Oracle Blog

Adam Leventhal's Weblog

inside the sausage factory

« OpenSolaris on LugRa... | Main | DTrace in ACM Queue »

DTrace for Linux

By ahl on Dec 13, 2005

With BrandZ, it's now possible to use DTrace on Linux applications. For the uninitiated, DTrace is the dynamic tracing facility in OpenSolaris; it allows for **systemic** analysis of a scope and precision unequalled in the industry. With DTrace, administrators and developers can trace low level services like I/O and scheduling, up the system stack through kernel functions calls, system calls, and system library calls, and into applications written in C and C++ or any of a host of dynamic languages like Java, Ruby, Perl or php. One of my contributions to BrandZ was to extend DTrace support for Linux binaries executed in a branded Zone.

DTrace has several different **instrumentation providers** that know how to instrument a particular part of the system and provide relevant probes for that component. The io provider lets you trace disk I/O, the fbt (function boundary tracing) provider lets you trace any kernel function call, etc. A typical system will start with more than 30,000 probes but providers can create probes dynamically to trace new kernel modules or user-land processes. When strictly focused on a user-land application, the most useful providers are typically the syscall provider to examine system calls and the pid provider that can trace any instruction in a any process executing on the system.

For Linux processes, the pid provider just worked (well, once Russ built a library to understand the Linux run-time linker), and we introduced a new provider -- the **lx-syscall provider** -- to trace entry and return for emulated Linux system calls. With these providers it's possible to understand every facet of a Linux application's behavior and with the other DTrace probes it's possible to reason about an application's use of system resources. In other words, you can take that sluggish Linux application, stick it in a branded Zone, dissect it using Solaris tools, and then bring it back to a native Linux system with the fruits of your DTrace investigation[1].

To give an example of using DTrace on Linux applications, I needed an application to examine. I wanted a well known program that either didn't run on Solaris or operated sufficiently differently such examining the Linux version rather than the Solaris port made sense. I decided on `/usr/bin/top` partly because of the dramatic differences between how it operates on Linux vs. Solaris (due to the differences in /proc), but mostly because of what I've heard my colleague, Bryan, refer to as the "top problem": your system is slow, so you run `top`. What's the top process? Top!

Running `top` in the Linux branded zone, I opened a shell in the global (Solaris) zone to use DTrace. I started as I do on Solaris applications: I looked at system calls. I was interested to see which system calls were being executed most frequently which is easily expressed in DTrace:

```
bash-3.00# dtrace -n lx-syscall:::entry'/execname == "top"/{ @[probefunc] = count(); }'
dtrace: description 'lx-syscall:::entry' matched 272 probes
\^C

  fstat64                                                          322
  access                                                           323
  gettimeofday                                                     323
  gtime                                                            323
  llseek                                                           323
  mmap2                                                            323
  munmap                                                           323

  select                                                           323
  getdents64                                                      1289
  lseek                                                           1291
  stat64                                                          3545
  rt_sigaction                                                    5805
  write                                                           6459
  fcntl64                                                         6772
  alarm                                                           8708
  close                                                          11282
  open                                                           14827
  read                                                           14830
```

Note the use of the aggregation denoted with the '@'. Aggregations are the mechanism by which DTrace allows users to examine **patterns** of system behavior rather than examining each individual datum -- each system call for example. (In case you also noticed the strange discrepancy between the number of open and close system calls, many of those opens are failing so it makes sense that they would have no corresponding close. I used the lx-syscall provider to suss this out, but I omitted that investigation in a vain appeal for brevity.)

There may well be something fishy about this output, but nothing struck me as so compellingly fishy to explore immediately. Instead, I fired up vi and wrote a short D script to see which system calls were taking the most time:

lx-sys.d

```
#!/usr/sbin/dtrace -s

lx-syscall:::entry
/execname == "top"/
{
        self->ts = vtimestamp;
}

lx-syscall:::return
/self->ts/
{
        @[probefunc] = sum(vtimestamp - self->ts);
        self->ts = 0;
}
```

This script creates a table of system calls and the time spent in them (in nanoseconds). The results were fairly interesting.

```
bash-3.00# ./lx-sys.d
dtrace: script './lx-sys.d' matched 550 probes
\^C
```

```
  llseek                                                     4940978
  gtime                                                      5993454
  gettimeofday                                               6603844
  fstat64                                                   14217312
  select                                                    26594875
  lseek                                                     30956093
  mmap2                                                     43463946
  access                                                    49033498
  alarm                                                     72216971
  fcntl64                                                  188281402
  rt_sigaction                                             197646176
  stat64                                                   268188885
  close                                                    417574118
  getdents64                                               781844851
  open                                                    1314209040
  read                                                    1862007391
  write                                                   2030173630
  munmap                                                  2195846497
```

That seems like a lot of time spent in **munmap** for top. In fact, I'm rather surprised that there's any mapping and unmapping going on at all (I guess that should have raised an eyebrow after my initial system call count). Unmapping memory is a pretty expensive operation that gets more expensive on bigger systems as it requires the kernel to do some work on **every** CPU to completely wipe out the mapping.

I then modified lx-sys.d to record the total amount of time top spent on the CPU and the total amount of time spent in system calls to see how large a chunk of time these seemingly expensive unmap operations were taking:

lx-sys2.d

```
#!/usr/sbin/dtrace -s

lx-syscall:::entry
/execname == "top"/
{
        self->ts = vtimestamp;
}

lx-syscall:::return
/self->ts/
{
        @[probefunc] = sum(vtimestamp - self->ts);
        @["- all syscalls -"] = sum(vtimestamp - self->ts);
        self->ts = 0;
}

sched:::on-cpu
/execname == "top"/
{
        self->on = timestamp;
}

sched:::off-cpu
/self->on/
{
        @["- total -"] = sum(timestamp - self->on);
        self->on = 0;
}
```

I used the sched provider to see when top was going on and off of the CPU, and I added a row to record the

total time spent in all system call. Here were the results (keep in mind I was just hitting \^C to end the experiment after a few seconds so it's expected that these numbers would be different from those above; there are ways to have more accurately timed experiments):

```
bash-3.00# ./lx-sys2.d
dtrace: script './lx-sys2.d' matched 550 probes
\^C

  llseek                                                         939771
  gtime                                                         1088745
  gettimeofday                                                  1090020
  fstat64                                                       2494614
  select                                                        4566569
  lseek                                                         5186943
  mmap2                                                         7300830
  access                                                        8587484
  alarm                                                        11671436
  fcntl64                                                      31147636
  rt_sigaction                                                 33207341
  stat64                                                       45223200
  close                                                        69338595
  getdents64                                                  131196732
  open                                                        220188139
  read                                                        309764996
  write                                                       340413183
  munmap                                                      365830103
  - all syscalls -                                           1589236337
  - total -                                                  3258101690
```

So system calls are consuming roughly half of top's time on the CPU and the munmap syscall is consuming roughly a quarter of that. This was enough to convince me that there was probably room for improvement and further investigation might bear fruit.

Next, I wanted to understand what this mapped memory was being used for so I wrote a little script that traces all the functions called in the process between when memory is mapped using the mmap2(2) system call and when it's unmapped and returned to the system through the munmap(2) system call:

map.d

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

lx-syscall::mmap2:return
/pid == $target/
{
        self->ptr = arg1;
        self->depth = 10;

        printf("%\*.s <- %s syscall\\n", self->depth, "", probefunc);
}

pid$target:::entry
/self->ptr/
{
        self->depth++;
        printf("%\*.s -> %s`%s\\n", self->depth, "", probemod, probefunc);
}

pid$target:::return
/self->ptr/
```

```
{
        printf("%\*.s <- %s`%s\\n", self->depth, "", probemod, probefunc);
        self->depth--;
}

lx-syscall::munmap:entry
/arg0 == self->ptr/
{
        self->depth++;
        printf("%\*.s -> %s syscall\\n", self->depth, "", probefunc);

        self->ptr = 0;
        self->depth = 0;
        exit(0);
}
```

This script uses the $target variable which means that we need to run it with the -p option where is the process ID of top. When mmap2 returns, we set a thread local variable, 'ptr', which stores the address at the base of the mapped region; for every function entry and return in the process we call printf() if `self->ptr` is set; finally, we exit DTrace when munmap is called with that same address. Here are the results:

```
bash-3.00# ./map.d -p `pgrep top`
           <- mmap2 syscall
           <- LM2`libc.so.1`syscall
          <- LM2`lx_brand.so.1`lx_emulate
         <- LM2`lx_brand.so.1`lx_handler
        <- libc.so.6`mmap
       <- libc.so.6`malloc
       -> libc.so.6`memset
       <- libc.so.6`memset
       -> libc.so.6`cfree
       <- libc.so.6`cfree
       -> libc.so.6`munmap
       <- LM2`lx_brand.so.1`lx_handler_table
       -> LM2`lx_brand.so.1`lx_handler
        -> LM2`lx_brand.so.1`lx_emulate
         -> LM2`libc.so.1`syscall
          -> munmap syscall
```

I traced the probemod (shared object name) in addition to probefunc (function name) so that we'd be able to differentiate proper Linux functions (in this case all in libc.so.6) from functions in the emulation library (LM2`lx_brand.so.1). What we can glean from this is that the mmap call is a result of a call to malloc() and the unmap is due to a call to free(). What's unfortunate is that we're not seeing any function calls in top itself. For some reason, the top developer chose to strip this binary (presumably to save precious 2k the symbol table would have used on disk) so we're stuck with no symbolic information for top's functions and no ability to trace the individual function calls[2], but we can still reason about this a bit more.

A little analysis in mdb revealed that cfree (an alias for free) makes a tail-call to a function that calls munmap. It seems strange to me that freeing memory would immediately results in an unmap operation (since it would cause exactly the high overhead we're seeing here. To explore this, I wrote another script which looks at what proportion of calls to malloc result in a call to mmap():

malloc.d

```
#!/usr/sbin/dtrace -s

pid$target::malloc:entry
{
        self->follow = arg0;
}
```

```
lx-syscall::mmap2:entry
/self->follow/
{
        @["mapped"] = count();
        self->follow = 0;
}

pid$target::malloc:return
/self->follow/
{
        @["no map"] = count();
        self->follow = 0;
}
```

Here are the results:

```
bash-3.00# ./malloc.d -p `pgrep top`
dtrace: script './malloc.d' matched 11 probes
\^C

  mapped                                                        275
  no map                                                       3024
```

So a bunch of allocations result in a mmap, but not a huge number. Next I decided to explore if there might be a correlation between the size of the allocation and whether or not it resulted in a call to mmap using the following script:

malloc2.d

```
#!/usr/sbin/dtrace -s

pid$target::malloc:entry
{
        self->size = arg0;
}

lx-syscall::mmap2:entry
/self->size/
{
        @["mapped"] = quantize(self->size);
        self->size = 0;
}

pid$target::malloc:return
/self->size/
{
        @["no map"] = quantize(self->size);
        self->size = 0;
}
```

Rather than just counting the frequency, I used the quantize aggregating action to built a power-of-two histogram on the number of bytes being allocated (`self->size`). The output was quite illustrative:

```
bash-3.00# ./malloc2.d -p `pgrep top`
dtrace: script './malloc2.d' matched 11 probes
\^C

  no map
           value  ------------- Distribution ------------- count
               2 |                                         0
               4 |@@@@@@@                                  426
               8 |@@@@@@@@@@@@@@@                          852
              16 |@@@@@@@@@@@                              639
              32 |@@@@                                     213
              64 |                                         0
             128 |                                         0
             256 |                                         0
             512 |@@@@                                     213
            1024 |                                         0

  mapped
           value  ------------- Distribution ------------- count
          131072 |                                         0
          262144 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 213
          524288 |                                         0
```

All the allocations that required a mmap were **huge** -- between 256k and 512k. Now it makes sense why the Linux libc allocator would treat these allocations a little differently than reasonably sized allocations. And this is clearly a smoking gun for top performance: it would do much better to preallocate a huge buffer and grow it as needed (assuming it actually needs it at all) than to malloc it each time. Tracking down the offending line of code would just require a non-stripped binary and a little DTrace invocation like this:

```
# dtrace -n pid`pgrep top`::malloc:entry'/arg0 >= 262144/{@[ustack()] = count()}'
```

From symptoms to root cause on a Linux application in a few DTrace scripts -- and it took me approximately 1000 times longer to cobble together some vaguely coherent prose describing the scripts than it did for me to actually perform the investigation. BrandZ opens up some pretty interesting new vistas for DTrace. I look forward to seeing Linux applications being brought in for tune-ups on BrandZ and then reaping those benefits either back on their mother Linux or sticking around to enjoy the fault management, ZFS, scalability, and, of course, continued access to DTrace in BrandZ.

---

[1] Of course, results may vary since the guts of the Linux kernel differ significantly from those of the Solaris kernel, but they're often fairly similar. I/O or scheduling problems will be slightly different, but often not *so* different that the conclusions lack applicability.
[2] Actually, we **can** still trace function calls -- in fact, we can trace any instruction -- but it takes something of a heroic effort. We could disassemble parts of top to identify calls sites and then use esoteric `pid123::-:`*`address`* probe format to trace the stripped function. I said we could do it; I never said it would be pretty.

Technorati Tags: BrandZ DTrace Solaris OpenSolaris

Category: DTrace

Tags: none

Permanent link to this entry

« OpenSolaris on LugRa... | Main | DTrace in ACM Queue »
Comments:

Awsome! I've been waiting to try out dtrace for a long time now. Thanks!

Posted by **Andreas Bruse** on December 14, 2005 at 12:12 AM PST #

Rather than jump to profiling an application I'd much perfer to see the results of a three way comparison of the libMicro benchmark running on the same machine, Solaris/Linux/Linux_translated. At least this will give us an idea of how much overhead the translator has on system calls in a controlled condition. If there is a large anomaly a dtrace session would be interesting. ---Bob palowoda@fiver.net

Posted by **Bob Palowoda** on December 14, 2005 at 12:49 PM PST #

Bob,
That would be a different investigation entirely -- one I've started to do and will continue to do, no doubt -- but the point was to demonstrate how performance of a Linux application could be improved not that of the emulation environment.

Posted by **Adam Leventhal** on December 14, 2005 at 03:33 PM PST #

Agreed :) Its nice to see dtrace coming to linux. My only problem currently, is that the documentation for installing Brandz on a linux box is not very clear and i am somewhat confused on what to do with the required 2 packages on a Fedora Core server.

Posted by **Michael Weiner** on December 14, 2005 at 09:35 PM PST #

Adam, This is great work, thanks. Couple of observations: Looks like Linux is going to have a tool to do the similar analysis natively. Here is the link to their version of the script you tried. http://sourceware.org/ml/systemtap/2005-q4/msg00437.html Looking at the Linux native analysis, the problem you have experienced doesn't seem to be there when ran natively. That brings to my mind a question, is it really true that Solaris virtually environment and native Linux are so close that one can do performance problem analysis in one environment and apply in the other. Your thoughts on this and if possibly explanation of the difference in results would be appreciated. (I understand that the details of the machine configuration and software versions are missing hence that might be the difference in the results.)

Posted by **Johnd** on December 16, 2005 at 03:21 AM PST #

Michael,
If there's something in particular that needs clarification, could you post a question to brandz-discuss@opensolaris.org? We're looking for feedback from the community.

Johnd,
As you point out, differences in the hardware and software configurations could account for the difference in the results. It's possible that the performance problem could be exagerated in the BrandZ environment, but this result is still valid in that mapping and unmapping memory is surely not the most efficient way for the allocator to behave in this situation. We don't have much data to support this claim, but I do think analysis under BrandZ will yeild results which are applicable on a native Linux system. The results will tend to be more accurate for investigations that deal strictly with application code rather than interactions with the kernel. What I didn't see in that email thread you referenced was any sort of analysis of the user-land flow control (map.d and malloc.d above). Being able to follow your investigation accross the user/kernel boundary and into applications (in a bunch of different languages) is one of the key differences between DTrace and every other tracing framework. Without user-land tracing even if their analysis of the system call timing had matched mine, it's not clear that they would have been able to drive to the root cause.

Posted by **Adam Leventhal** on December 18, 2005 at 03:45 PM PST #

I havnt seen anything here that wouldnt have been possible with a combination of strace and ltrace. And I wouldnt have had to write any scripts to use them either. Congratulations on writing a cool utility, but this stuff is nothing new, and certainly not only possible on solaris.

Posted by **John** on December 18, 2005 at 04:58 PM PST #

John,
Perhaps you're right about these specific examples, but DTrace lets you do far more than strace and ltrace. For example, you can mix data from the kernel, system libraries and Java in a single stream as I describe here. You can't do that in Linux or on any other operating system for that matter. And that's just one example. You may want to understand a bit more about what DTrace is capable of before you discount it.

Posted by **Adam Leventhal** on December 18, 2005 at 08:59 PM PST #

To avoid a stripped binary, build and install procps like this:

<tt>make CFLAGS=-g3 install</tt>

I think I have a fairly good handle on problems actually seen on Linux. The munmap() problem does not exist; the name "Slowaris" exists for a reason. (be thankful, because performance problems give Sun a reason to pay your salary)

Every system has it's cruddy part though, and /proc is surely where Linux collects cruft.

What top is required to do is quite ugly:

First, it must support up to 4 million tasks. Probably top can only do this at a reduced refresh rate, but anyway, this means that top can't leave the file descriptors open.

The per-task info is about 400 to 600 bytes. This is not nice to the cache or TLB. At only 1000 tasks, that's half a meg of RAM to cache and a full 128 TLB entries. Ouch. That's not anywhere near 4 million tasks! Just playing around with the order of fields in a struct can make a huge difference for top.

Then there is the problem of doing lots of 64-bit division on a register-starved 32-bit processor. This happens on both sides of the user/kernel boundry, creating and parsing numbers as ASCII decimal. Timestamps also require 64-bit division.

A custom stack-like allocator would help. The glibc obstacks stuff would work great. (hey, add that to Solaris and send it off to POSIX to be set in stone as a standard)

Smaller data structures would sure help, in the short run at least, but top may be getting support for more data columns in the future. This also reduces code sharing with the ps program, making maintenence hard.

Posted by **Albert Cahalan** on December 20, 2005 at 12:51 PM PST #

Post a Comment:
Comments are closed for this entry.

About

Adam Leventhal, Fishworks engineer

Search

Enter search term:

☑ Search only this blog

Recent Posts

- Leaving Oracle
- Fishworks history of SSDs
- Farewell to Bryan Cantrill
- What is RAID-Z?
- A Logzilla for your ZFS box
- 2010.Q1 simulator
- The need for triple-parity RAID
- Logzillas: to mirror or stripe?
- 2009.Q3 Storage Configuration
- Flash Memory Summit 2009

Top Tags

- 2009.q3
- apple
- arc
- asplos
- asplos2009
- cacm
- cddl
- ddrdrive
- dtrace
- dtrace.conf
- firefox
- fishworks
- flash
- gpl
- gzip
- hybridstoragepool
- iscsi
- itunes
- java
- javaone
- javaone2005
- knockoff
- l2arc
- leopard
- linux
- logzilla
- mac
- memory
- mozilla
- nand
- netapp
- opensolaris
- os
- oscon
- pid
- pid2proc
- port
- proc

- qa
- raid
- raid-z
- raidz
- raidz2
- ruby
- scoble
- ssd
- sunstorage7000
- systemtap
- x
- zfs

Categories

- DTrace
- Fishworks
- OpenSolaris
- Other
- Software
- ZFS

Archives

« March 2014

Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Today

DTrace Links

- InfoWorld Innovators 2005
- DTrace Home
- DTrace Schedule
- Discussion Forum
- Expert Exchange
- LinuxInsider story
- NY Times Solaris story
- Register story
- SysAdmin story
- eWEEK story
- sun.com story

Fishworks Engineering

- Bill Pijewski
- Brendan Gregg
- Bryan Cantrill
- Cindi McGuire
- Dave Pacheco
- Eric Schrock
- Greg Price
- Keith Wesolowski
- Mike Shapiro
- Todd Patrick

The DTrace Three

- Adam Leventhal
- Bryan Cantrill
- Mike Shapiro

Menu

- Blogs Home
- Weblog
- DTrace Schedule
- Login

Feeds

RSS

- All
- /DTrace
- /Fishworks
- /OpenSolaris
- /Other
- /Software
- /ZFS
- Comments

Atom

- All
- /DTrace
- /Fishworks
- /OpenSolaris
- /Other
- /Software
- /ZFS
- Comments