Kernel logging: APIs and implementation

From the kernel to user space logs

M. Tim Jones
Independent author

30 September 2010

In kernel development, we use `printk` for logging without much thought. But have you considered the process and underlying implementation of kernel logging? Explore the entire process of kernel logging, from `printk` to insertion into the user space log file.

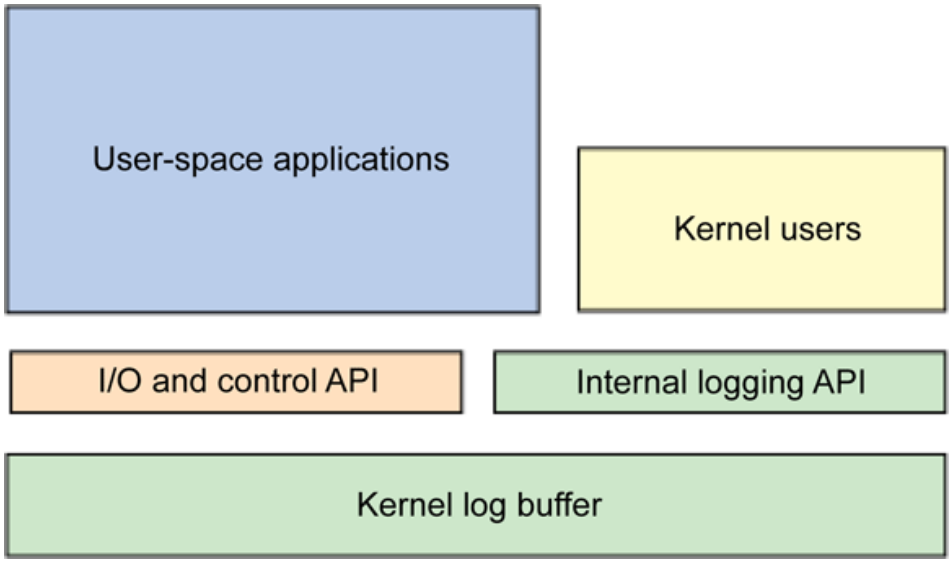
Connect with Tim

Tim is one of our most popular and prolific authors. Browse all of Tim's articles on developerWorks. Check out Tim's profile and connect with him, other authors, and fellow readers in My developerWorks.

The use of logs for debugging is as old as computing itself. Logs are useful not only for understanding the internal operation of a system but also the timing and relationships of activities within the system through the time-ordered messages within a time-stamped log.

This article begins its exploration of logging in the kernel by looking at the application programming interfaces (APIs) used to configure and collect the log information (see Figure 1 for a view of the overall framework and components). It then looks at movement of log data from the kernel into user space. Finally, the article explores the target of kernel-based log data: the log management framework in user space with `rsyslog`.

Figure 1. Kernel logging ecosystem and major components



Kernel API

Logging within the kernel is performed using the `printk` function, which shares similarity with its user space counterpart, `printf` (print formatted). The `printf` command has a long history in languages, most recently in the `C` language but going much farther back into the 1950s and 1960s in the Fortran (`PRINT` and `FORMAT` statements), BCPL (`writf` function; BCPL was a precursor of `C`), and ALGOL 68 languages (`printf`, `putf`).

Within the kernel, `printk` (print kernel) is used to write formatted messages into a buffer using a format almost identical to the `printf` function. You can find the format of `printk` in ./linux/include/linux/kernel.h (and its implementation in ./linux/kernel/printk.c):

```
int printk( const char * fmt, ... );
```

This format indicates that a string is used to define the text and format (just like `printf`) and is accompanied by a variable set of arguments (identified by the ellipsis [`...`]).

Kernel configuration and errors

Logging through `printk` is enabled through kernel configuration using the `CONFIG_PRINTK` kernel configuration option. Although `CONFIG_PRINTK` is typically enabled, system calls made to a kernel that does not include this option return in an `ENOSYS` error return.

One of the first differences you'll see in the use of `printk` is more about protocol and less about function. This feature uses an obscure aspect of the `C` language to simplify the specification of message level or priority. The kernel allows each message to be classified with a log level (one of eight that define the severity of the particular message). These levels communicate whether the system has become unusable (an emergency message), a critical condition has occurred (a critical message), or the message is simply informational. The kernel code simply defines the log level as the first argument of the message, as illustrated in the following example for a critical message:

```
printk( KERN_CRIT "Error code %08x.\n", val );
```

Note that the first argument is not an argument at all, as no comma (`,`) separates the level (`KERN_CRIT`) from the format string. The `KERN_CRIT` is nothing more than a string itself (in fact, it represents the string `"<2>"`; see Table 1 for a full list of log levels). As part of the preprocessor, `c` automatically combines those two strings in a capability called *string literal concatenation.* The result is a single string that incorporates the log level and user-specified format string as a single string. Note that if a caller does not provide a log level within `printk`, a default of `KERN_WARNING` is automatically used (meaning that only log messages of `KERN_WARNING` and higher priority will be logged).

Table 1. Log levels, symbolics, and uses

Symbolic	String	Usage
`KERN_EMERG`	`<0>`	Emergency messages (precede a crash)
`KERN_ALERT`	`<1>`	Error requiring immediate attention
`KERN_CRIT`	`<2>`	Critical error (hardware or software)
`KERN_ERR`	`<3>`	Error conditions (common in drivers)
`KERN_WARNING`	`<4>`	Warning conditions (could lead to errors)
`KERN_NOTICE`	`<5>`	Not an error but a significant condition
`KERN_INFO`	`<6>`	Informational message
`KERN_DEBUG`	`<7>`	Used only for debug messages
`KERN_DEFAULT`	`<d>`	Default kernel logging level
`KERN_CONT`	`<c>`	Continuation of a log line (avoid adding new time stamp)

The `printk` call can be called from any context in the kernel. The call begins in ./linux/kernel/printk.c in the `printk` function, which calls `vprintk` (in the same source file) after resolving the variable-length arguments using `va_start`.

Logging helper functions

The kernel also provides some helper functions for logging that can simplify their use. Each log level has its own function, which expands as a macro for the `printk` function. For example, when using `printk` with the `KERN_EMERG` log level, you can use the `pr_emerg`, instead. Each macro is listed in ./linux/include/linux/kernel.h.

The `vprintk` function performs a number of management-level checks (looking for recursion), and then grabs the lock for the log buffer (`__log_buf`). Next, the incoming string is checked for the log-level string; if found, the log level is set accordingly. Finally, `vprintk` grabs the current time (using the function `cpu_clock`) and converts it into a string using `sprintf` (not the standard library version but an internal kernel version implemented in ./linux/lib/vsprintf.c). The string passed into `printk` is then copied into the kernel log buffer using a special function that manages the bounds of the ring (`emit_log_char`). At the end of this function, a gratuitous acquisition and release of the console semaphore is performed that emits the next log message to the console (performed within `release_console_sem`). The size of the kernel ring buffer was originally 4KB but in recent kernels is sized at 16KB (and up to 1MB, depending on the architecture).

At this point, you've explored the API used to insert log messages into the kernel ring buffer. Now, let's look at the method used to migrate data from the kernel into the host.

Kernel logging and interface

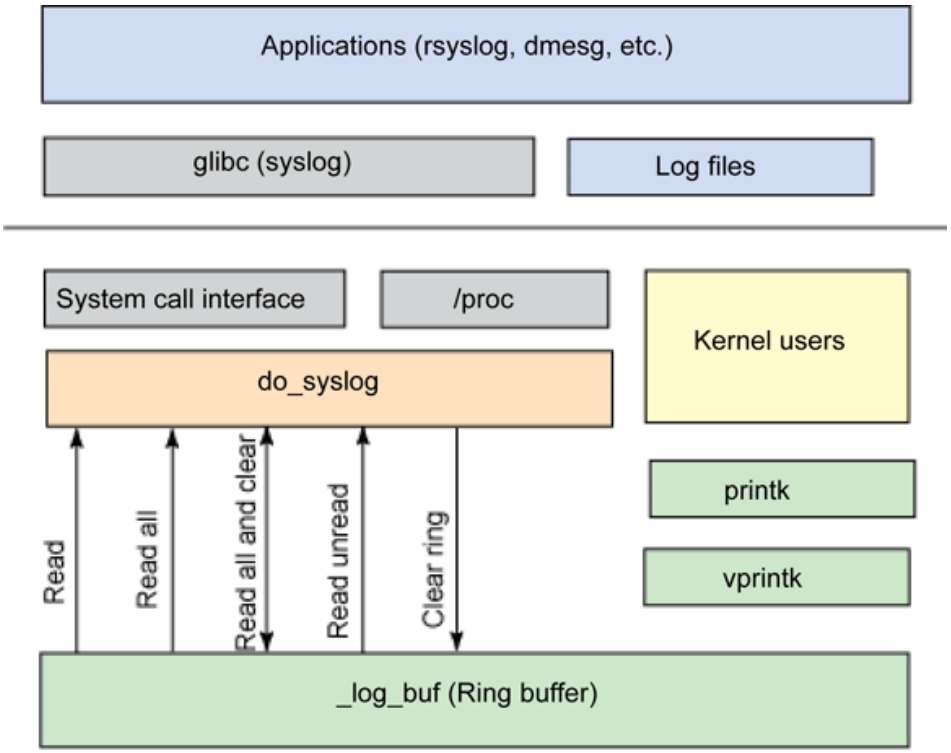
Access to the log buffer is provided at the core through the multi-purpose `syslog` system call. This single call implements a variety of actions that can all be performed from user space but only one action for non-root users. The prototype for the `syslog` system call is defined in ./linux/include/linux/syslog.h; its implementation is in ./linux/kernel/printk.c.

syslog(2) vs. syslog(3)

Note that the `syslog` defined here (`syslog(2)`) is different from the API for sending messages to the system logger (`syslog(3)`). The latter allows messages to be sent to the `syslog` (through functions to `open`, `close`, and `write` to the log using a particular priority).

The `syslog` call serves as the input/output (I/O) and control interface to the kernel's log message ring buffer. From the `syslog` call, an application can read log messages (partial, in their entirety, or only new messages) as well as control the behavior of the ring buffer (clear contents, set the level of messages to be logged, enable or disable console, and so on). Figure 2 provides a graphical illustration of the logging stack with some of the major components discussed.

Figure 2. Kernel logging stack identifying the major components



The `syslog` call (called `do_syslog` within the kernel in ./linux/kernel/printk.c) is a relatively small function that provides the ability to read and control the kernel ring buffer. Note that in glibc 2.0, this function is called `klogctl` because of overuse of the term *syslog,* which refers to a variety of calls and applications. The prototype function (in user space) for `syslog` and `klogctl` is defined as:

```
int syslog( int type, char *bufp, int len );
int klogctl( int type, char *bufp, int len );
```

The `type` argument communicates the command to perform and is associated with an optional buffer with its length. Some commands (such as clearing the ring buffer) ignore the `bufp` and `len` arguments. Although the first two command types perform no action within the kernel, the rest are used to read log messages or control aspects of logging. Three commands are used to read log messages. The `SYSLOG_ACTION_READ` command is used to block until log messages are available, and then return them in the provided buffer. This command consumes the messages (older messages will not appear in subsequent calls to this command). The `SYSLOG_ACTION_READ_ALL` command reads the last *n* characters from the log (where *n* is defined as the *'len'* parameter passed to `klogctl`). The `SYSLOG_ACTION_READ_CLEAR` command performs the `SYSLOG_ACTION_READ_ALL` action followed by a `SYSLOG_ACTION_CLEAR` command (clear the ring buffer). `SYSLOG_ACTION_CONSOLE` ON and OFF manipulate the log level to enable or disable log messages to the console, where `SYSLOG_CONSOLE_LEVEL` allows the caller to define the level of log messages for the console to accept. Finally, `SYSLOG_ACTION_SIZE_BUFFER` returns the size of the kernel ring buffer, and `SYSLOG_ACTION_SIZE_UNREAD` returns the number of characters currently available to be read in the kernel ring buffer. The complete list of `SYSLOG` commands is shown in Table 2.

Table 2. Commands implemented with the syslog/klogctl system call

Command/opcode	Purpose
`SYSLOG_ACTION_CLOSE (0)`	Close the log (unimplemented)
`SYSLOG_ACTION_OPEN (1)`	Open the log (unimplemented)
`SYSLOG_ACTION_READ (2)`	Read from the log
`SYSLOG_ACTION_READ_ALL (3)`	Read all messages from the log (non-destructively)
`SYSLOG_ACTION_READ_CLEAR (4)`	Read and clear all messages from the log
`SYSLOG_ACTION_CLEAR (5)`	Clear the ring buffer
`SYSLOG_ACTION_CONSOLE_OFF (6)`	Disable `printk`s to the console
`SYSLOG_ACTION_CONSOLE_ON (7)`	Enable `printk`s to the console
`SYSLOG_ACTION_CONSOLE_LEVEL (8)`	Set level of messages the console accepts
`SYSLOG_ACTION_SIZE_UNREAD (9)`	Return the number of unread characters in the log
`SYSLOG_ACTION_SIZE_BUFFER (10)`	Return the size of the kernel ring buffer

Implemented above the `syslog`/`klogctl` layer, the kmsg proc file system is a I/O path (implemented in ./linux/fs/proc/kmsg.c) that provides a binary interface for reading log messages from the kernel buffer. This is commonly read by a daemon (`klogd` or `rsyslogd`) that consumes the messages and passes them to `rsyslog` for routing to the appropriate log file (based on its configuration).

The file /proc/kmsg implements a small number of file operations that equate to internal `do_syslog` operations. Internally, the `open` call relates to the `SYSLOG_ACTION_OPEN` and the `release` call to `SYSLOG_ACTION_CLOSE` (each of which is implemented as a No Operation Performed [NOP]). The poll operation allows the wait for activity on the file, and then invokes `SYSLOG_ACTION_SIZE_UNREAD`

to identify the number of characters available to read. Finally, the `read` operation maps to `SYSLOG_ACTION_READ` to consume the available log messages. Note that the /proc/kmsg file is not useful to users: It is used by a single daemon to grab log messages and route them to the necessary log file in the /var space.

User space applications

User space provides a number of access points for reading and managing kernel logging. Let's begin with the lower-level interfaces (such as the /proc file system configuration elements), and then expand to the higher-level applications.

The /proc file system exports more than just a binary interface for accessing log messages (`kmsg`). It also presents a number of configuration elements both related and independent of those discussed through the `syslog`/`klogctl`. Listing 1 shows an exploration of these parameters.

Listing 1. Exploring the printk configuration parameters in /proc

```
mtj@ubuntu:~$ cat /proc/sys/kernel/printk
4 4 1 7
mtj@ubuntu:~$ cat /proc/sys/kernel/printk_delay
0
mtj@ubuntu:~$ cat /proc/sys/kernel/printk_ratelimit
5
mtj@ubuntu:~$ cat /proc/sys/kernel/printk_ratelimit_burst
10
```

From Listing 1, the first entry defines the log levels currently used in the `printk` API. These log levels represent the console log level, default message log level, minimum console log level, and default console log level. The `printk_delay` value represents the number of milliseconds to delay between `printk` messages (to add readability in some scenarios). Note here that it's set to zero, and it cannot be set through /proc. The `printk_ratelimit` defines the minimum length of time allowed between messages (currently defined as some number of kernel messages every 5 seconds). The number of messages is defined by `printk_ratelimit_burst` (currently defined as 10). This is particularly useful if you have a chatty kernel but a bandwidth-constrained console device (such as over a serial port). Note that within the kernel, rate limiting is caller controlled and is not implemented within `printk`. A `printk` user who desires rate limiting calls the `printk_ratelimit` function.

The `dmesg` command can also be used to print and control the kernel ring buffer. This command uses the `klogctl` system call to read the kernel ring buffer and emit it to standard output (stdout). The command can also be used to clear the kernel ring buffer (using the `-c` option), set the level for logging to the console (the `-n` option), and define the size of the buffer used to read the kernel log messages (the `-s` option). Note that if the buffer size is not specified, `dmesg` identifies the proper buffer size using the `SYSLOG_ACTION_SIZE_BUFFER` operation to `klogctl`.

Finally, the mother of all logging applications is `syslog`, a standardized logging framework that is implemented in major operating systems (including Linux® and Berkeley Software Distribution [BSD]). `syslog` has its own protocol used to convey event notification messages over a variety of transport protocols (dividing components into originators, relays, and collectors). In many cases,

all three are implemented in a single host. In addition to `syslog`'s many interesting features, it specifies how logging information is collected and filtered as well as where to store it. `syslog` has gone through numerous changes and evolved. You've probably heard of `syslog`, `klog`, or `sysklogd`. In more recent distributions of Ubuntu, a new version of `syslog` called `rsyslog` is used (based upon the original `syslog`), which refers to the reliable and extended `syslogd`.

The `rsyslogd` daemon, through its configuration file in /etc/rsyslog.conf), understands the /proc file system `kmsg` interface and uses it to extract kernel logging messages. Note that internally, all log levels are written through /proc/kmsg so that instead of the kernel defining which log levels to transport, the task is left to `rsyslog` itself. The kernel log messages are then stored in /var/log/kern.log (among other configured files). In /var/log, you'll find a plethora of log files that include general message and system-related calls (/var/log/messages), system boot log (/var/log/boot.log), authentication logs (/var/log/auth.log), and others.

Although the logs are available for your review, you can also use them for automated audits and forensics. A variety of log file analyzers exists for troubleshooting or compliance with security regulations and automatically look for problems using techniques such as pattern recognition or correlation analysis (even across systems).

Going farther

This article gave a glimpse into kernel logging and applications—from kernel log message creation in the kernel to its storage within the kernel's ring buffer to its transport into user space through `syslog`/`klogctl` or /proc/kmsg to its routing through the `rsyslog` logging framework to its final resting place in the /var/log subtree. Linux provides a rich and flexible framework for logging (both in the kernel and external).

Resources

Learn

- Read about `rsyslog` (the new system logging framework to replace `syslog` and `klog`) at its manual page and also its wiki site.
- Ubuntu maintains a useful page on logging focused on `rsyslog`. This white paper provides a detailed introduction to logging and `rsyslog`, including configuration and complex multi-host logging networks.
- The `syslog(2)`man page provides a great introduction to `syslog(2)` and its various options and configuration.
- The `printk` function relies on a feature of the `C` language called string *literal concatenation.* This `C` language page from Wikipedia introduces this technique.
- The `syslog` protocol is actually a standardized protocol through the Internet Engineering Task Force's Request for Comments (RFC) process. Read about syslog RFC 5424.
- Log file analysis is a hot topic in machine learning and monitoring tools. Learn more about general log analysis at Wikipedia and about an active log file monitoring tool called Swatch at SourceForge.
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools, as well as IT industry trends.
- Watch developerWorks on-demand demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.
- Follow developerWorks on Twitter, or subscribe to a feed of Linux tweets on developerWorks.

Get products and technologies

- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

M. Tim Jones

M. Tim Jones is an embedded firmware architect and the author of *Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective*. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.