

Solutions for tracing UNIX applications

Skill Level: Intermediate

[Martin Brown \(mc@mcslp.com\)](mailto:mc@mcslp.com)

Professional writer

Freelance

31 Mar 2009

If you are developing a UNIX® application, then you can trace and debug the running application and extract the information you need from it. But what if you want to know what is going on inside a UNIX application and you don't have access to the source code? This tutorial looks at some systems that enable you to trace the execution of applications and work out what they are doing without having to make any modifications to the source code, and even without having to stop and restart the application.

Section 1. Before you start

This tutorial is for UNIX system developers and administrators who are looking for the best ways to troubleshoot applications running on their systems. To get the most out of this tutorial, you should have a basic knowledge of the UNIX operating system and how it operates. Some basic programming experience is useful but is not required.

About this tutorial

Most developers and systems administrators know what should happen in their operating system and with their applications, but sadly, this isn't always the case. There are times when an application has failed, or is not behaving as you expect, and you need to find out more information. By using your existing knowledge of how your application should work and some basic UNIX skills, you can trace the application to find out what is causing the problem. This tutorial will teach you the basic techniques of using tracing tools to find out what your application is doing

behind the scenes.

First, the tutorial looks at the distinction between debugging and tracing, and how the two solutions differ. Then it examines some specific examples of where tracing can be used to solve problems in your application. DTrace provides elements of both system tracing and debugging, and also provides you with the ability to time and benchmark applications. Finally, the tutorial shows how to trace the information being exchanged between network computers to help find problems in network applications.

Section 2. Tracing overview

There are times when you need to know what is going on beneath the front-end of the application. For example, you may have a failing application, with no useful error message, with a system service that does not operate the way you expect. In these cases, you may not have the application source code, and therefore cannot perform a typical debug process to find out what is wrong. Tracing provides you with an alternative method.

Debugging

As a developer, the obvious solution to finding a UNIX application problem is to use the debugging feature of your development environment or operating system to examine the source code and find out what is causing the problem.

Most debugging systems allow you to monitor and examine both the execution process as individual lines are executed, as well as to monitor the values of individual variables and structures. With a debugger you can set a specific breakpoint within the code where the execution will stop and you can get information about the callstack -- the current route to the function -- and the variables and their values at that point.

Let's examine, as an example, an application that generates a calculation of a person's age by using the day they were born, and also taking into account leap years and other factors. To debug the application, you need the source code and you also need to compile the application with debugging enabled, as shown here: `$ gcc -g ageindays.c -o ageindays.`

To run the application, you supply the user's date of birth and the target date when you want to make a comparison (see Listing 1).

Listing 1. Making a comparison

```
$ ./ageindays 24/1/1980 22/2/2009
You have been alive 10622 days
You were born on 24/1/1980 which is a Thursday
```

To debug the application, you start by suspecting that the problem is located within a function called `calc_diff`, which compares the difference between the first and second dates. You might next create a sequence like the one shown in Listing 2.

Listing 2. Debugging the `calc_diff` function

```
$ gdb ageindays
GNU gdb 6.3.50-20050815 (Apple version gdb-962) (Sat Jul 26 08:14:40 UTC 2008)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-apple-darwin"...Reading symbols for shared
libraries ... done

(gdb) b calc_diff
Breakpoint 1 at 0x1bd7: file ageindays.c, line 27.
(gdb) r 24/1/1980 26/3/2009
Starting program: /nfs/MC/UnixSrc/c/bio/ageindays 24/1/1980 26/3/2009
Reading symbols for shared libraries ++. done

Breakpoint 1, calc_diff (day=26, month=3, year=2009) at ageindays.c:27
27 unsigned long days_diff=0;
(gdb) bt
#0 calc_diff (day=26, month=3, year=2009) at ageindays.c:27
#1 0x00001e3d in main (argc=3, argv=0xbffff708) at ageindays.c:89
(gdb) p days_diff
$1 = 8041
(gdb)
```

You can see from the output in [Listing 2](#) that you open the debugger, set a breakpoint in the `calc_diff()` function by specifying it by name, and then run the program within the debugger, supplying the same arguments as you would on the command line.

Once the debugger hits the breakpoint you created, execution of the application stops, and you can examine the code of the application and the functions being called. Using the debugger, you can see the arguments supplied to the function with their values (in this case, the date information supplied for the target date). Once execution stops, you can get a stack trace and see the exact line in the code that called the `calc_diff` function, and you can obtain the value of the `days_diff` variable. Because the execution of the application has been paused, you can also modify the value of a variable. This allows you to try out different values within the application to find potential problems.

All of this information is made available because the specific debugging information

(the symbols that make up the function and variable names) and other metadata such as the line within the code where the function is defined.

The specific debugging information has to be added to the binary application during building, and more importantly, you have to access the source to be able to compile the application with this debug information included. Debugging a program without being able to identify the function names and variable names is almost impossible.

Tracing compared to debugging

As a systems administrator, and often as a developer, you are more likely to be interested in finding a fault with a program. This can be why a particular program is causing other problems (such as memory and other errors), or why an application is not behaving as it should, and how it has in the past. Debugging the specifics of the application in this instance are not often useful. Instead, you need to examine how the application is being executed by the operating system.

With debugging, you are examining the execution of the individual functions defined within the application. Debugging concentrates on the application and the functions and structures within it, but typically ignores the system calls and calls to library functions that are made by the application to the operating system. Debugging provides a wealth of information about your application, but not necessarily how the operating system is executing that application.

With tracing, you are monitoring the interaction between the application and the operating system, usually by looking at the operating system functions that are called by the application during its execution.

Despite these semantic differences, the major difference between tracing and debugging is that tracing can take place without you having to have access to the source code, or to compile the application in any special way. This means that you can trace an application that comes with the operating system, or from a third-party vendor to find out what is wrong.

Tracing an application enables you to find out:

- Memory usage and calls to map memory
- Files being opened and closed during execution
- Reads and writes to different files
- Libraries loaded for a given application

Start by examining the output from `truss`, tools that are available on Solaris and AIX®.

Section 3. Using truss and strace

The truss tool is available on Solaris and AIX and provides the ability to trace system calls and signals within an application. The strace tool, available on Linux® provides similar functionality. On different systems, there are also tools that provide similar information, including ktrace (FreeBSD) and trace.

Overview of truss/strace

The truss and strace tools both provide similar levels of information, although the command-line options may be slightly different in each case. With both tools, the standard way to use them is to prefix the tool to the command you would ordinarily execute.

For example, Listing 3 shows the output of truss on the ageindays program mentioned earlier in this tutorial.

Listing 3. Output of truss

```
$ truss ./ageindays 24/1/1980 26/3/2009
execve("ageindays", 0x08047BBC, 0x08047BCC)  argc = 3
mmap(0x00000000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, -1, 0)
    = 0xFEFB0000
resolvepath("/usr/lib/ld.so.1", "/lib/ld.so.1", 1023) = 12
getcwd("/root", 1013) = 0
resolvepath("/root/ageindays", "/root/ageindays", 1023) = 15
xstat(2, "/root/ageindays", 0x08047880) = 0
open("/var/ld/ld.config", O_RDONLY) = 3
fxstat(2, 3, 0x08047760) = 0
mmap(0x00000000, 144, PROT_READ, MAP_SHARED, 3, 0) = 0xFEFA0000
close(3) = 0
sysconf(_CONFIG_PAGESIZE) = 4096
xstat(2, "/usr/lib/libc.so.1", 0x08046FA0) = 0
resolvepath("/usr/lib/libc.so.1", "/lib/libc.so.1", 1023) = 14
open("/usr/lib/libc.so.1", O_RDONLY) = 3
mmap(0x00010000, 32768, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_ALIGN, 3, 0)
    = 0xFE90000
mmap(0x00010000, 1413120, PROT_NONE, MAP_PRIVATE|MAP_NORESERVE|MAP_ANON|MAP_ALIGN, -1, 0)
    = 0xFEE30000
mmap(0xFEE30000, 1302809, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_TEXT, 3, 0)
    = 0xFEE30000
mmap(0xFE7F000, 30862, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
    MAP_INITDATA, 3, 1306624) = 0xFE7F000
mmap(0xFE78000, 4776, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_ANON,
    -1, 0) = 0xFE78000
munmap(0xFE6F000, 65536) = 0
mcntl(0xFEE30000, 187632, MC_ADVISE, MADV_WILLNEED, 0, 0) = 0
close(3) = 0
mmap(0x00010000, 24576, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON|MAP_ALIGN,
    -1, 0)
    = 0xFEE20000
```

```

munmap(0xFEf90000, 32768) = 0
getcontext(0x080475D0)
getrlimit(RLIMIT_STACK, 0x080475C8) = 0
getpid() = 15691 [15690]
lwp_private(0, 1, 0xFEE22A00) = 0x000001C3
setustack(0xFEE22A60)
sysi86(SI86FPSTART, 0xFEf879BC, 0x0000133F, 0x00001F80) = 0x00000001
ioctl(1, TCGETA, 0x08046C20) = 0
fstat64(1, 0x08046B80) = 0
You have been alive 10654 days
write(1, " Y o u   h a v e   b e e"..., 31) = 31
You were born on 24/1/1980 which is a Thursday
write(1, " Y o u   w e r e   b o r"..., 47) = 47
_exit(134511508)

```

By comparison, Listing 4 shows the output from strace on Linux.

Listing 4. Output from strace

```

$ strace ./ageindays 24/1/1980 26/3/2009
execve("./ageindays", [".:/ageindays", "24/1/1980", "26/3/2009"],
[/* 50 vars */]) = 0
brk(0) = 0x602000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f47db185000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f47db184000
access("/etc/ld.so.preload", R_OK)
= -1 ENOENT (No such file or directory)
open("/usr/lib/tls/x86_64/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/lib/tls/x86_64", 0x7ffff31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/lib/tls/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/lib/tls", 0x7ffff31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/lib/x86_64/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/lib/x86_64", 0x7ffff31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/lib/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/lib", {st_mode=S_IFDIR|0755, st_size=53248, ...}) = 0
open("/usr/local/lib/tls/x86_64/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/local/lib/tls/x86_64", 0x7ffff31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/local/lib/tls/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/local/lib/tls", 0x7ffff31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/local/lib/x86_64/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/local/lib/x86_64", 0x7ffff31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/local/lib/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/local/lib", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
open("/usr/local/qt/lib/tls/x86_64/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/local/qt/lib/tls/x86_64", 0x7ffff31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/local/qt/lib/tls/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)

```

```

stat("/usr/local/qt/lib/tls", 0x7fffe31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/local/qt/lib/x86_64/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/local/qt/lib/x86_64", 0x7fffe31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/local/qt/lib/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/local/qt/lib", 0x7fffe31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/X11R6/lib/tls/x86_64/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/X11R6/lib/tls/x86_64", 0x7fffe31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/X11R6/lib/tls/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/X11R6/lib/tls", 0x7fffe31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/X11R6/lib/x86_64/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/X11R6/lib/x86_64", 0x7fffe31858f0)
= -1 ENOENT (No such file or directory)
open("/usr/X11R6/lib/libc.so.6", O_RDONLY)
= -1 ENOENT (No such file or directory)
stat("/usr/X11R6/lib", 0x7fffe31858f0)
= -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=85050, ...}) = 0
mmap(NULL, 85050, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f47db16f000
close(3) = 0
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300\345\1\0\0\0\0\0@"...,
832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1330352, ...}) = 0
mmap(NULL, 3437208, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
= 0x7f47dac24000
mprotect(0x7f47dad63000, 2093056, PROT_NONE) = 0
mmap(0x7f47daf62000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x13e000) = 0x7f47daf62000
mmap(0x7f47daf67000, 17048, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x7f47daf67000
close(3) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f47db16e000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f47db16d000
arch_prctl(ARCH_SET_FS, 0x7f47db16d6f0) = 0
mprotect(0x7f47daf62000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ) = 0
mprotect(0x7f47db186000, 4096, PROT_READ) = 0
munmap(0x7f47db16f000, 85050) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f47db183000
write(1, "You have been alive 10654 days\n"..., 31You have been alive 10654 days
) = 31
write(1, "You were born on 24/1/1980 which "..., 47You were born on 24/1/1980
which is a Thursday
) = 47
exit_group(0)

```

In both cases, each line of output equates to the application executing one function call and shows the arguments to the function and the return value of the function call. Unlike the debugging example, each function call listed is a function within the system or system libraries and therefore represents a lower-level interface to the

functions called. For example, opening a file within an application may use the `fopen()` function within C or C++, but this is in fact a wrapper to the more primitive `open()` function.

You don't need to understand the specifics of each function to get a general idea of what the applications are doing. Many of the lines of the output are related to the initialization that the operating system applies in order to load and execute the program. The fundamentals of the two traces are the same:

- The `execve()` function is called to start a new program.
- Libraries for the program are loaded. With the Solaris output, the libraries first look for the use of `resolvepath()` and then open using `open()`. For Linux, `stat()` is used to check whether the library exists, and then `open()` is used to open it.
- Some memory is reserved and allocated for the process. Some of this will be the stack space reserved for the application, some will be used to hold the program, and others to hold the variables used by the program.
- Finally, the program is executed, and the `write()` function is called to print out the age and day of the birth information.

If you perform a trace and want to understand the specifics of each step, use the `man` command to read the manual page for each function.

Identifying application startup issues

A typical problem when starting an application is that the program fails to initialize properly, but terminates with an incomplete or misleading error message. Running a trace on the application can often highlight the problem. For example, Listing 5 shows how a test application has failed.

Listing 5. Application fails

```
$ ./errnoacc  
ERROR: Application failed to initialize
```

The error message provides no specific information about why the application failed to start. In this case, the problem has been deliberately introduced, but the same issue could occur with any command or application you were using, and the error message could be equally misleading or sometimes non-existent.

Running a trace on the application might give us some more clues (see Listing 6).

Listing 6. Running a trace

```

$ truss ./errnoacc
execve("/usr/lib/ld.so.1", 0x08047B20, 0x08047B28) argc = 1
mmap(0x00000000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, -1, 0)
= 0xFEFB0000
resolvepath("/usr/lib/ld.so.1", "/lib/ld.so.1", 1023) = 12
getcwd("/export/home/mc", 1014) = 0
resolvepath("/export/home/mc/errnoacc", "/export/home/mc/errnoacc", 1023) = 24
xstat(2, "/export/home/mc/errnoacc", 0x080477E4) = 0
open("/var/ld/ld.config", O_RDONLY) = 3
fxstat(2, 3, 0x080476C4) = 0
mmap(0x00000000, 144, PROT_READ, MAP_SHARED, 3, 0) = 0xFEFA0000
close(3) = 0
sysconf(_CONFIG_PAGESIZE) = 4096
xstat(2, "/usr/lib/libc.so.1", 0x08046F04) = 0
resolvepath("/usr/lib/libc.so.1", "/lib/libc.so.1", 1023) = 14
open("/usr/lib/libc.so.1", O_RDONLY) = 3
mmap(0x00010000, 32768, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_ALIGN, 3, 0) = 0xFEF90000
mmap(0x00010000, 1413120, PROT_NONE, MAP_PRIVATE|MAP_NORESERVE|MAP_ANON|MAP_ALIGN,
-1, 0) = 0xFEE30000
mmap(0xFEE30000, 1302809, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_TEXT, 3, 0)
= 0xFEE30000
mmap(0xFEF7F000, 30862, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
MAP_INITDATA, 3, 1306624) = 0xFEF7F000
mmap(0xFEF87000, 4776, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_ANON,
-1, 0) = 0xFEF87000
munmap(0xFEF6F000, 65536) = 0
mectl(0xFEE30000, 187632, MC_ADVISE, MADV_WILLNEED, 0, 0) = 0
close(3) = 0
mmap(0x00010000, 24576, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON|MAP_ALIGN,
-1, 0) = 0xFEE20000
munmap(0xFEF90000, 32768) = 0
getcontext(0x08047534)
getrlimit(RLIMIT_STACK, 0x0804752C) = 0
getpid() = 15727 [15726]
lwp_private(0, 1, 0xFEE22A00) = 0x000001C3
setustack(0xFEE22A60)
sysi86(SI86FPSTART, 0xFEF879BC, 0x0000133F, 0x00001F80) = 0x00000001
open("/etc/shadow", O_RDONLY) Err#13 EACCES [file_dac_read]
ioctl(1, TCGETA, 0x08046BB0) = 0
fstat64(1, 0x08046B10) = 0
ERROR: Application failed to initialize
write(1, " E R R O R :   A p p l i"., 40) = 40
_exit(0)

```

The offending issue is in this line: `open("/etc/shadow", O_RDONLY) Err#13 EACCES [file_dac_read]`.

Here the application is trying to open a file to which the user running the application has no access because the file permissions have secured the file. The application then terminates because the file cannot be opened, writing the error message in the process.

Tracing live applications

Often when you want to trace an application, it is because the application has already been started, and you want to find out why the application is not working. Like the initialization example, often the error message or other information provided

by the application may not describe the exact problem.

Locks, or attempting to access a resource that is currently in use by another process, can cause an application to apparently freeze and become unresponsive.

Both `strace` and `truss` provide the ability to "attach" to an already-running process. Attaching to the process works like running the process from the command line, thus producing a list of the system functions that are being executed by the program. The actual trace starts from the function being executed when the trace was started; a program that has "hung" during execution should show you the function the program is waiting to complete.

To trace a running program, you need to specify the Process ID (PID) of the process that you want to trace. For example, in Listing 6, the program being traced has stopped but not reported an error. Here, the `ps` tool has been used to determine the running processes (see Listing 7).

Listing 7. The `ps` tool is used to determine the running processes

```
$ ps -ef|grep errlock
mc 15779 15747  0 18:26:59 pts/2      0:00 ./errlock
mc 15742  680  0 18:26:36 pts/3      0:00 ./errlock
mc 15817 15784  0 18:28:44 pts/4      0:00 grep errlock
mc 15734  680  0 18:25:00 pts/3      0:01 /usr/bin/emacs-nox errlock.c
$ truss -p 15779
fcntl(3, F_SETLKW, 0x08047AC4) (sleeping...)
```

From this output, you can see that the `fcntl()` function has been called to set a lock on a file. In this case, the function has been set to wait until the lock has been set before continuing. Unfortunately, another process already has locked this file, so the second application will wait until the first application has finished with the file and has released the lock.

The limitation of `truss` in this instance is that you cannot tell what file is locked, or what file is currently locked and is holding up the execution of the second program. This is because the tracing process did not start until after the function call that opened the file had already been called. Both `truss` and `strace` only trace the functions as they are being executed; they cannot go back in time and work out the functions already called.

Getting a stack trace

As you have seen, `truss` can be useful when you want to monitor an entire program, but may have limited use when a program has already been started. If you are using a SVR4-based UNIX, such as Solaris or AIX, the `pstack` command may help.

The `pstack` command is actually part of a wider set of commands that output

information on running processes. Other tools in the set include pfiles, which outputs the list of files used by a process, and psig, which show the list of signals and signal handlers.

To use each command, you specify the PID of the process. The pstack command outputs the call stack for a running process, showing the list of functions called before the process reached the current function. For example, using pstack on the process waiting on a locked file produces the following from Listing 8.

Listing 8. Using pstack on the process waiting on a locked file

```
$ pstack 15828
15828: ./errlock
feef0877 fcntl      (3, 7, 8047ac4)
feedcd49 fcntl      (3, 7, 8047ac4, 8050e74) + 91
08050f10 main        (1, 8047b24, 8047b2c) + d8
08050cdc _start      (1, 8047c08, 0, 8047c12, 8047c7d, 8047c8e) + 80
```

In this case, it doesn't provide you with the information you need. Now try pfiles (see Listing 9).

Listing 9. Using pfiles

```
$ pfiles 15856
15856: ./errlock
Current rlimit: 256 file descriptors
0: S_IFCHR mode:0620 dev:292,0 ino:989038936 uid:101 gid:7 rdev:24,3
  O_RDWR|O_NOCTTY|O_LARGEFILE
  /dev/pts/3
1: S_IFCHR mode:0620 dev:292,0 ino:989038936 uid:101 gid:7 rdev:24,3
  O_RDWR|O_NOCTTY|O_LARGEFILE
  /dev/pts/3
2: S_IFCHR mode:0620 dev:292,0 ino:989038936 uid:101 gid:7 rdev:24,3
  O_RDWR|O_NOCTTY|O_LARGEFILE
  /dev/pts/3
3: S_IFREG mode:0666 dev:182,65545 ino:198 uid:101 gid:10 size:0
  O_RDWR
  advisory write lock set by process 15828
  /export/home/mc/lockdemo
```

This time the output is more useful. You can see that the file that has been opened by the process is called lockdemo, and since truss showed that the file is waiting for a lock, it is probably this file causing the problem.

Both truss and strace are examples of passive tracing. You can watch the functions as they are executed, but you can't extract more detail about what is going on, or make more detailed decisions about what to trace, and what information to output when the trace occurs.

Section 4. Dynamic tracing with DTrace

The Dynamic Tracing (DTrace) functionality built into Solaris, FreeBSD, and Mac OS X provides, as the name suggests, a more dynamic tracing environment. Unlike `truss` and similar tools, DTrace enables you to examine the internals of the programs you are running, and not just the system calls. Furthermore, with DTrace you can script the tracing of an application to customize the information you want to extract during the tracing process.

DTrace overview

DTrace combines many of the principles of the tracing that you have already seen with `truss` and `strace`, but adds a great deal of flexibility to the method and mechanism used to trace an application.

Unlike `truss` and `strace`, which list only functions within the kernel space, DTrace can be used to show the function names of the application, any libraries on which it relies, and the functions in the kernel that are called. This extends the usability of the tracing almost to the same level as a debugger. However, unlike a debugger, you cannot modify the values, and you cannot pause or otherwise alter the execution of the application. You are limited to tracing the execution, not controlling it.

The other key element of DTrace compared to both tracing and the majority of debuggers is that you can write a script that defines what to trace within the application, and information should be reported when the application executes. For example, you can specify that DTrace report only on a specific function, and that it prints out a single argument from the function call.

In addition to printing specific information, DTrace also provides built-in support for a number of utility values and functions. Within a script, for example, you can record a timestamp of when the function was called, and then compare the timestamp value with the timestamp when the function completes. By comparing the two values you can obtain the execution time for a specific function, or operation, and use the tracing information to provide execution and performance statistics.

Probes and providers

DTrace works by adding instrumentation to the system that identifies different points of execution. These points are called probes and include probes defined within the kernel, libraries, and programs. All functions within the kernel, libraries, and user programs can be identified as a probe. In addition, statically defined probes can be

used to identify special points of interest. For example, the kernel includes probes that can be used to identify when data is written to a disk. As a developer, you can add specific probes to your program to enable tracing by your users. These probes are identified as User-land Statically Defined Tracing, or USDT.

Probes are identified using the following structure:

`provider:module:function:name`, where *provider* is the name of the provider (for example, the name of a program, or a specific part of the kernel or operating system). The *module* is the kernel module or library, and the *function* is usually the function name within the module or program, and the *name* identifies the probe. The provider also has a special identifier, the PID provider, which is used to identify any running program and can be used to trace any function within a running program.

The name is usually the name of a specific probe within the system or program that has been specifically defined. DTrace also supports function boundary tracing (FBT), which enables you to trace the entry and return from any function within the kernel, library or an application. With FBT, the entry probe is triggered when the function is called, and the return probe is triggered when the function returns or completes.

Any of the components within the probe specification can either be omitted (in which case the probe specification matches all items) or with a wildcard. For example, you can specify all of the probes within a given provider using: `provider:::`. Or you can specify only the entry probes on a PID provider: `pid$target:::entry`.

The above probe specification is similar to the operation of `truss`, except that the functions will cover both the program, the operating system, and any libraries on which the program relies. This provides a much wider scope than `truss`.

When using the PID provider, the module can be used to identify the program. Without this specification, you trace every function called by the program. By specifying the program name, you limit the output to the functions defined within the program. For example, to trace the execution of the functions within the `ageindays` command, you might use: `pid$target:ageindays::`.

Any program can be traced using `dtrace`; you do not compile the program in a specific way. This means that unlike `truss`, you can gain a deeper understanding of an application even if you don't own the source to the program in question.

You can obtain a list of probes using the `dtrace` tool; the `-l` command line option will list all of the probes defined in the system: `$ dtrace -l`.

The list can be many thousands of lines long.

One-line tracing

As already noted, you can trace any application and any function within that application using DTrace. There are three ways to use dtrace from the command line: specifying a command; specifying a process ID; or specifying a named, static, probe.

Specify a command on the command-line to be executed (using `-c`), and specify the probe you want to monitor using the `-n` option (see Listing 10).

Listing 10. Specifying the probe you want to monitor

```
$ dtrace -n 'pid$target:ageindays::entry' -c './ageindays 24/1/1980 26/3/1980'
dtrace: description 'pid$target:ageindays::entry' matched 7 probes
You have been alive 62 days
You were born on 24/1/1980 which is a Thursday
dtrace: pid 15925 has exited
CPU      ID          FUNCTION:NAME
  1  57147      _start:entry
  1  57148      __fsr:entry
  1  57153      main:entry
  1  57152      check_day:entry
  1  57152      check_day:entry
  1  57151      calc_diff:entry
  1  57150      leap_year:entry
...
```

Now specify the process ID of a program already running, and the probe that you want to trace. For example, the line in Listing 11 traces calls to the syslog system by the inetd daemon.

Listing 11. Tracing calls to the syslog system

```
$ dtrace -n 'pid$target::syslog:entry { printf("%d %s", arg0, copyinstr(arg1)) }'
-p `pgrep -x inetd`
```

Or you can specify the name of a static probe, which will match all of the probes within any running process. For example, the line in Listing 12 looks for any exec functions being called (for example, when a user runs a command in a shell). Because you haven't specified a specific process, this probe will work for any call to the function by any user on the system (see Listing 12).

Listing 12. Specifying the name of a static probe

```
$ dtrace -n 'syscall::exec*:entry'
dtrace: description 'syscall::exec*:entry' matched 2 probes
CPU      ID          FUNCTION:NAME
  0  56750      exece:entry
  0  56750      exece:entry
  0  56750      exece:entry
```

In all of these examples, you have only used the basic output, which just outputs the

function or probe name, and the CPU and process ID where the probe was triggered. More extensive and selective probing and custom output is available by using the D scripting language.

Writing a DTrace script

The DTrace scripting language provides a simple mechanism for executing specific operations (called actions) when a probe is triggered by a provider. The language is limited in that you do not have the flexibility of a full language environment such as PHP or Perl, but you do have the ability to record information into variables, perform basic calculations, and support basic decision making.

A basic example that prints out the values when a specific probe is executed is shown below. The arguments to a probe or function are available using the aliases `arg0`, `arg1`, and so on for each argument. For this example, you are monitoring all of the functions starting with those open for a given process, and printing out the file opened in each case.

Listing 13. Monitoring all functions starting with open for a given process

```
#!/bin/dtrace -s
#pragma D option quiet
pid$target::open*:entry
{
    printf("Opened: %s\n", copyinstr(arg0));
}
```

You can save this text into a file called `open.t`, and then set the execute bit on the file: `$ chmod +x open.t`.

The file is now an executable script. To use it, supply the process ID with the `-p` command line option. For example, running it against a shell, you get the list of files the shell opens during execution (see Listing 14).

Listing 14. Using the `-p` option

```
$ ./open.t -p 15930
Opened: /root/.bash_path
Opened: /root/.bash_vars
Opened: /root/.bashrc
Opened: /root/.bash_aliases
...
```

A more typical use for DTrace scripts is to aggregate and summarize information, and provide counters and time differences between different operations. The probes within a DTrace script are executed sequentially within a single thread, which means

that you can monitor the execution of a series of probes. Many probes are provided in pairs, with the start probe indicating where an operation begins, and the done probe where an operation completes.

By recording the time when the start probe was triggered, and then when the probe completed, you can determine how long the operation took to complete. Within a DTrace script, the `self` variable can be used to hold the start time. For example, the script in Listing 15 uses named probes within the MySQL database system to monitor the execution time of a query.

Listing 15. Using named probes within MySQL to monitor execution time of a query

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
dtrace:::BEGIN
{
    printf("%-80s %6s\n", "Query", "Duration (ms)");
}
mysql*:::query-start
{
    self->query = copyinstr(arg5);
    self->querystart = timestamp;
}
mysql*:::query-done
{
    this->elapsed = (timestamp - self->querystart) / 1000000;
    printf("%-80s %6d\n", self->query, this->elapsed);
}
```

The trace shows a number of different elements of a typical DTrace script. First, the `BEGIN` block outputs some header information, which is useful when you are printing out tabulated data.

The `query-start` probes include a number of arguments, and the sixth argument (`arg5`) contains the full text of the original query.

When running the query on a server running MySQL, you can get useful statistics about the time taken to run each query (see Listing 16).

Listing 16. Getting statistics about the time taken to run each query

```
$ ./basic.d
Query                                     Duration (ms)
show tables                               203
select * from t1 where i < 5              131526
```

These examples have barely scratched the surface of what DTrace is capable of

achieving. See [Resources](#) for more examples and information.

Section 5. Tracing network packets

Tracing the application code can tell you what is going on within the application, and it can show you what a system is doing, but what if you want to monitor a network application and find out what information is being transferred over the network? There are many tools available, but the snoop tool in Solaris, iptrace in AIX and the Ethereal tool for many different platforms all provide the ability to view the packets going by on the network.

Network tracing basics

Nearly all of the different tools for tracing network packets work in the same basic fashion. Normally, your computer reads packets from the network and only processes and acts on the network packets that were specifically sent to your computer. With older Ethernet networks, all packets were sent to all computers. With a network switch, if you want to view packets other than those for your computer, you may need to connect to the management port on your network.

The methods for using these tools and the information they can provide differ slightly, but the fundamentals are the same. With AIX, the iptrace is a background daemon, so you must explicitly start and stop the tool to switch the process on and off. For example, to start, type: `# startsrc -s iptrace -a "-i tr0 /home/user/iptrace/log1"`. To stop the tool, type: `# stopsrc -s iptrace`.

On Solaris, the snoop tool is an application that you can execute at will: `# snoop`.

However, the number of packets on a typical network may be too much to be decoded and displayed on the fly. In this case, you can either specify a limiting scan, by setting the port name or number, or the computer name or number, or output the raw data to a file and then post-process the file. You can record the information by setting the output file: `$ snoop -o networkdump.log`.

To read in the saved data, type: `$ snoop -i networkdump.log`.

When using snoop for tracing applications, you will probably want to be more specific about your searches.

Scanning for a specific host with snoop

Typical uses for snoop during tracing are to find if an application is communicating over the network, or to find the specifics of what information is being exchanged.

For example, if you are trying to diagnose a problem between an NFS server and the NFS client, you might want to ensure that the two machines are actually exchanging information. One way to do this is to use snoop to monitor the data exchanged. On the client, you could use snoop to log the data being transferred to and from the server.

For example, Listing 17 looks for traffic to the host called bear.

Listing 17. Looking for traffic to host called bear

```
$ snoop host bear
Using device rge0 (promiscuous mode)
tweedledee.mcslp.pri -> bear.mcslp.pri TCP D=2049 S=1014 Syn Seq=1160567073
  Len=0 Win=49640 Options=<mss 1460,nop,wscale 0,nop,nop,sackOK>
bear.mcslp.pri -> tweedledee.mcslp.pri TCP D=1014 S=2049 Syn Ack=1160567074
  Seq=498630824 Len=0 Win=5840 Options=<mss 1460,nop,nop,sackOK,nop,wscale 7>
tweedledee.mcslp.pri -> bear.mcslp.pri TCP D=2049 S=1014 Ack=498630825
  Seq=1160567074 Len=0 Win=49640
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C ACCESS3 FH=EC09 (read,lookup,
  modify,extend,delete)
bear.mcslp.pri -> tweedledee.mcslp.pri TCP D=1014 S=2049 Ack=1160567230
  Seq=498630825 Len=0 Win=54
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R ACCESS3 OK (read,lookup,modify,
  extend,delete)
tweedledee.mcslp.pri -> bear.mcslp.pri TCP D=2049 S=1014 Ack=498630949
  Seq=1160567230 Len=0 Win=49640
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=EC09
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri TCP D=2049 S=1014 Ack=498631065
  Seq=1160567382 Len=0 Win=49640
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=02E4
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=8F7F
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=2764
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=FD7F
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=FF7F
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=8F7F
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=725E
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=8D7F
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=0C64
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=6CEC
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri NFS C GETATTR3 FH=5997
bear.mcslp.pri -> tweedledee.mcslp.pri NFS R GETATTR3 OK
tweedledee.mcslp.pri -> bear.mcslp.pri TCP D=2049 S=1014 Ack=498632341
  Seq=1160569142 Len=0 Win=49640
```

Fortunately, you can see that data is being exchanged. Snoop is capable of decoding the contents of the raw packets to determine what data is actually being

exchanged. In this case, the GETATTR3 network packet with NFS is being used, which gets the attribute data for a file (such as the name, size, and ownership information).

There is a lot of additional information being shown, so you could also be more explicit and select only the NFS packets. You can combine the expression used to specify what data to select by combining them like this: `$ snoop host bear and protocol nfs`.

To get more detailed information, you can use the verbose mode to output the actual data being exchanged. With the verbose mode enabled, snoop will debug each component of the packet, from the raw Ethernet header data to the individual TCP and protocol information. A single packet showing the output when filesystem state data is requested over NFS is shown in Listing 18.

Listing 18. A single packet showing the output when filesystem state data is requested over NFS

```
ETHER: ----- Ether Header -----
ETHER:
ETHER: Packet 2 arrived at 16:09:4.99083
ETHER: Packet size = 142 bytes
ETHER: Destination = 0:1a:ee:1:1:c0,
ETHER: Source       = 0:1d:60:1b:9a:2d,
ETHER: Ethertype    = 0800 (IP)
ETHER:
IP: ----- IP Header -----
IP:
IP: Version = 4
IP: Header length = 20 bytes
IP: Type of service = 0x00
IP:   xxx. .... = 0 (precedence)
IP:   ...0 .... = normal delay
IP:   .... 0... = normal throughput
IP:   .... .0.. = normal reliability
IP:   .... ..0. = not ECN capable transport
IP:   .... ...0 = no ECN congestion experienced
IP: Total length = 128 bytes
IP: Identification = 61800
IP: Flags = 0x4
IP:   .1.. .... = do not fragment
IP:   ..0. .... = last fragment
IP: Fragment offset = 0 bytes
IP: Time to live = 64 seconds/hops
IP: Protocol = 6 (TCP)
IP: Header checksum = c7b7
IP: Source address = 192.168.0.2, bear.mcslp.pri
IP: Destination address = 192.168.0.5, tweedledee.mcslp.pri
IP: No options
IP:
TCP: ----- TCP Header -----
TCP:
TCP: Source port = 2049
TCP: Destination port = 1013 (Sun RPC)
TCP: Sequence number = 2161119694
TCP: Acknowledgement number = 1253508400
TCP: Data offset = 20 bytes
TCP: Flags = 0x18
TCP:   0... .... = No ECN congestion window reduced
```

```

TCP:      .0.. .... = No ECN echo
TCP:      ..0. .... = No urgent pointer
TCP:      ...1 .... = Acknowledgement
TCP:      .... 1... = Push
TCP:      .... .0.. = No reset
TCP:      .... ..0. = No Syn
TCP:      .... ...0 = No Fin
TCP:      Window = 348
TCP:      Checksum = 0xec08
TCP:      Urgent pointer = 0
TCP:      No options
TCP:
RPC:      ----- SUN RPC Header -----
RPC:
RPC:      Record Mark: last fragment, length = 84
RPC:      Transaction id = 485864481
RPC:      Type = 1 (Reply)
RPC:      This is a reply to frame 1
RPC:      Status = 0 (Accepted)
RPC:      Verifier   : Flavor = 0 (None), len = 0 bytes
RPC:      Accept status = 0 (Success)
RPC:
NFS:      ----- Sun NFS -----
NFS:
NFS:      Proc = 18 (Get filesystem statistics)
NFS:      Status = 0 (OK)
NFS:      Post-operation attributes: (not available)
NFS:      Total space = 488217268224 bytes
NFS:      Available space = 137675571200 bytes
NFS:      Available space - this user = 112875532288 bytes
NFS:      Total file slots = 60555264
NFS:      Available file slots = 58563011
NFS:      Available file slots - this user = 58563011
NFS:      Invariant time = 0 sec
NFS:

```

Looking for specific information can also be useful if you want to determine what different applications are doing. For example, you may want to monitor what Web sites are being accessed from a host, which you can do by looking for the HTTP protocol and only outputting the HTTP lines from the snoop output (see Listing 19).

Listing 19. Monitoring what Web sties are being accessed from a host

```

$ snoop -v port 80 |egrep '^HTTP'
...
HTTP: ----- HyperText Transfer Protocol -----
HTTP:
HTTP: GET / HTTP/1.0
HTTP: User-Agent: Wget/1.10.2
HTTP: Accept: */*
HTTP: Host: www.bbc.co.uk
HTTP: Connection: Keep-Alive
HTTP: [...]
...

```

The `egrep` selects only the lines starting with HTTP. Here we can see somebody has accessed the BBC Web site.

Section 6. Summary

Conclusion

Tracing, and the need to trace your application, can come from many different areas. As a developer, tracing an application can be a quicker alternative than resorting to a full debugger when you are trying to diagnose a problem. Tools like DTrace can be even more intrusive and retrieve some very specific, and extensive, information about your application.

For administrators, tracing provides the only method for locating and finding the information that you need to diagnose problems. Without access to the source code, tracing is often your only means of determining what an application is doing. This tutorial showed that with tracing alone, you can find and diagnose problems with just a few commands.

Downloads

Description	Name	Size	Download method
ageindays application	ageindays.zip	5KB	HTTP

[Information about download methods](#)

Resources

- [System Administration Toolkit: Network Scanning](#) (Martin Brown, developerWorks, December 2007): Get more tips on network scanning.
- [Solve application problems with tracing](#) (developerWorks): Get information on using truss, trace, and similar tools.
- For more detailed information on using DTrace, see <http://docs.sun.com/app/docs/doc/819-5488>.
- Read [System Administration Toolkit: Standardizing your UNIX command-line tools](#) (Martin Brown, developerWorks, May 2006) to learn how to use the same command across multiplied machines.
- For an article series that will teach you how to program in bash, see [Bash by example, Part 1: Fundamental programming in the Bourne again shell \(bash\)](#) (Daniel Robbins, developerWorks, March 2000), [Bash by example, Part 2: More bash programming fundamentals](#) (Daniel Robbins, developerWorks, April 2000), and [Bash by example, Part 3: Exploring the ebuild system](#) (Daniel Robbins, developerWorks, May 2000).
- [System Administration Toolkit](#): Check out other parts in this series.
- [Making Unix and Linux work together](#) (Martin Brown, developerWorks, April 2006) is a guide to getting traditional Unix distributions and Linux working together.
- Different systems use different tools, and the IBM Redbook [Solaris to Linux Migration: A Guide for System Administrators](#) will help you identify some key tools.
- [Exploring the Linux memory model](#) (Vikram Shukla, developerWorks, January 2006) helps you understand how Linux uses memory, swap space and exchanges pages and processes between the two.
- [New to AIX and UNIX](#): Visit the New to AIX and UNIX page to learn more about AIX and UNIX.
- The [developerWorks AIX and UNIX zone](#) hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials.
- [AIX](#): A collaborative environment for technical information related to AIX.
- [Technology bookstore](#) Browse this site for books and other technical topics.
- [developerWorks technical events and webcasts](#): Stay current with developerWorks technical events and webcasts.
- [Podcasts](#): Tune in and catch up with IBM technical experts.

About the author

Martin Brown

Martin Brown has been a professional writer for over eight years. He is the author of numerous books and articles across a range of topics. His expertise spans myriad development languages and platforms -- Perl, Python, Java, JavaScript, Basic, Pascal, Modula-2, C, C++, Rebol, Gawk, Shellscrip, Windows, Solaris, Linux, BeOS, Mac OS/X and more -- as well as Web programming, systems management and integration. Martin is a regular contributor to ServerWatch.com, LinuxToday.com and IBM developerWorks, and a regular blogger at Computerworld, The Apple Blog and other sites, as well as a Subject Matter Expert (SME) for Microsoft. He can be contacted through his Web site at <http://www.mcslp.com>.