(struct kmem_cache *
   int nid = zone_to_nid(*z);

   if (zone_idx(*z) <= ZONE_NORMAL &&
-      cpuset_zone_allowed(*z, flags) &&
+      memset_zone_allowed(*z, flags) &&
       cache->nodelists[nid])
```

```
    obj = __cache_alloc_node(cache,
      flags | __GFP_THISNODE, nid);
Index: container-2.6.19-rc6/mm/vmscan.c
===================================================================
--- container-2.6.19-rc6.orig/mm/vmscan.c
+++ container-2.6.19-rc6/mm/vmscan.c
@@ -31,7 +31,7 @@
 #include <linux/rmap.h>
 #include <linux/topology.h>
 #include <linux/cpu.h>
-#include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/notifier.h>
 #include <linux/rwsem.h>
 #include <linux/delay.h>
@@ -983,7 +983,7 @@ static unsigned long shrink_zones(int pr
   if (!populated_zone(zone))
    continue;

-  if (!cpuset_zone_allowed(zone, __GFP_HARDWALL))
+  if (!memset_zone_allowed(zone, __GFP_HARDWALL))
    continue;

   note_zone_scanning_priority(zone, priority);
@@ -1033,7 +1033,7 @@ unsigned long try_to_free_pages(struct z
  for (i = 0; zones[i] != NULL; i++) {
   struct zone *zone = zones[i];

-  if (!cpuset_zone_allowed(zone, __GFP_HARDWALL))
+  if (!memset_zone_allowed(zone, __GFP_HARDWALL))
    continue;

   lru_pages += zone->nr_active + zone->nr_inactive;
@@ -1088,7 +1088,7 @@ out:
  for (i = 0; zones[i] != 0; i++) {
   struct zone *zone = zones[i];

-  if (!cpuset_zone_allowed(zone, __GFP_HARDWALL))
+  if (!memset_zone_allowed(zone, __GFP_HARDWALL))
    continue;

   zone->prev_priority = priority;
@@ -1349,7 +1349,7 @@ void wakeup_kswapd(struct zone *zone, in
   return;
  if (pgdat->kswapd_max_order < order)
   pgdat->kswapd_max_order = order;
- if (!cpuset_zone_allowed(zone, __GFP_HARDWALL))
+ if (!memset_zone_allowed(zone, __GFP_HARDWALL))
```

```
   return;
  if (!waitqueue_active(&pgdat->kswapd_wait))
   return;
Index: container-2.6.19-rc6/init/main.c
===================================================================
--- container-2.6.19-rc6.orig/init/main.c
+++ container-2.6.19-rc6/init/main.c
@@ -38,6 +38,7 @@
 #include <linux/writeback.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/container.h>
 #include <linux/efi.h>
 #include <linux/taskstats_kern.h>
@@ -571,6 +572,7 @@ asmlinkage void __init start_kernel(void
  vfs_caches_init_early();
  container_init_early();
  cpuset_init_early();
+ memset_init_early();
  mem_init();
  kmem_cache_init();
  setup_per_cpu_pageset();
@@ -602,6 +604,7 @@ asmlinkage void __init start_kernel(void
 #endif
  container_init();
  cpuset_init();
+ memset_init();
  taskstats_init_early();
  delayacct_init();

Index: container-2.6.19-rc6/fs/proc/array.c
===================================================================
--- container-2.6.19-rc6.orig/fs/proc/array.c
+++ container-2.6.19-rc6/fs/proc/array.c
@@ -73,6 +73,7 @@
 #include <linux/file.h>
 #include <linux/times.h>
 #include <linux/cpuset.h>
+#include <linux/memset.h>
 #include <linux/rcupdate.h>
 #include <linux/delayacct.h>

@@ -306,6 +307,7 @@ int proc_pid_status(struct task_struct *
  buffer = task_sig(task, buffer);
  buffer = task_cap(task, buffer);
  buffer = cpuset_task_status_allowed(task, buffer);
+ buffer = memset_task_status_allowed(task, buffer);
```

```
 #if defined(CONFIG_S390)
  buffer = task_show_regs(task, buffer);
 #endif
Index: container-2.6.19-rc6/init/Kconfig
===================================================================
--- container-2.6.19-rc6.orig/init/Kconfig
+++ container-2.6.19-rc6/init/Kconfig
@@ -260,7 +260,7 @@ config CPUSETS
  help
    This option will let you create and manage CPUSETs which
    allow dynamically partitioning a system into sets of CPUs and
-   Memory Nodes and assigning tasks to run only within those sets.
+   assigning tasks to run only within those sets.
    This is primarily useful on large SMP or NUMA systems.

    Say N if unsure.
@@ -269,6 +269,18 @@ config PROC_PID_CPUSET
  bool "Include legacy /proc/<pid>/cpuset file"
  depends on CPUSETS

+config MEMSETS
+ bool "Memset support"
+ depends on SMP
+ select CONTAINERS
+ help
+   This option will let you create and manage Memsets which
+   allow dynamically partitioning a system into sets of
+   Memory Nodes and assigning tasks to run only within those sets.
+   This is primarily useful on large SMP or NUMA systems.
+
+   Say N if unsure.
+
 config CONTAINER_CPUACCT
  bool "Simple CPU accounting container subsystem"
  select CONTAINERS

--
```