Subject: [PATCH 2.6.24-rc8-mm1 13/15] (RFC) IPC/semaphores: per <pid> semundo file in procfs
Posted by Pierre Peiffer on Tue, 29 Jan 2008 16:02:42 GMT
View Forum Message <> Reply to Message

From: Pierre Peiffer <pierre.peiffer@bull.net>

This patch adds a new procfs interface to display the per-process semundo
data.

A new per-PID file is added, named "semundo".
It contains one line per semaphore IPC where there is something to undo for
this process.
Then, each line contains the semid followed by each undo value
corresponding to each semaphores of the semaphores array.

This interface will be specially useful to allow a user to access
these data, for example for checkpointing a process

Signed-off-by: Pierre Peiffer <pierre.peiffer@bull.net>
Acked-by: Serge Hallyn <serue@us.ibm.com>

```
 fs/proc/base.c     |    3 +
 fs/proc/internal.h |    1
 ipc/sem.c          |  153 +++++++++++++++++++++++++++++++++++++++++++++++++++++
 3 files changed, 157 insertions(+)

Index: b/fs/proc/base.c
===================================================================
--- a/fs/proc/base.c
+++ b/fs/proc/base.c
@@ -2255,6 +2255,9 @@ static const struct pid_entry tgid_base_
 #ifdef CONFIG_TASK_IO_ACCOUNTING
  INF("io", S_IRUGO, pid_io_accounting),
 #endif
+#ifdef CONFIG_SYSVIPC
+ REG("semundo",   S_IRUGO, semundo),
+#endif
 };

 static int proc_tgid_base_readdir(struct file * filp,
Index: b/fs/proc/internal.h
===================================================================
--- a/fs/proc/internal.h
+++ b/fs/proc/internal.h
@@ -64,6 +64,7 @@ extern const struct file_operations proc
 extern const struct file_operations proc_smaps_operations;
```

```
 extern const struct file_operations proc_clear_refs_operations;
 extern const struct file_operations proc_pagemap_operations;
+extern const struct file_operations proc_semundo_operations;

 void free_proc_entry(struct proc_dir_entry *de);

Index: b/ipc/sem.c
===================================================================
--- a/ipc/sem.c
+++ b/ipc/sem.c
@@ -1435,4 +1435,157 @@ static int sysvipc_sem_proc_show(struct
      sma->sem_otime,
      sma->sem_ctime);
 }
+
+
+/* iterator */
+static void *semundo_start(struct seq_file *m, loff_t *ppos)
+{
+ struct sem_undo_list *undo_list = m->private;
+ struct sem_undo *undo;
+ loff_t pos = *ppos;
+
+ if (!undo_list)
+  return NULL;
+
+ if (pos < 0)
+  return NULL;
+
+ /* If undo_list is not NULL, it means that we've successfully grabbed
+  * a refcnt in semundo_open. That prevents the undo_list itself and the
+  * undo elements to be freed
+  */
+ spin_lock(&undo_list->lock);
+ undo = undo_list->proc_list;
+ while (undo) {
+  if ((undo->semid != -1) && !(pos--))
+   break;
+  undo = undo->proc_next;
+ }
+ spin_unlock(&undo_list->lock);
+
+ return undo;
+}
+
+static void *semundo_next(struct seq_file *m, void *v, loff_t *ppos)
+{
+ struct sem_undo *undo = v;
```

```
+ struct sem_undo_list *undo_list = m->private;
+
+ /*
+  * No need to protect against undo_list being NULL, if we are here,
+  * it can't be NULL.
+  * Moreover, by releasing the lock between each iteration, we allow the
+  * list to change between each iteration, but we only want to guarantee
+  * to have access to some valid data during the _show, not to have a
+  * full coherent view of the whole list.
+  */
+ spin_lock(&undo_list->lock);
+ do {
+  undo = undo->proc_next;
+ } while (undo && (undo->semid == -1));
+ ++*ppos;
+ spin_unlock(&undo_list->lock);
+
+ return undo;
+}
+
+static void semundo_stop(struct seq_file *m, void *v)
+{
+}
+
+static int semundo_show(struct seq_file *m, void *v)
+{
+ struct sem_undo_list *undo_list = m->private;
+ struct sem_undo *u = v;
+ int nsems, i;
+ struct sem_array *sma;
+
+ /*
+  * This semid has been deleted, ignore it.
+  * Even if we skipped all sem_undo belonging to deleted semid
+  * in semundo_next(), some more deletions may have happened.
+  */
+ if (u->semid == -1)
+  return 0;
+
+ seq_printf(m, "%10d", u->semid);
+
+ sma = sem_lock(undo_list->ns, u->semid);
+ if (IS_ERR(sma))
+  goto out;
+
+ nsems = sma->sem_nsems;
+ sem_unlock(sma);
+
```

```
+ for (i = 0; i < nsems; i++)
+  seq_printf(m, " %6d", u->semadj[i]);
+
+out:
+ seq_putc(m, '\n');
+ return 0;
+}
+
+static struct seq_operations semundo_op = {
+ .start = semundo_start,
+ .next = semundo_next,
+ .stop = semundo_stop,
+ .show = semundo_show
+};
+
+/*
+ * semundo_open: open operation for /proc/<PID>/semundo file
+ */
+static int semundo_open(struct inode *inode, struct file *file)
+{
+ struct task_struct *task;
+ struct sem_undo_list *undo_list = NULL;
+ int ret = 0;
+
+ /*
+  * We use RCU to be sure that the sem_undo_list will not be freed
+  * while we are accessing it. This may happen if the target task
+  * exits. Once we get a ref on it, we are ok.
+  */
+ rcu_read_lock();
+ task = get_pid_task(PROC_I(inode)->pid, PIDTYPE_PID);
+ if (task) {
+  undo_list = rcu_dereference(task->sysvsem.undo_list);
+  if (undo_list)
+   ret = !atomic_inc_not_zero(&undo_list->refcnt);
+  put_task_struct(task);
+ }
+ rcu_read_unlock();
+
+ if (!task || ret)
+  return -EINVAL;
+
+ ret = seq_open(file, &semundo_op);
+ if (!ret) {
+  struct seq_file *m = file->private_data;
+  m->private = undo_list;
+  return 0;
+ }
```

```
+
+ if (undo_list && atomic_dec_and_test(&undo_list->refcnt))
+  free_semundo_list(undo_list);
+ return ret;
+}
+
+static int semundo_release(struct inode *inode, struct file *file)
+{
+ struct seq_file *m = file->private_data;
+ struct sem_undo_list *undo_list = m->private;
+
+ if (undo_list && atomic_dec_and_test(&undo_list->refcnt))
+  free_semundo_list(undo_list);
+
+ return seq_release(inode, file);
+}
+
+const struct file_operations proc_semundo_operations = {
+ .open  = semundo_open,
+ .read  = seq_read,
+ .llseek  = seq_lseek,
+ .release = semundo_release,
+};
 #endif

--
Pierre Peiffer
```