Insider Tips: Inter-Process Communication Beyond Simple Signaling

Official Calls

Unix-style operating Systems clearly distinguish between processes, while at the same time allowing for freedom of communication. While troubleshooting a system, an admin needs a clear view of the scenario to help the Linux process reactivate the interrupted dialog.

BY MARC ANDRÉ SELIG

When processes need to talk, signals are the easiest way of doing so [1]. No matter how effective this technology may be, it still has a number of drawbacks. Only a few signals exist, and this restricts the possible content of the messages involved. Also, users – with the exception of root – are only permitted to signal their own processes. They are not allowed to communicate with foreign processes.

There are several other channels that allow processes to talk: from pipes, named pipes and sockets, through lock files and file locks, to System V interprocess communications and shared memory.

Pipes

Standard input and output channels are assigned to each process on a Unix-style operating system. The process expects to receive data via the standard input channel. When a user composes an email message in the command line, the client, *mailx*, will use standard input to read that message. In a similar way, a process will output its results to the standard output channel, unless instructed otherwise. This produces text output in a terminal window. Also, each process has a channel for standard error output, and uses this channel to indicate error states.

A pipe simply attaches the standard output from one process to the standard input of another, allowing one process to read what another process has written. This efficient and direct communication method assumes that both programs are running at the same time.

Searching a logfile for specific strings provides a good example of how to use a pipe. The following command line uses *grep* to find all the *GET* entries in an Apache logfile. It then uses *cut* to separate the first field (containing the client's IP address), and sorts the results numerically before going on to call *uniq* to avoid outputting each hostname more than once.

```
grep GET access_log | cut -d ➥
" " -f 1 | sort -n | uniq > ➥
hosts.txt
```

The pipe only affects the standard input and output. It does not redirect the standard error channel. If one of the programs were to output an error message or alert, the message would not be sent to *hosts.txt*. Instead, it would display directly in the shell. If you need to redirect standard error output, you have to do this explicitly:

```
find /etc -type f 2>/dev/null ➥
-print0 | xargs -0 grep -i ➥
imap 2>&1 | less
```

Unix-style operating systems assign a consecutive number to each open file. Standard input is 0, standard output 1, and standard error 2. The normal redirect *>hosts.txt* refers to the file descriptor 1, that is, standard output, whereas *2>/dev/null* redirects descriptor 2, the standard error channel. Note that modern Linux operating systems provide pseudo files for these streams: */dev/fd/0*, */dev/fd/1*, and */dev/fd/2*.

The statement we just looked at first finds regular files in */etc*. If error mes-

sages occur, they are sent to */dev/null*, that is, a black hole. The statement then searches for the *imap* string in the files it found. The search is not case-sensitive. *2>&1* copies any errors that occur to standard output, and redirects both outputs to less. Incidentally, the combination of *find … -print0*, and *xargs -0* ensures that this chain of commands can handle filenames that include blanks.

Pipes are simple and efficient. However, their use is limited. All of the processes involved need to be launched simultaneously, by the same user, on the same computer.

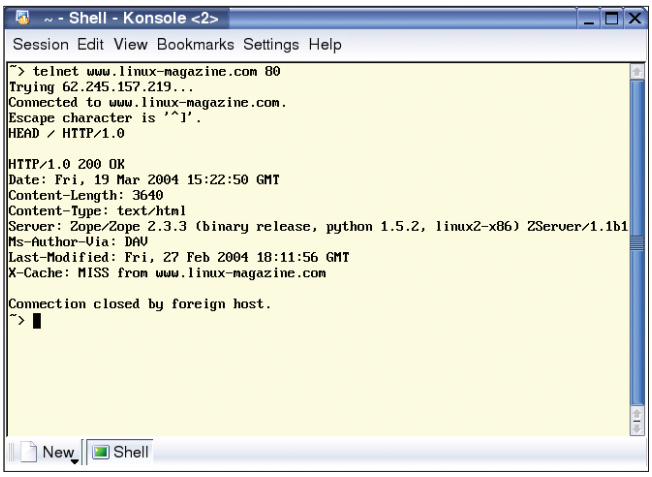

Figure 1: Admins can use a telnet command to talk to their servers directly. Typing *HEAD / HTTP/1.0*, followed by an empty line, tells this Apache server to reveal some information on the Linux Magazine homepage.

Named Pipes

So-called named pipes are a (FIFO, first in first out) variant of the pipe theme. Instead of linking two programs directly, they use special files for reading and writing. *mkfifo* is used to create the files. The advantage is that users do not need to launch the programs at the same time. The processes do no even need to belong to the same user. A file created by *mkfifo /tmp/mas/test* looks like this, when we type *ls -l /tmp/mas/test*:

```
prw-------  1 mas  users  0 Mar
 7 13:17  /tmp/mas/test
```

The read and write privileges have the same meanings as for regular files. The named pipe reacts more or less like a normal file. You can redirect the output of an arbitrary command to the named pipe:

```
ls /etc/mail/spamassassin
>/tmp/mas/test
```

However, the *ls* command will freeze in this case, as it will wait for the output to be read, and this is not happening at present. *cat /tmp/mas/test* in another terminal takes care of this, and leads to the following output:

```
local.cf
no-osiru.cf
```

Named pipes provide an extremely practical approach to passing messages to the user working at the X11 console. Displaying the messages directly on the screen background would distract the user – besides which, XFree does not support asynchronous text output outside of windows.

It is common to create a FIFO construct called */dev/xconsole*, and configure the system (*/etc/syslog.conf*) to send messages to that location:

```
# Copy messages to the X console
*.*  |/dev/xconsole
```

Admins, or the current user, can simply type *cat /dev/xconsole* to view messages.

Listing 1: A simple TCP server in Perl

```
01 #!/usr/bin/perl -w
02 use strict;
03 use IO::Socket;
04
05 my $socket = IO::Socket::INET-
   >new(
06   Listen => 5,
07   Proto => "tcp",
08   LocalPort => 2345,
09   ReuseAddr => 1,
10 )
11 or die "Problem: $!";
12
13 while (my $client = $socket-
   >accept) {
14   my $line = <$client>;
15   print "Connection from " .
   $client->peerhost . ": $line";
16   print $client "Demo\r\n";
17 }
```

Sockets

In comparison to named pipes, sockets add another level of abstraction to the communication between two processes. The processes do not even need to be running on the same computer in this case. Sockets provide an interface between various communication protocols, especially between the typical Internet protocols, TCP (Transmission Control Protocol), and UDP (User Datagram Protocol). Other variants, such as Unix domain sockets only work locally, that is, within a single system.

Sockets are the typical communication method on the Internet. Users surfing the Web, exchanging email messages, or establishing terminal connections all use sockets.

Most socket-based applications follow the client-server paradigm. Telnet or Mozilla are examples of clients, inetd or Apache (see Figure 1), the appropriate servers. Listing 1 provides a very minimalistic example of a TCP-based server. In real-life applications, processes do not typically handle the data which is passed to them directly. Instead the server will typically hand the information to a child process, allowing the server to get on with its job, such as waiting for the next connection.

Sockets exist in many variants and with many options. The distinction between Unix domain and Internet domain sockets is an important one. Internet domain sockets are used for global communication. Unix domain sockets are far quicker, and although restricted to the local computer, cause less overhead.

For example, MySQL will automatically use Unix domain sockets, if the server is running on the same computer as the client. Typing *ls -l /var/lib/mysql/mysql.sock* displays the following:

```
srwxrwxrwx 1 mysql mysql 0 Feb
13 14:24  /var/lib/mysql/
mysql.sock
```

Lockfiles

The inter-process communication variants discussed thus far have become

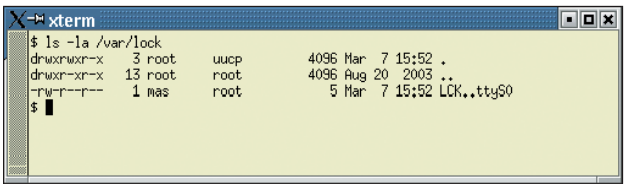

Figure 2: Unix-style operating systems have a directory called */var/lock* for lockfiles. The convention is to allow the *uucp* group write access.

more abstract as we went on. An Internet domain socket can send data more or less anywhere. Real-life applications often require solutions that provide something complex than simple signaling, but with a smaller footprint, and far less overhead than a convoluted socket.

Lockfiles or traditional semaphores can provide these techniques. They indicate that a specific resource is currently in use. For example, a program that uses the serial port can store its process ID in a special file, and thus indicate that it has reserved the port for its own use. Locks of this type are typically found in */var/lock*.

As Figure 2 above shows, the *uucp* group traditionally has write permissions for */var/lock*. The UUCP (Unix to Unix copy) program provides asynchronous file transfers between machines and was commonly used for copying files from one machine to another across modem connections.

Users who need access to semaphore controlled resources are typically added to the *uucp* group by the root user, allowing them to create lockfiles as required. Our example shows a file called *LCK..ttyS0*. The file indicates that the user *mas* needs exclusive access to the */dev/ttyS0* device.

Other processes can check whether this information is still applicable, by reading the lockfile. The file contains the PID of the calling process. If the process still exists, you can assume that the resource is still in use. If the process whose ID is stored in the lockfile has terminated, a new process can remove the existing lockfile and create a new one.

Malevolent processes could remove the lockfile immediately, and an ignorant process will not bother checking. This is an important aspect of many locking techniques. Locks are often advisory. Friendly programs will take this advice, but the system will not prevent anyone with appropriate privileges from accessing the resources, regardless of the lock status.

File Locks

If the locked resource is a normal file on a local filesystem, a modern Unix-style operating system can do without the lockfile. Instead, the process can simply lock the file. The process can additionally decide whether to allow any other processes read access, or whether it is necessary to lock down access to the file completely.

File locks of this type are often seen in the context of email systems, for the mail spools in */var/mail*, or */var/spool/mail*. Careful use of locks is particularly important here, as email is typically handled asynchronously. When a user deletes a message, that user's client rewrites the mail spool file, and removes the messages. If a new message were to arrive at the same time, the results could be dramatic, if not for locking. If you were lucky, the deleted message might still exist. In the worst case, the new message would end up somewhere in the middle of the spool file and arbitrarily delete parts of other messages.

Listing 2: Semaphore-based mail lock

```
01 #!/bin/sh
02 lockfile -ml
03 echo I can now delete
   messages.
04 lockfile -mu
```

Listing 3: Mail lock with *flock()*

```
01 #!/usr/bin/perl
02 use Fcntl ':flock';
03 open (MAILBOX,
   ">>/var/mail/$ENV{'USER'}")
04     or die "Cannot write to
   mailbox: $!";
05 flock(MAILBOX, LOCK_EX);
06 print "I can now delete
   messages.\n";
07 flock(MAILBOX, LOCK_UN);
08 close MAILBOX;
```

The convention is to have the Mail User Agent (mail client) lock the mail spool before it starts performing changes of this kind. The Mail Delivery Agent will respects existing locks, and not deliver mail without locking the spool file first.

There are a number of locking variants for email handling. Listings 2 and 3 show the important alternatives, which are often used parallel to one another. A semaphore, just like the ones discussed earlier (*/var/lock*), can be written to */var/mail* and has the advantage of working perfectly well on distributed filesystems such as NFS or AFS. In contrast, file locks (*flock()* or *fcntl()*) typically fail, or provide unreliable results, across networked filesystems. If they work, file locks have the advantage of being quicker and more efficient than lock files, and the processes involved do not require write access to the mail directory. ■

System V IPC

Modern System V Unix-type systems, such as Linux, have a wide range of IPC (Inter-Process Communication) facilities in addition to the ones discussed so far. The *ipc()* library function provides semaphores, and message queues. Many mechanisms produce similar results, but are anchored in the BSD tradition.

Shared memory is another interesting technology, that allows two processes to share a common memory area. This allows for simple and efficient data exchanges.

INFO

[1] Marc André Selig, "Admin Workshop: Signal-Based Inter-Process Communication": Linux Magazine, Issue 42, May 2004

THE AUTHOR

Marc André Selig spends half of his time working as a scientific assistant at the University of Trier and as an ongoing medical doctor in the Schramberg hospital. If he happens to find time for it, his currenty preoccupation is programing web-based databases on various Unix platforms.