Subject: [RFC][PATCH] Containers: Pagecache accounting and control subsystem (v1)
Posted by Vaidyanathan Srinivas on Tue, 20 Feb 2007 13:52:26 GMT
View Forum Message <> Reply to Message

This patch adds pagecache accounting and control on top of
Paul's container subsystem v7 posted at
http://lkml.org/lkml/2007/2/12/88

Comments, suggestions and criticisms are welcome.

Features:
---
* New subsystem called 'pagecache_acct' is registered with containers
* Container pointer is added to struct address_space to keep track of
  associated container
* In filemap.c and swap_state.c, the corresponding container's
  pagecache_acct subsystem is charged and uncharged whenever a new
  page is added or removed from pagecache
* The accounting number include pages in swap cache and filesystem
  buffer pages apart from pagecache, basically everything under
  NR_FILE_PAGES is counted as pagecache. However this excluded
  mapped and anonymous pages
* Limits on pagecache can be set by echo 100000 > pagecache_limit on
  the /container file system. The unit is in kilobytes
* If the pagecache utilisation limit is exceeded, pagecache reclaim
  code is invoked to recover dirty and clean pagecache pages only.

Advantages:
---
* Does not add container pointers in struct page

Limitations:
---
* Code is not safe for container deletion/task migration
* Pagecache page reclaim needs performance improvements
* Global LRU is churned in search of pagecache pages

Usage:
---

* Add patch on top of Paul container (v7) at kernel version 2.6.20
* Enable CONFIG_CONTAINER_PAGECACHE_ACCT in 'General Setup'
* Boot new kernel
* Mount container filesystem
 mount -t container /container
 cd /container
* Create new container

```
 mkdir mybox
 cd /container/mybox
* Add current shell to container
 echo $$ > tasks
* There are two files pagecache_usage and pagecache_limit
* In order to set limit, echo value in kilobytes to pagecache_limit
 echo 100000 > pagecache_limit
 #This would set 100MB limit on pagecache usage
* Trash the system from current shell using scp/cp/dd/tar etc
* Watch pagecache_usage and /proc/meminfo to verify behavior

* Only unmapped pagecache data will be accounted and controlled.
  These are memory used by cp, scp, tar etc.  While file mmap will
  be controlled by Balbir's RSS controller.

ToDo:
----

* Merge with container RSS controller and eliminate redundant code
* Support and test task migration and container deletion
* Review reclaim performance
* Optimise page reclaim

Signed-off-by: Vaidyanathan Srinivasan <svaidy@linux.vnet.ibm.com>
---
 include/linux/fs.h            |    6
 include/linux/pagecache_acct.h |   52 +++++
 init/Kconfig                  |    7
 kernel/Makefile               |    1
 kernel/pagecache_acct.c       |  376 ++++++++++++++++++++++++++++++++++++++++++
 mm/filemap.c                  |    8
 mm/vmscan.c                   |   76 ++++++++
 7 files changed, 524 insertions(+), 2 deletions(-)

--- linux-2.6.20.orig/include/linux/fs.h
+++ linux-2.6.20/include/linux/fs.h
@@ -447,6 +447,12 @@ struct address_space {
  spinlock_t  private_lock; /* for use by the address_space */
  struct list_head private_list; /* ditto */
  struct address_space *assoc_mapping; /* ditto */
+
+#ifdef CONFIG_CONTAINER_PAGECACHE_ACCT
+ struct container *container;  /* Charge page to the right container
+        using page->mapping */
+#endif
+
 } __attribute__((aligned(sizeof(long))));
  /*
```

```
   * On most architectures that alignment is already the case; but
--- /dev/null
+++ linux-2.6.20/include/linux/pagecache_acct.h
@@ -0,0 +1,52 @@
+/*
+ * Pagecache controller - "Account and control pagecache usage"
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ * This program is distributed in the hope that it will be useful,
+ * but WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+ * GNU General Public License for more details.
+ *
+ * You should have received a copy of the GNU General Public License
+ * along with this program; if not, write to the Free Software
+ * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
+ *

+ *
+ * Author: Vaidyanathan Srinivasan <svaidy@linux.vnet.ibm.com>
+ *
+ */
+
+#ifndef _LINUX_PAGECACHE_ACCT_H
+#define _LINUX_PAGECACHE_ACCT_H
+
+#include <linux/container.h>
+
+#ifdef CONFIG_CONTAINER_PAGECACHE_ACCT
+extern void pagecache_acct_init_page_ptr(struct page *page);
+extern void pagecache_acct_charge(struct page *page);
+extern void pagecache_acct_uncharge(struct page *page);
+extern int pagecache_acct_page_overlimit(struct page *page);
+extern int pagecache_acct_mapping_overlimit(struct address_space *mapping);
+extern int pagecache_acct_cont_overlimit(struct container *cont);
+extern int pagecache_acct_shrink_used(unsigned long nr_pages);
+#else
+static inline void pagecache_acct_init_page_ptr(struct page *page) {}
+static inline void pagecache_acct_charge(struct page *page) {}
+static inline void pagecache_acct_uncharge(struct page *page) {}
+static inline int pagecache_acct_page_overlimit(
+  struct page *page) { return 0; }
+static inline int pagecache_acct_mapping_overlimit(
+  struct address_space *mapping) { return 0; }
```

```
+static inline int pagecache_acct_cont_overlimit(
+  struct container *cont) { retrun 0; }
+static inline int pagecache_acct_shrink_used(
+  unsigned long nr_pages) { retrun 0; }
+#endif
+
+#endif
+
--- linux-2.6.20.orig/init/Kconfig
+++ linux-2.6.20/init/Kconfig
@@ -306,6 +306,13 @@ config CONTAINER_NS
          for instance virtual servers and checkpoint/restart
          jobs.

+config CONTAINER_PAGECACHE_ACCT
+ bool "Simple PageCache accounting container subsystem"
+ select CONTAINERS
+ help
+   Provides a simple Resource Controller for monitoring the
+   total pagecache consumed by the tasks in a container
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help
--- linux-2.6.20.orig/kernel/Makefile
+++ linux-2.6.20/kernel/Makefile
@@ -40,6 +40,7 @@ obj-$(CONFIG_CONTAINERS) += container.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_CONTAINER_CPUACCT) += cpu_acct.o
 obj-$(CONFIG_CONTAINER_NS) += ns_container.o
+obj-$(CONFIG_CONTAINER_PAGECACHE_ACCT) += pagecache_acct.o
 obj-$(CONFIG_IKCONFIG) += configs.o
 obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
 obj-$(CONFIG_AUDIT) += audit.o auditfilter.o
--- /dev/null
+++ linux-2.6.20/kernel/pagecache_acct.c
@@ -0,0 +1,376 @@
+
+/*
+ * Pagecache controller - "Account and control pagecache usage"
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License as published by
+ * the Free Software Foundation; either version 2 of the License, or
+ * (at your option) any later version.
+ *
+ * This program is distributed in the hope that it will be useful,
+ * but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+ * GNU General Public License for more details.
+ *
+ * You should have received a copy of the GNU General Public License
+ * along with this program; if not, write to the Free Software
+ * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
+ *

+ *
+ * Author: Vaidyanathan Srinivasan <svaidy@linux.vnet.ibm.com>
+ *
+ */
+
+#include <linux/module.h>
+#include <linux/container.h>
+#include <linux/fs.h>
+#include <linux/mm.h>
+#include <linux/mm_types.h>
+#include <linux/uaccess.h>
+#include <asm/div64.h>
+#include <linux/klog.h>
+#include <linux/pagecache_acct.h>
+
+/*
+ * Convert unit from pages to kilobytes
+ */
+#define K(x) ((x) << (PAGE_SHIFT - 10))
+/*
+ * Convert unit from kilobytes to pages
+ */
+#define K_to_pages(x) ((x) >> (PAGE_SHIFT -10))
+
+/* Limits for user string */
+
+#define MAX_LIMIT_STRING  25
+
+/* nr_pages above limit to start reclaim */
+
+#define NR_PAGES_RECLAIM_THRESHOLD 64
+
+struct pagecache_acct {
+ struct container_subsys_state css;
+ spinlock_t lock;
+ atomic_t count; /*Pagecache pages added*/
+ atomic_t removed_count; /*Pagecache pages removed*/
+ unsigned int limit; /* Pagecache usage limit in kiB */
+};
+
```

```
+/*Failure counters for debugging*/
+static atomic_t failed_count; /*Page charge failures?*/
+static atomic_t failed_removed_count; /*Page uncharge failure?*/
+static atomic_t reclaim_count; /*Overlimit direct page reclaim run count */
+
+static struct container_subsys pagecache_acct_subsys;
+
+static inline struct pagecache_acct *container_pca(struct container *cont)
+{
+ return container_of(
+   container_subsys_state(cont, &pagecache_acct_subsys),
+       struct pagecache_acct, css);
+}
+
+static int pagecache_acct_create(struct container_subsys *ss,
+    struct container *cont)
+{
+ struct pagecache_acct *pca = kzalloc(sizeof(*pca), GFP_KERNEL);
+ if (!pca)
+  return -ENOMEM;
+ spin_lock_init(&pca->lock);
+ cont->subsys[pagecache_acct_subsys.subsys_id] = &pca->css;
+ return 0;
+}
+
+static void pagecache_acct_destroy(struct container_subsys *ss,
+     struct container *cont)
+{
+ kfree(container_pca(cont));
+}
+
+static unsigned int pagecache_get_usage(struct pagecache_acct *pca)
+{
+ unsigned int count, removed_count, pagecache_used_kB;
+
+ count = (unsigned int) atomic_read(&pca->count);
+ removed_count = (unsigned int) atomic_read(&pca->removed_count);
+ /* Take care of roll over in the counters */
+ if (count >= removed_count)
+  pagecache_used_kB = count - removed_count;
+ else
+  pagecache_used_kB = ~0UL - (removed_count - count) + 1;
+
+ /* Convert unit from pages into kB */
+ pagecache_used_kB = K(pagecache_used_kB);
+
+ return pagecache_used_kB;
+}
```

```
+
+static ssize_t pagecache_usage_read(struct container *cont,
+
+        struct cftype *cft,
+        struct file *file,
+        char __user *buf,
+        size_t nbytes, loff_t *ppos)
+{
+ struct pagecache_acct *pca = container_pca(cont);
+ char usagebuf[64];
+ char *s = usagebuf;
+
+ s += sprintf(s, "%u kB \n", pagecache_get_usage(pca));
+ return simple_read_from_buffer(buf, nbytes, ppos, usagebuf,
+      s - usagebuf);
+}
+
+static ssize_t pagecache_debug_read(struct container *cont,
+        struct cftype *cft,
+        struct file *file,
+        char __user *buf,
+        size_t nbytes, loff_t *ppos)
+{
+ struct pagecache_acct *pca = container_pca(cont);
+ char usagebuf[64];
+ char *s = usagebuf;
+
+ s += sprintf(s, "%u kB Cnt: %u, %u Failed: %u, %u Reclaim: %u\n",
+    pagecache_get_usage(pca),
+    (unsigned int) atomic_read(&pca->count),
+    (unsigned int) atomic_read(&pca->removed_count),
+    (unsigned int) atomic_read(&failed_count),
+    (unsigned int) atomic_read(&failed_removed_count),
+    (unsigned int) atomic_read(&reclaim_count));
+
+ return simple_read_from_buffer(buf, nbytes, ppos, usagebuf,
+      s - usagebuf);
+}
+
+static ssize_t pagecache_limit_read(struct container *cont,
+        struct cftype *cft,
+        struct file *file,
+        char __user *buf,
+        size_t nbytes, loff_t *ppos)
+{
+ struct pagecache_acct *pca = container_pca(cont);
+ char usagebuf[64];
+ char *s = usagebuf;
```

```
+
+ s += sprintf(s, "%u kB\n", pca->limit);
+
+ return simple_read_from_buffer(buf, nbytes, ppos, usagebuf,
+       s - usagebuf);
+}
+
+static ssize_t pagecache_limit_write(struct container *cont,
+         struct cftype *cft,
+         struct file *file,
+         const char __user *buf,
+         size_t nbytes, loff_t *ppos)
+{
+ struct pagecache_acct *pca = container_pca(cont);
+ char buffer[MAX_LIMIT_STRING];
+ unsigned int limit;
+ unsigned int nr_pages;
+
+ if( nbytes > (MAX_LIMIT_STRING-1) || nbytes < 1 )
+  return -EINVAL;
+
+ if (copy_from_user(buffer, buf, nbytes))
+  return -EFAULT;
+
+ buffer[nbytes] = '\0'; /* Null termination */
+ limit = simple_strtoul(buffer, NULL, 10);
+ /* Round it off to lower 4K page boundary */
+ limit &= ~0x3;
+
+ /* Set the value in pca struct (atomic store). No read-modify-update */
+ pca->limit = limit;
+
+ /* Check for overlimit and initiate reclaim if needed */
+ /* The limits have changed now */
+ if ((nr_pages = pagecache_acct_cont_overlimit(cont))) {
+  pagecache_acct_shrink_used(nr_pages);
+ }
+ return nbytes;
+}
+
+static struct cftype cft_usage = {
+ .name = "pagecache_usage",
+ .read = pagecache_usage_read,
+};
+
+static struct cftype cft_debug = {
+ .name = "pagecache_debug",
+ .read = pagecache_debug_read,
```

```
+};
+
+static struct cftype cft_limit = {
+ .name = "pagecache_limit",
+ .read = pagecache_limit_read,
+ .write = pagecache_limit_write,
+};
+
+static int pagecache_acct_populate(struct container_subsys *ss,
+      struct container *cont)
+{
+ int rc;
+ rc = container_add_file(cont, &cft_usage);
+ if (rc)
+  return rc;
+ rc = container_add_file(cont, &cft_debug);
+ if (rc)
+  /* Cleanup with container_remove_file()? */
+  return rc;
+ rc = container_add_file(cont, &cft_limit);
+ return rc;
+}
+
+static struct container *mapping_container(struct address_space *mapping)
+{
+ if ((unsigned long) mapping & PAGE_MAPPING_ANON)
+  mapping = NULL;
+ if(!mapping)
+  return NULL;
+ if (!mapping->container) {
+  printk( KERN_DEBUG "Null Container in mapping: %p\n", mapping);
+ }
+ return mapping->container;
+}
+
+static struct container *page_container(struct page *page)
+{
+ return mapping_container(page->mapping);
+}
+
+void pagecache_acct_init_page_ptr(struct page *page)
+{
+ struct address_space *mapping;
+ mapping = page->mapping;
+ if ((unsigned long) mapping & PAGE_MAPPING_ANON)
+  mapping = NULL;
+ BUG_ON(!mapping);
+ if (current) {
```

```
+  if(!mapping->container)
+   mapping->container = task_container(current, &pagecache_acct_subsys);
+ } else
+  mapping->container = NULL;
+}
+
+void pagecache_acct_charge(struct page *page)
+{
+ struct container *cont;
+ struct pagecache_acct *pca;
+ unsigned int nr_pages;
+
+ if (pagecache_acct_subsys.subsys_id < 0) return;
+ cont = page_container(page);
+ if (cont) {
+  pca = container_pca(cont);
+  BUG_ON(!pca);
+  atomic_inc(&pca->count);
+ } else {
+  /* page->container is null??? */
+  printk(KERN_WARNING "pca_charge:page_container null\n");
+  atomic_inc(&failed_count);
+ }
+ /* Check for overlimit and initiate reclaim if needed */
+ if ((nr_pages = pagecache_acct_page_overlimit(page))) {
+  pagecache_acct_shrink_used(nr_pages);
+ }
+}
+
+void pagecache_acct_uncharge(struct page * page)
+{
+ struct container *cont;
+ struct pagecache_acct *pca;
+
+ if (pagecache_acct_subsys.subsys_id < 0) return;
+ cont = page_container(page);
+ if (cont) {
+  pca = container_pca(cont);
+  BUG_ON(!pca);
+  atomic_inc(&pca->removed_count);
+ } else {
+  /* page->container is null??? */
+  printk(KERN_WARNING "pca_uncharge:page_container null\n");
+  atomic_inc(&failed_removed_count);
+ }
+}
+
+int pagecache_acct_page_overlimit(struct page *page)
```

```
+{
+ struct container *cont;
+
+ if (pagecache_acct_subsys.subsys_id < 0)
+  return 0;
+ cont = page_container(page);
+ if (!cont)
+  return 0;
+ return pagecache_acct_cont_overlimit(cont);
+}
+
+int pagecache_acct_mapping_overlimit(struct address_space *mapping)
+{
+ struct container *cont;
+
+ if (pagecache_acct_subsys.subsys_id < 0)
+  return 0;
+ cont = mapping_container(mapping);
+ if (!cont)
+  return 0;
+ return pagecache_acct_cont_overlimit(cont);
+}
+
+int pagecache_acct_cont_overlimit(struct container *cont)
+{
+ struct pagecache_acct *pca;
+ unsigned int used, limit;
+
+ if (pagecache_acct_subsys.subsys_id < 0)
+  return 0;
+
+ if (!cont)
+  return 0;
+ pca = container_pca(cont);
+ used = pagecache_get_usage(pca);
+ limit = pca->limit;
+ if( limit && (used > limit) )
+  return K_to_pages(used - limit);
+ else
+  return 0;
+}
+
+extern unsigned long shrink_all_pagecache_memory(unsigned long nr_pages);
+
+int pagecache_acct_shrink_used(unsigned long nr_pages)
+{
+ unsigned long ret = 0;
+ atomic_inc(&reclaim_count);
```

```
+
+ /* Don't call reclaim for each page above limit */
+ if (nr_pages > NR_PAGES_RECLAIM_THRESHOLD) {
+  ret += shrink_all_pagecache_memory(nr_pages);
+ }
+
+ return 0;
+}
+
+static struct container_subsys pagecache_acct_subsys = {
+ .name = "pagecache_acct",
+ .create = pagecache_acct_create,
+ .destroy = pagecache_acct_destroy,
+ .populate = pagecache_acct_populate,
+ .subsys_id = -1,
+};
+
+int __init init_pagecache_acct(void)
+{
+ int id = container_register_subsys(&pagecache_acct_subsys);
+ int rc = 0; /* Default to success */
+ /* Flag failure */
+ if (id < 0)
+  rc = id;
+ return rc;
+}
+
+__initcall(init_pagecache_acct);
+
--- linux-2.6.20.orig/mm/filemap.c
+++ linux-2.6.20/mm/filemap.c
@@ -30,6 +30,7 @@
 #include <linux/security.h>
 #include <linux/syscalls.h>
 #include <linux/cpuset.h>
+#include <linux/pagecache_acct.h>
 #include "filemap.h"
 #include "internal.h"

@@ -117,6 +118,8 @@ void __remove_from_page_cache(struct pag
  struct address_space *mapping = page->mapping;

  radix_tree_delete(&mapping->page_tree, page->index);
+ /* Uncharge before the mapping is gone */
+ pagecache_acct_uncharge(page);
  page->mapping = NULL;
  mapping->nrpages--;
  __dec_zone_page_state(page, NR_FILE_PAGES);
```

```
@@ -451,6 +454,11 @@ int add_to_page_cache(struct page *page,
    __inc_zone_page_state(page, NR_FILE_PAGES);
   }
   write_unlock_irq(&mapping->tree_lock);
+  /* Unlock before charge, because we may reclaim this inline */
+  if (!error) {
+   pagecache_acct_init_page_ptr(page);
+   pagecache_acct_charge(page);
+  }
   radix_tree_preload_end();
  }
  return error;
--- linux-2.6.20.orig/mm/vmscan.c
+++ linux-2.6.20/mm/vmscan.c
@@ -37,6 +37,7 @@
 #include <linux/delay.h>
 #include <linux/kthread.h>
 #include <linux/freezer.h>
+#include <linux/pagecache_acct.h>

 #include <asm/tlbflush.h>
 #include <asm/div64.h>
@@ -66,6 +67,9 @@ struct scan_control {
  int swappiness;

  int all_unreclaimable;
+
+ int reclaim_pagecache_only;  /* Set when called from
+     pagecache controller */
 };

 /*
@@ -470,6 +474,16 @@ static unsigned long shrink_page_list(st
    goto keep;

   VM_BUG_ON(PageActive(page));
+  /* Take it easy if we are doing only pagecache pages */
+  if (sc->reclaim_pagecache_only) {
+   /* Check if this is a pagecache page they are not mapped */
+   if (page_mapped(page))
+    goto keep_locked;
+   /* Check if this container has exceeded pagecache limit */
+   if (!pagecache_acct_page_overlimit(page))
+    goto keep_locked;
+  }
+

   sc->nr_scanned++;
```

```
@@ -518,7 +532,8 @@ static unsigned long shrink_page_list(st
   }

   if (PageDirty(page)) {
-   if (referenced)
+   /* Reclaim even referenced pagecache pages if over limit */
+   if (!pagecache_acct_page_overlimit(page) && referenced)
     goto keep_locked;
    if (!may_enter_fs)
     goto keep_locked;
@@ -832,6 +847,14 @@ force_reclaim_mapped:
   cond_resched();
   page = lru_to_page(&l_hold);
   list_del(&page->lru);
+  /* While reclaiming pagecache make it easy */
+  if (sc->reclaim_pagecache_only) {
+   if (page_mapped(page) || !pagecache_acct_page_overlimit(page)) {
+    list_add(&page->lru, &l_active);
+    continue;
+   }
+  }
+
   if (page_mapped(page)) {
    if (!reclaim_mapped ||
        (total_swap_pages == 0 && PageAnon(page)) ||
@@ -1027,6 +1050,7 @@ unsigned long try_to_free_pages(struct z
   .swap_cluster_max = SWAP_CLUSTER_MAX,
   .may_swap = 1,
   .swappiness = vm_swappiness,
+  .reclaim_pagecache_only = 0,
  };

  count_vm_event(ALLOCSTALL);
@@ -1131,6 +1155,7 @@ static unsigned long balance_pgdat(pg_da
   .may_swap = 1,
   .swap_cluster_max = SWAP_CLUSTER_MAX,
   .swappiness = vm_swappiness,
+  .reclaim_pagecache_only = 0,
  };
  /*
   * temp_priority is used to remember the scanning priority at which
@@ -1361,7 +1386,7 @@ void wakeup_kswapd(struct zone *zone, in
  wake_up_interruptible(&pgdat->kswapd_wait);
 }

-#ifdef CONFIG_PM
+#if defined (CONFIG_PM) || defined (CONFIG_CONTAINER_PAGECACHE_ACCT)
```

```
 /*
  * Helper function for shrink_all_memory().  Tries to reclaim 'nr_pages' pages
  * from LRU lists system-wide, for given pass and priority, and returns the
@@ -1436,6 +1461,7 @@ unsigned long shrink_all_memory(unsigned
   .swap_cluster_max = nr_pages,
   .may_writepage = 1,
   .swappiness = vm_swappiness,
+  .reclaim_pagecache_only = 0,
  };

  current->reclaim_state = &reclaim_state;
@@ -1510,6 +1536,52 @@ out:

  return ret;
 }
+
+unsigned long shrink_all_pagecache_memory(unsigned long nr_pages)
+{
+ unsigned long ret = 0;
+ int pass;
+ struct reclaim_state reclaim_state;
+ struct scan_control sc = {
+  .gfp_mask = GFP_KERNEL,
+  .may_swap = 0,
+  .swap_cluster_max = nr_pages,
+  .may_writepage = 1,
+  .swappiness = 0, /* Do not swap, only pagecache reclaim */
+  .reclaim_pagecache_only = 1, /* Flag it */
+ };
+
+ current->reclaim_state = &reclaim_state;
+
+ /*
+  * We try to shrink LRUs in 5 passes:
+  * 0 = Reclaim from inactive_list only
+  * 1 = Reclaim from active list but don't reclaim mapped
+  * 2 = 2nd pass of type 1
+  * 3 = Reclaim mapped (normal reclaim)
+  * 4 = 2nd pass of type 3
+  */
+ for (pass = 0; pass < 5; pass++) {
+  int prio;
+
+  for (prio = DEF_PRIORITY; prio >= 0; prio--) {
+   unsigned long nr_to_scan = nr_pages - ret;
+   sc.nr_scanned = 0;
+   ret += shrink_all_zones(nr_to_scan, prio, pass, &sc);
+   if (ret >= nr_pages)
```

```
+    goto out;
+
+   if (sc.nr_scanned && prio < DEF_PRIORITY - 2)
+    congestion_wait(WRITE, HZ / 10);
+  }
+ }
+
+
+out:
+ current->reclaim_state = NULL;
+
+ return ret;
+}
 #endif

 /* It's optimal to keep kswapds on the same CPUs as their memory, but
```