;login:

THE MAGAZINE OF USENIX & SAGE

April 2001 • volume 26 • number 2

inside:

OPEN SOURCE

A Logging and Tracing Facility for an Embedded Source Code UNIX Product

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

a logging and tracing facility for an embedded source code UNIX product

by Bob Gray

Bob Gray is co-founder of Boulder Labs, a digital video company. Designing architectures for performance has been his focus ever since he built an image processor system on UNIX in the late 1970s. He has a Ph.D. in computer science from the University of Colorado.

<bob@cs.colorado.edu>

I find myself designing yet another real-time, priority-based, multi-tasking system to use in a manufacturing plant. We are leveraging general purpose PC hardware and Source Code UNIX to control valves, relays, and motors and to monitor feedback with optical sensors and analog-to-digital converters. I've come to rely on a handful of tried-and-true methodologies for designing and building complicated systems.

This month I want to describe a debugging and tracing facility for multi-processing systems that I first used with PDP-11s in the late 1970s. The refined, current version, which runs under FreeBSD, is available from <*http://www.boulderlabs.com*>. The code should be trivial to port to any machine with a GNU C compiler and Sys V type shared memory. But regardless of your particular environment, the concepts are applicable to any real-time, multi-tasking system.

In an earlier article, I described the advantages of using a Source Code UNIX for embedded products. See my April 1999 article, (;login: vol. 24, No. 2) "Embedding Source Code UNIX in the Product" (<*http://www.boulderlabs.com/6.embedding*>). In this article, and successor articles, I'll share a number of design and coding methodologies that I have found successful.

A multi-tasking system often has complicated behavior. Various processes react to outside stimuli and internal interprocess communication. Tracing the system's activity is difficult when everything is working, and even harder during the development cycle when components aren't fully functional. The software described in this article is designed to track what is happening when the system is running, both during development and during product deployment.

The basic concept is that we wish to have control over the amount of logging data generated and present that information to the user in chronological order. During a debug cycle, we want to be able to increase the detail of logging in certain areas and decrease the detail in other areas.

The goals of the logging and tracing system are:

1. Low overhead so as to not skew what is being measured.
2. Easy visualization of what each process is doing and when.
3. A mechanism that not only allows the developers to control the amount of logging, but also to choose which messages get logged.
4. A mechanism that tracks the last *N* events without the need to flush data to disk.

Let's look at some sample output.

A	B	C : D	E	F	G
21:55:34.0258	C	177:1	chef_con.c	480:	This is the Chef version: 1.1
21:55:34.0272	C	0:2	chef_con.c	482:	chef_control running as CHEF
21:55:34.0278	F	177:1	frame.c	2537:	NOW START CHILDREN
21:55:34.0280	F	177:1	frame.c	1894:	before SendMsg: chData->numChildren=4
21:55:34.0285	M	177:1	motor.c	496:	Motor is alive waiting for event

Shared memory is the interprocess communication mechanism of choice when you're striving for minimal time impact of communicating

Column A is a high-resolution timestamp. You'll always see logging lines in chronological order. Column B is a symbol indicator of which process logged the message. Columns C and D are orthogonal parameters that control which messages appear in the log buffer. Details are given later. Columns E and F are the file name and source code line number. Column G is the string of the log message.

To satisfy goal number 1, we want a very low overhead logging/tracing mechanism. Ideally, there would be zero I/O and zero system calls. I'm willing to tolerate the cost of writing a message to memory, but a disk write is too expensive.

A naïve logging mechanism would have a buffer in each process of the multi-tasking system. This presents two problems. First, every process would have to provide an I/O mechanism to ultimately get the log information to the user. If a process were to crash, the logging information could easily be lost. Second, the relative sequencing of events in multiple processes is not easily visualized if every process logs its own data. (Granted, a post-processing step could merge the individual logs.)

Our mechanism designates one process as the parent that creates a piece of shared memory. All other processes "attach" to the shared memory. Any process can log information by acquiring write-access to the shared memory and copying its logging data. A circular buffer is implemented in the shared memory. One benefit of the circular buffer is that it keeps track of the last *N* events your system logged without any I/O. Like the "black box" in an aircraft accident, the last few things before the crash are usually enough. Optionally, you can request that the circular buffer be periodically flushed to disk or to a terminal. The code detects when the circular buffer wraps around and gives the user a clear indication that old data has been overwritten.

The logging software is leveraged from the BSD kernel circular buffer. If you look at /usr/include/sys/msgbuf.h on a FreeBSD system, you will see the code's heritage. I changed that base code to work in user-land instead of kernel space, and I added shared memory constructs to support the multiple process debugging. Shared memory is the interprocess communication mechanism of choice when you're striving for minimal time impact of communicating. I first used several of the techniques mentioned in this article when I programmed a message-passing system under DEC's RSX11M operating system in 1977. Several years later, shared memory made it into mainstream UNIX with Sys V (shmat, shmctl) and 4.2BSD (mmap).

Here is the essence of the logging:

```
circWrite(const char *p, int n) {

    lockResource(&mbp->msg_lock, &mbp->msg_busycnt);
    for(i=0; i<n; i++){
        mbp->msg_bufc[mbp->msg_bufx++] = *p++;
        ...
    }
    freeResource(&mbp->msg_lock, &mbp->msg_busycnt);
}
```

To enforce exclusive access, a process must first acquire the shared memory circular buffer by calling the lockResource function. Fortunately, most architectures (including the x86) have a "test and set" instruction that allows the lockResource function to be implemented without kernel assistance for an uncontested lock. My code uses the x86 cmpxchg instruction. Therefore, typically, only a few instructions are executed to gain access. (Thanks to Ron G. Minnich for posting his fastlock code many years ago on the Internet.)

Then, a tight loop copies a preformatted message into the circular buffer at memory writing speed. The freeResource routine is also fast. All of the code to log a message requires less than a microsecond on today's hardware. (One hundred instructions on a 100MIPS machine take 1 microsecond.) The chances of two processes needing to log at the same time are very low. For the contested access to the shared memory, we require the overhead of a system call which causes the process to context switch.

Over the years, I've learned that you want to build diagnostic tracing into your code from the start – it's harder to add it later. While a debugger, such as GDB, is invaluable in the development process, you also need logging and tracing in software products.

Over the years, I've learned that you want to build diagnostic tracing into your code from the start – it's harder to add it later.

Many years ago, I found Eric Allman's sendmail trace facility (see contrib/sendmail/src/trace.c). At runtime, it allows the user to turn on various debugging facilities at various priority levels. As a programmer, you sprinkle logging messages throughout your code, each message at a particular priority with a particular integer that Eric calls a "module ID." He chooses the following conventions for priorities:

```
#define FATAL 0
#define WARN 1
#define DIAG2 2
#define DIAG3 3
 ...
```

Module IDs are orthogonal to priorities. Each log message has both a priority and a module ID. At runtime, you can control which messages get logged by setting a priority level for each module ID. Once this vector has been initialized, at runtime, a message is logged if, for its module ID, the specified priority is lower than that of the corresponding vector entry.

It's time for an example. Assume that by a command line argument, I set the module ID vector as follows:

modID	Priority
0	2
1	2
2	2
3	2
4	2
5	3
6	1

And assume, sprinkled through my code, I have the following lines:

```
LOG(DIAG2, 3, 'M', "Got START from MOTOR");
LOG(DIAG3, 4, 'F', "Call to handle_TCP_read');
LOG(WARN,  5, 'G', "Command too long (%d)\n", length);
```

LOG is a macro with the following definition:

```
#define LOG(pri, mId, mod_name, fmt, args...)    \
{    \
if (mId>=0 && mId<TVLEN && tTdvect[mId] >= pri)        \
{    \
Log (pri, mod_name, mId, __FILE__, __LINE__, fmt , ## args);\}    \
}
```

You can see that we will "Log" the message "Got START from MOTOR" because tTdvect[mId] >= pri. Note also the third argument, 'M'. It's a symbol for the motor control

As products are deployed, we have found that these kinds of log files are often sufficient to diagnose a problem without any on-site visit.

process. "Call to handle_TCP_read" will not be logged because tTdvect[4] < DIAG3. "Command too long (%d)\n" from the 'G' process will be logged because tTdvect[5] >= WARN.

The Log function formats the message and has the basic form:

```
Log (int pri, int mod_name, int modId, char *file, int line, char *fmt, ...)
{
    gettimeofday(.....);     /* get system time stamp */

    n = snprintf (mbuf,LOGBUFSIZE,
       "%02d:%02d:%02d.%04ld %c%3d:%1d %-10.10s %4d: %s%s%s\n",
       t->tm_hour, t->tm_min, t->tm_sec, timeStamp.tv_usec / 100,
       mod_name, modId, pri, file, line, buf, errStr ? ": " : "",
       errStr ? errStr : "");
    circWrite(mbuf,n);
}
```

We made the decision to pay for one system call, gettimeofday, that gets an accurate time-stamp. The argument, module, is a letter that gives a symbolic identity to the process that logged to shared memory. For example, 'M' represents the motor control process. File and line are obtained from the C pre-processor using the __FILE__, __LINE__ directives. The variable number of arguments following the fmt string is handled with the va_list, va_start, vsnprintf mechanism.

The logging vector is initialized to priority 2 for all entries. That is, as code is executed, lines with a priority level of FATAL, WARN, and DIAG2 will cause logging activity. From the command line, we can override these defaults. For example, the parent process (the one that created the shared memory) could be started with the following debugging options which would be passed to the respective children processes, 'G' and 'M' (one is a GUI, the other is Motor control):

```
-dG3:25-33  -dM4:0-
```

Then the 'G' process would be told to adjust the vector entries 25–33 to priority level 3 and the 'M' process would be told to adjust its vector entries from 0 through the last, to priority 4. (Note, each process has its own tTdvect array.) Hence, logging would increase as a result of these runtime options. The syntax is:

```
debug_option   : '-d' [mod_name] pri ':' range
mod_name       : 'C' | 'D' | 'F' | 'G' | 'M'   (adjust for your system)
pri            : uchar
range          : uchar '-' uchar | uchar '-'
```

Finally, there are options to control the frequency of circular buffer flushing, if any, the file or file descriptor it goes to, the frequency of flushing, and the size of the circular buffer. Typically, it is the parent process that has this responsibility.

Over a couple of projects, we have found this logging and tracing facility invaluable during the development cycle. As products are deployed, we have found that these kinds of log files are often sufficient to diagnose a problem without any on-site visit. In the spirit of collaborative Internet development, we hope you find this software useful.

Thanks to Dave Clements and Tom Poindexter.