File management in the Linux kernel

This document describes how locking for files (struct file) and file descriptor table (struct files) works.

Up until 2.6.12, the file descriptor table has been protected with a lock (files->file_lock) and reference count (files->count). ->file_lock protected accesses to all the file related fields of the table. ->count was used for sharing the file descriptor table between tasks cloned with CLONE_FILES flag. Typically this would be the case for posix threads. As with the common refcounting model in the kernel, the last task doing a put_files_struct() frees the file descriptor (fd) table. The files (struct file) themselves are protected using reference count (->f_count).

In the new lock-free model of file descriptor management, the reference counting is similar, but the locking is based on RCU. The file descriptor table contains multiple elements - the fd sets (open_fds and close_on_exec, the array of file pointers, the sizes of the sets and the array etc.). In order for the updates to appear atomic to a lock-free reader, all the elements of the file descriptor table are in a separate structure - struct fdtable. files_struct contains a pointer to struct fdtable through which the actual fd table is accessed. Initially the fdtable is embedded in files_struct itself. On a subsequent expansion of fdtable, a new fdtable structure is allocated and files->fdtab points to the new structure. The fdtable structure is freed with RCU and lock-free readers either see the old fdtable or the new fdtable making the update appear atomic. Here are the locking rules for the fdtable structure -

1. All references to the fdtable must be done through the files_fdtable() macro:

   ```
   struct fdtable *fdt;

   rcu_read_lock();

   fdt = files_fdtable(files);
   ....
   if (n <= fdt->max_fds)
           ....
   ...
   rcu_read_unlock();
   ```

   files_fdtable() uses rcu_dereference() macro which takes care of the memory barrier requirements for lock-free dereference. The fdtable pointer must be read within the read-side critical section.

2. Reading of the fdtable as described above must be protected by rcu_read_lock()/rcu_read_unlock().
3. For any update to the fd table, files->file_lock must be held.
4. To look up the file structure given an fd, a reader must use either lookup_fd_rcu() or files_lookup_fd_rcu() APIs. These take care of barrier requirements due to lock-free lookup.

   An example:

   ```
   struct file *file;

   rcu_read_lock();
   file = lookup_fd_rcu(fd);
   if (file) {
           ...
   }
   ....
   rcu_read_unlock();
   ```

5. Handling of the file structures is special. Since the look-up of the fd (fget()/fget_light()) are lock-free, it is possible that look-up may race with the last put() operation on the file structure. This is avoided using atomic_long_inc_not_zero() on ->f_count:

   ```
   rcu_read_lock();
   file = files_lookup_fd_rcu(files, fd);
   if (file) {
           if (atomic_long_inc_not_zero(&file->f_count))
                   *fput_needed = 1;
           else
           /* Didn't get the reference, someone's freed */
                   file = NULL;
   }
   rcu_read_unlock();
   ....
   return file;
   ```

   atomic_long_inc_not_zero() detects if refcounts is already zero or goes to zero during increment. If it does, we fail fget()/fget_light().

6. Since both fdtable and file structures can be looked up lock-free, they must be installed using rcu_assign_pointer() API. If they are looked up lock-free, rcu_dereference() must be used. However it is advisable to use files_fdtable() and lookup_fd_rcu()/files_lookup_fd_rcu() which take care of these issues.

7. While updating, the fdtable pointer must be looked up while holding files->file_lock. If ->file_lock is dropped, then another thread expand the files thereby creating a new fdtable and making the earlier fdtable pointer stale.

   For example:

   ```
   spin_lock(&files->file_lock);
   fd = locate_fd(files, file, start);
   if (fd >= 0) {
           /* locate_fd() may have expanded fdtable, load the ptr */
           fdt = files_fdtable(files);
           __set_open_fd(fd, fdt);
           __clear_close_on_exec(fd, fdt);
           spin_unlock(&files->file_lock);
   .....
   ```

   Since locate_fd() can drop ->file_lock (and reacquire ->file_lock), the fdtable pointer (fdt) must be loaded after locate_fd().