DTrace Workshop

Context-Switch presents,

DTrace on Solaris 10

London, June, 2006

Brendan Gregg
This document has been provided on an “as is” basis, WITHOUT WARRANTY OF ANY KIND, either expressed or implied, without even the implied warranty of merchantability or fitness for a particular purpose. This document could include technical inaccuracies, typographical errors and even spelling errors (or at the very least, Australian spelling).

Various content, including diagrams and material from the DTraceToolkit, are copyright © 2006 by Brendan Gregg.

This document was NOT written by Sun Microsystems, and opinions expressed are not those of Sun Microsystems unless by coincidence.
This Workshop

DAY 1
• Solaris 9 Performance Monitoring
• Introducing DTrace

DAY 2
• Programming in D
• The DTrace Mentality

DAY 3
• Fixing DTrace Faults
Chapters

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTraceToolkit
Chapter 10 – DTrace Mentality 2
Some topics presented are based on,

- **Solaris Performance and Tools**: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (McDougall/Mauro/Gregg)
These slides cover key topics

I will cover many side topics, elaborate further, answer questions, run demos, ...

as such, when I deviate from the slides – don't panic! It's part of the course.
Chapter 1

Solaris 9 Performance Monitoring

- What can be observed
- What *can't* be observed
- Key DTrace wins
- Start here
Example Fault

• Where has my CPU gone?

  using Solaris 9 tools

  ... interactive demo ...
The Solaris Toolkit

Consists of many, many tools

- ps, prstat, ptree, pstack, pmap
- vmstat, mpstat, iostat, netstat
- sar
- kstat, ndd, mdb
- truss, sotruss, apptrace, prex
- cpustat, trapstat, lockstat
- ...

These is no single tool to rule them all (not even DTrace)
I have tried sysperfstat

$ sysperfstat 1

<table>
<thead>
<tr>
<th>Time</th>
<th>%CPU</th>
<th>%Mem</th>
<th>%Disk</th>
<th>%Net</th>
<th>CPU</th>
<th>Mem</th>
<th>Disk</th>
<th>Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>23:27:41</td>
<td>0.85</td>
<td>44.11</td>
<td>2.40</td>
<td>0.19</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>23:27:42</td>
<td>3.00</td>
<td>80.98</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>23:27:43</td>
<td>2.00</td>
<td>80.98</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>23:27:44</td>
<td>17.00</td>
<td>80.98</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>23:27:45</td>
<td>46.00</td>
<td>80.49</td>
<td>22.05</td>
<td>54.20</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>23:27:46</td>
<td>50.00</td>
<td>79.83</td>
<td>14.19</td>
<td>78.18</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>23:27:47</td>
<td>48.00</td>
<td>79.39</td>
<td>8.04</td>
<td>80.94</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>23:27:48</td>
<td>54.00</td>
<td>79.62</td>
<td>3.06</td>
<td>70.89</td>
<td>4.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>23:27:49</td>
<td>39.00</td>
<td>79.43</td>
<td>6.78</td>
<td>74.52</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Such a tool serves one role but not all. In this case, it provides the “view from 20,000 feet”.
Coping With Many Tools

- Categorise by Approach
- Categorise by Resource Type
- Checklists of tools (these slides)
- Study some, be aware what else exists
  just remembering that something is doable is valuable!
Categorise by Approach

1. Monitoring
   Monitoring multiple hosts
   Gathering long term data

2. Identification
   Examining system-wide health

3. Analysis
   Focusing on details
Examples

1. Monitoring
   SNMP, sar, SunMC

2. Identification
   kstat (vmstat, mpstat, iostat)
   procfs (ps, prstat)
   mnttab (df)

3. Analysis
   truss, apptrace, prex, lockstat, ...
Hints

• Be careful when using sar:
  • sar has sampling issues
  • the default configuration needs tuning
  • try `truss -ft ioctl sar -u 1 5` (yes, sar reads everything. compare to mpstat)

• Solaris 10 now uses Net-SNMP
  • USM – usernames, passwords, encryption
  • VACM – restricting views
    (these greatly improve security)
Categorise by Resource Type

- **CPU**
  - vmstat, mpstat, sar, prstat, ps

- **Memory**
  - vmstat, swap, ::memstat, prstat, ps, pmap

- **Disk**
  - iostat, taztool, iosnoop, iotop

- **Network**
  - netstat, kstat, ndd, nx.se, nicstat

- ...
The following list of tools serves to:

- provide a checklist
- show what is doable – you'll remember later
- show what can be done – the right tool for the job
- show what can't be done – where DTrace can help
- provide starting points for using DTrace
Tools Checklist

- uptime
- ps
- /usr/ucb/ps
- prstat
- prstat -m
- vmstat
- mpstat
- iostat
- prex/tnf*
- psio
- netstat -i
- nx.se
- nicstat
- netstat -rn
- sar
- sar -u
- sar -q
- kstat
- K9Toolkit
- ndd
- checkcable
- cpustat
- CacheKit
- busstat
- trapstat
- lockstat
- lockstat -l
- truss
- sotruss
- apptrace
- truss -ua.out
- adb
- mdb -k
- mdb -p
- lastcomm
- BSM auditing
- snmpwalk
$ uptime
2:38am  up 347 day(s),  9:08,  5 users,  load average: 0.01, 0.02, 0.03

- load average: 1, 5 and 15 minute averages.
  - These were once the average length of the combined dispatcher queues + currently running threads, sampled during clock().
  - It is now microstate accounting based, and is all the threads usr + sys + CPU latency times.
  - load averages tend to over-simplify CPU issues.

- Can't customise the interval
- DTrace can trace scheduler activity

- BTW – 347 days is nothing; I maintain the “Sun Book of Records” (search google), where the current record is 2001 days!
Useful to check long-term processes

Default fields not hugely useful, use -o

DTrace can access similar procfs statistics, and gather many additional statistics
$ /usr/ucb/ps auxww

<table>
<thead>
<tr>
<th>USER</th>
<th>PID</th>
<th>%CPU</th>
<th>%MEM</th>
<th>SZ</th>
<th>RSS</th>
<th>TT</th>
<th>S</th>
<th>START</th>
<th>TIME</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>3768</td>
<td>3.0</td>
<td>0.8</td>
<td>3216</td>
<td>2856</td>
<td>pts/2</td>
<td>O</td>
<td>16:24:40</td>
<td>0:00</td>
<td>/usr/ucb/ps -auxww</td>
</tr>
<tr>
<td>root</td>
<td>3453</td>
<td>0.2</td>
<td>0.8</td>
<td>4544</td>
<td>2904</td>
<td>?</td>
<td>S</td>
<td>15:59:10</td>
<td>0:01</td>
<td>/usr/lib/ssh/sshd</td>
</tr>
<tr>
<td>root</td>
<td>498</td>
<td>0.1</td>
<td>0.8</td>
<td>4680</td>
<td>2944</td>
<td>?</td>
<td>S</td>
<td>May 03</td>
<td>0:29</td>
<td>/usr/lib/ssh/sshd</td>
</tr>
<tr>
<td>brendan</td>
<td>17545</td>
<td>0.1</td>
<td>0.6</td>
<td>2648</td>
<td>2080</td>
<td>pts/2</td>
<td>S</td>
<td>May 03</td>
<td>0:29</td>
<td>/bin/bash</td>
</tr>
<tr>
<td>brendan</td>
<td>3769</td>
<td>0.1</td>
<td>0.3</td>
<td>984</td>
<td>808</td>
<td>pts/2</td>
<td>S</td>
<td>16:24:40</td>
<td>0:00</td>
<td>head</td>
</tr>
<tr>
<td>brendan</td>
<td>2408</td>
<td>0.1</td>
<td>1.4</td>
<td>6800</td>
<td>5112</td>
<td>?</td>
<td>S</td>
<td>May 28</td>
<td>92:28</td>
<td>SCREEN</td>
</tr>
<tr>
<td>root</td>
<td>2037</td>
<td>0.1</td>
<td>0.3</td>
<td>2024</td>
<td>800</td>
<td>?</td>
<td>S</td>
<td>May 28</td>
<td>168:21</td>
<td>/usr/sbin/in.routed</td>
</tr>
<tr>
<td>root</td>
<td>3765</td>
<td>0.1</td>
<td>0.3</td>
<td>1016</td>
<td>816</td>
<td>?</td>
<td>S</td>
<td>16:24:28</td>
<td>0:00</td>
<td>sleep 15</td>
</tr>
<tr>
<td>nobody</td>
<td>22112</td>
<td>0.1</td>
<td>22.79150484352</td>
<td>22.79150484352</td>
<td>pts/2</td>
<td>?</td>
<td>S</td>
<td>Feb 15</td>
<td>231:44</td>
<td>(squid)</td>
</tr>
</tbody>
</table>

- Sorting by %CPU is nice
- *Fields colliding is not nice*
- *pargs* is better at viewing full arg listings
$ prstat

PID USERNAME  SIZE   RSS STATE PRI NICE      TIME  CPU PROCESS/NLWP
3453 root     4544K 2904K sleep   59    0   0:00:01 0.7% sshd/1
2408 brendan  6800K 5112K sleep   59    0   1:32:29 0.7% screen-3.9.8/1
5558 irc      8152K 3832K sleep   59    0   0:15:41 0.4% irssi/1
3944 brendan  4584K 4224K cpu0    39    0   0:00:00 0.4% prstat/1
25442 brendan 2312K 296K sleep   59    0   0:05:05 0.1% telnet/1
3939 root     1016K 816K sleep   59    0   0:00:00 0.0% sleep/1
17545 brendan 2648K 2080K sleep   49    0   0:00:04 0.0% bash/1
2037 root     2024K 800K sleep   59    0   2:48:21 0.0% in.routed/1

Total: 114 processes, 197 lwps, load averages: 0.04, 0.02, 0.03

- Great summary view
- Doesn't wallop the CPU
- *DTrace can also provide updating summaries, of a simple nature*
```bash
$ prstat -m

<table>
<thead>
<tr>
<th>PID</th>
<th>USERNAME</th>
<th>USR</th>
<th>SYS</th>
<th>TRP</th>
<th>TFL</th>
<th>DFL</th>
<th>LCK</th>
<th>SLP</th>
<th>LAT</th>
<th>VCX</th>
<th>ICX</th>
<th>SCL</th>
<th>SIG</th>
<th>PROCESS/NLWP</th>
</tr>
</thead>
<tbody>
<tr>
<td>3999</td>
<td>brendan</td>
<td>0.4</td>
<td>1.3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>98</td>
<td>0.0</td>
<td>9</td>
<td>0</td>
<td>259</td>
<td>0</td>
<td>prstat/1</td>
</tr>
<tr>
<td>2408</td>
<td>brendan</td>
<td>0.7</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>99</td>
<td>0.0</td>
<td>6</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>screen-3.9.8/1</td>
</tr>
<tr>
<td>19763</td>
<td>brendan</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100</td>
<td>0.0</td>
<td>5</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>sdtperfmeter/1</td>
</tr>
<tr>
<td>2150</td>
<td>root</td>
<td>0.0</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>8</td>
<td>0</td>
<td>26</td>
<td>0</td>
<td>nscd/25</td>
</tr>
<tr>
<td>78</td>
<td>root</td>
<td>0.0</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>picld/4</td>
</tr>
<tr>
<td>73</td>
<td>root</td>
<td>0.0</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>syseventd/14</td>
</tr>
<tr>
<td>2061</td>
<td>root</td>
<td>0.0</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>rpcbind/1</td>
</tr>
<tr>
<td>22112</td>
<td>root</td>
<td>0.0</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>squid/1</td>
</tr>
</tbody>
</table>

Total: 114 processes, 197 lwps, load averages: 0.00, 0.02, 0.02

- Microstate accounting – great breakdown of process time (very useful!)
- **Restricted to pre-determined accounting states**
- **DTrace can measure custom states**
$ vmstat 1 3

kthr memory page disk faults cpu
r b w swap free re mf pi po fr de sr dd dd f0 s3 in sy cs us sy id
0 0 21 422864 68800 3 16 5 0 0 0 0 0 0 0 0 126 106 135 1 0 98
0 0 51 434616 55560 32 174 0 0 0 0 0 0 0 0 403 442 123 1 3 96
0 0 51 434616 55560 0 0 0 0 0 0 0 0 0 0 0 0 0 126 401 250 116 0 0 100

$ vmstat -p 1 3

memory page executable anonymous filesystem
swap free re mf fr de sr epi epo epf api apo apf fpi fpo fpf
422864 68800 3 16 0 0 0 0 0 0 0 0 0 0 0 5 0 0
434616 55560 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0
434616 55560 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- Best system-wide view
- Can't view statistics by-process, by-zone, ...
- DTrace can!
mpstat

$ mpstat 5

<table>
<thead>
<tr>
<th>CPU</th>
<th>minf</th>
<th>mjf</th>
<th>xcal</th>
<th>intr</th>
<th>ithr</th>
<th>csw</th>
<th>icsw</th>
<th>migr</th>
<th>smtx</th>
<th>srw</th>
<th>syscl</th>
<th>usr</th>
<th>sys</th>
<th>wt</th>
<th>idl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>127</td>
<td>26</td>
<td>135</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>106</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>149</td>
<td>29</td>
<td>155</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>123</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>98</td>
<td>17</td>
<td>112</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>94</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>104</td>
<td>23</td>
<td>144</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>112</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>97</td>
<td></td>
</tr>
</tbody>
</table>

...  

- Good by-CPU summary  
- DTrace can also measure by-CPU
$ iostat -xnmPz 1

**extended device statistics**

<table>
<thead>
<tr>
<th>r/s</th>
<th>w/s</th>
<th>kr/s</th>
<th>kw/s</th>
<th>wait</th>
<th>actv</th>
<th>wsvc_t</th>
<th>asvc_t</th>
<th>%w</th>
<th>%b</th>
<th>device</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>19.4</td>
<td>11.6</td>
<td>0</td>
<td>0</td>
<td>c0t0d0s0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>56.4</td>
<td>11.4</td>
<td>0</td>
<td>0</td>
<td>c0t0d0s1</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>94.9</td>
<td>15.6</td>
<td>0</td>
<td>0</td>
<td>c0t0d0s3</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>92.4</td>
<td>14.6</td>
<td>0</td>
<td>0</td>
<td>c0t0d0s4</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>37.7</td>
<td>10.0</td>
<td>0</td>
<td>0</td>
<td>c0t0d0s5</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>15.8</td>
<td>12.1</td>
<td>0</td>
<td>0</td>
<td>c0t0d0s7 (/data)</td>
</tr>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.4</td>
<td>0.4</td>
<td>0.0</td>
<td>0.0</td>
<td>1.8</td>
<td>4.3</td>
<td>0</td>
<td>0</td>
<td>c0t2d0s0 (/)</td>
</tr>
<tr>
<td>0.1</td>
<td>0.2</td>
<td>4.1</td>
<td>3.1</td>
<td>0.0</td>
<td>0.0</td>
<td>37.4</td>
<td>6.7</td>
<td>0</td>
<td>0</td>
<td>c0t2d0s1 (/export/home)</td>
</tr>
<tr>
<td>0.1</td>
<td>0.0</td>
<td>0.2</td>
<td>0.5</td>
<td>0.0</td>
<td>0.0</td>
<td>3.3</td>
<td>12.7</td>
<td>0</td>
<td>0</td>
<td>c0t2d0s3 (/squidcache)</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>4.4</td>
<td>0</td>
<td>0</td>
<td>mars:vold(pid2295)</td>
</tr>
</tbody>
</table>

- Excellent per-disk/partition/controller summary
- *No by-process summary, or event details*
- *DTrace solves both*
prex/tnfxtract/tnfdump

- prex: enables static probes to record event details to a buffer
- tnfxtract: extracts buffer contents
- tnfdump: converts buffer contents to text
- DTrace has this exact functionality, and more (dynamic)
# tnfdump out1.tnf

probe   tnf_name: "pagein" tnf_string: "keys vm pageio io;file ../../../common/os/bio.c;line 1333;"

probe   tnf_name: "strategy" tnf_string: "keys io blockio;file ../../../common/os/driver.c;line 411;"

probe   tnf_name: "biodone" tnf_string: "keys io blockio;file ../../../common/os/bio.c;line 1222;"

<table>
<thead>
<tr>
<th>Elapsed (ms)</th>
<th>Delta (ms)</th>
<th>PID</th>
<th>LWPID</th>
<th>TID</th>
<th>CPU</th>
<th>Probe Name Data / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000000</td>
<td>0.000000</td>
<td>5926</td>
<td>1</td>
<td>0xd423fe00</td>
<td>0</td>
<td>pagein vnode: 0xd678de40 offset: 0 size: 8192</td>
</tr>
<tr>
<td>0.029433</td>
<td>0.029433</td>
<td>5926</td>
<td>1</td>
<td>0xd423fe00</td>
<td>0</td>
<td>strategy device: 26738688 block: 1616 size: 8192 buf: 0xd7c32aa8 flags: 34078801</td>
</tr>
<tr>
<td>12.298443</td>
<td>12.269010</td>
<td>0</td>
<td>0</td>
<td>0xd3a00de0</td>
<td>0</td>
<td>biodone device: 26738688 block: 1616 buf: 0xd7c32aa8</td>
</tr>
</tbody>
</table>
psio

# ./psio 1

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>%I/O</th>
<th>STIME</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>brendan</td>
<td>13271</td>
<td>10093</td>
<td>65.4</td>
<td>23:20:16</td>
<td>pts/20</td>
<td>0:01</td>
<td>grep brendan contents</td>
</tr>
<tr>
<td>root</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>Mar 16</td>
<td>?</td>
<td>0:16</td>
<td>sched</td>
</tr>
<tr>
<td>root</td>
<td>1</td>
<td>0</td>
<td>0.0</td>
<td>Mar 16</td>
<td>?</td>
<td>0:10</td>
<td>/etc/init -</td>
</tr>
<tr>
<td>root</td>
<td>2</td>
<td>0</td>
<td>0.0</td>
<td>Mar 16</td>
<td>?</td>
<td>0:00</td>
<td>pageout</td>
</tr>
</tbody>
</table>

... 

- [http://www.brendangregg.com/psio.html](http://www.brendangregg.com/psio.html)
- Solved %Disk I/O by-process
- *Could only run for short intervals*
- *DTrace takes this much further – tracing events, by-process info, ...*
### netstat -i

```
$ netstat -i 1 5

<table>
<thead>
<tr>
<th>packets</th>
<th>errs</th>
<th>packets</th>
<th>errs</th>
<th>colls</th>
<th>packets</th>
<th>errs</th>
<th>packets</th>
<th>errs</th>
<th>colls</th>
</tr>
</thead>
<tbody>
<tr>
<td>206999694</td>
<td>32</td>
<td>223490272</td>
<td>0</td>
<td>0</td>
<td>326931321</td>
<td>32</td>
<td>343421899</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>37</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>54</td>
<td>0</td>
<td>0</td>
<td>52</td>
<td>0</td>
<td>56</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

- An approximation of activity
- No further details for analysis
- Packets are not bytes
- Both kstat and DTrace provide more info
$ se nx.se 1
Current tcp RtoMin is 400, interval 1, start Thu May 11 18:22:37 2006

18:22:38 Iseg/s Oseg/s InKB/s OuKB/s Rst/s Atf/s Ret% Icn/s Ocn/s
tcp      2.0    2.0   0.06   0.13   0.00   0.00   0.0   0.00   0.00
Name    Ipkt/s Opkt/s InKB/s OuKB/s IErr/s OErr/s Coll% NoCP/s Defr/s
hme0     3.0    3.9   0.22   0.34  0.000  0.000   0.0   0.00   0.00

18:22:39 Iseg/s Oseg/s InKB/s OuKB/s Rst/s Atf/s Ret% Icn/s Ocn/s
tcp      1.0    1.0   0.04   0.32   0.00   0.00   0.0   0.00   0.00
Name    Ipkt/s Opkt/s InKB/s OuKB/s IErr/s OErr/s Coll% NoCP/s Defr/s
hme0     3.0    3.9   0.22   0.34  0.000  0.000   0.0   0.00   0.00
^C

- From the SE Toolkit
- Provides many more useful details
$ nicstat 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Int</th>
<th>rKb/s</th>
<th>wKb/s</th>
<th>rPk/s</th>
<th>wPk/s</th>
<th>rAvs</th>
<th>wAvs</th>
<th>%Util</th>
<th>Sat</th>
</tr>
</thead>
<tbody>
<tr>
<td>18:23:49</td>
<td>hme0</td>
<td>4.12</td>
<td>4.69</td>
<td>6.88</td>
<td>7.43</td>
<td>613.61</td>
<td>645.92</td>
<td>0.07</td>
<td>0.00</td>
</tr>
<tr>
<td>18:23:50</td>
<td>hme0</td>
<td>0.19</td>
<td>0.23</td>
<td>2.01</td>
<td>2.01</td>
<td>98.00</td>
<td>116.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>18:23:51</td>
<td>hme0</td>
<td>1.41</td>
<td>4.45</td>
<td>13.00</td>
<td>17.00</td>
<td>111.38</td>
<td>267.88</td>
<td>0.05</td>
<td>0.00</td>
</tr>
<tr>
<td>18:23:52</td>
<td>hme0</td>
<td>0.99</td>
<td>3.64</td>
<td>9.00</td>
<td>11.00</td>
<td>112.89</td>
<td>339.09</td>
<td>0.04</td>
<td>0.00</td>
</tr>
<tr>
<td>18:23:53</td>
<td>hme0</td>
<td>0.10</td>
<td>0.23</td>
<td>1.00</td>
<td>2.00</td>
<td>98.00</td>
<td>116.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

- [http://www.brendangregg.com/k9toolkit.html](http://www.brendangregg.com/k9toolkit.html)
- Uses kstat, written in both C and Perl
- Provides bytes, %Utilisation
$ netstat -rn

Routing Table: IPv4

<table>
<thead>
<tr>
<th>Destination</th>
<th>Gateway</th>
<th>Flags</th>
<th>Ref</th>
<th>Use</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>220.244.170.56</td>
<td>220.244.170.58</td>
<td>U</td>
<td>1</td>
<td>18146</td>
<td>hme0:1</td>
</tr>
<tr>
<td>192.168.0.0</td>
<td>192.168.0.2</td>
<td>U</td>
<td>1</td>
<td>12537</td>
<td>hme0:2</td>
</tr>
<tr>
<td>192.168.1.0</td>
<td>192.168.1.1</td>
<td>U</td>
<td>1</td>
<td>32151</td>
<td>hme0</td>
</tr>
<tr>
<td>224.0.0.0</td>
<td>192.168.1.1</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>hme0</td>
</tr>
<tr>
<td>default</td>
<td>220.244.170.57</td>
<td>UG</td>
<td>1</td>
<td>326156</td>
<td></td>
</tr>
<tr>
<td>127.0.0.1</td>
<td>127.0.0.1</td>
<td>UH</td>
<td>9116268837</td>
<td>lo0</td>
<td></td>
</tr>
</tbody>
</table>

- Check for unexpected routes
- You discover problems after the fact
- DTrace can snoop route changes live
$ sar -u 1 5

SunOS mars 5.9 Generic_118558-05 sun4u 05/11/2006

18:33:46 %usr %sys %wio %idle
18:33:47 2 2 0 96
18:33:48 4 1 0 95
18:33:49 3 4 5 88
18:33:50 0 2 0 98
18:33:51 4 3 3 90

- System Activity Reporter
- Prints a variety of kstats
- Can collect historic data, to identify long term patterns
- Has several issues
sar -u

$ sar -u

SunOS mars 5.9 Generic_118558-05 sun4u 05/11/2006

<table>
<thead>
<tr>
<th>Time</th>
<th>%usr</th>
<th>%sys</th>
<th>%wio</th>
<th>%idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00:01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>01:00:01</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>02:00:01</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>03:00:01</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>04:00:00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>05:00:01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>99</td>
</tr>
</tbody>
</table>

- **This shows historic data from sar -u**
- **%wio is always zero in Solaris 10**
- **Note that the system is idle**
sar -q

$ sar -q

SunOS mars 5.9 Generic_118558-05 sun4u 05/11/2006

00:00:01 runq-sz %runocc swpq-sz %swpocc
01:00:01 1.4 0 51.0 100
02:00:01 1.0 1 51.0 100
03:00:01 1.2 0 51.0 100
04:00:00 1.4 0 51.0 100
05:00:01 1.3 0 51.0 100
...

- sar -q shows run queue sizes
- This is the same period as before, and shows a > 1.0 run queue size. huh?
$ kstat -pm cpu_info
  cpu_info:0:cpu_info0:chip_id    0
  cpu_info:0:cpu_info0:class      misc
  cpu_info:0:cpu_info0:clock_MHz  333
  cpu_info:0:cpu_info0:cpu_type   sparcv9
  cpu_info:0:cpu_info0:crtime     4.553718221
  cpu_info:0:cpu_info0:device_ID  0
  cpu_info:0:cpu_info0:fpu_type   sparcv9
  cpu_info:0:cpu_info0:implementation UltraSPARC-IIi
...

- Great resource
- libkstat, Sun::Solaris::Kstat, or /usr/bin/kstat
  - netstat -k is now deprecated
- Many kstats are not documented
- kstat finds problems, DTrace analyses them
K9Toolkit

- http://www.brendangregg.com/k9toolkit.html
- A Perl kstat collection
- Contains
  - sysperfstat
  - checkcable
  - nicstat
  - prtdevs
  - ...

ndd

# ndd /dev/hme link_speed
1

# ndd /dev/rtl5 link_speed
operation failed: Invalid argument

- Accesses read/write settings for network related drivers
- **consistency issues between interface types**
# checkcable

<table>
<thead>
<tr>
<th>Interface</th>
<th>Link</th>
<th>Duplex</th>
<th>Speed</th>
<th>AutoNEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>hme0</td>
<td>UP</td>
<td>FULL</td>
<td>100</td>
<td>ON</td>
</tr>
<tr>
<td>hme1</td>
<td>UP</td>
<td>HALF</td>
<td>10</td>
<td>ON</td>
</tr>
</tbody>
</table>

- http://www.brendangregg.com
- Translates both ndd and kstats
- *May not be needed in the long term (interface kstats are getting better)*
cpustat

# cpustat -c pic0=EC_ref,pic1=EC_hit 1 5
  time  cpu  event    pic0    pic1
1.008  0  tick    77992    63504
2.008  0  tick    81972    65364
3.008  0  tick    76869    62250
4.008  0  tick    77347    62518
5.008  0  tick    76023    62405
5.008  1  total   390203   316041

- Examine CPU PICs
- Observe I$, D$, E$ performance
- Use cputrack for per-process info
- *Info available is CPU-type dependant*
# ccachestat 5 5

<table>
<thead>
<tr>
<th></th>
<th>I-Cache</th>
<th>D-Cache</th>
<th>E-Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>miss %hit</td>
<td>total miss %hit</td>
<td>total miss %hit</td>
</tr>
<tr>
<td>7424k</td>
<td>107k 98.55</td>
<td>2476k 39k 98.39</td>
<td>879k 200k 77.20</td>
</tr>
<tr>
<td>7941k</td>
<td>107k 98.65</td>
<td>3502k 53k 98.46</td>
<td>860k 211k 75.43</td>
</tr>
<tr>
<td>39082k</td>
<td>504k 98.71</td>
<td>15243k 293k 98.08</td>
<td>3420k 487k 85.75</td>
</tr>
<tr>
<td>19926k</td>
<td>248k 98.75</td>
<td>4985k 103k 97.92</td>
<td>1531k 277k 81.89</td>
</tr>
<tr>
<td>11028k</td>
<td>146k 98.67</td>
<td>5340k 95k 98.21</td>
<td>1856k 326k 82.42</td>
</tr>
</tbody>
</table>

- [http://www.brendangregg.com/cachekit.html](http://www.brendangregg.com/cachekit.html)
- Digs out useful info, calculates hit ratios
- *Needs updates to measure newer CPUs*
### busstat

```sh
# busstat -w ac,pic0=钟cycles,pic1=mem_bank0_rds 2 100
```

<table>
<thead>
<tr>
<th>time</th>
<th>dev</th>
<th>event0</th>
<th>pic0</th>
<th>event1</th>
<th>pic1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ac0</td>
<td>clock_cycles</td>
<td>167242902</td>
<td>mem_bank0_rds</td>
<td>3144</td>
</tr>
<tr>
<td>2</td>
<td>ac1</td>
<td>clock_cycles</td>
<td>167254476</td>
<td>mem_bank0_rds</td>
<td>1392</td>
</tr>
<tr>
<td>4</td>
<td>ac0</td>
<td>clock_cycles</td>
<td>168025190</td>
<td>mem_bank0_rds</td>
<td>40302</td>
</tr>
</tbody>
</table>

- Measures bus PICs
- *No public PIC documentation (yet)*

Anyone actually used this outside of Sun?
```
# trapstat -t 10 l

<table>
<thead>
<tr>
<th>CPU</th>
<th>iTLB-miss</th>
<th>%tim</th>
<th>ITSb-miss</th>
<th>%tim</th>
<th>DTLB-miss</th>
<th>%tim</th>
<th>DTsb-miss</th>
<th>%tim</th>
<th>%tim</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 u</td>
<td>2393</td>
<td>0.1</td>
<td>3950</td>
<td>0.7</td>
<td>11926</td>
<td>0.6</td>
<td>4205</td>
<td>0.8</td>
<td>2.2</td>
</tr>
<tr>
<td>0 k</td>
<td>105</td>
<td>0.0</td>
<td>0</td>
<td>0.0</td>
<td>96358</td>
<td>5.2</td>
<td>214</td>
<td>0.1</td>
<td>5.3</td>
</tr>
<tr>
<td>TTL</td>
<td>2498</td>
<td>0.2</td>
<td>3950</td>
<td>0.7</td>
<td>108284</td>
<td>5.7</td>
<td>4419</td>
<td>0.9</td>
<td>7.5</td>
</tr>
</tbody>
</table>
```

# trapstat

trapstat: not implemented on i86pc

- MMU statistics
- Check this to see if MPSS needs tuning
# lockstat -k sleep 5

Adaptive mutex block: 6 events in 5.638 seconds (1 events/sec)

<table>
<thead>
<tr>
<th>Count</th>
<th>indiv</th>
<th>cuml</th>
<th>rcnt</th>
<th>nsec</th>
<th>Lock</th>
<th>Caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>33%</td>
<td>33%</td>
<td>0.00</td>
<td>71995</td>
<td>0x30000e723c0</td>
<td>clock</td>
</tr>
<tr>
<td>1</td>
<td>17%</td>
<td>50%</td>
<td>0.00</td>
<td>85133</td>
<td>0x300055ffcb0</td>
<td>rctl_test_entity</td>
</tr>
<tr>
<td>1</td>
<td>17%</td>
<td>67%</td>
<td>0.00</td>
<td>77176</td>
<td>0x300040b2888</td>
<td>rctl_test_entity</td>
</tr>
<tr>
<td>1</td>
<td>17%</td>
<td>83%</td>
<td>0.00</td>
<td>70159</td>
<td>0x30000e72280</td>
<td>clock</td>
</tr>
<tr>
<td>1</td>
<td>17%</td>
<td>100%</td>
<td>0.00</td>
<td>73192</td>
<td>0x30000e722c0</td>
<td>clock</td>
</tr>
</tbody>
</table>

- Kernel lock statistics
lockstat -l

# lockstat -kIi 997 sleep 5

Profiling interrupt: 5199 events in 5.208 seconds (998 events/sec)

<table>
<thead>
<tr>
<th>Count</th>
<th>indiv</th>
<th>cuml</th>
<th>rcnt</th>
<th>nsec</th>
<th>CPU+PIL</th>
<th>Caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>1962</td>
<td>38%</td>
<td>38%</td>
<td>0.00</td>
<td>6751</td>
<td>cpu[0]</td>
<td>(usermode)</td>
</tr>
<tr>
<td>264</td>
<td>5%</td>
<td>43%</td>
<td>0.00</td>
<td>7605</td>
<td>cpu[0]</td>
<td>mutex_enter</td>
</tr>
<tr>
<td>187</td>
<td>4%</td>
<td>46%</td>
<td>0.00</td>
<td>7970</td>
<td>cpu[0]</td>
<td>dnlc_lookup</td>
</tr>
<tr>
<td>170</td>
<td>3%</td>
<td>50%</td>
<td>0.00</td>
<td>7271</td>
<td>cpu[0]</td>
<td>utl0</td>
</tr>
<tr>
<td>142</td>
<td>3%</td>
<td>52%</td>
<td>0.00</td>
<td>6866</td>
<td>cpu[0]</td>
<td>lookupppnvp</td>
</tr>
<tr>
<td>119</td>
<td>2%</td>
<td>55%</td>
<td>0.00</td>
<td>6476</td>
<td>cpu[0]</td>
<td>syscall_mstate</td>
</tr>
<tr>
<td>112</td>
<td>2%</td>
<td>57%</td>
<td>0.00</td>
<td>7142</td>
<td>cpu[0]</td>
<td>ufs_lookup</td>
</tr>
<tr>
<td>99</td>
<td>2%</td>
<td>59%</td>
<td>0.00</td>
<td>2041</td>
<td>cpu[0]</td>
<td>idle</td>
</tr>
</tbody>
</table>

- `-I` samples the kernel
- *DTrace can both sample and trace*
$ truss date
execve("/usr/bin/date", 0xFFF8754, 0xFFF875C) argc = 1
resolvepath("/usr/bin/date", "/usr/bin/date", 1023) = 13
resolvepath("/usr/lib/ld.so.1", "/usr/lib/ld.so.1", 1023) = 16
stat("/usr/bin/date", 0xFFF8528) = 0
open("/var/ld/ld.config", O_RDONLY) Err#2 ENOENT
stat("/usr/lib/libc.so.1", 0xFFF8030) = 0
resolvepath("/usr/lib/libc.so.1", "/usr/lib/libc.so.1", 1023) = 18
...

• Trace syscalls and signals
• truss is “violent”. can slow target by 70%
• Can't trace all processes at the same time
• truss behaves synchronously to syscalls
• DTrace buffers system-wide syscall details and reads the buffer asynchronously. slows target < 1%
$ sotruss date

- date -> libc.so.1:*atexit(0xff3bfaac, 0x21800, 0x0)
- date -> libc.so.1:*atexit(0x11654, 0x21800, 0x0)
- date -> libc.so.1:*setlocale(0x6, 0x116dc, 0xff3509e0)
- date -> libc.so.1:*textdomain(0x116e0, 0x116dc, 0xff3509e0)
- date -> libc.so.1:*getopt(0x1, 0xffbff684, 0x116f0)
- date -> libc.so.1:*time(0x21d98, 0xffbff684, 0x116f0)
- date -> libc.so.1:*nl_langinfo(0x3a, 0xffbff684, 0x21d98)
- date -> libc.so.1:*localtime(0x21d98, 0x3a, 0x21d98)
- date -> libc_psr.so.1:*memcpy(0xffbff5f4, 0xff2427b0, 0x24)
- date -> libc.so.1:*strftime(0x21d9c, 0x400, 0xff22b094)

- Shared library tracing
- Added in Solaris 2.6
- Not so clever with function arguments
$ apptrace date

date     -> libc.so.1:atexit(func = 0xff3bfaac) = 0x0

date     -> libc.so.1:atexit(func = 0x11654) = 0x0

date     -> libc.so.1:setlocale(category = 0x6, locale = "") = "C"

date     -> libc.so.1:textdomain(domainname = "SUNW_OST_OSCMD") =

    "SUNW_OST_OSCMD"

date     -> libc.so.1:getopt(argc = 0x1, argv = 0xffbfff67c, optstring = "a:u")

    = 0xffffffff errno = 0 (Error 0)

...

date     -> libc.so.1:strftime(s = "Fri May 12 17:16:12 ", maxsize = 0x400,

    format = "%a %b %e %T %Z %Y", timeptr = 0xffbfff5ec) = 0x1c

...

• Shared library tracing
• Added in Solaris 8
• Evaluates arguments
• DTrace has this functionality
$ truss -ua.out banner hello
...
-> _init(0x20f84, 0x21000, 0x0, 0x0)
<- _init() = 0x20f84
-> main(0x2, 0xffbff744, 0xffbff750, 0x21000)
brk(0x000216A0) = 0
brk(0x000236A0) = 0
-> banner(0xffbff84b, 0x2144c, 0x0, 0x0)
   -> banset(0x20, 0x2144c, 0x0, 0x0)
<- banset() = 32
   -> convert(0x68, 0x2144c, 0x0, 0x2164a)
<- convert() = 104
...

• Traces user functions
• -u can also trace library calls
adb/mdb

- Perform core dump analysis – processes
- Perform clash dump analysis – kernel
- mdb was added in Solaris 8
## mdb -k

### Loading modules:
```
[ unix krtld genunix specfs dtrace ufs ip sctp usba
random fctl nca cpc nfs sppp sd audiosup crypto ]
```

###_mappings

<table>
<thead>
<tr>
<th>BASE</th>
<th>LIMIT</th>
<th>SIZE</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>18f4000</td>
<td>8f4000</td>
<td>ktextseg</td>
</tr>
<tr>
<td>18f4000</td>
<td>1c80000</td>
<td>38c000</td>
<td>kvalloc</td>
</tr>
<tr>
<td>70000000</td>
<td>80000000</td>
<td>10000000</td>
<td>kvseg32</td>
</tr>
<tr>
<td>edd00000</td>
<td>f0000000</td>
<td>2300000</td>
<td>kdebugseg</td>
</tr>
<tr>
<td>2a10000000000</td>
<td>2a13ead80000</td>
<td>3ead8000</td>
<td>kpseg</td>
</tr>
<tr>
<td>2a75000000000</td>
<td>2a753adc0000</td>
<td>3adc000</td>
<td>kmapseg</td>
</tr>
<tr>
<td>3000000000000</td>
<td>7000000000000</td>
<td>4000000000000</td>
<td>kvseg</td>
</tr>
<tr>
<td>70000000000000</td>
<td>70000091600000</td>
<td>9160000</td>
<td>kmem64</td>
</tr>
<tr>
<td>fffffffffffffffff</td>
<td>fffffffffffffffe</td>
<td>4000000000000</td>
<td>kpmseg</td>
</tr>
</tbody>
</table>

- Visit kernel-land,
mdb -k

> ::pgrep syslog | ::print proc_t
{
  p_exec = 0x30002a822c0
  p_as = 0x300011326c8
  p_lockp = 0x30000e72780
  p_crlock = {
    _opaque = [ 0 ]
  }
  p_cred = 0x3000034ebe8
  p_swapcnt = 0
  p_stat = '\002'
  p_wcode = '\0'
  p_pidflag = 0

... walk kernel structures

* DTrace does this too
mdb -p

# mdb -p `pgrep -o syslogd`
Loading modules: [ ld.so.1 libc.so.1 ]
> ::mappings
    BASE   LIMIT   SIZE   NAME
 10000   22000   12000 /usr/sbin/syslogd
 32000   34000   2000  /usr/sbin/syslogd
 34000   64000   30000 /usr/sbin/syslogd
   fe87a000 fe87c000   2000  [ anon ]
   fe97a000 fe97c000   2000  [ anon ]
   fe980000 fe982000   2000  [ anon ]
   fe982000 fe99c000   1a000  [ anon ]
   fe99c000 fe99e000   2000  [ anon ]
   fe99e000 fe9b8000   1a000  [ anon ]
...

- Visit user-land
- *DTrace does this too (copyin/copyout)*
process accounting

# lastcomm

<table>
<thead>
<tr>
<th>Command</th>
<th>User</th>
<th>Terminal</th>
<th>Duration</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>man</td>
<td>root</td>
<td>term/a</td>
<td>0.02 secs</td>
<td>Fri May 12</td>
</tr>
<tr>
<td>sh</td>
<td>root</td>
<td>term/a</td>
<td>0.00 secs</td>
<td>Fri May 12</td>
</tr>
<tr>
<td>more</td>
<td>root</td>
<td>term/a</td>
<td>0.01 secs</td>
<td>Fri May 12</td>
</tr>
<tr>
<td>sh</td>
<td>root</td>
<td>term/a</td>
<td>0.00 secs</td>
<td>Fri May 12</td>
</tr>
<tr>
<td>mv</td>
<td>root</td>
<td>term/a</td>
<td>0.01 secs</td>
<td>Fri May 12</td>
</tr>
<tr>
<td>sh</td>
<td>root</td>
<td>term/a</td>
<td>0.00 secs</td>
<td>Fri May 12</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Log process creation
- Extended accounting adds task and flow
- *Limited details, no process args*
- *DTrace does this much better for short intervals*
BSM auditing

- Logs plenty of details
- Should be customised
- *Not really an as-needed troubleshooting tool*
- *DTrace serves this role*
Net-SNMP

$ snmpwalk -v1 -c public localhost|more
SNMPv2-MIB::sysDescr.0 = STRING: SunOS jupiter 5.10 Generic i86pc
SNMPv2-MIB::sysObjectID.0 = OID: NET-SNMP-MIB::netSnmpAgentOIDs.3
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (774102766) 89 days,
   14:17:07.66
SNMPv2-MIB::sysContact.0 = STRING: "System administrator"
SNMPv2-MIB::sysName.0 = STRING: jupiter
SNMPv2-MIB::sysLocation.0 = STRING: "System administrators office"
SNMPv2-MIB::sysServices.0 = INTEGER: 72
SNMPv2-MIB::sysORLastChange.0 = Timeticks: (90) 0:00:00.90
SNMPv2-MIB::sysORID.1 = OID: IF-MIB::ifMIB
SNMPv2-MIB::sysORID.2 = OID: SNMPv2-MIB::snmpMIB
SNMPv2-MIB::sysORID.3 = OID: TCP-MIB::tcpMIB
SNMPv2-MIB::sysORID.4 = OID: IP-MIB::ip
SNMPv2-MIB::sysORID.5 = OID: UDP-MIB::udpMIB
...

• Added to Solaris 10
... Unanswered Questions

- disk I/O by process. easily.
- network I/O by process.
- short-lived process analysis. easily.
- interrupt driver on-CPU time

- So what else couldn't you answer on older Solaris?
End of Chapter 1

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – *Introducing DTrace*
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTrace Toolkit
Chapter 10 – DTrace Mentality 2
Chapter 2

Introducing DTrace

- What DTrace is
- What role DTrace plays
- A one-liner Demonstration
- DTrace Scripting
- DTrace resources
DTrace

- Dynamic Tracing
- Solaris 10 & OpenSolaris
- Difficult to fully explain! We will use,
  a) features
  b) what it is like
  c) what is “DTrace”
  d) when to use it
  e) demonstrations
DTrace Features

- A framework for performance observability and debugging in real-time
- Examines from user space to the kernel,
  - user-land functions and instructions
  - library calls and instructions
  - system calls
  - kernel functions
  - device driver functions
- ... from the same tool
- Is safe to use in production
DTrace Features

- Can examine applications without restarting them
- Has low impact when running, and zero impact when not
- Dynamically inserts trace points into the kernel and running applications, called probes
- Can trace over 30,000 kernel probes
DTrace Features

- Can trace as many application probes as it finds functions and instructions (possibly millions)
- Can trace system boot activity – before init!
- Provides a C-like language for writing custom scripts
- Has solved countless long-term performance issue mysteries – often easily
- One of the most significant additions to the field of Operating Systems
What DTrace is like

DTrace has similar features to the following,

- truss  
  tracing system calls
- apptrace  
  tracing library calls
- truss -ua.out  
  tracing user functions
- prex/tnf*  
  tracing kernel functions
- lockstat  
  profiling the kernel
- mdb -k  
  access to kernel VM
- mdb -p  
  access to user VM
- C, awk  
  programming languages
- ...  
  + more
What is “DTrace”

The DTrace Framework

Providers

syscall

io

dtrace

Probes

...
When to use DTrace

Sysadmins can use DTrace for performance analysis and troubleshooting,

1. **Monitoring**
   - SNMP, sar, SunMC, ...

2. **Identification**
   - kstat (vmstat, mpstat, iostat)
   - procfs (ps, prstat), ...

3. **Analysis**
   --> DTrace <--
When to use DTrace

Application developers can use DTrace for code profiling and fault finding,

1. Development
   IDEs, vim/emacs

2. Testing
   Compiler profiling, coded statistics
   --> DTrace <--

3. Production
   coded statistics
   --> DTrace <--
Demonstration

- This demonstration will introduce key DTrace components and terminology
- The aim is to explain the following,

```bash
# dtrace -n 'syscall::exec*:return { trace(execname); }'
dtrace: description 'syscall::exec*:return ' matched 2 probes

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION::NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return staroffice</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return grep</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return grep</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return soffice</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return dirname</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return expr</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return basename</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return expr</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return sopatchlevel.sh</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return dirname</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return expr</td>
</tr>
</tbody>
</table>

...
Listing Probes

# dtrace -l

<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dtrace</td>
<td></td>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>dtrace</td>
<td></td>
<td>END</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>dtrace</td>
<td></td>
<td>ERROR</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>fbt</td>
<td>pool</td>
<td>pool_info</td>
<td>entry</td>
</tr>
<tr>
<td>5</td>
<td>fbt</td>
<td>pool</td>
<td>pool_info</td>
<td>return</td>
</tr>
<tr>
<td>6</td>
<td>fbt</td>
<td>pool</td>
<td>pool_detach</td>
<td>entry</td>
</tr>
</tbody>
</table>

# dtrace -l | wc -l

44797

- dtrace -l lists probes
- For this demonstration, there were 44796 probes. This number varies depending on the OS build, and which providers have been recently used.
Finding Probes

- Since there are so many probes, being able to find useful probes is a crucial task.
- grep can be used to filter the `dtrace -l` output, since grep is well known and RE's are powerful.

```
# dtrace -l | grep syscall | grep exec
1089 syscall exec entry
1090 syscall exec return
5991 syscall exece entry
5992 syscall exece return
```

- Here we matched two forms of exec()
### Specifying Probes #1

<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1089</td>
<td>syscall</td>
<td>syscall</td>
<td>exec entry</td>
</tr>
</tbody>
</table>

```
# dtrace -ln 'syscall::exec:entry'
```

```
# dtrace -ln 'syscall::exec*:'
```

- First, a single probe was matched by specifying its fully qualified probe name
- Second, multiple probes were matched. “*” is a wildcard, as are blank fields “::”
Specifying Probes #2

- Short cuts exist – dropping left-hand fields is the same as wildcards.
- For example, the following are the same
  - `dtrace:::BEGIN`
  - `:::BEGIN`
  - `BEGIN`
  - `dtrace:::BEGIN` fires when DTrace begins tracing, much the same as awk's `BEGIN` (and used for the same reasons too – printing headers).
Specifying Probes #3

- The following are *not* the same,
  - syscall::read:entry
  - read:entry

# dtrace -ln 'read:entry'

<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1073</td>
<td>syscall</td>
<td></td>
<td>read entry</td>
</tr>
<tr>
<td>12978</td>
<td>fbt</td>
<td>genunix</td>
<td>read entry</td>
</tr>
<tr>
<td>49474</td>
<td>pid581</td>
<td>libc.so.1</td>
<td>read entry</td>
</tr>
</tbody>
</table>

- If we only intended to match the syscall read(),
  this shortcut matches others by mistake.

**Brendan's Style Hint:**
Use fully-qualified probe names, they are safest
# dtrace -ln 'syscall::exec:entry'

Provider
A library of probes. The provider defines the remaining fields.

## To best understand these last fields,
1. choose provider
2. see chapter in the DTrace Guide
Tracing

• After finding the desired probes \((-1)\), you are ready to trace activity,

```bash
# dtrace -n 'syscall::exec*:

  dtrace: description 'syscall::exec*:' matched 4 probes

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5991</td>
<td>exece:entry</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return</td>
</tr>
<tr>
<td>0</td>
<td>5991</td>
<td>exece:entry</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return</td>
</tr>
<tr>
<td>0</td>
<td>5991</td>
<td>exece:entry</td>
</tr>
</tbody>
</table>
...
```

• This is the default output

• Each output line represents a probe firing
# dtrace -n 'syscall::exec*:'

Dtrace Probe-id. not so interesting

DTrace

CPU ID
0 5991
0 5992

FUNCTION:NAME
exece:entry
exece:return

The CPU-id. if this changes, the output may be shuffled!

Last 2 fields of probe that fired

How many probes your probe description actually matched

matched 4 probes
Actions

- A custom action can be performed when a probe fires

```bash
# dtrace -n 'syscall::exec*: { trace(execname); }'
dtrace: description 'syscall::exec*: ' matched 4 probes
CPU       ID        FUNCTION:NAME
0  5991          exece:entry   bash
0  5992          exece:return   date
0  5991          exece:entry   bash
0  5992          exece:return   ls
...
```

- `trace()` prints one argument
- `execname` is the process name
Process Creation

- fork variants: fork1(2), forkall(2), vfork(2)
- exec variants: exec(2), exece(2)
Both syscall entry and return can be traced
Tracing Process Creation

```
# dtrace -n 'syscall::exec*:return { trace(execname); }'
dtrace: description 'syscall::exec*:return' matched 2 probes
CPU     ID                    FUNCTION:NAME
 0   5992                     exece:return   staroffice
 0   5992                     exece:return   grep
 0   5992                     exece:return   grep
 0   5992                     exece:return   soffice
 0   5992                     exece:return   dirname
 0   5992                     exece:return   expr
 0   5992                     exece:return   basename
 0   5992                     exece:return   expr
 0   5992                     exece:return   sopatchlevel.sh
 0   5992                     exece:return   dirname
...
```

• For exec() the return probe is traced, as this is when
  the destination execname is available on-CPU
• This shows new processes from running “soffice”
Terminology

- A summary of DTrace terminology,

```bash
# dtrace -n 'syscall::exec*:return { trace(execname); }'
```

Consumer  
Probe  
Action

Provider  
Module  
Function  
Name

It's important to be able to talk-the-talk...
There are many providers,

- **syscall**  System call entry and return probes
- **vminfo**  Virtual Memory statistic probes
- **sysinfo** Classic sysinfo statistic probes
- **io**  Disk and NFS events
- **sched**  system scheduling events
- **profile**  fixed sampling
- **dtrace**  program BEGIN/END probes
- **pid**  user-level tracing
- **fbt**  raw kernel tracing
- ...
Provider Questions

1. Where are syscalls documented?

2. The pid provider traces user-level activity:
   Where are library functions documented?
   Where are user functions documented?

3. The fbt provider traces kernel-level activity:
   Where are kernel functions documented?

4. DTrace provides its own custom providers. Where would they be documented?
Provider Documentation

Documentation
- man(2)
- DTrace Guide
- man(3C)
- source code

Providers
- syscall
- vminfo
- sched
- io
- profile
- dtrace
- ...
- pid
- fbt

Solaris Internals 2nd Ed
(cvs.opensolaris.org)

“The Best Of”
(Stable)
Action Intro

- `trace()` prints 1 argument
- `printf()` may also be used
- DTrace variable builtins include,
  - `execname` process name
  - `pid` process ID
  - `timestamp` time since boot, nanoseconds
  - `probefunc` probe function component
  - `probename` probe name component
  - `curthread` pointer to current thread
  - `curpsinfo` pointer to psinfo like structure
printf Example

- Enhancing our previous one-liner,

```c
# dtrace -n 'syscall::exec*:return { printf("%6d %s", pid, execname); }'
dtrace: description 'syscall::exec*:return ' matched 2 probes

CPU     ID                    FUNCTION:NAME
 0   5992                     exece:return  28751 staroffice
 0   5992                     exece:return  28752 grep
 0   5992                     exece:return  28755 grep
 0   5992                     exece:return  28753 soffice
 0   5992                     exece:return  28753 soffice
...
```

- format characters include,

  `\%d` integers  
  `\%f` floats  
  `\%s` strings  
  `\%S` safe strings – escaped binary characters  
  `\%Y` text formatted time (use with “walltimestamp”)

Style: spaces after commas.
# dtrace -n 'syscall::exec*:return { printf("%-20Y %6d %s", walltimestamp, pid, execname); }'

dtrace: description 'syscall::exec*:return' matched 2 probes

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return 2006 May 22 02:12:36 28762 staroffice</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return 2006 May 22 02:12:36 28763 grep</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return 2006 May 22 02:12:36 28765 grep</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return 2006 May 22 02:12:36 28766 soffice</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return 2006 May 22 02:12:36 28767 dirname</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return 2006 May 22 02:12:36 28764 expr</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return 2006 May 22 02:12:36 28768 basename</td>
</tr>
</tbody>
</table>

• **Our output still contains the DTrace defaults.** This can be eliminated using -q.

• **Our command is > 80 characters.** This can be better managed using a script file.
quiet mode

# dtrace -qn 'syscall::exec*::return { printf("%-20Y %6d %s\n", walltimestamp, pid, execname); }'
2006 May 22 02:14:36 28773 staroffice
2006 May 22 02:14:36 28774 grep
2006 May 22 02:14:36 28776 grep
2006 May 22 02:14:36 28777 soffice
2006 May 22 02:14:36 28778 dirname
2006 May 22 02:14:36 28775 expr
2006 May 22 02:14:36 28779 basename
2006 May 22 02:14:36 28780 expr
2006 May 22 02:14:36 28781 sopatchlevel.sh
2006 May 22 02:14:36 28782 dirname
^C

- dtrace -q suppresses the default output
- A \n is now needed to terminate each line
DTrace Script File

# cat -n exec.d
1  #!/usr/sbin/dtrace -qs
2
3  syscall::exec*:return
4  {
5       printf("%20Y %6d %s\n", walltimestamp, pid, execname);
6  }

# chmod 755 exec.d
#
# ./exec.d
2006 May 22 02:20:59  28788 staroffice
2006 May 22 02:20:59  28789 grep
2006 May 22 02:20:59  28791 grep
2006 May 22 02:20:59  28792 soffice
2006 May 22 02:20:59  28793 dirname
...

• So far, so good. The script file is easier to edit.
• Some improvements can be made...
# cat -n exec.d
 1 #!/usr/sbin/dtrace -s
 2
 3 #pragma D option quiet
 4
 5 dtrace:::BEGIN
 6 { 
 7     printf("%-20s %6s %s\n", "TIME", "PID", "CMD");
 8 }
 9
 10 syscall::exec*::return 
 11 { 
 12     printf("%-20Y %6d %s\n", walltimestamp, pid, execname); 
 13 } 

# /
exec.d

<table>
<thead>
<tr>
<th>TIME</th>
<th>PID</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006 May 22</td>
<td>28801</td>
<td>staroffice</td>
</tr>
<tr>
<td>2006 May 22</td>
<td>28802</td>
<td>grep</td>
</tr>
<tr>
<td>2006 May 22</td>
<td>28804</td>
<td>grep</td>
</tr>
<tr>
<td>2006 May 22</td>
<td>28805</td>
<td>soffice</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DTrace Scripting

- Seems like awk!
- “pragma D option quiet” is the same as -q, however is more obvious (especially when more options are used).
- Printing a heading serves to both label the output, and to indicate when DTrace has began tracing.
  - dtrace:::BEGIN probe fires at start (headings)
  - dtrace:::END probe fires at end (reports)
curpsinfo->pr_psargs

- To print more than just the execname: try,

```sh
# dtrace -qn 'syscall::exec*:return { printf("%6d %s\n", pid,
    curpsinfo->pr_psargs); }'
```

```
29396 /bin/sh /usr/bin/soffice
29397 grep StarOffice 7 /export/home/brendan/.sversionrc
29398 grep StarOffice 7 /export/home/brendan/.sversionrc
29396 /bin/sh /usr/staroffice7/program/soffice
29400 /usr/bin/sh /usr/bin/dirname /usr/staroffice7/program/soffice
29400 /usr/bin/expr /usr/staroffice7/program/soffice/ : \((/\))*/[^/]*//*$ | /usr/staroffice7/program/soffice
29402 /usr/bin/sh /usr/bin/basename /usr/staroffice7/program/soffice
29402 /usr/bin/expr //usr/staroffice7/program/soffice : \(.*[^/\)]/\)/*/.*$ : . */\(.*\) : 
29404 /bin/sh /usr/staroffice7/program/sopatchlevel.sh
29405 /usr/bin/sh /usr/bin/dirname /
    /usr/staroffice7/program/sopatchlevel.sh
29405 /usr/bin/expr /usr/staroffice7/program/sopatchlevel.sh/ : \((/\))/^[^/]*//^[^/]*//^[^/]*$ | /
29406 uname -s
^C
```
DTrace Resources

- OpenSolaris DTrace Community
  http://www.opensolaris.org/os/community/dtrace

- DTrace Guide
  http://docs.sun.com/app/docs/doc/817-6223

- DTrace Tools
  http://www.brendangregg.com/dtrace.html

- Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris. McDougall/Mauro/Gregg
  ISBN: 0-13-156819-1
End of Chapter 2

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTraceToolkit
Chapter 10 – DTrace Mentality 2
Chapter 3

Command Line DTrace

- The syscall Provider
- Entry/Return Arguments
- Predicates
- Aggregations
- Stack tracing
- Intro to sysinfo Provider
- Intro to profile Provider
syscall Provider

- This provider traces syscall entry and returns
- syscalls are the interface between user-land and the kernel, and reflect much of an applications behaviour
- syscalls are well documented – man(2)
- This provider is a great place to start learning DTrace
Entry Arguments

- The syscall entry arguments are available as unsigned 64-bit ints, with the names: arg0, arg1, arg2, ...
- These arguments can be casted if need be to the appropriate types. eg, (int)arg0
- Typed versions may be provided as args[0], args[1], args[2], ...
- Arguments are listed in the syscall's man page
Entry Argument Examples

- syscall::read::entry,
  - arg0 (int) file descriptor
  - arg1 (void *) buffer
  - arg2 (size_t) requested read size

- syscall::mkdir::entry,
  - arg0 (char *) path
  - arg1 (mode_t) mode

- String pointers can't be read unless you dereference them. Use copyinstr().
Entry Arguments: Integers

```
# dtrace -n 'syscall::read:entry { trace(arg2); }'
dtrace: description 'syscall::read:entry ' matched 1 probe

CPU     ID                    FUNCTION:NAME
0   1073                       read:entry              8192
0   1073                       read:entry              8192
0   1073                       read:entry              8192
0   1073                       read:entry              1495
0   1073                       read:entry              1329
0   1073                       read:entry               785
0   1073                       read:entry              4096
0   1073                       read:entry              4096

...`
# dtrace -n 'syscall::open:entry { trace(copyinstr(arg0)); }'

dtrace: description 'syscall::open: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>1077</td>
<td>open:entry /var/ld/ld.config</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry /usr/lib/libc.so.1</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry /var/ld/ld.config</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry /usr/lib/libgen.so.1</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry /usr/lib/libc.so.1</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry /var/ld/ld.config</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry /usr/lib/libc.so.1</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry /var/ld/ld.config</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry /usr/lib/libc.so.1</td>
</tr>
</tbody>
</table>

... 

- `copyinstr()` is a convenient function to fetch strings from user-land to the kernel.
- `copyin()` also exists, for fetching any data.
Return Code

- The syscall return code is available from the return probe as arg0.
- The man page for the syscall should explain what the return code is.
- In general, the return code is,
  -1 failure
  >= 0 success
- If a syscall fails, the error code is available as the DTrace built-in “errno”. See /usr/include/sys/errno.h for a description.
Return Code Example

- For example, syscall::read:return has, 
  arg0  number of successful bytes read, 
or -1 for failure

```shell
# dtrace -n 'syscall::read:return { trace(arg0); }'
dtrace: description 'syscall::read:return' matched 1 probe
```

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1074</td>
<td>read:return</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>0</td>
<td>1074</td>
<td>read:return</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>0</td>
<td>1074</td>
<td>read:return</td>
<td></td>
<td>63</td>
</tr>
<tr>
<td>0</td>
<td>1074</td>
<td>read:return</td>
<td></td>
<td>63</td>
</tr>
<tr>
<td>0</td>
<td>1074</td>
<td>read:return</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>0</td>
<td>1074</td>
<td>read:return</td>
<td></td>
<td>315</td>
</tr>
<tr>
<td>0</td>
<td>1074</td>
<td>read:return</td>
<td></td>
<td>80</td>
</tr>
</tbody>
</table>

... 

- Most actual bytes read are < 1 Kbyte.
Return Code Hints

• The return code is often typed as an int, but sometimes (x86) isn't,

```plaintext
# dtrace -n 'syscall::open:return { trace(arg0); }'
dtrace: description 'syscall::open:return ' matched 1 probe
CPU     ID                    FUNCTION:NAME
     0   1078                      open:return        4294967295
     0   1078                      open:return         3
     0   6280                    open64:return        4294967295
^C
# dtrace -n 'syscall::open:return { trace((int)arg0); }'
dtrace: description 'syscall::open:return ' matched 1 probe
CPU     ID                    FUNCTION:NAME
     0   1078                      open:return        -1
     0   1078                      open:return         3
     0   6280                    open64:return        -1

• Here a “cat /etc/shadow” was run as non-root on x86, with arg0 uncasted and casted.
Questions

• Would it be useful to only trace syscalls that failed? (ie, return arg0 == -1)
• Would it be useful to trace syscalls with unusual entry arguments: eg, read()s of a size less than 8 bytes?
• Would it be useful to match on execname and pid?
• You could just dump everything and grep later. But there is a better way...
Predicates

- Allow filtering of trace data
- Are a boolean expression used to determine if an action is performed
- For example,

```plaintext
# dtrace -n 'syscall::read:entry /arg2 < 8/ { printf("%-16s %d", execname, arg2); }

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

CPU     ID                    FUNCTION:NAME
0   1073                       read:entry bash             1
0   1073                       read:entry bash             1
0   1073                       read:entry bash             1
...

This matches < 8 byte requested read()s.
```
Predicates

- General Syntax,

\[
\text{probe-description} \\
/\text{predicate}/ \\
\{
  \text{action;}
  \text{action;}
  \ldots
\}
\]

- This is the most common way DTrace does if-then-else statements (which DTrace does not have directly).

Style:
place the probe, predicate and braces on separate lines.
Predicate Examples

- `/execname == "ls"/` process is “ls” - will match multiple processes, all called “ls”
- `/(int)arg0 < 0/` arg0 as int is less than 0
- `/pid != 0/` not the kernel
- `/pid/` same as above
- `/execname != "dtrace" && pid != 0/` neither the dtrace command nor the kernel
truss-like DTrace

# dtrace -n 'syscall:::entry /execname == "ls"/ { printf("%x %x %x", arg0, arg1, arg2); }'
dtrace: description 'syscall:::entry ' matched 231 probes

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6247</td>
<td>resolvepath:entry 8047fed 804738c 3ff</td>
</tr>
<tr>
<td>0</td>
<td>6119</td>
<td>sysconfig:entry 6 468d fec1d444</td>
</tr>
<tr>
<td>0</td>
<td>6247</td>
<td>resolvepath:entry d27fd9dc 804738c 3ff</td>
</tr>
<tr>
<td>0</td>
<td>6093</td>
<td>xstat:entry 2 8047fed 80477b8</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry d27f7a24 0 0</td>
</tr>
<tr>
<td>0</td>
<td>6093</td>
<td>xstat:entry 2 d27fbf38 8047070</td>
</tr>
<tr>
<td>0</td>
<td>6247</td>
<td>resolvepath:entry d27fbf38 80470f8 3ff</td>
</tr>
<tr>
<td>0</td>
<td>1077</td>
<td>open:entry d27fbf38 0 0</td>
</tr>
</tbody>
</table>

- This snoops syscalls from all processes called “ls”
- It prints three arguments in hex (or tries to)
- The output is verbose. Much more so for bigger apps. Remember truss -c?
Aggregations

- A DTrace variable type.
- In rough order of complexity,
  - char
  - int
  - long
  - double
  - float
  - pointer
  - array
  - associative array
  - aggregation
  - struct
  - object
Why Aggregations First

• Aggregations are introduced early because we want you to use them as much as possible!

• Their advantages are:
  • They summarise the data in often the most appropriate way – providing the final report
  • Aggregations are fast – they do not lock between CPUs in the same way as other data types
# dtrace -n 'syscall:::entry { @fred = count(); }'

```bash
dtrace: description 'syscall:::entry' matched 231 probes
^C
```

11783

- The Aggregation name is “@fred”
- The Aggregation function is “count()” - which counts occurrences
- When Ctrl-C is hit, DTrace prints @fred. This is for convenience. Manually printing aggregations is possible if desired.
- Here, DTrace observed 11783 system calls.
Aggregation by execname

```
# dtrace -n 'syscall:::entry { @fred[execname] = count(); }'
dtrace: description 'syscall:::entry ' matched 231 probes
^C

svc.configd           1
svc.startd            3
nautilus              3
httpd                 6
nscd                  38
java                  73
bash                  95
dtrace                119
sshd                  128
acroread              321
find                  15068
```

- The key is now execname, using "@fred[execname]"
- This is producing a frequency count by execname report
- Here, "find" processes caused 15,068 syscalls
Aggregation by syscall

```
# dtrace -n 'syscall:::entry /execname == "find"/ { @fred[probefunc] =
    count(); }'
dtrace: description 'syscall:::entry ' matched 231 probes
^C

write 47
fcntl 127
fsat 127
close 129
fstat64 254
fchdir 256
getdents64 261
acl 3131
gtime 3131
lstat64 3132
```

- “probefunc” is the 3rd probe field, which for the syscall provider is the syscall name
- Here, we can see which syscalls find is calling
Aggregation Syntax

- The general syntax is, 
  \[@name[\text{key}] = \text{function(args)}\]
- The name and the key are optional
- Examples,
  - \[@\text{num}[\text{execname}] = \text{count()};\]
  - \[@\text{execname}] = \text{count()};\]
  - \[@\text{total} = \text{count()};\]
- Multiple keys can be used. Eg,
  - \[@\text{num}[\text{pid, execname}] = \text{count()};\]

Style:
I like to name my aggregates.
Multiple Keys

```bash
# dtrace -n 'syscall:::entry { @num[pid, execname] = count(); }'
dtrace: description 'syscall:::entry ' matched 231 probes
^C

3104  gnome-terminal                                                   2
3153  gnome-terminal                                                   2
3098  nautilus                                                         3
4804  java                                                            10
599   sshd                                                            24
8117  acroread                                                        45
28921 dtrace                                                          71
113   nscd                                                           270
28920 find                                                           3418
```

- The key is now [pid, execname]
- DTrace has printed them in neat columns

Any fancier key combinations, and DTrace may not be so neat – we'll need to tweak the output.
Aggregating Functions

- So far we have only seen `count()`. Which is certainly fairly useful.
- Aggregating functions,
  - `count()` count occurrences
  - `sum(value)` sum value
  - `avg(value)` average value
  - `min(value)` find value minimum
  - `max(value)` find value maximum
  - `quantize(value)` power-of-2 distribution plot
  - `lquantize(value, min, max, step)` linear distribution plot
Aggregation Demos

• The Aggregating functions will be used to analyse the behaviour of the following “wc” command:

# wc /usr/share/man/windex

• This file is 931886 bytes (910 Kbytes)
• The “wc” command will read through the input file – so there should be many syscalls to trace.
Aggregation sum()

# dtrace -n 'syscall::read:entry { @bytes[execname] = sum(arg2); }'
dtrace: description 'syscall::read:entry ' matched 1 probe
^C

bash 4
sshd 98304
wc 942080

- This sums the requested read() bytes by execname
- “wc” requested 942080 bytes. This almost matches the file,

# ls -l /usr/share/man/windex
-rw-r--r-- 1 root root 931886 Dec 22 14:23 /usr/share/man/windex

- Why would there be a difference?
sum() on return

- Summing the return is now attempted,

  # dtrace -n 'syscall::read:return { @bytes[execname] = sum(arg0); }'
dtrace: description 'syscall::read:return ' matched 1 probe

  ^C

  sshd 188
  wc 931886

- Perfect match!

  # ls -l /usr/share/man/windex
  -rw-r--r-- 1 root root 931886 Dec 22 14:23 /usr/share/man/windex

- Although be careful: syscalls can return -1 for failure – which would wreck our sum() value. A predicate could be used to avoid this.
Questions

- What would be the size of each read?
- How many reads would occur?
Aggregation avg()

- The average read size can be aggregated:

  ```
  # dtrace -n 'syscall::read:return { @bytes[execname] = avg(arg0); }'
  dtrace: description 'syscall::read:return' matched 1 probe
  ^C
  
  bash  1
  sshd  37
  wc    8103
  ```

- The average read size is 8103 bytes.
- Since 8 Kbytes == 8192 bytes, it sounds like there is an 8 Kbyte ceiling. Could we check this?
Aggregation min(), max()

• The following fetches the min then max:

```bash
# dtrace -n 'syscall::read:return { @bytes[execname] = min(arg0); }'
dtrace: description 'syscall::read:return ' matched 1 probe
^R
^C

wc                                           0
bash                                         1
sshd                                         2

# dtrace -n 'syscall::read:return { @bytes[execname] = max(arg0); }'
dtrace: description 'syscall::read:return ' matched 1 probe
^C

bash                                         1
sshd                                         52
wc                                           8192

• 8.00 Kbytes is indeed the maximum.
Aggregation count()

- Counting syscalls is always useful:
  
  ```
  # dtrace -n 'syscall::read:entry { @num[execname] = count(); }'
  dtrace: description 'syscall::read:entry ' matched 1 probe
  ^C
  
  bash  4
  sshd  8
  wc 115
  ```

- 115 reads()s occurred.
- This makes sense,
  - average read size: 8103 bytes
  - file size: 931886 bytes
  - $931886 / 8103 = 115.005$
Aggregations so far

• By running,
  • count()
  • sum()
  • min(), max()
  • avg()
• we have learnt much about the behaviour of the wc command.
• Similar knowledge can be gained much more quickly using quantize().
Aggregation quantize()

# dtrace -n 'syscall::read:return { @dist[execname] = quantize(arg0); }'
dtrace: description 'syscall::read:return ' matched 1 probe
^C
...

wc

<table>
<thead>
<tr>
<th>value</th>
<th>------------ Distribution ------------</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>113</td>
</tr>
<tr>
<td>16384</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
## Distribution Plots

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>###</td>
<td>113</td>
</tr>
<tr>
<td>16384</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- Conveys a count, min, max, avg, (and with some mental calculations) a sum; all at once!

**occurrence was at least this value**

**number of occurrences**

**mostly 8192 to 16383 byte reads**
Aggregation lquantize()

- When power-of-2 distributions are not suitable, lquantize provides linear customisable distributions.
- The arguments are, lquantize(value, min, max, step)
Aggregation lquantize()

```c
# dtrace -n 'syscall::read:entry { @dist[execname] = lquantize(arg0, 0, 128, 1); }'
dtrace: description 'syscall::read:entry' matched 1 probe
^C

bash
value  --------------- Distribution --------------- count
< 0 |                        0
    0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
    1 |                        0
...

wc
value  --------------- Distribution --------------- count
  2 |                        0
  3 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 115
  4 |                        0

• This is read() file discriptor. wc used FD 3, while bash using FD 0 (STDIN).```
Remember that raw data can always be dumped, for other post processing:

```bash
# dtrace -n 'syscall::read:return /execname == "wc"/ { trace(arg0); }'
dtrace: description 'syscall::read:return ' matched 1 probe
CPU     ID                    FUNCTION:NAME
0   1074                      read:return              8192
0   1074                      read:return              8192
0   1074                      read:return              8192
...                      
0   1074                      read:return              8192
0   1074                      read:return              6190
0   1074                      read:return              8192
^C
```

Aggregations are great summaries, but sometimes you just want the raw data.
Stack Traces

- DTrace provides 3 stack trace functions:
  - `stack()` kernel stack trace
  - `ustack()` user stack trace
  - `jstack()` java stack trace

- These explain why your event is occurring – the ancestry.

- `dtrace(1M)` needs the process to still exist to evaluate symbols for `ustack()` and `jstack()`.
ustack()

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

CPU     ID                    FUNCTION:NAME
0   1074                      read:return
    libc.so.1`_read+0x15
    bash`rl_getc+0x1f
    bash`rl_read_key+0xad
    bash`readline_internal_char+0x5c
    bash`0x80abf72
    bash`0x80abf8c
    bash`readline+0x37
    bash`0x80675ad
    bash`0x8067525
    bash`0x8067d2b
    bash`0x80686b4
    bash`0x8068275
    bash`yyparse+0x12f
    bash`parse_command+0x56
    bash`read_command+0x8c
    bash`reader_loop+0xdd
    bash`main+0x638
    bash`0x806395a

^C
ustack() short-lived processes

- Now we will run ustack() on the read() from the short-lived “wc” command:

```bash
# dtrace -n 'syscall::read:return /execname == "wc"/ { ustack(); }'
dtrace: description 'syscall::read:return ' matched 1 probe
CPU     ID                    FUNCTION:NAME
0   1074                      read:return
    0xd279e465                  
    0xd27856de
    0x8050c70
    0x8050a56
```

- wc completes and dissapears before the dtrace(1M) command can read its symbol table.
ustack() and mdb

- One technique is to use mdb to set a breakpoint on exit:

```
$ mdb /usr/bin/wc
  > exit:b
  > :r /usr/share/man/windex
     11821 113846 931886 /usr/share/man/windex
mdb: stop at 0x805090c
mdb: target stopped at:
PLT:exit: jmp *0x8062014
> 

# dtrace -n 'syscall::read:return /execname == "wc"/ { ustack(); }'
dtrace: description 'syscall::read:return ' matched 1 probe
CPU     ID                    FUNCTION:NAME
 0   1074                      libc.so.1`_read+0x15
     libc.so.1`fread+0xb6
     wc`main+0x184
     wc`0x8050a56
```

...
ustack() aggregations

- The entire stack trace can be used as a key.
- This measures the most frequent stack trace that caused the probe.
- The stack can be truncated using an integer argument: eg, ustack(5) for 5 lines only.

```bash
# dtrace -n 'syscall::read:return /execname == "zsh"/ { @num[ustack()] = count(); }
```

dtrace: description 'syscall::read:return ' matched 1 probe

```
^C
```

```
libc.so.1`_read+0x15
libc.so.1`getlogin+0x2a
zsh`createparamtable+0x1e2
zsh`setupvals+0x5bd
zsh`zsh_main+0x1ae
zsh`main+0xe
zsh`0x8059b9e
1
```

This stack trace occurred only once
... continued

... 

```
libc.so.1`_read+0x15
zle.so`0xd251fd22
zle.so`getkey+0x130
zle.so`0xd251f520
zle.so`getkeymapcmd+0x48
zle.so`getkeycmd+0x2c
zle.so`zlecore+0x6b
zle.so`zleread+0x403
zsh`autoload_zleread+0x3e
zsh`0x8079c59
zsh`ingetc+0x6d
zsh`0x80734fa
zsh`gettok+0x18
zsh`yylex+0x17
zsh`parse_event+0x23
zsh`loop+0x91
zsh`zsh_main+0x1d0
zsh`main+0xe
zsh`0x8059b9e
20
```

This entire stack trace occurred 20 times
Back to the Syscall Provider

- So far we have just used the syscall provider
- There are some difficulties to be aware of:
  - when tracing writes – be aware that there is `write()`, `writev()`, `pwrite()`, `pwrite64()`
  - when tracing reads – be aware that there is `read()`, `readv()`, `pread()`, `pread64()`
    There is also `readlink()` - for symlinks
  - other variants exist. `open()`, `open64()`...

- `readv()` and `writev()` cause problems –
  fetching the size from an iovec isn't easy
sysinfo Provider Intro

- sysinfo provides probes that trace regular system activity
- sysinfo:::readch traces all the read variants, and provides the successful bytes read as arg0
- sysinfo:::writech traces all the write variants, and provides the successful bytes written as arg0
- Don't need to worry about syscall return of -1
Bytes Read by Process Name

- sysinfo does this easily,

  ```
  # dtrace -n 'sysinfo:::readch { @bytes[execname] = sum(arg0); }'
  dtrace: description 'sysinfo:::readch ' matched 4 probes
  ^C
  
  bash 14
  sshd 591
  ```

- Handles read variants
- This could use quantize(), or trace writech
- We should now be able to construct many useful one-liners using syscall, sysinfo and aggregations
profile Provider Intro

- profile samples activity at a custom rate
- The maximum sample rate is 5000 Hertz

```bash
# dtrace -n 'profile:::profile-100hz { @num[execname] = count(); }'
dtrace: description 'profile:::profile-100hz' matched 1 probe
^C

dirname       1
bash          2
sopatchlevel.sh 2
uname         2
expr          3
soffice       3
staroffice    3
javaldx       4
soffice.bin   8
pagein        84
sched         389
```

- This is sampling execname at 100 Hertz
profile and ustack()

```c
# dtrace -n 'profile::profile-100hz /execname == "prstat"/ {
    @num[ustack(5)] = count(); }
'dtrace: description 'profile::profile-100hz' matched 1 probe
^C

libc.so.1`memcpy+0x41
prstat`0x8053a54
prstat`main+0x791
prstat`0x8051dc2
  1
...

libc.so.1`_pread+0x15
prstat`0x8053606
prstat`0x80537d3
prstat`main+0x791
prstat`0x8051dc2
  39
```

- This can help identify where a process is spending its time
Using profile

• The profile provider allows simple and useful one-liners and scripts to be written

• The overheads of using profile are fixed to the sampling rate; unlike other providers, where a high frequency of traced events can slow the target.

• Be aware of sampling issues. If accurate measurements are desired, use other providers to measure on an event-by-event basis.
End of Chapter 3

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTrace Toolkit
Chapter 10 – DTrace Mentality 2
Chapter 4

DTrace one-liners

- List useful one-liners
- Discuss when to use them
One-Liners Are Great

- Remember “handy one-liners for sed”?
- One-liners can provide simple solutions to common problems
- One-liners may not need the same arduous approval procedures that scripts may require
- Someone once emailed me to say:

  thanks for the DTraceToolkit. I got as far as the one-liners, and found I could do everything I needed to.

  (so much for writing 100+ scripts!)
Useful One-Liners

- The DTrace one-liners list is kept in the DTraceToolkit under Docs/oneliners.txt (http://www.opensolaris.org/os/community/dtrace/dtracetoolkit)
- So far we have covered enough DTrace syntax to understand most of them.
- Many probes they use will be new, but readily understandable in such a practical context.
- I will demonstrate many of these live...
# New processes with arguments,
`dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'`

# Files opened by process name,
`dtrace -n 'syscall::open*:entry { printf("%s %s",execname,copyinstr(arg0)); }'`

# Files created using creat() by process name,
`dtrace -n 'syscall::creat*:entry { printf("%s %s",execname,copyinstr(arg0)); }'`

# Syscall count by process name,
`dtrace -n 'syscall:::entry { @num[execname] = count(); }'`

# Syscall count by syscall,
`dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'`

# Syscall count by process ID,
`dtrace -n 'syscall:::entry { @num[pid,execname] = count(); }'`

# Read bytes by process name,
`dtrace -n 'sysinfo:::readch { @bytes[execname] = sum(arg0); }'`

# Write bytes by process name,
`dtrace -n 'sysinfo:::writech { @bytes[execname] = sum(arg0); }'`
One-liners #2

```bash
# Read size distribution by process name,
dtrace -n 'sysinfo:::readch { @dist[execname] = quantize(arg0); }'

# Write size distribution by process name,
dtrace -n 'sysinfo:::writech { @dist[execname] = quantize(arg0); }'

# Disk size by process ID,
dtrace -n 'io:::start { printf("%d %s %d",pid,execname,args[0]->b_bcount); }'

# Disk size aggregation

dtrace -n 'io:::start { @size[execname] = quantize(args[0]->b_bcount); }'

# Pages paged in by process name,
dtrace -n 'vminfo:::pgpgin { @pg[execname] = sum(arg0); }'

# Minor faults by process name,
dtrace -n 'vminfo:::as_fault { @mem[execname] = sum(arg0); }'

# Interrupts by CPU,
dtrace -n 'sdt:::interrupt-start { @num[cpu] = count(); }'

# CPU cross calls by process name,
dtrace -n 'sysinfo:::xcalls { @num[execname] = count(); }'
```
One-liners #3

# Lock time by process name,
dtrace -n 'lockstat:::adaptive-block { @time[execname] = sum(arg1); }'

# Lock distribution by process name,
dtrace -n 'lockstat:::adaptive-block { @time[execname] = quantize(arg1); }'

# Kernel function calls by module

dtrace -n 'fbt:::entry { @calls[probemod] = count(); }'

# Stack size for processes

dtrace -n 'sched:::on-cpu { @[execname] = max(curthread->t_procp->p_stksize); }'

# Kill all top processes when they are invoked,
dtrace -wn 'syscall::exece:return /execname == "top"/ { raise(9); }'

# New processes with arguments and time,
dtrace -qn 'syscall::exec*:return { printf("%Y %s
", walltimestamp, curpsinfo->pr_psargs); }'

# Successful signal details,
dtrace -n 'proc:::signal-send /pid/ { printf("%s -%d %d", execname, args[2], args[1]->pr_pid); }'
End of Chapter 4

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTraceToolkit
Chapter 10 – DTrace Mentality 2
Chapter 5

DTrace Mentality 1

- Approach Strategies
- Thinking in DTrace
Strategy #1: snoop or summary

- **snoop**
  - watch events as they occur
  - raw data shows all details
  - may be too verbose
  - achieved using trace() or printf()

- **summary**
  - produce a summary report of data
  - often the fastest way to process data
  - may lose information due to summarising
  - achieved using aggregates @
# 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>5992</td>
<td>exece:return staroffice</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return grep</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return grep</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return soffice</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return dirname</td>
</tr>
<tr>
<td>0</td>
<td>5992</td>
<td>exece:return expr</td>
</tr>
</tbody>
</table>

• Output “chugs” along at 1 Hertz. To improve:
  • dtrace -x switchrate=10hz command line
  • #pragma D option switchrate=10hz script
• Output can be shuffled slightly – check the CPU column, or print timestamp and post-process.
# dtrace -n 'syscall::exece:return { @num[execname] = count(); }'
dtrace: description 'syscall::exece:return' matched 1 probe
^C

soffice.bin
pagein
javaldx
basename
sopatchlevel.sh
soffice
staroffice
dirname
grep
expr
uname

• The most frequent occurring item is often an issue, but not always.
Strategy #2: drill down analysis

- Start broad, then focus on potential issues
- For example:

```
# vmstat 1

kthr      memory            page            disk          faults      cpu
r b w   swap  free  re  mf pi po fr de sr cd s0 -- --   in   sy   cs us sy id
0 0 41 857772 109964 20 86 24 1 1 0 4 2 0 0 0 277 416 216 1 1 98
0 0 51 707120 75408  0  36  0 0 0 0 0 0 0 0 0 290 1218 268 1 3 96
0 0 51 707120 75408  0  0 0 0 0 0 0 0 0 0 0 0 0 279 1285 269 1 4 95
0 0 51 707120 75408  0  0 0 0 0 0 0 0 0 0 0 0 0 285 1188 263 0 3 97
0 0 51 707120 75408  0  0 0 0 0 0 0 0 0 0 0 0 0 276 1154 234 1 3 96
```

- vmstat provides a general system-wide view - a good starting point. Chapter 1 lists others.
- **What may be interesting from this output?**
drill down analysis

- vmstat showed many syscalls
- DTrace can identify if a single process (name) is responsible:

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

```
dtrace: description 'syscall:::entry ' matched 231 probes
^C
```

```
svc.configd 1
inetd 1
nscd 13
httpd 15
svc.startd 46
sshd 56
java 155
dtrace 167
acroread 659
top 4615
```
drill down analysis

- top is calling most of the system calls
- Now the syscall type is identified:

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

dtrace: description 'syscall:::entry ' matched 231 probes
^C

pollsys 7
write 7
gtime 7
sysconfig 7
getuid 7
uadmin 7
llseek 14
getdents64 14
ioctl 28
open 2121
read 2121
close 2121
```
drill down analysis

- open(), read(), close() were all called 2121 times – perhaps by the same function.
- DTrace can now aggregate the user stack for these functions:

```bash
# dtrace -n 'syscall::open:entry /execname == "top"/ { 
    @num[ustack()] = count(); }'
dtrace: description 'syscall::open:entry ' matched 1 probe
^C

libc.so.1`__open+0x15
libc.so.1`open+0x77
top`getptable+0xe5
top`get_process_info+0x14
top`main+0x695
top`_start+0x80
2424
```
The same stack trace is seen for all syscalls

Great – we have identified the function responsible for calling so many syscalls

What can be done next?
Further drill down analysis

- Other analysis that can then be performed:
  - Examine arguments to syscalls to understand the nature of the activity.
  - Measure elapsed and on-cpu timestamps to prove that this large number of syscalls really is an issue (it may not be!)
  - Read the source code to the application (if available) to fully understand the issue and suggest a fix.
  - Examine activity from other providers other than the syscall layer.

- A lot more can be done. We have only just begun!
Strategy #3: frequency count

• When in doubt, frequency count.

```bash
# dtrace -n 'sysinfo::: /execname == "top"/ { @num[probename] = count(); }'
dtrace: description 'sysinfo::: ' matched 59 probes
^C

trap 2
writech 9
syswrite 9
outch 9
inv_swtch 48
pswitch 68
readch 2727
sysread 2727
namei 2727
```

• Be aware that this is counting occurrences that a probe was fired. ie, the readch statistic probe was called 2727 times. Don't infer too much...
Frequency Count

```bash
# dtrace -n 'sysinfo::: /execname == "top"/ { @num[probenam... sum(arg0); }'
dtrace: description 'sysinfo::: ' matched 59 probes
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>syswrite</td>
<td>8</td>
</tr>
<tr>
<td>inv_switche</td>
<td>38</td>
</tr>
<tr>
<td>pswitch</td>
<td>56</td>
</tr>
<tr>
<td>sysread</td>
<td>2424</td>
</tr>
<tr>
<td>namei</td>
<td>2424</td>
</tr>
<tr>
<td>writech</td>
<td>3135</td>
</tr>
<tr>
<td>outch</td>
<td>3158</td>
</tr>
<tr>
<td>readch</td>
<td>814464</td>
</tr>
</tbody>
</table>

- Now we can see the value of the statistic. readch is at 814464 bytes.
Frequency Count

# dtrace -ln mib::: | wc
  486    2430   41475

• Why study 485 probes if most are *never* called? If it's never called, it's never called!

  # dtrace -n 'mib::: { @num[probename] = count(); }'
dtrace: description 'mib:::' matched 485 probes
^C

  udpInDatagrams                       1
  ipInDelivers                         1
  tcpInAckBytes                        74
  tcpRttUpdate                         74
  tcpInDataInorderSegs                 74
  tcpInAckSegs                         74
  tcpInDataInorderBytes                74
  ipOutRequests                        81
  tcpOutDataBytes                      81
  tcpOutDataSegs                       81
  ipInReceives                         142
Strategy #4: known count

- Cause a fault or workload a known number of times. Probes of interest will occur that known number of times (or a multiple).

- Here a non-root user modifies /etc/motd and tries to save 17 times:

```bash
# dtrace -n 'syscall:::entry /execname == "vi"/ { @num[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 231 probes
^C
g getpid
  17
stat64
  17
ioctl
  34
read
  51
write
  102
```
I like to create 17 events, as 17 is not a commonly occurring number, and is more likely to be related to your test. A similar moderately sized prime would be 23.

The number of eyelashes on a yellow pig is supposedly 17.

Hence, the “Yellow Pig number” is 17.

“Yellow Pig Day” is July 17th, celebrated by mathematicians.

(I'm not making this up - try google!)
Strategy #5: aggregate stacks

- Stack Traces can be aggregated in at least two ways:

  A) Event probes
  - when an event of interest occurs, the stack is aggregated
  - this can explain why the event is taking place – the functions that lead up to the event

  B) Profile sampling
  - the stack trace is aggregated at 100hz (at least)
  - this is a crude but often effective way to show where the application is spending most of its time
Thinking in DTrace

- DTrace itself is an easy language, with the syntax well defined in the DTrace Guide
- Being successful with DTrace is about the application – being able to think in DTrace
- Tips for thinking in DTrace
  - practise, practise, practise
  - know the existing tools inside out (chapter 1)
  - know the OS well (Solaris Internals 2nd edition)
  - study source code, and how applications are constructed and operate
  - think outside the box!
End of Chapter 5

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTraceToolkit
Chapter 10 – DTrace Mentality 2
Chapter 6

Providers

- Provider Guide
- Using fbt
- Using pid
<table>
<thead>
<tr>
<th>Provider</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syscall</td>
<td>System call entry and return probes</td>
</tr>
<tr>
<td>sysinfo</td>
<td>Classic sysinfo statistic probes</td>
</tr>
<tr>
<td>vminfo</td>
<td>Virtual Memory statistic probes</td>
</tr>
<tr>
<td>io</td>
<td>Disk and NFS events</td>
</tr>
<tr>
<td>proc</td>
<td>Process events such as creation</td>
</tr>
<tr>
<td>sched</td>
<td>System scheduling events</td>
</tr>
<tr>
<td>lockstat</td>
<td>Kernel synchronisation lock events</td>
</tr>
<tr>
<td>plockstat</td>
<td>User synchronisation lock events</td>
</tr>
<tr>
<td>mib</td>
<td>MIB statistic updates</td>
</tr>
<tr>
<td>fasttrap</td>
<td>User location tracing</td>
</tr>
<tr>
<td>fpuinfo</td>
<td>SPARC FPU simulation events</td>
</tr>
</tbody>
</table>
Providers

- **sdt** Statically defined tracing
- **profile** Fixed sampling
- **dtrace** Program BEGIN/END probes
- **fbt** Raw kernel tracing
- **pid** User-level tracing

- Check the DTrace Guide for additional providers
syscall

- Example one-liner,

```
# Syscall count by syscall,
 dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'
```

- Probes the interface between user-land and the kernel
- Provides many insights into application behaivour
- syscalls are documented in section 2 of the man pages
sysinfo

- Example one-liner,

```bash
# Read bytes by process name,
dtrace -n 'sysinfo:::readch { @bytes[execname] = sum(arg0); }'
```

- Probes the sysinfo statistics, which are provided by kstat to tools such as mpstat
- Provides many insights into application and system behaviour
- Documented in the DTrace Guide (chap 23)
- Also see /usr/include/sys/sysinfo.h, cpu_sysinfo.h
vminfo

- Example one-liner,

```
# Pages paged in by process name,
dtrace -n 'vminfo:::pgpgin { @pg[execname] = sum(arg0); }'
```

- Probes the vminfo statistics, which are provided by kstat to tools such as vmstat
- Provides many insights into virtual memory behaviour
- Documented in the DTrace Guide (chap 24)
- Also see /usr/include/sys/sysinfo.h, cpu_vminfo.h
Example one-liner,

```bash
# Disk size aggregation
dtrace -n 'io:::start { @sz[execname] = quantize(args[0]->b_bcount); }'
```

- Probes I/O events, for both disk and NFS
- Provides a probe for the start and finish of each I/O transaction
- Provides crucial data for understanding how applications are driving the disks
- Documented in the DTrace Guide (chap 27)
Example one-liner,

```
# New processes with arguments,
dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'
```

- Probes process events such as exec, exit, thread creation and exit, and signals
- Provides high level probes so that process analysis can be performed easily
- Documented in the DTrace Guide (chap 25)
sched

- Example one-liner,

```bash
# Times the scheduler begins to run a thread,
dtrace -n 'sched:::on-cpu { @on[execname] = count(); }'
```

- Probes thread scheduling events
- Provides high level probes so that scheduler analysis can be performed easily
- Documented in the DTrace Guide (chap 26)
- Also see Solaris Internals 2nd edition
Example one-liner,

```bash
# Kernel function calls by module
dtrace -n 'fbt:::entry { @calls[probemod] = count(); }'
```

Function Boundary Tracing – traces raw kernel function events

These probes are “unstable” - their names and arguments can and do change between minor releases of Solaris (which is why we have the higher level providers)

fbt is usually the bulk of the probes: 30,000+
Since fbt can trace the entire kernel, observing just about any kernel behaviour is possible.

Enabling all fbt probes (30,000+) will slow the kernel noticeably. Be selective.

Documentation for the fbt probes (which is really documentation for the kernel) is in:

- The DTrace Guide (chap 20)
- Solaris Internals 2nd edition
- cvs.opensolaris.org
• Example one-liner,

```bash
# Trace user-level function entries,
dtrace -n 'pid$target:a.out::entry' -c 'some_command'
```

• The pid provider runs a command (-c cmd), or traces a particular process (-p PID).

• The pid provider can:
  • trace user functions
  • trace user instructions
  • trace library functions
  • trace library instructions

• The pid provider can create millions of probes
• The provider name is,
  • pid\text{PID} \quad \text{Where PID is the target to trace}
  • pid81 \quad \text{Trace PID 81}
  • pid\$target \quad \text{Trace the target of -c or -p}

• The full syntax is,
  \text{pidPID:segment:function:name}

  • segment is the name of the mapping. eg,
    • libc.so.1 (or “libc”) \quad \text{a library}
    • a.out \quad \text{the binary}
  • function is the function name. eg, “main”
  • name is either: entry, return, or an instruction address
pid demonstrations

- user functions,

```bash
# dtrace -n 'pid$target:a.out::entry' -c 'ping mars'
dtrace: description 'pid$target:a.out::entry' matched 16 probes
mars is alive
dtrace: pid 29890 has exited
CPU   ID            FUNCTION:NAME
  0  39187           __fsr:entry
  0  39188           main:entry
  0  39191       send_scheduled_probe:entry
  0  39198           set_IPv4_options:entry
  0  39190           schedule_sigalrm:entry
  0  39199                check_reply:entry
  0  39195           seq_match:entry
  0  39191       send_scheduled_probe:entry
```
pid demonstrations

- user instructions,

```bash
# dtrace -n 'pid$target:a.out:set_IPV4_options:' -c 'ping mars'
dtrace: description 'pid$target:a.out:set_IPV4_options:' matched 169 probes
mars is alive
dtrace: pid 29900 has exited

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>39188</td>
<td>set_IPV4_options:entry</td>
</tr>
<tr>
<td>0</td>
<td>39189</td>
<td>set_IPV4_options:0</td>
</tr>
<tr>
<td>0</td>
<td>39190</td>
<td>set_IPV4_options:1</td>
</tr>
<tr>
<td>0</td>
<td>39191</td>
<td>set_IPV4_options:3</td>
</tr>
<tr>
<td>0</td>
<td>39192</td>
<td>set_IPV4_options:6</td>
</tr>
<tr>
<td>0</td>
<td>39193</td>
<td>set_IPV4_options:7</td>
</tr>
<tr>
<td>0</td>
<td>39194</td>
<td>set_IPV4_options:8</td>
</tr>
<tr>
<td>0</td>
<td>39195</td>
<td>set_IPV4_options:9</td>
</tr>
<tr>
<td>0</td>
<td>39196</td>
<td>set_IPV4_options:10</td>
</tr>
<tr>
<td>0</td>
<td>39197</td>
<td>set_IPV4_options:12</td>
</tr>
<tr>
<td>0</td>
<td>39198</td>
<td>set_IPV4_options:19</td>
</tr>
<tr>
<td>0</td>
<td>39199</td>
<td>set_IPV4_options:1b</td>
</tr>
<tr>
<td>0</td>
<td>39200</td>
<td>set_IPV4_options:1f</td>
</tr>
<tr>
<td>0</td>
<td>39301</td>
<td>set_IPV4_options:15d</td>
</tr>
<tr>
<td>0</td>
<td>39302</td>
<td>set_IPV4_options:15e</td>
</tr>
<tr>
<td>0</td>
<td>39303</td>
<td>set_IPV4_options:15f</td>
</tr>
<tr>
<td>0</td>
<td>39304</td>
<td>set_IPV4_options:160</td>
</tr>
<tr>
<td>0</td>
<td>39305</td>
<td>set_IPV4_options:162</td>
</tr>
<tr>
<td>0</td>
<td>39306</td>
<td>set_IPV4_options:163</td>
</tr>
<tr>
<td>0</td>
<td>39187</td>
<td>set_IPV4_options:return</td>
</tr>
</tbody>
</table>
```
pid demonstrations

- library functions,

```bash
# dtrace -n 'pid$target:::entry' -c 'ping mars'
dtrace: description 'pid$target:::entry' matched 5318 probes
mars is alive
dtrace: pid 29905 has exited
CPU     ID                    FUNCTION:NAME
  0  39320                 call_array:entry
  0  39321                  call_init:entry
  0  39352                      leave:entry
  0  39351                 fmap_setup:entry
  0  39296              rt_bind_clear:entry
  0  39287             _rt_bind_clear:entry
  0  39289                   _rt_null:entry
  0  39296             _rt_bind_clear:entry
  0  39287            _rt_bind_clear:entry
  0  65195                  libc_init:entry
  0  63314                     atexit:entry
  0  64852                lmalloc:entry
...```
## pid demonstrations

- library instructions,

```bash
# dtrace -n 'pid$target:libc:getpid:' -c 'ping mars'
dtrace: description 'pid$target:libc:getpid:' matched 9 probes
mars is alive
dtrace: pid 29914 has exited
```

```
<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>39188</td>
<td>getpid:entry</td>
</tr>
<tr>
<td>0</td>
<td>39189</td>
<td>getpid:0</td>
</tr>
<tr>
<td>0</td>
<td>39190</td>
<td>getpid:5</td>
</tr>
<tr>
<td>0</td>
<td>39191</td>
<td>getpid:6</td>
</tr>
<tr>
<td>0</td>
<td>39192</td>
<td>getpid:b</td>
</tr>
<tr>
<td>0</td>
<td>39193</td>
<td>getpid:d</td>
</tr>
<tr>
<td>0</td>
<td>39194</td>
<td>getpid:13</td>
</tr>
<tr>
<td>0</td>
<td>39195</td>
<td>getpid:15</td>
</tr>
<tr>
<td>0</td>
<td>39187</td>
<td>getpid:return</td>
</tr>
<tr>
<td>0</td>
<td>39188</td>
<td>getpid:entry</td>
</tr>
<tr>
<td>0</td>
<td>39189</td>
<td>getpid:0</td>
</tr>
<tr>
<td>0</td>
<td>39190</td>
<td>getpid:5</td>
</tr>
<tr>
<td>0</td>
<td>39191</td>
<td>getpid:6</td>
</tr>
</tbody>
</table>

...
End of Chapter 6

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTraceToolkit
Chapter 10 – DTrace Mentality 2
Chapter 7

The D Language

- Comments
- Variable Types
- Macros
- Script Options
- Pragmas
The D language is related to the C programming language. This includes comment syntax:

/* this is a one line comment */

/*
 * This is a
 * block comment.
 */
Variable Types

- DTrace tries to figure out the variable type on first declaration
- Variable types can be predeclared
- Types include:
  - integer
  - string
  - pointer
  - associative array
  - aggregate
  - thread-local
  - clause-local
Integers

- Using integers:
  - `int mycount;` declaration
  - `mycount = 1;` declaration / setting
  - `mycount++;` increment by 1
  - `/mycount > 10/` testing in a predicate

- A declaration of the form “int mycount;” must be placed outside of any action block. They are usually placed at the top of the program.
Strings

• Using strings:
  • `string name;` declaration
  • `name = "Fred Nurke";` declaration / setting
  • `/name == "Fred"/` predicate testing
  • `name = 0;` memory cleanup

• Strings are null terminated

• `printf()` supports,
  • `%s` string format
  • `%S` safe string format (escapes odd chars)
Pointers

- Used in the same way as C programming
- DTrace can walk most structures without casting (e.g., vnode_t, proc_t, ...)
- Special builtins,
  - curthread: pointer to the on-cpu thread
  - curpsinfo: pointer to a psinfo like structure
- E.g.,
  - curpsinfo->pr_psargs: 80 char arg list
  - curthread->t_procp->p_lwpcnt: # of LWPs
Associative Arrays

- Also known as “hashed arrays”, or “key/value arrays”

- Using associative arrays:
  - `string name[int];` declaration
  - `name[3] = "Fred";` declaration / setting
  - `/name[3] == "Fred"/` predicate testing
  - `name[3] = 0;` memory cleanup
  - `name["sd0", 1] = 4;` two key array

- *Try to use aggregates instead*
Aggregates

- Discussed in Chapter 3
- Using Aggregates:
  - `@name[keys] = func();` setting
  - `printa("%s... %@d", @name);` printing
  - `trunc(@name);` clearing
  - `trunc(@name, size);` truncating to size
  - `clear(@name);` clearing values
  - `normalize(@name, n);` dividing by n
- Aggregates perform well, and produce great reports
Thread-local

- A variable stored private to each thread.
- Using thread-local:
  - `self->start = timestamp;` setting
  - `self int start;` declaration
  - `/self->start > 0/` predicates
  - `self->start = 0;` memory cleanup
- Thread-local variables are crucial, and are used heavily when scripting in DTrace.
Clause-local

• A temporary variable for calculations within an action clause

• Using clause-local:
  • `this->delta = 5;` setting
  • `this int delta;` declaration

• These are used to improve performance, rather than using global integers or strings that require lock overheads between CPUs

• Only use these within one action
Kernel Variables

- Any symbol from the kernel can be read, along with its struct members
- Example kernel variables:
  - `freemem` free mem in pages
  - `utsname.nodename` hostname
- Often only the source code can explain what these really are
- Module variables can also be read (prefix with the module name)
Macro Variables

- DTrace provides some useful macro variables for scripting:
  - $1, $2, ... arguments to script as ints
  - $$1, $$2, ... arguments to script as strings
  - $pid PID of dtrace(1M)
  - $uid real UID of dtrace(1M)
  - $target PID target of dtrace (-c or -p)

- For the argument variables to assume default values (0 or ""), use:
  - #pragma D option defaultargs
When writing complex scripts, it can be useful for them to process options. Eg:

```
# ./DTraceToolkit-0.96/dvmstat -h
USAGE: dvmstat [-h] { -p PID | -n name | command }
   -p PID            # examine this PID
   -n name           # examine this process name

eg,
    dvmstat -p 1871   # examine PID 1871
    dvmstat -n tar    # examine processes called "tar"
    dvmstat df -h    # run and examine "df -h"
```

Currently this is best achieved by embedding the DTrace script in a shell script, and using the shell's getopts function. Be careful to write neatly and follow programming best practices – such scripts get messy fast.
Pragmas

- These change the behaviour of DTrace, and often an alternative to command line options
- Useful pragmas:
  - `#pragma D option quiet`
  - `#pragma D option switchrate=10hz`
  - `#pragma D option bufsize=16m`
  - `#pragma D option flowindent`
  - `#pragma D option destructive`
- Others are listed in the DTrace Guide (chapter 16)
Destructive Actions

- These do more than just read data:
  - `stop()` freeze the current process
  - `raise(sig)` send this sig to the current PID
  - `chill(ns)` pause for nanoseconds
  - `system(cmd)` run this shell command
  - `copyout(buf, addr, bytes)` copy this data to user-land

- Use extreme caution when using destructive actions
End of Chapter 7

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTrace Toolkit
Chapter 10 – DTrace Mentality 2
Chapter 8

Advanced Scripting

- Step through an example script
- Discuss script internals
The following is from the DTrace Toolkit:

```bash
# ./shortlived.d
Tracing... Hit Ctrl-C to stop.
^C
short lived processes:  0.924 secs
total sample duration:  4.308 secs

Total time by process name,
  soffice          3 ms
  sopatchlevel.sh 10 ms
  grep            14 ms
  uname           24 ms
  javaldx         33 ms
  expr            36 ms
  soffice.bin     93 ms
  pagein         696 ms

Total time by PPID,
  29854           30 ms
  625             93 ms
  29846           786 ms
```
shortlived.d

- shortlived.d calculates CPU time consumed by short-lived processes (processes that began and ended while DTrace was tracing).
- A common performance issue is short-lived processes hogging the CPU, which can be difficult to identify using traditional tools such as prstat.
- It is useful to discuss the entire internals of this script, from comments to code.
#!/usr/sbin/dtrace -qs
/*
 * shortlived.d - determine time spent by short lived processes.
 *                Written in DTrace (Solaris 10 3/05).
 *
 * 27-May-2006, ver 0.94
 *
 * USAGE: shortlived.d       # wait, then hit Ctrl-C
 *
 * Applications that run many short lived processes can cause load
 * on the system that is difficult to identify - the processes
 * aren't sampled in time by programs such as prstat. This program
 * illustrates how much time was spent processing those extra
 * processes, and a table of process name by total times for each.
 *
 * SEE ALSO: execsnoop
 *
 * Notes:
 * - The measurements are minimum values, not all of the overheads
 *   caused by process generation and destruction are measured (DTrace
 *   can do so, but the script would become seriously complex).
 * - The summary values are accurate, the by program and by PPID values
 *   are usually slightly smaller due to rounding errors.
*/
COPYRIGHT: Copyright (c) 2005, 2006 Brendan Gregg.

CDDL HEADER START

The contents of this file are subject to the terms of the Common Development and Distribution License, Version 1.0 only (the "License"). You may not use this file except in compliance with the License.

You can obtain a copy of the license at Docs/cddl1.txt or http://www.opensolaris.org/os/licensing.
See the License for the specific language governing permissions and limitations under the License.

CDDL HEADER END

22-Apr-2005 Brendan Gregg Created this.
shortlived.d

/**
 * Start
 */
dtrace:::BEGIN
{
    /* save start time */
    start = timestamp;

    /* this is time spent on shortlived processes */
    procs = 0;

    /* print header */
    printf("Tracing... Hit Ctrl-C to stop.\n");
}

/* 
 * Measure parent fork time 
 */
syscall::*fork*::entry
{
    /* save start of fork */
    self->fork = vtimestamp;
}
syscall::*fork*::return
/*arg0 != 0 && self->fork*/
{
    /* record on-CPU time for the fork syscall */
    procs += vtimestamp - self->fork;
    self->fork = 0;
}

/* 
 * Measure child processes time 
 */
syscall::*:fork*:return
/arg0 == 0/**
{
    /* save start of child process */
    self->start = vtimestamp;

    /* memory cleanup */
    self->fork = 0;
}

proc:::exit
/self->start/**
{
    /* record on-CPU time for process execution */
    this->oncpu = vtimestamp - self->start;
    procs += this->oncpu;

    /* sum on-CPU by process name and ppid */
    @Times_exec[execname] = sum(this->oncpu/1000000);
    @Times_ppid[ppid] = sum(this->oncpu/1000000);

    /* memory cleanup */
    self->start = 0;
}
/*
 * Print report
 */
dtrace:::END
{
    this->total = timestamp - start;
    printf("short lived processes: %%6d.%%03d secs\n",
            procs/1000000000, (procs%1000000000)/1000000);
    printf("total sample duration: %%6d.%%03d secs\n",
            this->total/1000000000, (this->total%1000000000)/1000000);
    printf("\nTotal time by process name,\n");
    printa("%18s %@12d ms\n", @Times_exec);
    printf("\nTotal time by PPID,\n");
    printa("%18d %@12d ms\n", @Times_ppid);
}
End of Chapter 8

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting

Chapter 9 – The DTrace Toolkit
Chapter 10 – DTrace Mentality 2
The DTraceToolkit

- Introduce the DTraceToolkit
- Explain Layout
- A Recipe For Getting Started
DTraceToolkit

- http://www.opensolaris.org/os/community/dtrace/dtracetoolkit
- Version 0.96 contains over 100 scripts
- Freeware (CDDL)
- Serves to:
  - Provide tools to analyse to common problems
  - Provide numerous demonstrations of programming in D
  - Promote DTrace
- Written for Solaris 10 3/05 as much as possible
Main Parts

• The DTrace Toolkit is currently 3 main parts:
  • The Scripts
  • Man Pages
  • Examples

• There is a man page for every script, and an example file for every script.

• It took considerable time to write 100+ man pages – so please read them!

• The example files are often more useful than the man pages
Installation

• Isn't required – the scripts can be run immediately after extraction
• If desired, an install script has been provided. It prompts for install options, and defaults to installing under /opt/DTT

# ./install
DTraceToolkit Installation
--------------------
DTraceToolkit version 0.96, 24-Apr-2006

hit Ctrl-C any time you wish to quit.

Enter target directory for installation [/opt/DTT]:
The scripts are positioned in a hierarchy: the most useful are in the top directory, and others are in meaningful subdirectories. The “Bin” directory contains symlinks to all the scripts.
The Scripts

- If they end in `.d`, they are pure DTrace. See the man page or try reading their header.
- If they don't end in `.d`, they are wrapped in shell or Perl. See their man page and try running them with `-h`.
- If they begin with,
  - **Tracing**... Hit Ctrl-C to end.
    they are tracing events
  - **Sampling**... Hit Ctrl-C to end.
    they are sampling events using the profile provider
The Man Pages

- Man Pages can be read using,

```bash
# MANPATH=$MANPATH:/opt/DTT/Man      # or wherever
# man iosnoop
```

Reformatting page. Please Wait... done

USER COMMANDS

iosnoop(1m)

NAME

iosnoop - snoop I/O events as they occur. Uses DTrace.

SYNOPSIS

```bash
iosnoop [-a|-A|-Deghinostv] [-d device] [-f filename] [-m mount_point] [-n name] [-p PID]
```

DESCRIPTION

iosnoop prints I/O events as they happen, with useful details such as UID, PID, block number, size, filename, etc.

This is useful to determine the process responsible for using the disks, as well as details on what activity the process is requesting. Behaviour such as random or sequential I/O can be observed by reading the block numbers.
In this example, bitesize.d was run for several seconds then Ctrl-C was hit. As bitesize.d runs it records how processes on the system are accessing the disks - in particular the size of the I/O operation. It is usually desirable for processes to be requesting large I/O operations rather than taking many small "bites".

The final report highlights how processes performed. The find command mostly read 1K blocks while the tar command was reading large blocks - both as expected.

```bash
# bitesize.d
Tracing... Hit Ctrl-C to end.
^C
```

```
PID   CMD
7110  -bash\0

value          Distribution          count
512            |                         0
1024           |@@@@@@@@@@@@@@@@@@@@@@@@@@  2
```
Script Subdirectories

- So far there is,

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bin/</td>
<td>Symlinks to the scripts</td>
</tr>
<tr>
<td>Apps/</td>
<td>Application specific scripts</td>
</tr>
<tr>
<td>Cpu/</td>
<td>Scripts for CPU analysis</td>
</tr>
<tr>
<td>Disk/</td>
<td>Scripts for disk I/O analysis</td>
</tr>
<tr>
<td>Extra/</td>
<td>Misc scripts</td>
</tr>
<tr>
<td>Kernel/</td>
<td>Scripts for kernel analysis</td>
</tr>
<tr>
<td>Locks/</td>
<td>Scripts for lock analysis</td>
</tr>
<tr>
<td>Mem/</td>
<td>Scripts for memory analysis</td>
</tr>
<tr>
<td>Net/</td>
<td>Scripts for network analysis</td>
</tr>
<tr>
<td>Proc/</td>
<td>Scripts for process analysis</td>
</tr>
<tr>
<td>System/</td>
<td>Scripts for system analysis</td>
</tr>
<tr>
<td>User/</td>
<td>Scripts for user based activity analysis</td>
</tr>
<tr>
<td>Zones/</td>
<td>Scripts for analysis by zone</td>
</tr>
</tbody>
</table>

- Remember to check the top-level scripts first
DTrace Toolkit Recipe

- Running the following scripts in this order has proved a useful recipe:
  1) execsnoop
  2) iosnoop
  3) opensnoop
  4) errinfo
  5) procsystime
  6) rwtop
  7) iotop
  8) dvmstat
  9) Disk/bitesize.d
This exists to capture DTrace data in the most broad possible way
- It creates a tar file containing the output from numerous DTrace commands
- If you only had 5 minutes to “DTrace” a server, dexplorer may be the best tool to run – it creates many files for later offline analysis
- A html-iser for dexplorer has yet to be written (it won't be hard to do)
Behind the DTraceToolkit

- The DTraceToolkit is a product of much testing – far more time has gone into testing the scripts than actually writing them.
- The scripts need to be tested on:
  - SPARC and x86
  - Solaris 10 3/05, Solaris 10 1/06, ...
  - Different builds of OpenSolaris (makes a huge difference for any fbt based scripts)
  - All possible workloads
- *If a script's output is 99% correct, is that “correct”?”*
A script's output that is incorrect 1% of the time, is incorrect.

Imagine a “ps -ef” that has 1 bad line out of every 100.

Only in some cases is this acceptable – where the data is clearly marked as an estimation.

For example, prstat's “SIZE” and “RSS” are not 100% accurate, but they are still useful estimates as we are aware that they are estimates.
End of Chapter 9

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTrace Toolkit
Chapter 10 – DTrace Mentality 2
Chapter 10

DTrace Mentality 2

- More DTrace Strategies
Strategy #6: Elapsed Time

- Elapsed time is the calculated using,
  
  ```
  start = timestamp;
  ...
  elapsed = timestamp - start;
  ```

- Elapsed time explains why an application is experiencing slow response times

- Elapsed time includes disk I/O times, network I/O times, scheduling latency, ...

- Elapsed times can be measured at the application, treating the OS as a black box
Measuring Times

• Some difficulties when measuring these:
  • Knowing what is the start probe and what is the end probe.
    • If you are lucky, the provider will already have it as :entry and :return, or :start and :done.
    • If you are unlucky, you will need to find the probes from what is available. Try a previous strategy – “known count”.
  • Associating the start probe to the end probe
    • If you are lucky, they will be in the same thread and you can use a thread-local variable. Eg, self->start
    • If you are unlucky, you'll need associative arrays and some other ID as the key.
Strategy #7: On-CPU Time

- On-CPU time is calculated using:
  
  ```
  start = vtimestamp;
  ...
  oncpu = vtimestamp - start;
  ```

- On-CPU time explains why the CPUs are busy.

- On-CPU time excludes disk I/O times, network I/O times, scheduling latency, ...
In the following example, syscall elapsed and overhead times are measured. Elapsed times represent the time from syscall start to finish; overhead times measure the time spent on the CPU,

```bash
# dtruss -eon bash
```

<table>
<thead>
<tr>
<th>PID/LWP</th>
<th>ELAPSD</th>
<th>CPU</th>
<th>SYSCALL(args)</th>
<th>SYSCALL(args) = return</th>
</tr>
</thead>
<tbody>
<tr>
<td>3911/1:</td>
<td>41</td>
<td>26</td>
<td>write(0x2, &quot;l\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>1001579</td>
<td>43</td>
<td>read(0x0, &quot;s\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>38</td>
<td>26</td>
<td>write(0x2, &quot;s\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>1019129</td>
<td>43</td>
<td>read(0x0, &quot; \001\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>38</td>
<td>26</td>
<td>write(0x2, &quot; \0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>998533</td>
<td>43</td>
<td>read(0x0, &quot;-\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>38</td>
<td>26</td>
<td>write(0x2, &quot;-\001\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>1094323</td>
<td>42</td>
<td>read(0x0, &quot;l\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>39</td>
<td>27</td>
<td>write(0x2, &quot;l\001\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>1210496</td>
<td>44</td>
<td>read(0x0, &quot;r\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>40</td>
<td>28</td>
<td>write(0x2, &quot;n\001\0&quot;, 0x1)</td>
<td>= 1 0</td>
</tr>
<tr>
<td>3911/1:</td>
<td>9</td>
<td>1 lwp_sigmask(0x3, 0x2, 0x0)</td>
<td>= 0xFFBFFFEFF 0</td>
<td></td>
</tr>
<tr>
<td>3911/1:</td>
<td>70</td>
<td>63</td>
<td>ioctl(0x0, 0x540F, 0x80F6D00)</td>
<td>= 0 0</td>
</tr>
</tbody>
</table>
Strategy #8: Milestones

- Try to understand the behaviour of the target by identifying application-level activity milestones. For example:
  - establishing a connection
  - beginning a transaction
  - completing a transaction
  - writing to a log

- These may be identified using:
  - the probe name – ie, the function name
  - usstack() when solaris statistics are triggered
  - engaging in “known count” strategy
  - dereferencing random strings
End of Chapter 10

Chapter 1 – Solaris 9 Performance Tools
Chapter 2 – Introducing DTrace
Chapter 3 – Command Line DTrace
Chapter 4 – DTrace one-liners
Chapter 5 – DTrace Mentality 1
Chapter 6 – Providers
Chapter 7 – The D Language
Chapter 8 – Advanced Scripting
Chapter 9 – The DTrace Toolkit
Chapter 10 – DTrace Mentality 2
Workshops

DTrace Workshops

- Now it's time for you to get your hands dirty solving a variety of problems
- I'll create problems on your systems at random
- When you solve them, don't shout out the answer!
- Many people find this is the best way to learn – you need to think on your own
- Good Luck!