Subject: Re: [PATCH 2.6.24-rc8-mm1 14/15] (RFC) IPC/semaphores: prepare semundo code to work on another task
Posted by serue on Wed, 30 Jan 2008 21:44:30 GMT
View Forum Message <> Reply to Message

Quoting pierre.peiffer@bull.net (pierre.peiffer@bull.net):
> From: Pierre Peiffer <pierre.peiffer@bull.net>
>
> In order to modify the semundo-list of a task from procfs, we must be able to
> work on any target task.
> But all the existing code playing with the semundo-list, currently works
> only on the 'current' task, and does not allow to specify any target task.
>
> This patch changes all these routines to allow them to work on a specified
> task, passed in parameter, instead of current.
>
> This is mainly a preparation for the semundo_write() operation, on the
> /proc/<pid>/semundo file, as provided in the next patch.
>
> Signed-off-by: Pierre Peiffer <pierre.peiffer@bull.net>
> ---
>
> ipc/sem.c | 90 ++----------
> 1 file changed, 68 insertions(+), 22 deletions(-)
>
> Index: b/ipc/sem.c
> ===
> --- a/ipc/sem.c
> +++ b/ipc/sem.c
> @@ -1017,8 +1017,9 @@ asmlinkage long sys_semctl (int semid, i
> }
>
> /* If the task doesn't already have a undo_list, then allocate one
> - * here. We guarantee there is only one thread using this undo list,
> - * and current is THE ONE
> + * here.
> + * The target task (tsk) is current in the general case, except when
> + * accessed from the procfs (ie when writting to /proc/<pid>/semundo)
>   *
>   * If this allocation and assignment succeeds, but later
>   * portions of this code fail, there is no need to free the sem_undo_list.
> @@ -1026,22 +1027,60 @@ asmlinkage long sys_semctl (int semid, i
>   * at exit time.
>   *
>   * This can block, so callers must hold no locks.
> + *
> + * Note: task_lock is used to synchronize 1. several possible concurrent
> + * creations and 2. the free of the undo_list (done when the task using it

> + * exits). In the second case, we check the PF_EXITING flag to not create
> + * an undo_list for a task which has exited.
> + * If there already is an undo_list for this task, there is no need
> + * to held the task-lock to retrieve it, as the pointer can not change
> + * afterwards.
> */
> -static inline int get_undo_list(struct sem_undo_list **undo_listp)
> +static inline int get_undo_list(struct task_struct *tsk,
> +   struct sem_undo_list **ulp)
> {
> - struct sem_undo_list *undo_list;
> + if (tsk->sysvsem.undo_list == NULL) {
> + struct sem_undo_list *undo_list;

Hmm, this is weird. If there was no undo_list and
tsk!=current, you set the refcnt to 2. But if there was an
undo list and tsk!=current, where do you inc the refcnt?

>
> - undo_list = current->sysvsem.undo_list;
> - if (!undo_list) {
> - undo_list = kzalloc(sizeof(*undo_list), GFP_KERNEL);
> + /* we must alloc a new one */
> + undo_list = kmalloc(sizeof(*undo_list), GFP_KERNEL);
>   if (undo_list == NULL)
>   return -ENOMEM;
> +
> + task_lock(tsk);
> +
> + /* check again if there is an undo_list for this task */
> + if (tsk->sysvsem.undo_list) {
> +   if (tsk != current)
> +   atomic_inc(&tsk->sysvsem.undo_list->refcnt);
> +   task_unlock(tsk);
> +   kfree(undo_list);
> +   goto out;
> + }
> +
>   spin_lock_init(&undo_list->lock);
> - atomic_set(&undo_list->refcnt, 1);
> - undo_list->ns = get_ipc_ns(current->nsproxy->ipc_ns);
> - current->sysvsem.undo_list = undo_list;
> + /*
> +   * If tsk is not current (meaning that current is creating
> +   * a semundo_list for a target task through procfs), and if
> +   * it's not being exited then refcnt must be 2: the target
> +   * task tsk + current.
> +   */

```
> +  if (tsk == current)
> +   atomic_set(&undo_list->refcnt, 1);
> +  else if (!(tsk->flags & PF_EXITING))
> +   atomic_set(&undo_list->refcnt, 2);
> +  else {
> +   task_unlock(tsk);
> +   kfree(undo_list);
> +   return -EINVAL;
> +  }
> +  undo_list->ns = get_ipc_ns(tsk->nsproxy->ipc_ns);
> +  undo_list->proc_list = NULL;
> +  tsk->sysvsem.undo_list = undo_list;
> +  task_unlock(tsk);
>    }
> - *undo_listp = undo_list;
> +out:
> + *ulp = tsk->sysvsem.undo_list;
>    return 0;
>  }
>
> @@ -1065,17 +1104,12 @@ static struct sem_undo *lookup_undo(stru
>    return un;
>  }
>
> -static struct sem_undo *find_undo(struct ipc_namespace *ns, int semid)
> +static struct sem_undo *find_undo(struct sem_undo_list *ulp, int semid)
>  {
>    struct sem_array *sma;
> - struct sem_undo_list *ulp;
>    struct sem_undo *un, *new;
> + struct ipc_namespace *ns;
>    int nsems;
> - int error;
> -
> - error = get_undo_list(&ulp);
> - if (error)
> -  return ERR_PTR(error);
>
>    spin_lock(&ulp->lock);
>    un = lookup_undo(ulp, semid);
> @@ -1083,6 +1117,8 @@ static struct sem_undo *find_undo(struct
>    if (likely(un!=NULL))
>     goto out;
>
> + ns = ulp->ns;
> +
>    /* no undo structure around - allocate one. */
>    sma = sem_lock_check(ns, semid);
```

>   if (IS_ERR(sma))
> @@ -1133,6 +1169,7 @@ asmlinkage long sys_semtimedop(int semid
>   struct sem_array *sma;
>   struct sembuf fast_sops[SEMOPM_FAST];
>   struct sembuf* sops = fast_sops, *sop;
> + struct sem_undo_list *ulp;
>   struct sem_undo *un;
>   int undos = 0, alter = 0, max;
>   struct sem_queue queue;
> @@ -1177,9 +1214,13 @@ asmlinkage long sys_semtimedop(int semid
>   alter = 1;
>   }
>
> + error = get_undo_list(current, &ulp);
> + if (error)
> + goto out_free;
> +
> retry_undos:
>   if (undos) {
> - un = find_undo(ns, semid);
> + un = find_undo(ulp, semid);
>   if (IS_ERR(un)) {
>   error = PTR_ERR(un);
>   goto out_free;
> @@ -1305,7 +1346,7 @@ int copy_semundo(unsigned long clone_fla
>   int error;
>
>   if (clone_flags & CLONE_SYSVSEM) {
> - error = get_undo_list(&undo_list);
> + error = get_undo_list(current, &undo_list);
>   if (error)
>   return error;
>   atomic_inc(&undo_list->refcnt);
> @@ -1405,10 +1446,15 @@ next_entry:
>   kfree(undo_list);
> }
>
> -/* called from do_exit() */
> +/* exit_sem: called from do_exit()
> + * task_lock is used to synchronize with get_undo_list()

Ok I had to think about this again. I'd like the comment
here to point out that the task_lock here acts as a barrier
between the prior setting of PF_EXITING and the undo_list
being freed here, so that get_undo_list() will either see
PF_EXITING is NOT in the tsk->flags, in which case it will
insert the undo_list before the task_lock() is grabbed here,
and with count=2, so that it gets correctly put here in

exit_sem, or it will see PF_EXITING set and cancel the
undo_list it was creating.

> + */
> void exit_sem(struct task_struct *tsk)
> {
> - struct sem_undo_list *ul = tsk->sysvsem.undo_list;
> + struct sem_undo_list *ul;
> + task_lock(tsk);
> + ul = tsk->sysvsem.undo_list;
> + task_unlock(tsk);
>   if (ul) {
>   rcu_assign_pointer(tsk->sysvsem.undo_list, NULL);
>   synchronize_rcu();
>
> --
> Pierre Peiffer
> _____
> Containers mailing list
> Containers@lists.linux-foundation.org
> https://lists.linux-foundation.org/mailman/listinfo/containers

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers