Subject: Re: [PATCH] Fix rmmod/read/write races in /proc entries
Posted by adobriyan on Wed, 24 Jan 2007 15:16:41 GMT
View Forum Message <> Reply to Message

On Tue, Jan 23, 2007 at 12:58:01PM -0800, Andrew Morton wrote:
> <head spins>
>
> Looks a bit hacky. Can this race not be fixed by addition of suitable
> locking, or possibly refcounting-under-locking?

I'll think about it.

> > @@ -76,6 +77,12 @@ proc_file_read(struct file *file, char _
> >   if (!(page = (char*) __get_free_page(GFP_KERNEL)))
> >   return -ENOMEM;
> >
> > + if (!dp->proc_fops)
> > + goto out_free;
> > + atomic_inc(&dp->pde_users);
> > + if (!dp->proc_fops)
> > + goto out_dec;
> > +
>
> You'll be shocked to know that I'd prefer more comments in there. Enough
> for a later maintainer to be able to understand what's going on.

Here is replacement patch with rewritten changelog and comments in
place. HTH.

[PATCH] Fix rmmod/read/write races in /proc entries

Current /proc creation interfaces suffer from at least two types of races:
---
1. Write via ->write_proc sleeps in copy_from_user(). Module disappears
   meanwhile.

   pde = create_proc_entry()
   if (!pde)
 return -ENOMEM;
   pde->write_proc = ...
    open
    write
    copy_from_user
   pde = create_proc_entry();
   if (!pde) {
 remove_proc_entry();

```
 return -ENOMEM;
 /* module unloaded */
   }
    *boom*
```

```
2. Read/write happens when PDE only partially initialized. ->data is NULL
   when create_proc_entry() returns. Almost all ->read_proc and
   ->write_proc handlers assume that ->data is valid.

   pde = create_proc_entry();
   if (pde) {
 /* which dereferences ->data */
 pde->write_proc = ...
    open
    write
 pde->data = ...
   }
```

The following plan is going to be executed (as per Al Viro's explanations):

PDE gets atomic counter counting reads and writes in progress done
via ->read_proc, ->write_proc, ->get_info . Generic proc code will bump
PDE's counter before calling into module-specific method and decrement
it after it returns.

remove_proc_entry() will wait until all readers and writers are done.
To do this reliably it will set ->proc_fops to NULL and generic proc
code won't call into module it it sees NULL ->proc_fops.

This patch implements part above. So far, no changes in proc users
required. Patch fixes races of type 1.

Unfortunately, fixing races of type #2 will require changing in some modules.

We need an indicator of PDE readinness of accepting reads and writes.
->proc_fops nicely fits. It is going to get new semantics:

* if ->proc_fops is valid, PDE will accept reads and writes via ->read_proc,
  ->write_proc, ->get_info.
* if ->proc_fops is NULL, PDE won't call into module's code.

remove_proc_entry() and only remove_proc_entry() will clear ->proc_fops.
create_proc_entry() will not set ->proc_fops. Helpers will be required.

Helpers will set ->proc_fops last (after ->data, particularly).

```
 set_proc_entry_data_fops(pde, data, fops);
 set_proc_entry_data_read_write(pde, data, read_proc, write_proc);
```

When all necessary helpers will be plugged, create_proc_entry will stop
setting default proc_fops and helpers will start setting it. Races of
type #2 will be fixed.

If module sets ->proc_fops only, or uses create_proc_read_entry(), or uses
create_proc_info_entry(), there won't be any changes for module.

```
 fs/proc/generic.c       |   61 ++++++++++++++++++++++++++++++++++++++++++++++---
 include/linux/proc_fs.h |   15 ++++++++++
 2 files changed, 73 insertions(+), 3 deletions(-)

--- a/fs/proc/generic.c
+++ b/fs/proc/generic.c
@@ -19,6 +19,7 @@ #include <linux/init.h>
 #include <linux/idr.h>
 #include <linux/namei.h>
 #include <linux/bitops.h>
+#include <linux/delay.h>
 #include <linux/spinlock.h>
 #include <asm/uaccess.h>

@@ -76,6 +77,25 @@ proc_file_read(struct file *file, char _
  if (!(page = (char*) __get_free_page(GFP_KERNEL)))
   return -ENOMEM;

+ if (!dp->proc_fops)
+  /*
+   * remove_proc_entry() marked PDE as "going away".
+   * No new readers allowed.
+   */
+  goto out_free;
+ /*
+  * We are going to call into module's code via ->get_info or
+  * ->read_proc. Bump refcount so that remove_proc_entry() will
+  * wait for read to complete.
+  */
+ atomic_inc(&dp->pde_users);
+ if (!dp->proc_fops)
+  /*
+   * While we're busy bumping refcount, remove_proc_entry()
+   * marked PDE as "going away". Obey.
+   */
+  goto out_dec;
```

```
+
  while ((nbytes > 0) && !eof) {
   count = min_t(size_t, PROC_BLOCK_SIZE, nbytes);

@@ -195,6 +215,9 @@ proc_file_read(struct file *file, char _
   buf += n;
   retval += n;
  }
+out_dec:
+ atomic_dec(&dp->pde_users);
+out_free:
  free_page((unsigned long) page);
  return retval;
 }
@@ -205,14 +228,33 @@ proc_file_write(struct file *file, const
 {
  struct inode *inode = file->f_path.dentry->d_inode;
  struct proc_dir_entry * dp;
+ ssize_t rv;

  dp = PDE(inode);

  if (!dp->write_proc)
   return -EIO;
+ /*
+  * remove_proc_entry() marked PDE as "going away".
+  * No new writers allowed.
+  */
+ if (!dp->proc_fops)
+  return -EIO;

- /* FIXME: does this routine need ppos?  probably... */
- return dp->write_proc(file, buffer, count, dp->data);
+ rv = -EIO;
+ /*
+  * We are going to call into module's code via ->write_proc .
+  * Bump refcount so that module won't dissapear while ->write_proc
+  * sleeps in copy_from_user(). remove_proc_entry() will wait for
+  * write to complete.
+  */
+ atomic_inc(&dp->pde_users);
+ if (dp->proc_fops)
+  /* PDE is ready, refcount bumped, call into module. */
+  /* FIXME: does this routine need ppos?  probably... */
+  rv = dp->write_proc(file, buffer, count, dp->data);
+ atomic_dec(&dp->pde_users);
+ return rv;
 }
```

```
@@ -717,12 +759,25 @@ void remove_proc_entry(const char *name,
  if (!parent && xlate_proc_name(name, &parent, &fn) != 0)
   goto out;
  len = strlen(fn);
-
+again:
  spin_lock(&proc_subdir_lock);
  for (p = &parent->subdir; *p; p=&(*p)->next ) {
   if (!proc_match(len, fn, *p))
    continue;
   de = *p;
+
+  /*
+   * Stop accepting new readers/writers. If you're dynamically
+   * allocating ->proc_fops, save a pointer somewhere.
+   */
+  de->proc_fops = NULL;
+  /* Wait until all readers/writers are done. */
+  if (atomic_read(&de->pde_users) > 0) {
+   spin_unlock(&proc_subdir_lock);
+   msleep(1);
+   goto again;
+  }
+
   *p = de->next;
   de->next = NULL;
   if (S_ISDIR(de->mode))
--- a/include/linux/proc_fs.h
+++ b/include/linux/proc_fs.h
@@ -56,6 +56,19 @@ struct proc_dir_entry {
  gid_t gid;
  loff_t size;
  struct inode_operations * proc_iops;
+ /*
+  * NULL ->proc_fops means "PDE is going away RSN" or
+  * "PDE is just created". In either case ->get_info, ->read_proc,
+  * ->write_proc won't be called because it's too late or too early,
+  * respectively.
+  *
+  * Valid ->proc_fops means "use this file_operations" or
+  * "->data is setup, it's safe to call ->read_proc, ->write_proc which
+  * dereference it".
+  *
+  * If you're allocating ->proc_fops dynamically, save a pointer
+  * somewhere.
+  */
```

```
  const struct file_operations * proc_fops;
  get_info_t *get_info;
  struct module *owner;
@@ -66,6 +79,8 @@ struct proc_dir_entry {
  atomic_t count;  /* use count */
  int deleted;  /* delete flag */
  void *set;
+ atomic_t pde_users; /* number of readers + number of writers via
+      * ->read_proc, ->write_proc, ->get_info */
 };

 struct kcore_list {
```