by Peter Collinson, **Hillside Systems**

Logs

UNIX supports a number of background processes, generally known as *daemons*, that tend to just sit there on the machine doing some task or other. Occasionally, and sometimes more than occasionally, some of these processes need to communicate with a human. We need to provide a mechanism where the daemons can complain that something is not functioning correctly and have a human fix the problem.

In the early days of computing, where batch systems prevailed, a log of all activity was written on the system console, and the system designer supposed that someone would be sitting there waiting to tend to the machine's every need. In the main, daemons on early UNIX systems also wrote to the console when they discovered a problem. Actually, the daemons just printed on their standard output channel. When launched at system start-up, the message was written to the console. If they were launched from some other terminal, then the message would be displayed there, annoying the user.

Scribbling on the console was often unsatisfactory. Sometimes the console was hidden in a machine room while users sat outside in noisy terminal rooms talking to the machine from screens over RS232 serial connections. No one was watching for problems on the console. Sometimes the console was also the system's printer and could be the machine's only hard-copy device. Users didn't want system messages cluttering up their nicely formatted output.

It was always easy to create files on UNIX, and so programmers started to write messages to log files. Initially, this was done by using file redirection when the command was started:

```
command > log&
```

However, increasingly programmers of daemons started to include code to create logs explicitly. Once the program designer knew the program was going to create a file, then it became possible to add a new class of informational messages saying "I've just done this" to the logging code. Suddenly you could discover when and what the daemons were doing. Administrators now had the ability to use the standard UNIX tool set to examine logs and confirm that some mail had been sent, that a dump process had saved some specific data and that the `news` system had transmitted a specific article.

Writing Files

There were technical problems, however, with creating logs that stemmed from the multiprocessing nature of UNIX. To understand the problems, you need to know a little about how a UNIX process writes files. A UNIX process that wants to write data to a file will first open it using the `open` system call. It's normally the case that the program wants to make a brand new file, so the `open` system call takes a flag that says "create this file if it doesn't exist or truncate it to zero length if it does."

In the original UNIX systems, there was a special `creat` system call that made a new file. When Ken Thompson is asked, "what would you have done differently when you designed the UNIX file system?" he usually says, "I would have added an 'e' to the 'creat' system call." The `creat` system call is not really used in modern UNIX systems; its action has been subsumed into the `open` system call by augmenting the mode parameter.

However, there are circumstances where we want to write to a pre-existing file, and in these cases, the programmer doesn't supply the "create" flag. Actually, opening a log file is probably the most common occurrence of this situation.

When `open` is called without the `create` flag, then the file must exist or the system call will fail. The call will also fail unless the user that's running the process is allowed to write to the file. Assuming all is well, the `open` system call will return a value called the "file descriptor."

The file descriptor is used in subsequent system calls to refer to the open file. When the program wants to write to the file, it loads a section of memory (a *buffer*) with the data to be written and uses the `write` system call to make the kernel write the data. The `write` system call is passed the number of bytes to be written, the buffer address and the file descriptor.

Notice that there is no positioning information in the `write` system call. For example, there's no way of saying add this data to the end of the file. In the normal case, when just dumping some data to a new file, the programmer will want to use several `write` calls to output the data, and it is assumed that each `write` call will add data to the file just after any data that was written previously. When the user closes the file, using the `close` system call on the file descriptor, there's then an assumption that the data in the file will appear in the same order that it was written.

These two assumptions seem obvious, but there are occasions where we don't want to obey them. A general mechanism is needed to allow us to deal with the need for random read and write access into files. What happens is that the kernel maintains a pointer into the file. The pointer is associated with the file descriptor and is used to give the starting position for the next read or write system call. Whenever a read or a write call is executed, the pointer is incremented by the number of bytes that have been moved between the kernel and the user process.

In the usual case of writing sequential data to the file, we now have the following sequence of events. First, the program opens the file. The pointer is set to the first byte. The program executes a `write` system call and data will be moved into the kernel's buffers, and possibly written to the file. The pointer is moved to the end of the data. Subsequent `write` calls will add new data after each previous piece of information, each time moving the pointer so that the data will be appended to the file.

OK, so what happens when we want to open a file and write data at the end? We need some way to position the pointer that the kernel maintains. The `lseek` system call is used to set the pointer to an offset in the file so that a subsequent `read` or `write` system call will operate from that pointer position. The arguments to the `seek` call are the file descriptor, an offset and a value that tells the system how to apply the offset: from the start of the file, from the end of the file or relative to the current position.

Because the offset can be a negative value, the maximum size of the offset in the original UNIX systems was 2^{15}. UNIX Version 7 changed the name of the `seek` call to `lseek` (long seek) and changed the offset value to 2^{31}. More recent systems have altered the size to 2^{63}. The change from 32 to 64 bits was effected without having to use the name `llseek`.

Problems with Multiprocessing

Armed with the `lseek` call we can now write an entry to a log file. We `open` the file, `lseek` to the end, `write` the data, `close` the file. Each of the words in courier in the previous sentence map directly onto a system call.

This works fine until more than one process attempts to write to the log file at the same time. The file descriptor is "per-process," (actually it's a little more complex than that, but let's ignore that complexity), so the pointer to the file can be set to point at different places in the file by several processes. Normally this is fine and we want exactly this behavior. We want processes to be able to read the same file and have position pointers set differently.

However, consider two processes attempting to write entries at the same time. First, they both use the open system call and obtain file descriptors with associated pointers that point to the start of the file. Actually on a single processor system, one process will execute the system call before the other, but let's assume the processes are moving in step. Then, they both seek to the end of the file, moving their own copy of the file pointer to the same place. Again, on a single processor system, one of the seeks will happen before the other. Now, they both execute the write call. One will get in first and write data, but the second call will overwrite that data because it's been told to write at a specific position. Now they both close the file and go away feeling happy.

The result is not so good. The log entry for one of the processes has been overwritten by data from the second. The log is not correct, and worse, the formatting in the file may be corrupt. The result is worse if the processes write part of a line in one write system call, followed by a second call to write the remainder of the same line. In this case, you may get parts of one log entry appearing with parts of another.

In early UNIX systems, it was hard to prevent this confusion from happening. The programmer can ensure that all the data for one line is written in one system call, which will help. But there is no good way of ensuring that the data is not completely overwritten. There was a need to lock the file, so that one process can know that when it performs the lseek call, then the file will not be altered until the process has successfully written its entry. It took some years before "official" file locking emerged, although many people created their own "home grown" solutions.

The BSD releases added a new solution. A mode was added to the open call that told the kernel whenever a write system call was made on the file, the data was to be added to the "real" end of the file. Assuming that the programmer arranged to write the data in a single system call, then the data would always be appended to the end of the file even if it had grown (or shrunk) between the time the file was opened and the write call was executed. Writing logs became considerably simpler.

Syslog

When logs started to emerge as a standard way of logging events, UNIX administrators had a huge headache. Programs would write files in places that the original designer had found convenient on their system, but usually in a place that was deeply inconvenient on the system run by the administrator. Unless the logs were tended, they could grow to immense sizes which could be embarrassing. I used to refer to log tending as "gardening:" it became a daily chore akin to pruning and general weeding. I obviously expressed my horror of this activity somewhat too much. My staff at that time commented by placing a small set of indoor gardening tools on top of the machine.

Random log placement and lack of gardening tools still goes on to some extent. Solstice Backup takes it upon itself to store log copies in its own tree and provides no way to stop them from growing. Actually, most of the standard system logs now come with an automatic pruning tool that restrains them from growing endlessly.

Eric Allman, the author of `sendmail`, (now CTO at Sendmail Inc., then at the University of California, Berkeley) began to examine the issue of maintaining logs because he wanted to make `sendmail` generate them. He had encountered the problems I've described above and wanted a logging system for `sendmail` that allowed a single process to write the log files, while getting around the problems with the `seek` and `write` calls. He also wanted to allow the system administrator to be able to configure where the logs are written. Much later on, a new need emerged. Having the log files spread across many machines was a pain. If the log entry could be translated into a network message, then it could be sent to some central log server that we know today as `syslogd`.

There are likely to be many clients of the logging server so some method is needed to identify the program that sent the message. Each message is tagged with a pair of numbers: a faci-lity and a level. The facility is used to separate the messages depending on the program that sent them. There are a small number of possible values. The `user` facility acts as a catchall for anything not fitting into the general scheme. Messages from the kernel are tagged with `kern`. There are several facilities used by groups of related programs: `daemon` is used by all system daemons, e.g., `ftpd` or `xntpd`; `cron` is used by the various programs making time managed actions, like `cron` or `at`; and `auth` by authorization programs, such as `login` or `su`.

There are also facilities used by single subsystems: `news` used by the news system; `mail` by the mail system; and `uucp` is reserved for UUCP. Finally, there are eight facility codes reserved for local use known as `local0`, `local1`, etc.

The names represent numbers that are compiled into `syslogd` and every program that uses the client interface. The numbers are cast in stone. I do wonder in retrospect whether the numeric coding was a good idea. Had the facilities been strings, then perhaps we would have seen them evolve more over time. I am interested in logging FTP in detail on my machine, for example, but don't care too much about `xntpd`. Both share the same facility name. OK, I can do things with `grep` on the log file to find the data that I need, but that's after the event. Eric shares this view, but it is too late to go back now.

For each facility code, there are several possible logging levels. In order of severity, these are: `emerg`, for panic conditions that will normally be broadcast to all users; `alert`, for conditions that should be corrected immediately; `crit`, for critical conditions such as hard device errors; `err`, for other errors; `warning`, for warning messages; `notice`, for bringing the user's attention to something; `info`, for information messages; and `debug`, for debugging messages.

Again, these are fixed numeric values. The severity levels provide a way for `syslogd` to choose to store or ignore messages. Clients are written to always send logging messages to the daemon. The daemon is then instructed to ignore messages over a certain level (`emerg` has zero value and `debug` is seven). This allows you to "turn debugging off" in `syslogd`.

Debugging messages are always being logged, but usually messages of this priority are ignored.

The real problem with this scheme is that it's hard to know what levels of message any particular program generates. Sadly, the manual pages for client daemons that use `syslogd` don't often document this information–and they should. Sun's manual pages are beginning to show this information. Also, in recent Solaris systems, Sun is logging the facility/severity pair that caused a message to appear in one of the log files. This only seems to work for programs running on the machine and gets confused when it's sent a message from my Cisco router.

Controlling syslogd

The daemon is controlled by the file `syslog.conf` that you will find lurking in `/etc`. Getting the format of the file right can sometimes be an arcane art and the logger program can help here. This program is used to send messages from the command line to the daemon. Consult your manual page.

The file consists of two columns separated by tabs. Note that the separator must be tabs and not spaces. The first column is a selector and the second a destination. The selector is matched with the facility/severity pair of the inbound message and the message is sent to the destination if the match succeeds. If any selector in the file does not match the message, it is simply junked. Multiple destinations are possible for the same message. So the lines:

```
kern.debug          /var/adm/messages
kern.debug          /dev/console
```

will write all messages from the kernel onto `/var/adm/messages` and also onto the console. By setting the level qualifier to lower valued severity levels, you can block out messages from the higher valued ones. For example, using `notice` (value 5) will cause `syslogd` to ignore `info` (6) and `debug` (7) messages that emanate from the kernel. (For a quick reminder of the numbers, see `/usr/include/sys/syslog.h`) Any target file must exist when syslogd is started, so a simple way of turning off logging is to simply remove the file being used for storage.

You can specify wild cards:

```
*.err          /var/adm/messages
```

will write all messages from all facilities that are of severity `err`, `crit`, `alert` or `emerg` onto the file. You can group messages for one destination by using a semi-colon:

```
*.err;kern.debug             /var/adm/messages
```

There's a special selector that can be used to inhibit messages from a particular source. So if the selector above reads:

```
*.err;kern.debug;user.none
```

it will inhibit any user messages from being written to the file.

You can group severity levels too:

```
daemon,auth.notice    /var/adm/messages
```

sends all messages below, including notice for the daemon and `auth` facilities to the file.

Destinations can be more than just simple files. A destination can be a comma-separated list of user names that are informed of the message by scribbling on their terminal if they are logged onto the system. But don't use this lightly, it can be a pain. If the destination is a star, all logged in users are sent the message.

If the destination starts with an @, then it should be followed by a host name. In this case, all selected messages are sent to the named host and will be logged there.

Before being processed by the daemon, the `syslog.conf` file is passed through by the `m4` macro-processor. This allows you to have the same configuration file on all your machines but select different actions depending on the settings of variables. Sun provides support for having one centralized logging host. If your machine has the nickname of `loghost` in `/etc/hosts`, then `syslogd` will define the variable `LOGHOST` before invoking `m4`. This variable is used to select bits of the `config.sys` file. It allows different things to be done locally and on the main host logging machine.

If you change `syslog.conf`, then you need to send a HUP signal to the running process to tell it to reread the configuration file. You can do a `ps` to find the value, but to help you, the daemon writes its process ID in a file, `/etc/syslog.pid`. Then the command:

```
kill -HUP `cat /etc/syslog.pid`
```

will work nicely.

Finally

The `syslogd` daemon has become central to the operation of UNIX systems. It's so important that it's started very early on in the system bootstrap sequence. If you change `/etc/syslog.conf` to create a new logging file, don't forget to make sure that the data is removed periodically. The script `/usr/lib/newsyslog` can give you a guide here. Finally, if you want to watch a log growing, the command

```
tail -f file
```

will track the file and print any new data that's added.

***Peter Collinson** runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* pc@cpg.com.