Subject: Re: [PATCH 2.6.19-rc3] VFS: per-sb dentry lru list
Posted by vaverin on Tue, 14 Nov 2006 06:12:45 GMT
View Forum Message <> Reply to Message

Hello Neil

Neil Brown wrote:
> On Wednesday November 1, vvs@sw.ru wrote:
>> Currently we have 3 type of functions that works with dentry_unused list:
>>
>> 1) prune_dcache(NULL) -- called from shrink_dcache_memory, frees the memory and
>> requires global LRU. works well in current implementation.
>> 2) prune_dcache(sb) -- called from shrink_dcache_parent(), frees subtree, LRU
>> is not need here. Current implementation uses global LRU for these purposes, it
>> is ineffective, and patch from Neil Brown fixes this issue.
>> 3) shrink_dcache_sb() -- called when we need to free the unused dentries for
>> given super block. Current implementation is not effective too, and per-sb LRU
>> would be the best solution here. On the other hand patch from Neil Brown is much
>> better than current implementation.
>>
>> In general I think that we should approve Neil Brown's patch. We (I and Kirill
>> Korotaev) are ready to acknowledge it when the following remarks fill be fixed:
>
>
>> - it seems for me list_splice() is not required inside
>>   prune_dcache(),
>
> Yes, the list should be empty when we finish so you are right.
>
>> - DCACHE_REFERENCED dentries should not be removed from private list to
>> dentry_unused list, this flag should be ignored if the private list is used,
>
> Agreed.
>
>> - count argument should be ignored in this case too, we want to free all the
>> dentries in private list,
>
> Agreed.
>
>> - when we shrink the whole super block we should free per-sb anonymous dentries
>> too (please see Kirill Korotaev's letter)
>>
>
> Yes. Unfortunately I don't think it is as easy as it sounds.
> I'll have a closer look.
>
>
>> Then I'm going to prepare new patch that will enhance the shrink_dcache_sb()

>> performance:
>> - we can add new list head into struct superblock and use it in
>> shrink_dcache_sb() instead of temporal private list. We will check is it empty
>> in dput() and add the new unused dentries to per-sb list instead of
>> dentry_unused list.
>
> I think that makes sense. It means that you end up doing less work in
> select_parent, because the work has already been done in dput.
>
> How is the patch going?

Please see patches in attach files:
first one is incremental for your patch,
second one is merged version.

It would be very interesting to know your opinion about these changes.

thank you,
 Vasily Averin

```
--- linux-2.6.19-rc4/fs/dcache.c.vshud1 2006-11-02 14:23:03.000000000 +0300
+++ linux-2.6.19-rc4/fs/dcache.c 2006-11-03 14:57:58.000000000 +0300
@@ -171,8 +171,14 @@ repeat:
   if (d_unhashed(dentry))
   goto kill_it;
    if (list_empty(&dentry->d_lru)) {
-    dentry->d_flags |= DCACHE_REFERENCED;
-    list_add(&dentry->d_lru, &dentry_unused);
+  struct list_head *list;
+
+  list = &dentry->d_sb->s_shrink_list;
+  if (list_empty(list)) {
+   list = &dentry_unused;
+     dentry->d_flags |= DCACHE_REFERENCED;
+  }
+    list_add(&dentry->d_lru, list);
     dentry_stat.nr_unused++;
    }
   spin_unlock(&dentry->d_lock);
@@ -394,10 +400,6 @@ static void prune_one_dentry(struct dent
  *
  * This function may fail to free any resources if
  * all the dentries are in use.
- *
- * Any dentries that were not removed due to the @count
- * limit will be splice on to the end of dentry_unused,
- * so they should already be founded in dentry_stat.nr_unused.
  */
```

```
 static void prune_dcache(int count, struct list_head *list)
@@ -409,7 +411,7 @@ static void prune_dcache(int count, stru
   /* use the dentry_unused list */
   list = &dentry_unused;

- for (; count ; count--) {
+ for (; have_list || count-- ;) {
   struct dentry *dentry;
   struct list_head *tmp;
   struct rw_semaphore *s_umount;
@@ -434,6 +436,17 @@ static void prune_dcache(int count, stru
     spin_unlock(&dentry->d_lock);
    continue;
   }
+  /*
+   * If this dentry is for "my" filesystem, then I can prune it
+   * without taking the s_umount lock: either I already hold it
+   * (called from shrink_dcache_sb) or are called from some
+   * filesystem opernations and therefore cannot race with
+   * generic_shutdown_super().
+   */
+  if (have_list) {
+   prune_one_dentry(dentry);
+   continue;
+  }
   /* If the dentry was recently referenced, don't free it. */
   if (dentry->d_flags & DCACHE_REFERENCED) {
    dentry->d_flags &= ~DCACHE_REFERENCED;
@@ -443,21 +456,6 @@ static void prune_dcache(int count, stru
    continue;
   }
   /*
-   * If the dentry is not DCACHED_REFERENCED, it is time
-   * to remove it from the dcache, provided the super block is
-   * NULL (which means we are trying to reclaim memory)
-   * or this dentry belongs to the same super block that
-   * we want to shrink.
-   */
-  /*
-   * If this dentry is for "my" filesystem, then I can prune it
-   * without taking the s_umount lock (I already hold it).
-   */
-  if (have_list) {
-   prune_one_dentry(dentry);
-   continue;
-  }
-  /*
```

```
    * ...otherwise we need to be sure this filesystem isn't being
    * unmounted, otherwise we could race with
    * generic_shutdown_super(), and end up holding a reference to
@@ -483,27 +481,9 @@ static void prune_dcache(int count, stru
    list_add(&dentry->d_lru, &dentry_unused);
   dentry_stat.nr_unused++;
  }
- /* split any remaining entries back onto dentry_unused */
- if (have_list)
-  list_splice(list, dentry_unused.prev);
  spin_unlock(&dcache_lock);
 }

-/**
- * shrink_dcache_sb - shrink dcache for a superblock
- * @sb: superblock
- *
- * Shrink the dcache for the specified super block. This
- * is used to reduce the dcache presence of a file system
- * before re-mounting, and when invalidating the device
- * holding a file system.
- */
-
-void shrink_dcache_sb(struct super_block * sb)
-{
- shrink_dcache_parent(sb->s_root);
-}
-
 /*
  * destroy a single subtree of dentries for unmount
  * - see the comments on shrink_dcache_for_umount() for a description of the
@@ -706,11 +686,10 @@ positive:
  * drop the lock and return early due to latency
  * constraints.
  */
-static int select_parent(struct dentry * parent, struct list_head *new)
+static void select_parent(struct dentry * parent, struct list_head *new)
 {
  struct dentry *this_parent = parent;
  struct list_head *next;
- int found = 0;

  spin_lock(&dcache_lock);
 repeat:
@@ -732,7 +711,6 @@ resume:
   if (!atomic_read(&dentry->d_count)) {
    list_add_tail(&dentry->d_lru, new);
    dentry_stat.nr_unused++;
```

```
-   found++;
   }

   /*
@@ -740,7 +718,7 @@ resume:
    * ensures forward progress). We'll be coming back to find
    * the rest.
    */
-  if (found && need_resched())
+  if (!list_empty(new) && need_resched())
    goto out;

   /*
@@ -761,7 +739,37 @@ resume:
  }
 out:
  spin_unlock(&dcache_lock);
- return found;
+}
+
+/**
+ * select_anon - further prune the cache
+ * @sb: superblock
+ *
+ * Prune the dentries that are anonymous
+ */
+
+static void select_anon(struct super_block *sb)
+{
+ struct hlist_node *lp;
+ struct hlist_head *head = &sb->s_anon;
+
+ spin_lock(&dcache_lock);
+ hlist_for_each(lp, head) {
+  struct dentry *this = hlist_entry(lp, struct dentry, d_hash);
+  if (!list_empty(&this->d_lru)) {
+   dentry_stat.nr_unused--;
+   list_del_init(&this->d_lru);
+  }
+  /*
+   * move only zero ref count dentries to the end
+   * of list for prune_dcache
+   */
+  if (!atomic_read(&this->d_count)) {
+   list_add(&this->d_lru, &sb->s_shrink_list);
+   dentry_stat.nr_unused++;
+  }
+ }
```

```
+ spin_unlock(&dcache_lock);
 }

 /**
@@ -773,11 +781,38 @@ out:

 void shrink_dcache_parent(struct dentry * parent)
 {
- int found;
  LIST_HEAD(list);

- while ((found = select_parent(parent, &list)) != 0)
-  prune_dcache(found, &list);
+ for (;;) {
+  select_parent(parent, &list);
+  if (list_empty(&list))
+   break;
+  prune_dcache(0, &list);
+ }
+}
+
+/**
+ * shrink_dcache_sb - shrink dcache for a superblock
+ * @sb: superblock
+ *
+ * Shrink the dcache for the specified super block. This
+ * is used to reduce the dcache presence of a file system
+ * before re-mounting, and when invalidating the device
+ * holding a file system.
+ */
+
+void shrink_dcache_sb(struct super_block * sb)
+{
+ struct list_head *list;
+
+ list = &sb->s_shrink_list;
+ for (;;) {
+  select_parent(sb->s_root, list);
+  select_anon(sb);
+  if (list_empty(list))
+   break;
+  prune_dcache(0, list);
+ }
 }

 /*
--- linux-2.6.19-rc4/include/linux/fs.h.vshud1 2006-11-03 15:28:04.000000000 +0300
+++ linux-2.6.19-rc4/include/linux/fs.h 2006-11-03 14:52:16.000000000 +0300
```

```
@@ -941,6 +941,7 @@ struct super_block {
  struct hlist_head s_anon;  /* anonymous dentries for (nfs) exporting */
  struct list_head s_files;

+ struct list_head s_shrink_list;
  struct block_device *s_bdev;
  struct list_head s_instances;
  struct quota_info s_dquot; /* Diskquota specific options */

--- linux-2.6.19-rc4/fs/dcache.c.vdch 2006-11-03 14:22:54.000000000 +0300
+++ linux-2.6.19-rc4/fs/dcache.c 2006-11-03 14:57:58.000000000 +0300
@@ -171,8 +171,14 @@ repeat:
   if (d_unhashed(dentry))
   goto kill_it;
    if (list_empty(&dentry->d_lru)) {
-    dentry->d_flags |= DCACHE_REFERENCED;
-    list_add(&dentry->d_lru, &dentry_unused);
+  struct list_head *list;
+
+  list = &dentry->d_sb->s_shrink_list;
+  if (list_empty(list)) {
+   list = &dentry_unused;
+     dentry->d_flags |= DCACHE_REFERENCED;
+  }
+    list_add(&dentry->d_lru, list);
     dentry_stat.nr_unused++;
    }
   spin_unlock(&dentry->d_lock);
@@ -384,8 +390,8 @@ static void prune_one_dentry(struct dent
 /**
  * prune_dcache - shrink the dcache
  * @count: number of entries to try and free
- * @sb: if given, ignore dentries for other superblocks
- *        which are being unmounted.
+ * @list: If given, remove from this list instead of
+ *        from dentry_unused.
  *
  * Shrink the dcache. This is done when we need
  * more memory, or simply when we need to unmount
@@ -396,33 +402,27 @@ static void prune_one_dentry(struct dent
  * all the dentries are in use.
  */

-static void prune_dcache(int count, struct super_block *sb)
+static void prune_dcache(int count, struct list_head *list)
 {
+ int have_list = list != NULL;
+
```

```
  spin_lock(&dcache_lock);
- for (; count ; count--) {
+ if (!have_list)
+  /* use the dentry_unused list */
+  list = &dentry_unused;
+
+ for (; have_list || count-- ;) {
   struct dentry *dentry;
   struct list_head *tmp;
   struct rw_semaphore *s_umount;

   cond_resched_lock(&dcache_lock);

-  tmp = dentry_unused.prev;
-  if (sb) {
-   /* Try to find a dentry for this sb, but don't try
-    * too hard, if they aren't near the tail they will
-    * be moved down again soon
-    */
-   int skip = count;
-   while (skip && tmp != &dentry_unused &&
-       list_entry(tmp, struct dentry, d_lru)->d_sb != sb) {
-    skip--;
-    tmp = tmp->prev;
-   }
-  }
-  if (tmp == &dentry_unused)
+  tmp = list->prev;
+  if (tmp == list)
    break;
   list_del_init(tmp);
-  prefetch(dentry_unused.prev);
+  prefetch(list->prev);
   dentry_stat.nr_unused--;
   dentry = list_entry(tmp, struct dentry, d_lru);

@@ -436,6 +436,17 @@ static void prune_dcache(int count, stru
    spin_unlock(&dentry->d_lock);
    continue;
   }
+  /*
+   * If this dentry is for "my" filesystem, then I can prune it
+   * without taking the s_umount lock: either I already hold it
+   * (called from shrink_dcache_sb) or are called from some
+   * filesystem opernations and therefore cannot race with
+   * generic_shutdown_super().
+   */
+  if (have_list) {
```

```
+   prune_one_dentry(dentry);
+   continue;
+  }
    /* If the dentry was recently referenced, don't free it. */
    if (dentry->d_flags & DCACHE_REFERENCED) {
     dentry->d_flags &= ~DCACHE_REFERENCED;
@@ -445,21 +456,6 @@ static void prune_dcache(int count, stru
     continue;
    }
    /*
-   * If the dentry is not DCACHED_REFERENCED, it is time
-   * to remove it from the dcache, provided the super block is
-   * NULL (which means we are trying to reclaim memory)
-   * or this dentry belongs to the same super block that
-   * we want to shrink.
-   */
-  /*
-   * If this dentry is for "my" filesystem, then I can prune it
-   * without taking the s_umount lock (I already hold it).
-   */
-  if (sb && dentry->d_sb == sb) {
-   prune_one_dentry(dentry);
-   continue;
-  }
-  /*
     * ...otherwise we need to be sure this filesystem isn't being
     * unmounted, otherwise we could race with
     * generic_shutdown_super(), and end up holding a reference to
@@ -489,67 +485,6 @@ static void prune_dcache(int count, stru
 }

 /*
- * Shrink the dcache for the specified super block.
- * This allows us to unmount a device without disturbing
- * the dcache for the other devices.
- *
- * This implementation makes just two traversals of the
- * unused list.  On the first pass we move the selected
- * dentries to the most recent end, and on the second
- * pass we free them.  The second pass must restart after
- * each dput(), but since the target dentries are all at
- * the end, it's really just a single traversal.
- */
-
-/**
- * shrink_dcache_sb - shrink dcache for a superblock
- * @sb: superblock
- *
```

```
- * Shrink the dcache for the specified super block. This
- * is used to free the dcache before unmounting a file
- * system
- */
-
-void shrink_dcache_sb(struct super_block * sb)
-{
- struct list_head *tmp, *next;
- struct dentry *dentry;
-
- /*
-  * Pass one ... move the dentries for the specified
-  * superblock to the most recent end of the unused list.
-  */
- spin_lock(&dcache_lock);
- list_for_each_safe(tmp, next, &dentry_unused) {
-  dentry = list_entry(tmp, struct dentry, d_lru);
-  if (dentry->d_sb != sb)
-   continue;
-  list_move(tmp, &dentry_unused);
- }
-
- /*
-  * Pass two ... free the dentries for this superblock.
-  */
-repeat:
- list_for_each_safe(tmp, next, &dentry_unused) {
-  dentry = list_entry(tmp, struct dentry, d_lru);
-  if (dentry->d_sb != sb)
-   continue;
-  dentry_stat.nr_unused--;
-  list_del_init(tmp);
-  spin_lock(&dentry->d_lock);
-  if (atomic_read(&dentry->d_count)) {
-   spin_unlock(&dentry->d_lock);
-   continue;
-  }
-  prune_one_dentry(dentry);
-  cond_resched_lock(&dcache_lock);
-  goto repeat;
- }
- spin_unlock(&dcache_lock);
-}
-
-/*
  * destroy a single subtree of dentries for unmount
  * - see the comments on shrink_dcache_for_umount() for a description of the
  *   locking
```

```
@@ -739,7 +674,7 @@ positive:

 /*
  * Search the dentry child list for the specified parent,
- * and move any unused dentries to the end of the unused
+ * and move any unused dentries to the end of a new unused
  * list for prune_dcache(). We descend to the next level
  * whenever the d_subdirs list is non-empty and continue
  * searching.
@@ -751,11 +686,10 @@ positive:
  * drop the lock and return early due to latency
  * constraints.
  */
-static int select_parent(struct dentry * parent)
+static void select_parent(struct dentry * parent, struct list_head *new)
 {
  struct dentry *this_parent = parent;
  struct list_head *next;
- int found = 0;

  spin_lock(&dcache_lock);
 repeat:
@@ -775,9 +709,8 @@ resume:
    * of the unused list for prune_dcache
    */
   if (!atomic_read(&dentry->d_count)) {
-   list_add_tail(&dentry->d_lru, &dentry_unused);
+   list_add_tail(&dentry->d_lru, new);
    dentry_stat.nr_unused++;
-   found++;
   }

   /*
@@ -785,7 +718,7 @@ resume:
    * ensures forward progress). We'll be coming back to find
    * the rest.
    */
-  if (found && need_resched())
+  if (!list_empty(new) && need_resched())
    goto out;

   /*
@@ -806,7 +739,37 @@ resume:
  }
 out:
  spin_unlock(&dcache_lock);
- return found;
+}
```

```
+
+/**
+ * select_anon - further prune the cache
+ * @sb: superblock
+ *
+ * Prune the dentries that are anonymous
+ */
+
+static void select_anon(struct super_block *sb)
+{
+ struct hlist_node *lp;
+ struct hlist_head *head = &sb->s_anon;
+
+ spin_lock(&dcache_lock);
+ hlist_for_each(lp, head) {
+  struct dentry *this = hlist_entry(lp, struct dentry, d_hash);
+  if (!list_empty(&this->d_lru)) {
+   dentry_stat.nr_unused--;
+   list_del_init(&this->d_lru);
+  }
+  /*
+   * move only zero ref count dentries to the end
+   * of list for prune_dcache
+   */
+  if (!atomic_read(&this->d_count)) {
+   list_add(&this->d_lru, &sb->s_shrink_list);
+   dentry_stat.nr_unused++;
+  }
+ }
+ spin_unlock(&dcache_lock);
 }

 /**
@@ -818,10 +781,38 @@ out:

 void shrink_dcache_parent(struct dentry * parent)
 {
- int found;
+ LIST_HEAD(list);

- while ((found = select_parent(parent)) != 0)
-  prune_dcache(found, parent->d_sb);
+ for (;;) {
+  select_parent(parent, &list);
+  if (list_empty(&list))
+   break;
+  prune_dcache(0, &list);
+ }
```

```
+}
+
+/**
+ * shrink_dcache_sb - shrink dcache for a superblock
+ * @sb: superblock
+ *
+ * Shrink the dcache for the specified super block. This
+ * is used to reduce the dcache presence of a file system
+ * before re-mounting, and when invalidating the device
+ * holding a file system.
+ */
+
+void shrink_dcache_sb(struct super_block * sb)
+{
+ struct list_head *list;
+
+ list = &sb->s_shrink_list;
+ for (;;) {
+  select_parent(sb->s_root, list);
+  select_anon(sb);
+  if (list_empty(list))
+   break;
+  prune_dcache(0, list);
+ }
 }

 /*
--- linux-2.6.19-rc4/include/linux/fs.h.vdch 2006-11-03 14:22:12.000000000 +0300
+++ linux-2.6.19-rc4/include/linux/fs.h 2006-11-03 14:52:16.000000000 +0300
@@ -941,6 +941,7 @@ struct super_block {
  struct hlist_head s_anon;  /* anonymous dentries for (nfs) exporting */
  struct list_head s_files;

+ struct list_head s_shrink_list;
  struct block_device *s_bdev;
  struct list_head s_instances;
  struct quota_info s_dquot; /* Diskquota specific options */
```