DTrace Topics:
Introduction

Brendan Gregg
Sun Microsystems
April 2007
DTrace Topics: Introduction

• This presentation is an introduction to DTrace, and is part of the “DTrace Topics” collection.
  > Difficulty: ☕
  > Audience: Everyone

• These slides cover:
  > What is DTrace
  > What is DTrace for
  > Who uses DTrace
  > DTrace Essentials
  > Usage Features
What is DTrace

- DTrace is a dynamic troubleshooting and analysis tool first introduced in the Solaris 10 and OpenSolaris operating systems.
- DTrace is many things, in particular:
  > A tool
  > A programming language interpreter
  > An instrumentation framework
- DTrace provides observability across the entire software stack from one tool. This allows you to examine software execution like never before.
DTrace example #1

- Tracing new processes system-wide,

```
# dtrace -n 'syscall::exece:return { trace(execname); }'
dtrace: description 'syscall::exece:return ' matched 1 probe

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return man</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return sh</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return neqn</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return tbl</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return nroff</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return col</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return sh</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return mv</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return sh</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return more</td>
</tr>
</tbody>
</table>
```

System calls are only one layer of the software stack.
The Entire Software Stack

- How did you analyze these?

Examples:
- Java, JavaScript, ...
- /usr/bin/*
- /usr/lib/*
- man -s2
- VFS, DNLC, UFS, ZFS, TCP, IP, ...
- sd, st, hme, eri, ...
- disk data controller
The Entire Software Stack

- It was possible, but difficult:

  Dynamic Languages
  User Executable
  Libraries
 Syscall Interface

  Kernel
  Device Drivers
  Scheduler
  File Systems

  Memory allocation

Previously:
- debuggers
- truss -ua.out
- apptrace, sotruss
- truss
- prex; tnf*
- lockstat
- mdb
- kstat, PICs, guesswork
The Entire Software Stack

• DTrace is all seeing:

- Dynamic Languages
- User Executable
- Libraries
- Syscall Interface
- Kernel
- Device Drivers
- Scheduler
- Memory allocation
- File Systems
- Hardware

DTrace visibility:
- Yes, with providers
- Yes
- Yes
- Yes
- Yes
- No. Indirectly, yes
What DTrace is like

- DTrace has the combined capabilities of numerous previous tools and more:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>truss -ua.out</td>
<td>tracing user functions</td>
</tr>
<tr>
<td>apptrace</td>
<td>tracing library calls</td>
</tr>
<tr>
<td>truss</td>
<td>tracing system calls</td>
</tr>
<tr>
<td>prex; tnf*</td>
<td>tracing some kernel functions</td>
</tr>
<tr>
<td>lockstat</td>
<td>profiling the kernel</td>
</tr>
<tr>
<td>mdb -k</td>
<td>accessing kernel VM</td>
</tr>
<tr>
<td>mdb -p</td>
<td>accessing process VM</td>
</tr>
</tbody>
</table>

Plus a programming language similar to C and awk.
Syscall Example

- Using truss:

```bash
$ truss date
execve("/usr/bin/date", 0x08047C9C, 0x08047CA4) argc = 1
resolvepath("/usr/lib/ld.so.1", "/lib/ld.so.1", 1023) = 12
resolvepath("/usr/bin/date", "/usr/bin/date", 1023) = 13
xstat(2, "/usr/bin/date", 0x08047A58) = 0
open("/var/ld/ld.config", O_RDONLY) = 3
fxstat(2, 3, 0x08047988) = 0
mmap(0x00000000, 152, PROT_READ, MAP_SHARED, 3, 0) = 0xFEFB0000
close(3) = 0
mmap(0x00000000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, -1
sysconfig(_CONFIG_PAGESIZE) = 4096
[...]
```

truss slows down the target

Only examine 1 process

Output is limited to provided options
# Syscall Example

## Using DTrace:

```bash
# dtrace -n 'syscall:::entry { printf("%16s %x %x", execname, arg0, arg1); }'
dtrace: description 'syscall:::entry ' matched 233 probes

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
<th>NAME</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75943</td>
<td>read:entry</td>
<td>Xorg f</td>
<td>8047130</td>
</tr>
<tr>
<td>1</td>
<td>76211</td>
<td>setitimer:entry</td>
<td>Xorg 0</td>
<td>8047610</td>
</tr>
<tr>
<td>1</td>
<td>76143</td>
<td>writev:entry</td>
<td>Xorg 22</td>
<td>80477f8</td>
</tr>
<tr>
<td>1</td>
<td>76255</td>
<td>pollsys:entry</td>
<td>Xorg 8046da0 1a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>75943</td>
<td>read:entry</td>
<td>Xorg 22</td>
<td>85121b0</td>
</tr>
<tr>
<td>1</td>
<td>76035</td>
<td>ioctl:entry</td>
<td>soffice.bin 6 5301</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>76035</td>
<td>ioctl:entry</td>
<td>soffice.bin 6 5301</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>76255</td>
<td>pollsys:entry</td>
<td>soffice.bin 8047530 2</td>
<td></td>
</tr>
</tbody>
</table>
```

- **Minimum performance cost**
- **Watch every process**
What is DTrace for

• Troubleshooting software bugs
  > Proving what the problem is, and isn't.
  > Measuring the magnitude of the problem.

• Detailed observability
  > Observing devices, such as disk or network activity.
  > Observing applications, whether they are from Solaris, 3rd party, or in-house.

• Capturing profiling data for performance analysis
  > If there is latency somewhere, DTrace can find it
What isn't DTrace

• DTrace isn't a replacement for kstat or SMNP
  > kstat already provides inexpensive long term monitoring.
• DTrace isn't sentient, it needs to borrow your brain to do the thinking
• DTrace isn't “dTrace”
Who is DTrace for

• Application Developers
  > Fetch in-flight profiling data without restarting the apps, even on customer production servers.
  > Detailed visibility of all the functions that they wrote, and the rest of the software stack.
  > Add static probes as a stable debug interface.

• Application Support
  > Provides a comprehensive insight into application behavior.
  > Analyze faults and root-cause performance issues.
  > Prove where issues are, and measure their magnitude.
Who is DTrace for

• System Administrators
  > Troubleshoot, analyze, investigate where never before.
  > See more of your system - fills in many observability gaps.

• Database Administrators
  > Analyze throughput performance issues across all system components.

• Security Administrators
  > Customized short-term auditing
  > Malware deciphering
Who is DTrace for

• Kernel Engineers
  > Fetch kernel trace data from almost every function.
  > Function arguments are auto-casted providing access to all struct members.
  > Fetch nanosecond timestamps for function execution.
  > Troubleshoot device drivers, including during boot.
  > Add statically defined trace points for debugging.
How to use DTrace

• DTrace can be used by either:
  > Running prewritten one-liners and scripts
    – DTrace one-liners are easy to use and often useful, http://www.solarisinternals.com/dtrace
    – The DtraceToolkit contains over 100 scripts ready to run, http://www.opensolaris.org/os/community/dtrace/dtracetoolkit
  > Writing your own one-liners and scripts
    – Encouraged – the possibilities are endless
    – It helps to know C
    – It can help to know operating system fundamentals
DTrace wins

• Finding unnecessary work
  > Having deep visibility often finds work being performed that isn't needed. Eliminating these can produce the biggest DTrace wins – 2x, 20x, etc.

• Solving performance issues
  > Being able to measure where the latencies are, and show what their costs are. These can produce typical performance wins – 5%, 10%, etc.
DTrace wins

• Finding bugs
  > Many bugs are found though static debug frameworks; DTrace is a dynamic framework that allows custom and comprehensive debug info to be fetched when needed.

• Proving performance issues
  > Many valuable DTrace wins have no immediate percent improvement, they are about gathering evidence to prove the existence and magnitude of issues.
Example scenario: The past

• Take a performance issue on a complex customer system,

  Customer:
  “Why is our system slow?”

• With previous observability tools, customers could often find problems but not take the measurements needed to prove that they found the problem.

> What is the latency cost for this issue? As a percent?
Example scenario: The past

- Application Vendor: “The real problem may be the database.”
- Database Vendor: “The real problem may be the OS.”
- OS Vendor: “The real problem may be the application.”

• The “blame wheel”
Example scenario: The past

- The lack of proof can mean stalemate.

Customer:
“I think I've found the issue in the application code.”

Application Vendor:
“That issue is costly to fix. We are happy to fix it, so long as you can prove that this is the issue.”
Example scenario: The future

A happy ending

• With DTrace, all players can examine all of the software themselves.

  Customer:
  “I measured the problem, it is in the application.”

  Application Vendor:
  “I'd better fix that right away.”

  Example: “80% of the average transaction time is spent in the application waiting for user-level locks.”
Example scenario: The future

An alternate happy ending for application vendors

Application Vendor: “We measured the problem and found it was in the OS.”

OS Vendor: “We'd better fix that right away.”

– Example: “80% of our average transaction time is consumed by a bug in libc.”
Answers to initial questions

- DTrace is not available for Solaris 9.
- You need to be root, or have the correct privileges, to run /usr/sbin/dtrace.
- There is a GUI called chime.
- DTrace is safe for production use, provided you don't deliberately try to cause harm.
- DTrace has low impact when in use, and zero impact when not.
What's next:

• We just covered:
  > *What is DTrace*
  > *What is DTrace for*
  > *Who uses DTrace*

• Next up is:
  > DTrace Essentials
  > Usage Features
Terminology

- **Example #1**

```bash
# dtrace -n 'syscall::exece:return { trace(execname); }'
dtrace: description 'syscall::exece:return' matched 1 probe

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return man</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return sh</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return neqn</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return tbl</td>
</tr>
<tr>
<td>0</td>
<td>76044</td>
<td>exece:return nroff</td>
</tr>
</tbody>
</table>
```

---

**Diagram:**

```
consumer

probe

action
```

- consumer
- probe
- action

```bash
# dtrace -n 'syscall::exece:return { trace(execname); }'
dtrace: description 'syscall::exece:return' matched 1 probe
```
Consumer

- Consumers of libdtrace(3LIB),
  - `dtrace` command line and scripting interface
  - `lockstat` kernel lock statistics
  - `plockstat` user-level lock statistics
  - `intrstat` run-time interrupt statistics

- libdtrace is currently a private interface and not to be used directly (nor is there any great reason to); the supported interface is `dtrace(1M)`.

  > NOTE: You are still encouraged to use `libkstat(3LIB)` and `proc(4)` directly, rather than wrapping `/usr/bin` consumers.
Privileges

- Non-root users need certain DTrace privileges to be able to use DTrace.
- These privileges are from the Solaris 10 “Least Privilege” feature.

```
$ id
uid=1001(user1) gid=1(other)
$ /usr/sbin/dtrace -n 'syscall::exece::return'
dtrace: failed to initialize dtrace: DTrace requires additional privileges
```
Probes

- Data is generated from instrumentation points called “probes”.
- DTrace provides thousands of probes.
- Probe examples:

<table>
<thead>
<tr>
<th>Probe Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syscall::read::entry</td>
<td>A read() syscall began</td>
</tr>
<tr>
<td>proc:::exec-success</td>
<td>A process created successfully</td>
</tr>
<tr>
<td>io:::start</td>
<td>An I/O was issued (disk/vol/NFS)</td>
</tr>
<tr>
<td>io:::done</td>
<td>An I/O completed</td>
</tr>
</tbody>
</table>
Probe Names

• Probe names are a four-tuple:

Provider    Module    Function    Name

syscall::exece::return

> Provider      A library of related probes.
> Module        The module the function belongs to, either a kernel module or user segment.
> Function      The function name that contains the probe.
> Name          The name of the probe.
Listing Probes

- `dtrace -l` lists all currently available probes that you have privilege to see, with one probe per line:

  ```
  # dtrace -l
  ID   PROVIDER            MODULE                          FUNCTION NAME
  1     dtrace                                                     BEGIN
  2     dtrace                                                     END
  3     dtrace                                                     ERROR
  4      sched                FX                          fx_yield schedctl-yi
  [...] 
  # dtrace -l | wc -l
  69880
  ```

- Here the root user sees 69,879 available probes.
- The probe count changes – it is dynamic (DTrace).
Tracing Probes

- **dtrace -n** takes a probe name and enables tracing:

```bash
# dtrace -n syscall::exece:return
dtrace: description 'syscall::exece:return' matched 1 probe
CPU     ID                    FUNCTION:NAME
0  76044                     exece:return
0  76044                     exece:return
^C
```

- The default output contains:
  - **CPU** CPU id that event occurred on (if this changes, the output may be shuffled)
  - **ID** DTrace probe id
  - **FUNCTION:NAME** Part of the probe name
## Providers

- **Examples of providers:**

<table>
<thead>
<tr>
<th>Provider</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syscall</td>
<td>system call entries and returns</td>
</tr>
<tr>
<td>proc</td>
<td>process and thread events</td>
</tr>
<tr>
<td>sched</td>
<td>kernel scheduling events</td>
</tr>
<tr>
<td>sysinfo</td>
<td>system statistic events</td>
</tr>
<tr>
<td>vminfo</td>
<td>virtual memory events</td>
</tr>
<tr>
<td>io</td>
<td>system I/O events</td>
</tr>
<tr>
<td>profile</td>
<td>fixed rate sampling</td>
</tr>
<tr>
<td>pid</td>
<td>user-level tracing</td>
</tr>
<tr>
<td>fbt</td>
<td>raw kernel tracing</td>
</tr>
</tbody>
</table>
## Providers

- Example of probes:

<table>
<thead>
<tr>
<th>Provider</th>
<th>Example probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>syscall</td>
<td>syscall::read:entry</td>
</tr>
<tr>
<td>proc</td>
<td>proc::exec-success</td>
</tr>
<tr>
<td>sched</td>
<td>sched::on-cpu</td>
</tr>
<tr>
<td>sysinfo</td>
<td>sysinfo::readch</td>
</tr>
<tr>
<td>vminfo</td>
<td>vminfo::maj_fault</td>
</tr>
<tr>
<td>io</td>
<td>io::start</td>
</tr>
<tr>
<td>profile</td>
<td>profile::profile-1000hz</td>
</tr>
<tr>
<td>pid</td>
<td>pid172:libc:fopen:entry</td>
</tr>
<tr>
<td></td>
<td>pid172:a.out:main:entry</td>
</tr>
<tr>
<td>fbt</td>
<td>fbt::bdev_strategy:entry</td>
</tr>
</tbody>
</table>
Providers

• Providers are documented in the DTrace Guide, as separate chapters.

• Providers are dynamic, the number of available probes can vary.

• Some providers are “unstable interface”, such as fbt and sdt.
  > This means that their probes, while useful, may vary in name and arguments between Solaris versions.
  > Try to use stable providers instead (if possible).
Provider Documentation

• Some providers assume a little background knowledge, other providers assume a lot. Knowing where to find supporting documentation is important.

• Where do you find documentation on:
  > Syscalls?
  > User Libraries?
  > Application Code?
  > Kernel functions?
Provider Documentation

- Additional documentation may be found here:

<table>
<thead>
<tr>
<th>Target</th>
<th>Provider</th>
<th>Additional Docs</th>
</tr>
</thead>
<tbody>
<tr>
<td>syscalls</td>
<td>syscall</td>
<td>man(2)</td>
</tr>
<tr>
<td>libraries</td>
<td>pid:lib*</td>
<td>man(3C)</td>
</tr>
<tr>
<td>app code</td>
<td>pid:a.out</td>
<td>source code?</td>
</tr>
<tr>
<td>raw kernel</td>
<td>fbt</td>
<td>Solaris Internals 2\textsuperscript{nd} Ed, (\text{<a href="http://cvs.opensolaris.org%7D%5C">http://cvs.opensolaris.org}\</a>)</td>
</tr>
</tbody>
</table>
Actions

- When a probe fires, an action executes.
- Actions are written in the D programming language.
- Actions can:
  > print output
  > save data to variables, and perform calculations
  > walk kernel or process memory
- With destruction actions allowed, actions can:
  > raise signals on processes
  > execute shell commands
  > write to some areas of memory
trace() Example

The trace() action accepts one argument and prints it when the probe fired.

```bash
# dtrace -n 'syscall::exece:return { trace(execname); }'
dtrace: description 'syscall::exece:return ' matched 1 probe
CPU     ID                    FUNCTION:NAME
0  76044                     exece:return   man
0  76044                     exece:return   sh
0  76044                     exece:return   neqn
0  76044                     exece:return   tbl
0  76044                     exece:return   nroff
0  76044                     exece:return   col
[...]```
printf() Example

```c
# dtrace -n 'syscall::exece:return { printf("%6d %s\n", pid, execname); }'
dtrace: description 'syscall::exece:return ' matched 1 probe

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>74415</td>
<td>exece:return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4301 sh</td>
</tr>
<tr>
<td>0</td>
<td>74415</td>
<td>exece:return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4304 neqn</td>
</tr>
<tr>
<td>0</td>
<td>74415</td>
<td>exece:return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4305 nroff</td>
</tr>
<tr>
<td>0</td>
<td>74415</td>
<td>exece:return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4306 sh</td>
</tr>
<tr>
<td>0</td>
<td>74415</td>
<td>exece:return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4308 sh</td>
</tr>
</tbody>
</table>
```

- DTrace ships with a powerful printf(), to print formatted output.
Default Variables

- Numerous predefined variables can be used, eg:
  - `pid, tid` Process ID, Thread ID
  - `timestamp` Nanosecond timestamp since boot
  - `probefunc` Probe function name (3rd field)
  - `execname` Process name
  - `arg0, ...` Function arguments and return value
  - `errno` Last syscall failure error code
  - `curpsinfo` Struct containing current process info, eg, `curpsinfo->pr_psargs` – process + args

- Pointers and structs! DTrace can walk memory using C syntax, and has kernel types predefined.
curthread

- `curthread` is a pointer to current `kthread_t`

  From here you can walk kernel memory and answer endless questions about OS internals.

- Eg, the current process `user_t` is,

  ```c
  curthread->t_procp->p_user
  ```

- You might not ever use `curthread`, but it is good to know that you can. (And there are other ways to get inside the kernel).

**Opinion:**
`curthread` is like the down staircase in nethack, angband, moria, ...
Variable Types

• DTrace supports the following variable types:
  > Integers
  > Structs
  > Pointers
  > Strings
  > Associative arrays
  > Aggregates

• Including types from /usr/include/sys, eg uint32_t.
Aggregations

- A great feature of DTrace is to process data as it is captured, such as using aggregations.
- E.g., frequency counting syscalls:

```
# dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 233 probes
^C
[...]
  writev            170
  write             257
  read             896
  pollsys         959
  ioctl          1253
```

@num is the aggregation variable, probefunc is the key, and count() is the aggregating function.
Aggregating Functions

• These include:
  > `count()`  count events, useful for frequency counts
  > `sum(value)`  sum the value
  > `avg(value)`  average the value
  > `min(value)`  find the value minimum
  > `max(value)`  find the value maximum
  > `quantize(value)`  print power-2 distribution plots
Quantize

- Very cool function, here we quantize write sizes:

```bash
# dtrace -n 'sysinfo:::writech { @dist[execname] = quantize(arg0); }'
dtrace: description 'sysinfo:::writech ' matched 4 probes
^C
[...]
ls

value  ------------- Distribution ------------- count
  4  |                                    0
  8  |                                    2
 16  |                                    0
 32  |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 118
 64  |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 127
 96  |                                    0

[...]
```

- Here we see that `ls` processes usually write between 32 and 127 bytes. Makes sense?
### ls -l

```
# ls -l /etc
dttotal 793

<table>
<thead>
<tr>
<th>Mode</th>
<th>User</th>
<th>Group</th>
<th>Size</th>
<th>Date</th>
<th>Time</th>
<th>Symbol</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td></td>
<td>12 Mar</td>
<td>03:28</td>
<td></td>
<td>TIMEZONE -&gt; default/init</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>4 root</td>
<td>sys</td>
<td></td>
<td>6 Apr</td>
<td>06:59</td>
<td></td>
<td>X11</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>2 adm</td>
<td>adm</td>
<td></td>
<td>3 Mar</td>
<td>09:25</td>
<td></td>
<td>acct</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>3 root</td>
<td>root</td>
<td></td>
<td>3 Apr</td>
<td>23:11</td>
<td></td>
<td>ak</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td></td>
<td>12 Mar</td>
<td>03:28</td>
<td></td>
<td>aliases -&gt; mail/aliases</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>5 root</td>
<td>sys</td>
<td></td>
<td>5 Feb</td>
<td>23:29</td>
<td></td>
<td>amd64</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>7 root</td>
<td>bin</td>
<td></td>
<td>18 Mar</td>
<td>09:20</td>
<td></td>
<td>apache</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>4 root</td>
<td>bin</td>
<td></td>
<td>7 Feb</td>
<td>23:12</td>
<td></td>
<td>apache2</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>2 root</td>
<td>sys</td>
<td></td>
<td>5 Feb</td>
<td>23:27</td>
<td></td>
<td>apoc</td>
</tr>
<tr>
<td>-rw-r--r--</td>
<td>1 root</td>
<td>bin</td>
<td></td>
<td>1012 Mar</td>
<td>09:33</td>
<td></td>
<td>auto_home</td>
</tr>
<tr>
<td>-rw-r--r--</td>
<td>1 root</td>
<td>bin</td>
<td></td>
<td>1066 Mar</td>
<td>09:33</td>
<td></td>
<td>auto_master</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td></td>
<td>16 Mar</td>
<td>03:28</td>
<td></td>
<td>autopush -&gt; ../sbin/autopu</td>
</tr>
</tbody>
</table>

[...]
```

**ls** writes one line at a time, each around 80 chars long.
Predicates

- DTrace predicates are used to filter probes, so that the action fires when a conditional is true.

  probename /predicate/ { action }

- Eg, syscalls for processes called “bash”:

```bash
# dtrace -n 'syscall:::entry /execname == "bash"/ { @num[probefunc] = count(); }'

dtrace: description 'syscall:::entry ' matched 233 probes
^C

  exece 2
  [...] 
  read 29
  write 31 
  lwp_sigmask 42
  sigaction 62
```
Scripting

• If your one-liners get too long, write scripts. Eg, bash-syscalls.d:

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

syscall:::entry
/execname == "bash"/
{
    @num[probefunc] = count();
}
```

• Getting it running:

```bash
# chmod 755 bash-syscalls.d
# ./bash-syscalls.d
dtrace: script './bash-syscalls.d' matched 233 probes
[...]```
What's next:

• We just covered:
  > What is DTrace
  > What is DTrace for
  > Who uses DTrace
  > DTrace Essentials

• Next up is:
  > Usage Features
Measuring Time

• Access to high resolution timestamps is of particular use for performance analysis.
  > timestamp  time since boot in nanoseconds
  > vtimestamp  thread on-CPU timestamp

• Measuring these for application and operating system function calls will answer:
  > timestamp  where is the latency?
  > vtimestamp  why are the CPUs busy?
Printing Stacks

• Printing user and kernel stack traces explains both why and the how something happened.

• Why is bash calling read()? Using `ustack()`:

```
# dtrace -n 'syscall::read:entry /execname == "bash"/ { ustack(); }'
```

```
dtrace: description 'syscall::read:entry ' matched 1 probe

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>74314</td>
<td>read:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libc.so.1`_read+0x7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bash`rl_getc+0x22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bash`rl_read_key+0xad</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bash`readline_internal_char+0x5f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bash`0x80b1171</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bash`0x80b118c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bash`readline+0x3a</td>
</tr>
</tbody>
</table>
```

Ahh, `readline()`
End of Intro

• DTrace is a big topic, but you don't need to know it all to get value from DTrace.

• To learn more, browse “DTrace Topics”, http://www.solarisinternals.com/dtrace.

Here you will find:

> A wiki version of this presentation
> The PDF for this presentation
> dozens of other DTrace Topics (eg, one-liners!)

• Also see the “Solaris Performance and Tools” book, http://www.sun.com/books/catalog/solaris_perf_tools.xml
Sampling

- DTrace isn't just about tracing events, DTrace can also sample at customized rates.
- Eg, sampling 5-level user stack traces from Xorg:

```bash
# dtrace -n 'profile-1001 /execname == "Xorg"/ { @[ustack(5)] = count(); }'
dtrace: description 'profile-1001 ' matched 1 probe
^C
libfb.so`fbSolid+0x2c6
libfb.so`fbFill+0xb8
libfb.so`fbPolyFillRect+0x1d5
nvidia_drv.so`0xfe09e87b
Xorg`miColorRects+0x124
  41
nvidia_drv.so`_nv000592X+0x3d
0x1016c00
  87
```

nvidia was on-CPU 87 times
See Also

• DTrace home: http://www.opensolaris.org/os/community/dtrace
  > Main site of links
  > DTrace-discuss mailing list

• Team DTrace blogs:
  > http://blogs.sun.com/bmc
  > http://blogs.sun.com/mws
  > http://blogs.sun.com/ahl

• DTrace Toolkit:
  > http://www.opensolaris.org/os/community/dtrace/dtracetoolkit
dtrace:::END

Brendan Gregg
brendan@sun.com