TRACE - A System Wide Diagnostic Tool

Stephen Foulkes, Ron Rechenmacher
Fermi National Accelerator Laboratory
Batavia, IL 60510 USA
Email: sfoulkes@fnal.gov

***Abstract*— TRACE is a system-wide diagnostic tool that allows one to gather timing information with a minimal impact on application(s) performance. TRACE supports a variety of architectures under Linux and VxWorks. This utility instruments code easily and is controllable through the /proc file system. It has hooks to be built into a larger monitoring/alarming/debugging framework as well as supporting architecture-dependent features such as performance measurement counters or registers.**

I. Introduction

TRACE is a freeware open source logging tool developed at Fermilab. Its number one goal is to minimize the amount of overhead that will be added to an application that uses it. TRACE supports the use of multiple logging backends as well as allowing the user better control over how and where messages are stored. TRACE also supports running on multiple operating systems and processor architectures. Currently, TRACE is running under Linux on x86, x86_64 and PowerPC. It also runs under VxWorks on PowerPC. This paper will focus on using TRACE on Linux, and the details of its implementation there.

II. TRACE on Linux Overview

TRACE is built into the Linux kernel. Configuration and status information, as well as the contents of the circular buffer, which is the default logging backend that comes with TRACE is exported from the kernel through the /proc pseudo filesystem. Messages can be sent to TRACE from inside the kernel as well as from userspace. Binaries compiled with TRACE calls will run on systems that do not have TRACE support. Currently, TRACE only supports C and C++ applications.

III. Installation

The TRACE sources can be obtained from the TRACE website [1], and then unpacked onto the target system. The user needs to take care of any configuration changes before installation because most of the parameters are hard codes, and can't be updated on a running system. These paraemeters include the maximum number of applications that can generate messages at one time, the maximum number of logging backends, the maximum size of a TRACE message and the maximum number of parameters that can be sent with a TRACE message. All of these are located inside trace.c which sits inside the TRACE source tree at src/init.

Once TRACE has been adequately configured, the TRACE installer can be run by executing a "make install" command from inside the TRACE source tree. The installer will try to locate the Linux kernel sources and also attempt to determine the version of the kernel. After the kernel sources have been located, the installer will find a patch that is closet in version to the kernel installed on the system and apply it. TRACE has patches for many 2.4.x and 2.6.x kernels, including the ones distributed with Scientific Linux 3.x and 4.x. If the installer is unable to locate the kernel sources the –kernel_dir parameter may be passed to the makefile with the correct location. If the kernel version can't be determined or is determined incorrectly the –kernel_version parameter may be used to specify the version. Once the patches have been applied the kernel can be rebuilt. No other configuration to the kernel is necessary.

On sytem startup, TRACE will post several debugging messages to the system console as it allocates memory for its internal buffers. There will also be a trace directory placed inside /proc that contains several files including control, level, buffer, raw and version. The TRACE installer also instruments the kernel with several TRACE calls. The messages these calls generate will be visible inside /proc/trace/buffer.

IV. The TRACE Circular Buffer

TRACE maintains a circular buffer inside the kernel that messages can be logged to. This circular buffer is the default logging backend for TRACE and the data contained within it does not persist between system restarts. Writes to the circular buffer will not block and are reentrant, making the circular buffer a good choice for logging messages from multi threaded applications or from interrupt service routines.

The circular buffer is accessed from the console, through the /proc/trace/buffer file. Cat-ing this file is all that is necessary to see the contents of the buffer:

```
[~]$ cat /proc/trace/buffer
PID    TraceName lvl   message
-----------------------------------------------
12208  KERNEL    24    trace_proc_control_write
12208  KERNEL    27    system call 4
12208  KERNEL    26    system call return 0
12208  KERNEL    27    system call 175
12208  KERNEL    26    system call return 0
[~]$
```

There are several parameters that the circular buffer logs, but does not necessarily display. These include the process ID, TRACE logging level, time stamp, message, and the name of the application generating the message. The full list of parameters that are logged is listed in /proc/trace/control on the "available heading" line. Changing the parameters that are displayed when the circular buffer is printed is accomplished by echoing a string containing the desired parameters to be displyed to /proc/trace/control:

```
echo print=timeStamp,PID,lvl,message > \
       /proc/trace/control
```

Executing that command will cause the time stamp, process ID, TRACE level and messages parameters to be printed when the TRACE buffer is accessed. The TRACE circular buffer can be cleared by echoing the reset command to /proc/trace/control:

```
echo reset > /proc/trace/control
```

The TRACE circular buffer can also be cleared programatically with the following function which is defined in the TRACE header:

```
void traceControl_Reset()
```

The current status of the circular buffer is displayed in /proc/trace/control. Information exported through that file includes the size, total number of entries in the circular buffer and the number of entries used. The maximum number of entries and maximum message size are hard coded inside src/init/trace.c which is located in the TRACE source tree. The defaults parameters that come with TRACE are reasonable, and should not have to be changed.

V. Instrumenting Source Code with TRACE

All that is necessary to start using TRACE in an application is to include the TRACE header. The main TRACE header file resides inside the TRACE source directory at src/include/linux/trace.h. This file is installed inside the kernel source tree at include/linux/trace.h.

The TRACE header only has one dependency, which is trace_intr.h. This file is architecture dependent, and is responsible for passing TRACE messages to the circular buffer between user space and kernel space. The TRACE installer will drop the appropriate copy of trace_intr.h into the correct asm directory inside the kernel source tree for each platform that TRACE supports (asm-i386, asm-ppc, etc). The kernel build system will symlink the asm directory to the appropriate architecture specific asm directory that represents the platform the kernel was built for, which allows trace.h to pickup the correct trace_intr.h.

If an application is going to be distributed with TRACE messages, the TRACE header file should be included with the source. This way, it will be possible to build the application on systems that do not have TRACE. In order to make this work, the trace.h file will have to be modified to include the appropriate trace_intr.h, which will also have to be distributed with the application.

A. The TRACE Macro

The TRACE macro is the method that is used to log messages with TRACE. It has the following prototype:

```
TRACE(int level, char *format_string,
      int param1, int param2,
      int param3, int param4,
      int param5, int param6);
```

The level parameter allows the user to group similar messages together. For example, error messages could be set to level zero while status message could be set to level one. That way, if the user only wishes to see error messages they can turn off all levels except level zero. The format string is similar to a printf format string and currently only integer parameters are supported. Integers can be printed with the usual printf() conversion specifications such as %d and %x. TRACE by default supports a maximum of six parameters.

VI. Filtering Messages

It is possible to specify a name for each source of messages that get passed to TRACE. This is useful in situations where multiple applications on a system are using TRACE, and the user wishes to filter messages to those produced by a specific application. To specify a TRACE name for a particular application, define the TRACE_NAME constant before the TRACE header is included:

```
#define TRACE_NAME ''myApplication''
#include <linux/trace.h>
```

It is possible to modify the TRACE_NAME at any point in time, which can be useful for giving forked processes a different name. The following function will reinitialize TRACE with a new name:

```
void traceControl_ReInit(char *name)
```

For each TRACE name, there are thirty two levels that messages can be posted to. Each level can be turned on and off individually. This is controlled through the /proc/trace/level file. That file looks like the following:

```
  0     KERNEL=0x00000000,0x0,0x0,0x0
198 NULL_NAME=0xffffffff,0x0,0x0,0x0
199      FULL=0xffffffff,0x0,0x0,0x0
```

The number on the far left is the TRACE_ID, which is a unique number that TRACE assigns to each declared name. The four thirty-two bit numbers across from each TRACE name are the level masks. Each bit in the mask corresponds to a level. Having a value of one will enable logging messages for that level, while a zero will disable logging. For example, a level mask of 0xF would enable TRACE messages for levels zero through three. There is one level mask for each TRACE logging backend. The first level mask always controls the circular buffer.

There are two ways to enable and disable the levels. The first is to echo a new level string to the /proc/trace/level file:

```
echo KERNEL=0xf,0x0,0x0,0x0>/proc/trace/level
```

That string will enable messages on levels zero through three for the kernel TRACE name to be sent to the TRACE circular buffer. Messages going to the other TRACE logging backends from the kernel TRACE name have been disabled. Note that TRACE applies filters to messages before they are logged, so messages that are generated on disabled levels will not be logged.

It is also possible to modify the levels that are enabled/disabled programatically:

```
 void traceControl_LevelSet(int function,
                            int mask)
 int traceControl_LevelGet(int function)
```

Those two functions are defined in the TRACE header, and allow the setting and retreiving of TRACE levels for a particular TRACE function.

VII. User Defined Logging Backends

TRACE has the capability to send messages to upto three user defined logging backends. User defined logging backends must have the following prototype:

```
void function_name(struct timeval *time, int lvl,
                   int id, char *msg, ...);
```

TRACE is informed about the user defined logging backends through the TRACE_FUNCTIONS constant, which must be defined before the TRACE header file is included:

```
#define TRACE_FUNCTIONS {user_logger1,
                         user_logger2}
#define TRACE_NAME ``demo4''
#include <linux/trace.h>
```

User defined logging backends are useful in cases where the user may want to redirect TRACE messages to some other logging mechanism, like printk() or printf(). The TRACE header defines a printf() like function already, and this can be used with the following:

```
#define TRACE_FUNCTIONS {trace_printf}
#include <linux/trace.h>
```

Besides the /proc/trace/level file, there is an overall mask which enables/disables TRACE logging backends. The TRACE mode, which is contained inside /proc/trace/control is a bitmask that enables the TRACE logging backends. Its default value is 0x1, which enables only the circular buffer. The mode can be changed by echoing a new mode to the /proc/trace/control file:

```
echo mode=0x3 > /proc/trace/control
```

The TRACE mode can also be manipulated programatically with the following functions:

```
void traceControl_Mode(int new_mode)
int traceControl_ModeGet()
```

References

[1] http://fermitools.fnal.gov/abstracts/trace/abstract.html