Insider Tips: Locks

Competitive Thinking

A genuine multitasking system like Linux runs many processes concurrently. Programs must compete for data access. Assigning locks to files ensures exclusive access and prevents the possibility of data.

BY MARC ANDRÉ SELIG

BMW AG

Most Linux machines have an MTA, a Mail Transfer Agent. This can be Postfix, Exim or even Sendmail. The MTA either uses a Fetchmail process or TCP to fetch email messages. When the MTA finds a message addressed to the local user, it passes the message on to the Local Delivery Agent (LDA). And the LDA stores the message in a mailbox file, after possibly taking a detour via a filter such as Procmail.

If two messages arrive at the same time, the MTA will hand both of them to an LDA process at the same (see Figure 1). Each process will then attempt to write to the same mailbox file, again at the same time. If you are lucky, the messages end up in the right file, but in the wrong order, but you are far more likely to lose a file as the processes overwrite each other's data.

Organized Access with Locks

Locks are the typical answer to this issue. A lock denies access to a resource while a process is using it. If you attempt to access a locked file, you might get an error message, or the file opening function might just wait for a while. The function will not return to the main program loop until the current process has released the file, allowing the next user to open it.

Thus, locks can solve the problem of email messages that arrive at the same time. The LDA with the first message simply locks the mailbox. The LDA with the second message gets an error when it attempts to open the mailbox, waits for a while, and tries again later. When the first LDA has written the message, closed the mailbox and removed the lock, there is nothing to stop the second message arriving safely.

Unfortunately, file locks also actually create a few problems. One of the most minor problems is that they can change the order in which things happen. The following scenario explains what can happen: imagine that the LDA is just delivering a message and has locked the mailbox; a second process is waiting for the file to be released as it needs to deliver a second message.

What happens if a third message just happens to arrive at exactly the same time the first LDA releases the mailbox. The third message again locks the file, and the second LDA has to wait: a classic example of a race condition.

Listing 1: File Locking with Perl

```
#!/usr/bin/perl -wT
use Fcntl ':flock';
sysopen(FH, "File", O_RDWR|O_CREAT) or die
 "Error sysopen: $!";
flock(FH, LOCK_EX) or die "Error flock: $!";

# ... Write file ...

flock(FH, LOCK_UN) or die "Error flock: $!";
close FH or die "Error close: $!";
```

Stale

Stale locks are a far more serious problem. Stale locks occur when a program fails to release files because it has crashed, or a user has *killed* the program.

This would mean other processes would have to wait forever for the file to be released.

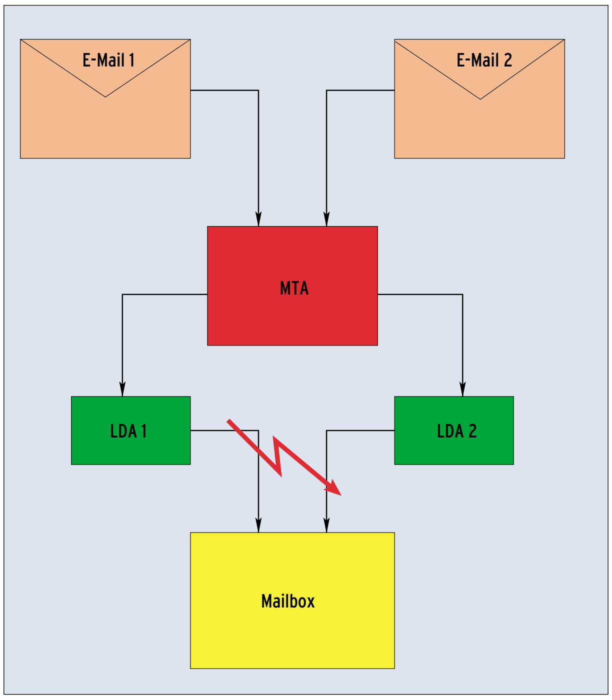

Figure 1: Without locking two email messages arriving at the same time would overwrite each other, as one LDA process is unaware of what the other is doing.

Listing 3: File Locking Primitive in the Shell

```
#!/bin/sh
# lockfile is part of the procmail distribution and
# available on most Linux distributions
lockfile file.lock
# ...
rm -f file.lock
```

Email messages cannot be delivered, and users cannot edit the mailbox. Stale locks are particularly dangerous on NFS servers; they can even cause the server to freeze.

When a program uses locks, it should check to see if there is an active process assigned to the locked file. Some locks disappear automatically when the accompanying process terminates. Other lockfiles use the process ID in their names, or as part of their contents.

Lock Variants

There are many different approaches to implementing file locking. The two basic categories are mandatory and advisory locks, and the kernel has system calls for both variants.

Processes cannot ignore mandatory locks. The kernel takes care of denying access to the locked resource until the lock is removed. This is basically a safe method, although it is susceptible to stale locks and fairly useless for network file systems such as NFS and AFS.

An advisory or discretionary lock is simply a note to the effect that a file is locked, although there is no saying whether programs will honor the lock. It always makes sense to be careful with locked files.

Soft locks of this kind can be implemented by the kernel and by user space libraries. It is also possible to use simple files to implement advisory locks. This method also works with NFS, assuming the program provides a more or less atomic file opening function.

The bad news is; if you are looking to write a program for use with any flavor of Unix, mandatory locks are not a good idea. On the upside, programs for use with any version of Linux are a lot simpler, since Linux has the POSIX-compatible *fcntl* function (which is identical to *lockf* on Linux) and the BSD-compatible *flock*. Libraries often facilitate the use of these monsters. Listing 1 shows an example in Perl.

Lockfiles

Primitive semaphores are simpler than the kernel locks we have looked at so far. Literally speaking, a semaphore is a visible signal used for ship to ship communication in days gone by. Under Unix, a semaphore is a file or data structure that indicates a state – the state of a resource, for example.

Programs typically indicate a locked mailbox by appending a *.lock* extension to the filename (for example, the mailbox file */var/mail/mas* would lead to a lockfile called */var/mail/mas.lock*). This technique even works on NFS, although this assumes that the program modifying the mailbox is permitted to create a new file in */var/mail/*.

Programs need to check for existing lockfiles, as mentioned previously. Lockfile checking and creation needs to be a single operation to avoid a race condition. The following pseudo-code shows you what a race condition looks like:

```
IF: not exists file.lock
  THEN: create file.lock
  ELSE: wait
```

If two programs run this code at exactly the same time, both enter the *THEN* branch of the condition and create the lock file. This undermines the whole principle of locking, as both programs will again access the mailbox at the same time. The following approach makes more sense:

```
IF: create exclusive file.lock
  THEN: do something
  ELSE: wait
```

This checks for and creates the lockfile in a single atomic operation. Even if two programs run this code at the same time, this will not cause a race condition because one of the programs will create the file before the other.

Listing 2 shows an implementation of the previous pseudo-code in Perl. The shell script in Listing 3 uses the *lockfile* program that is part of Procmail.

A word of warning: for networked filesystems, lockfiles are often unreliable. Don't depend on locks performing as expected over NFS exports. ■

Listing 2: File Locking in Perl

```
#!/usr/bin/perl -w
use Fcntl;
sysopen(FH, "file.lock", O_WRONLY|O_EXCL|O_CREAT)
  or die "Error sysopen: $!");
```

INFO

[1] Marc André Selig: "Official Calls", Linux Magazine #43 / June 2004, p62.