

DTrace Programming Workshop

*DTrace on Solaris 10,
OpenSolaris,
and other Operating Systems*

About this document

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*). The document was originally composed by Brendan Gregg, author of the DTrace Toolkit, with updates and amendments by Context-Switch Limited.

Various content, including diagrams and material from the DTrace Toolkit, are copyright © 2006-2008 by Brendan Gregg and used under exclusive license by Context-Switch Limited. The remaining portions of this document are copyright © 2006-2008 Context-Switch Limited.

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

The Workshop Agenda

- DAY 1
 - General Solaris Performance Monitoring
 - Introducing DTrace
- DAY 2
 - Programming in D
 - The DTrace Mentality
- DAY 3
 - Applying DTrace (*A day of workshop*)

Workshop Modules

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

Some topics presented are based on the book: "*Solaris Performance and Tools: DTrace and mdb Techniques for Solaris 10 and OpenSolaris*" written by Richard McDougall, Jim Mauro & Brendan Gregg

About the slides and presentation

- These slides cover key topics
- We will cover many side topics, elaborate further, answer questions, run demos, and discuss many performance-related topics
- As such, if we deviate from the slides –
don't panic!
It's all part of the workshop delivery.

MODULE 1:

General Solaris Performance Monitoring

- *What can be observed? And, how quickly can it be observed?*
 - *What can't be observed? And, why?*
 - *What is the impact of observation?*
- DTrace wins thanks to its ability to observe at such a fine level of detail with minimal system overhead - (*but, may not always be the required utility! - This is why we need to understand general utilities*)

An example fault...

- "Where has my CPU gone?" using standard Solaris tools

... An interactive demo using sar and vmstat ...

The Solaris Toolkit

- Consists of many, many tools

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

- There is no single tool to rule them all
(*not even DTrace!*)

Brendan Gregg tried...

```
$ sysperfstat 1
```

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

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

Coping with many tools:

We will...

- Categorise by Approach
- Categorise by Resource Type
- Create at a checklist of tools (*including the detail in these slides*)
- Study some, be aware what else exists (*just remembering that something is doable is valuable!*)

Categorize 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, centralised syslog, ...
- 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` *always reads every statistic it can report on!*
Now, compare this to how `mpstat` works)

Solaris 10 now uses SNMP:

- User-based Security Model (USM)
 - usernames, passwords, encryption
- View-based Access Control MIB (VACM)
 - restricting views
- These provide greatly improved security -
a trait of the Solaris 10 OS

Categorize Tools by Resource Type:

- CPU - `vmstat`, `mpstat`, `sar`, `prstat`, `ps`
- Memory - `vmstat`, `swap`, `prstat`,
::`memstat` (*mdb*), `ps`, `pmap`
- Disk - `iostat`, `taztool`, `iosnoop`, `iostat`
- Network - `netstat`, `kstat`, `ndd`, `nicstat`

Tools Checklist:

- 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*

Monitoring Tools and Utilities

- uptime
- /usr/bin/ps
- /usr/ucb/ps
- prstat
- prstat -m
- vmstat
- mpstat
- iostat
- prex/tnf
- psio *
- netstat -i
- nicstat *
- netstat -rn
- sar
- sar -u
- sar -q
- kstat
- K9Toolkit *
- ndd
- checkcable *
- cpustat
- CacheKit *
- busstat
- trapstat
- lockstat
- truss
- sotruss
- apptrace
- truss -ua.out
- adb
- mdb -k
- mdb -p
- lastcomm
- BSM auditing
- snmpwalk

**Those listed in blue are written by Brendan Gregg*

uptime

```
$ uptime
```

```
9:37am up 371 day(s), 16:07, 6 users, load average: 0.02, 0.02, 0.02
```

- 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 thread's usr + sys + CPU latency times.*
 - *load averages tend to over-simplify CPU issues!*
- Can't customise the interval
- DTrace can trace scheduler activity
 - BTW – 371 days is nothing; Brendan maintains the “Sun Book of Records” (google it), where the current record is more than 2001 days!*

/usr/bin/ps

```
$ ps -ef
```

```
UID    PID  PPID  C   STIME TTY  TIME  CMD
root    0     0    0   May 28 ?  0:08  sched
root    1     0    0   May 28 ?  0:48  /etc/init
root    2     0    0   May 28 ?  0:02  pageout
root    3     0    0   May 28 ? 89:39  fsflush
root 2316     1    0   May 28 ?  0:00  /usr/lib/saf/sac -t 300
nobody 22112 22111 0 Feb 15 ? 231:44 (squid)
root 2061     1    0   May 28 ?  0:12  /usr/sbin/rpcbind
root   73     1    0   May 28 ?  0:00  /usr/lib/sysevent/syseventd
root   78     1    0   May 28 ?  0:01  /usr/lib/picl/picld
...
```

- Useful to check long-term processes
- Default fields are not hugely useful, use `-o`
- DTrace can access similar *procfs* statistics, and gather many additional statistics

/usr/ucb/ps

```
$ /usr/ucb/ps auxww
```

```
USER      PID %CPU %MEM    SZ  RSS  TT      S   START    TIME COMMAND
root      3768  3.0  0.8   3216 2856 pts/2  O   16:24:40  0:00 /usr/ucb/ps -
      auxww
root      3453  0.2  0.8   4544 2904   ?    S   15:59:10  0:01
      /usr/lib/ssh/sshd
root      498   0.1  0.8   4680 2944   ?    S    May 03  0:29
      /usr/lib/ssh/sshd
brendan  17545  0.1  0.6   2648 2080 pts/2  S    May 08  0:04 /bin/bash
brendan   3769  0.1  0.3    984  808 pts/2  S   16:24:40  0:00 head
brendan   2408  0.1  1.4   6800 5112   ?    S    May 28  92:28 SCREEN
root      2037  0.1  0.3   2024  800   ?    S    May 28 168:21
      /usr/sbin/in.routed
root      3765  0.1  0.3   1016  816   ?    S   16:24:28  0:00 sleep 15
nobody  22112  0.1 22.7  91504 84352 ?    S    Feb 15 231:44 (squid)
```

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

prstat

```
$ 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
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
25442brendan  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
17545brendan  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

prstat -m

```
$ prstat -m
```

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

```
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 – of your own choice!

vmstat

```
$ vmstat 1 3
```

```
 kthr      memory          page          disk          faults          cpu
 r b w  swap  free  re  mf pi po fr de sr dd sl -- --  in  sy  cs us sy id
 0 0 0 1610024 679984 39 173 652 1 1 0 47 111 0 0 0 517 3930 822 13 5 83
 1 0 0 1537160 580800 25 445 3099 0 0 0 0 424 0 0 0 1097 10161 1023 73 11 16
 1 0 0 1536736 580352 24 348 2931 0 0 0 0 383 0 0 0 1042 10531 982 17 11 72
```

```
$ 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
1614624 686176 40 180 1 0 49 44 0 0 0 0 0 354 1 1
1525904 564008 16 72 0 0 0 103 0 0 0 0 0 11374 0 0
1523736 561936 35 412 0 0 0 78 0 0 0 0 0 10649 0 0
```

- Best system-wide view
- Can't view statistics by-process, by-zone, ...
- DTrace can take these stats and drill further – *answering who causes them and why*

mpstat

```
$ mpstat 5
```

```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0  149  65   0   491  389  713  153   0   1   1  3262  11  4   0  85
  0  363 161   0   685  583  733  285   0   1   0 21210  85 15   0  0
  0  284 263   0  1198 1097 1177  507   0   1   0 21892  78 22   0  0
  0  334 1198   0  2781 2679 2836 1342   0   0   0 15970  78 22   0  0
  0  323 1072   0  2495 2393 2570 1207   0   0   0 15327  79 21   0  0
  0  296 1087   0  2509 2407 2616 1228   0   1   0 14000  79 20   0  0
  0  261 897   0  2146 2044 2240 1047   0   0   0 10994  83 17   0  0
```

- Good by-CPU summary
- DTrace can also measure by-CPU
- DTrace finally explains where those *xcal/s* come from

iostat

```
$ iostat -xnmPz 1
```

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

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

prex/tnextract/tnfdump

- `prex`: *enables static probes to record event details to a (kernel-based) buffer*
- `tnextract`: *extracts buffer contents to a file*
- `tnfdump`: *converts buffer contents (in file) to text output (stdout)*
- DTrace has this exact functionality, and more (*dynamic!*)

prex/tnfextract/tnfdump

```
# tnfdump tnffile.tnf
```

```
probe    tnf_name: "syscall_end" tnf_string: "keys syscall thread;file
    ../../sparc/os/syscall.c;line 797;"
probe    tnf_name: "syscall_start" tnf_string: "keys syscall thread;file
    ../../sparc/os/syscall.c;line 494;"
probe    tnf_name: "thread_block" tnf_string: "keys synch;file
    ../../common/tnf/tnf_res.c;line 287;"
...
```

```
-----
```

Elapsed (ms)	Delta (ms)	PID	LWPID	TID	CPU	Probe Name

Data / Description . . .						

0.000000	0.000000	1028	1	0x30002532020	0	syscall_end
rval1: 0 rval2: 3298578425768	errno: 0					
0.077490	0.077490	1028	1	0x30002532020	0	syscall_start
sysnum: 4						
0.152817	0.075327	1028	1	0x30002532020	0	syscall_end
rval1: 6 rval2: 8195	errno: 0					
0.166153	0.013336	1028	1	0x30002532020	0	syscall_start
sysnum: 3						
0.218053	0.051900	1028	1	0x30002532020	0	thread_block
reason: 0x300046ee362	stack: <tnf_symbols>					
0.232830	0.014777	1028	1	0x30002532020	0	thread_state
state: 7						
0.240759	0.007929	1028	1	0x30002532020	0	thread_state
tid: 0x2a10001fcc0	state: 1					

psio

```
# ./psio 1
```

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

- <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
```

input		bge0	output			input (Total)		output		
packets	errs	packets	errs	colls	packets	errs	packets	errs	colls	
42158	0	44249	0	0	84626	0	88808	0	0	
21	0	18	0	0	42	0	36	0	0	
5	0	4	0	0	10	0	8	0	0	
4	0	4	0	0	8	0	8	0	0	
4	0	3	0	0	8	0	6	0	0	

- An approximation of activity
- No further details for analysis
- Packets are not bytes!
- Both `kstat` and `DTrace` provide more info

nicstat

```
$ nicstat 1
```

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

- <http://www.brendangregg.com/k9toolkit.html>
- Uses `kstat`, written in both C and Perl
- Provides bytes, %Utilisation

netstat -rn

```
$ netstat -rn
```

```
Routing Table: IPv4
```

Destination	Gateway	Flags	Ref	Use	Interface
220.244.170.56	220.244.170.58	U	1	18146	hme0:1
192.168.0.0	192.168.0.2	U	1	12537	hme0:2
192.168.1.0	192.168.1.1	U	1	32151	hme0
224.0.0.0	192.168.1.1	U	1	0	hme0
default	220.244.170.57	UG	1	3	
127.0.0.1	127.0.0.1	UH	4	81	lo0

- Check for unexpected routes
- You discover problems after the fact
- DTrace can snoop changes live (*OK, so can route monitor. Who has used route monitor?*)

sar

```
$ sar 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	9

- 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
00:00:01      %usr      %sys      %wio      %idle
01:00:01          1          1          0          98
02:00:01          2          1          0          98
03:00:01          1          1          0          98
04:00:00          1          1          0          98
05:00:01          0          0          0          99
...
```

- This shows historic data from `sar -u`
- `%wio` is always zero in Solaris 10
(it was never properly understood anyway)
- Note that the system is idle

sar -c

```
$ sar -c
SunOS mars 5.9 Generic_118558-05 sun4u 05/11/2006
17:39:05 scall/s sread/s swrit/s  fork/s  exec/s rchar/s wchar/s
17:39:10      3805      1262        43   12.57    6.39  359217   22116
17:39:15      2882        937        31    6.40    3.20  286867   19712
17:39:20      1550        579        68    3.40    2.40  545063  493898
17:39:25      3039      1580        41    2.60    1.60 1312734 162488
...
```

- This shows that processes are being created (*forks & execs*)
- *Does not help identify which processes are being started!*
 - *Is this where DTrace can be used?*

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 5 1.0 100
02:00:01 1.0 1 5 1.0 100
03:00:01 1.2 0 5 1.0 100
04:00:00 1.4 0 5 1.0 100
```

- `sar -q` shows run queue sizes
- This is the same period as before, and shows a > 1.0 run queue size. **Huh?**

kstat

```
# kstat -pm cpu_info
cpu_info:0:cpu_info0:brand      UltraSPARC-IIIi
cpu_info:0:cpu_info0:chip_id    0
cpu_info:0:cpu_info0:class      misc
cpu_info:0:cpu_info0:clock_MHz  1062
cpu_info:0:cpu_info0:core_id    0
cpu_info:0:cpu_info0:cpu_fru     hc:///component=MB
cpu_info:0:cpu_info0:cpu_type    sparcv9
cpu_info:0:cpu_info0:crttime     40.28150371
cpu_info:0:cpu_info0:device_ID   208525810884
cpu_info:0:cpu_info0:fpu_type    sparcv9
```

...

- Great resource
- `libkstat`, `Sun::Solaris::Kstat`, or `/usr/bin/kstat`
- `netstat -k` is deprecated from Solaris 10 onwards
- 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`
 - *and more...*

ndd

```
# ndd /dev/hme link_speed  
1  
# ndd /dev/rtls link_speed  
operation failed: Invalid argument
```

- Accesses read/write settings for network related drivers
- There are consistency issues between interface types

checkcable

```
# checkcable
```

Interface	Link	Duplex	Speed	AutoNEG
hme0	UP	FULL	100	ON
hme1	UP	HALF	10	ON

<http://www.brendangregg.com>

- Translates both `nfd` 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

CacheKit

```
# ccachestat 5 5
```

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

<http://www.brendangregg.com/cachekit.html>

- Digs out useful info, calculates hit ratios
- Needs updates to measure newer CPUs

busstat

```
# busstat -w ac,pic0=clock_cycles,pic1=mem_bank0_rds 2 100
time  dev  event0          pic0  event1          pic1
2     ac0  clock_cycles  167242902  mem_bank0_rds  3144
2     ac1  clock_cycles  167254476  mem_bank0_rds  1392
4     ac0  clock_cycles  168025190  mem_bank0_rds  40302
...
```

- Measures bus PICs
- No public PIC documentation (*yet*)
 - *Has anyone actually used this other than Sun-trained engineers?*

trapstat

```
# trapstat -t 10
```

```
cpu m| itlb-miss %tim itsb-miss %tim | dtlb-miss %tim dtsb-miss %tim |%tim
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
  0 u|          669  0.0           2  0.0 |           158  0.0           2  0.0 |  0.0
  0 k|           46  0.0           0  0.0 |          24594  1.2           13  0.0 |  1.2
=====+=====+=====+=====+=====+=====+=====+=====+=====+=====
ttl |           715  0.0           2  0.0 |          24752  1.2           15  0.0 |  1.3
```

```
# trapstat
```

```
trapstat: not implemented on i86pc
```

- MMU statistics
- Check this to see if MPSS needs tuning

lockstat

```
# lockstat -k sleep 5
```

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

Count	indv	cuml	rcnt	nsec	Lock	Caller
2	33%	33%	0.00	71995	0x30000e723c0	clock
1	17%	50%	0.00	85133	0x300055ffcb0	rctl_test_entity
1	17%	67%	0.00	77176	0x300040b2888	rctl_test_entity
1	17%	83%	0.00	70159	0x30000e72280	clock
1	17%	100%	0.00	73192	0x30000e722c0	clock

- Kernel lock statistics – quite useful
- What entirely different mode of operation does lockstat provide?

lockstat -I

```
# lockstat -kIi 997 sleep 5
```

```
Profiling interrupt: 5066 events in 5.062 seconds (1001 events/sec)
```

Count	indv	cuml	rcnt	nsec	CPU+PIL	Caller
4699	93%	93%	0.00	401	cpu[0]	idle
189	4%	96%	0.00	394	cpu[0]	generic_idle_cpu
38	1%	97%	0.00	1708	cpu[0]	ufs_scan_inodes
38	1%	98%	0.00	1938	cpu[0]	mutex_enter
20	0%	98%	0.00	2014	cpu[0]	caslong
10	0%	99%	0.00	1339	cpu[0]	(usermode)
5	0%	99%	0.00	3219	cpu[0]+10	set_anoninfo
...						
1	0%	100%	0.00	1246	cpu[0]	trap
1	0%	100%	0.00	1005	cpu[0]	pagefault

- -I samples the kernel
- DTrace can both sample *and* trace

truss

- Trace *syscalls* and *signals*. Uses the `procfs ctl` files.
- `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 - so slows the target < 1%

sotruss

- Shared library tracing
- Was added in Solaris 2.6

- It uses the dynamic linker to trace details
- Not so clever with function arguments

apptrace

- Shared library tracing
- Was added in Solaris 8

- Evaluates arguments
- DTrace has this functionality

truss -ua.out

```
# truss -ua.out banner hello
execve("/usr/bin/banner", 0xFFBFFCCC, 0xFFBFFCD8)  argc = 2
resolvepath("/usr/lib/ld.so.1", "/lib/ld.so.1", 1023) = 12
...
/1: getpid()                                     = 2202 [2201]
/1: setustack(0xFF3A2088)
/1@1:      -> _init(0x22088, 0xff36cbc0, 0xa7a48, 0xff2c08d0)
/1@1:      <- _init() = 0x22088
/1@1:      -> main(0x2, 0xffbfffccc, 0xffbfffcd8, 0x22000)
...
/1: memcntl(0xFF260000, 6576, MC_ADVISE, MADV_WILLNEED, 0, 0) = 0
/1: close(3)                                       = 0
/1: munmap(0xFF380000, 8192)                       = 0
/1@1:      -> banner(0xffbffd7, 0x2251c, 0x5c810, 0x0)
/1@1:      -> banset(0x20, 0x2251c, 0x0, 0x0)
/1@1:      <- banset() = 0
/1@1:      -> convert(0x68, 0x2276f, 0x2276e, 0x20)
/1@1:      <- convert() = 104
/1@1:      -> banfil(0x2245c, 0x2251c, 0xffffffff76, 0xffffffff6f)
...
```

- Traces user functions
- `-u` can also trace library calls

adb/mdb

- Perform core dump analysis – *processes*
- Perform crash dump analysis – *kernel*

- mdb was added in Solaris 8 and enhanced in Solaris 9

mdb -k

```
# mdb -k
```

```
Loading modules: [ unix krtld genunix specfs dtrace ufs pcisch  
ip sctp usba s1394 fcp fctl nca lofs zfs random crypto nfs  
audiosup logindmux sd ptm md cpc sPPP wrsmd fcip ]
```

```
> ::mappings
```

BASE	LIMIT	SIZE	NAME
1000000	18fa000	8fa000	ktextseg
18fa000	1e00000	506000	valloc
70000000	80000000	10000000	vseg32
edd00000	f0000000	2300000	debugseg
2a10000000	2a17e258000	7e258000	pseg
2a750000000	2a757680000	7680000	mapseg
30000000000	70000000000	40000000000	vseg
70000000000	70001246000	1246000	mem64
800000000000000000	800010000000000000	100000000000	pmseg

```
> ::quit
```

- Visit kernel-land

mdb -k

```
> ::pgrep syslog | ::print proc_t
{
  p_exec = 0x300059d7e40
  p_as = 0x30003c05a38
  p_lockp = 0x30001178c40
  p_crlock = {
    _opaque          = [ 0 ]
  }
  p_cred = 0x3000037c168
  p_swapcnt =      0
  p_stat = '\002'
  p_wcode = '\0'
  p_pidflag =      0
  p_wdata = 0
  p_ppid = 0x1
  p_link = 0
  p_parent = 0x30000343360
  p_child = 0
  p_sibling =      0x300053bec18
  p_psibling = 0x300053c6fc8
  ...
}
```

- Walk kernel structures
- DTrace does this too

mdb -p

```
# mdb -p `pgrep -o syslogd`
Loading modules: [ ld.so.1 libc.so.1 libutil.so.1 ]
> ::mappings
    BASE    LIMIT    SIZE  NAME
    10000    22000    12000 /usr/sbin/syslogd
    32000    34000     2000 /usr/sbin/syslogd
    34000    82000    4e000 /usr/sbin/syslogd
fe77a000 fe77c000     2000 [ anon ]
fe87a000 fe87c000     2000 [ anon ]
...
ff370000 ff388000    18000 /lib/libscf.so.1
ff398000 ff39a000     2000 /lib/libscf.so.1
ff3a4000 ff3a6000     2000 [ anon ]
ff3b0000 ff3de000    2e000 /lib/ld.so.1
ff3ee000 ff3f0000     2000 /lib/ld.so.1
ff3f0000 ff3f2000     2000 /lib/ld.so.1
...
```

- Visit *userland*
- DTrace does this too (*copyin/copyout*)

Process Accounting

```
# lastcomm
man      root      term/a      0.02 secs Fri May 12 18:25
sh       root      term/a      0.00 secs Fri May 12 18:25
more     root      term/a      0.01 secs Fri May 12 18:25
sh       root      term/a      0.00 secs Fri May 12 18:25
mv       root      term/a      0.01 secs Fri May 12 18:25
sh       root      term/a      0.00 secs Fri May 12 18:25
...
```

- 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::netSnmAgentOIDs.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
-
- *What else couldn't you answer on older versions of Solaris?*

The source of your data is...

prodfs (/proc)

/usr/bin/ps
/usr/ucb/ps
prstat
ptree
pmap
pstack
pldd
...<pfiles>...

kstat

iostat
kstat
mpstat
sar
vmstat

TNF

prex
tnfextract
tnfdump
tnfmerge
tnfview
taztool
SWAT

various

BSM audit
trapstat
lockstat
mdb
truss
apptrace
pacct
lastcomm

END OF MODULE 1

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

Module 2

Introducing DTrace

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

What is Dtrace?

- Exists in Solaris 10 (*and later releases*)
- Also available with OpenSolaris (*and more...*)

Check out Mac OS X Leopard!

- To run the `dtrace` utility, you must be the **root** user or have access to a role with the appropriate privileges (*set using RBAC*)

Dtrace

- DTrace can be difficult to fully explain!
The more you use it, the more potential you should see for its application.

In the workshop, we will describe and make use of the following...:

- a) DTrace features and options*
- b) The detail that it allows you to view*
- 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*
 - ... all from the same tool
- Is safe to use in production (*if applied correctly!*)

DTrace Features

- Can examine applications without restarting them (*but can also stop a running process!*)
- Dynamically inserts trace points into the kernel and running applications, called *probes*
- Has low impact when running (*if applied to a relatively small number of probes that are firing*), and zero impact when no probes are firing

- Can trace any number of over 40,000

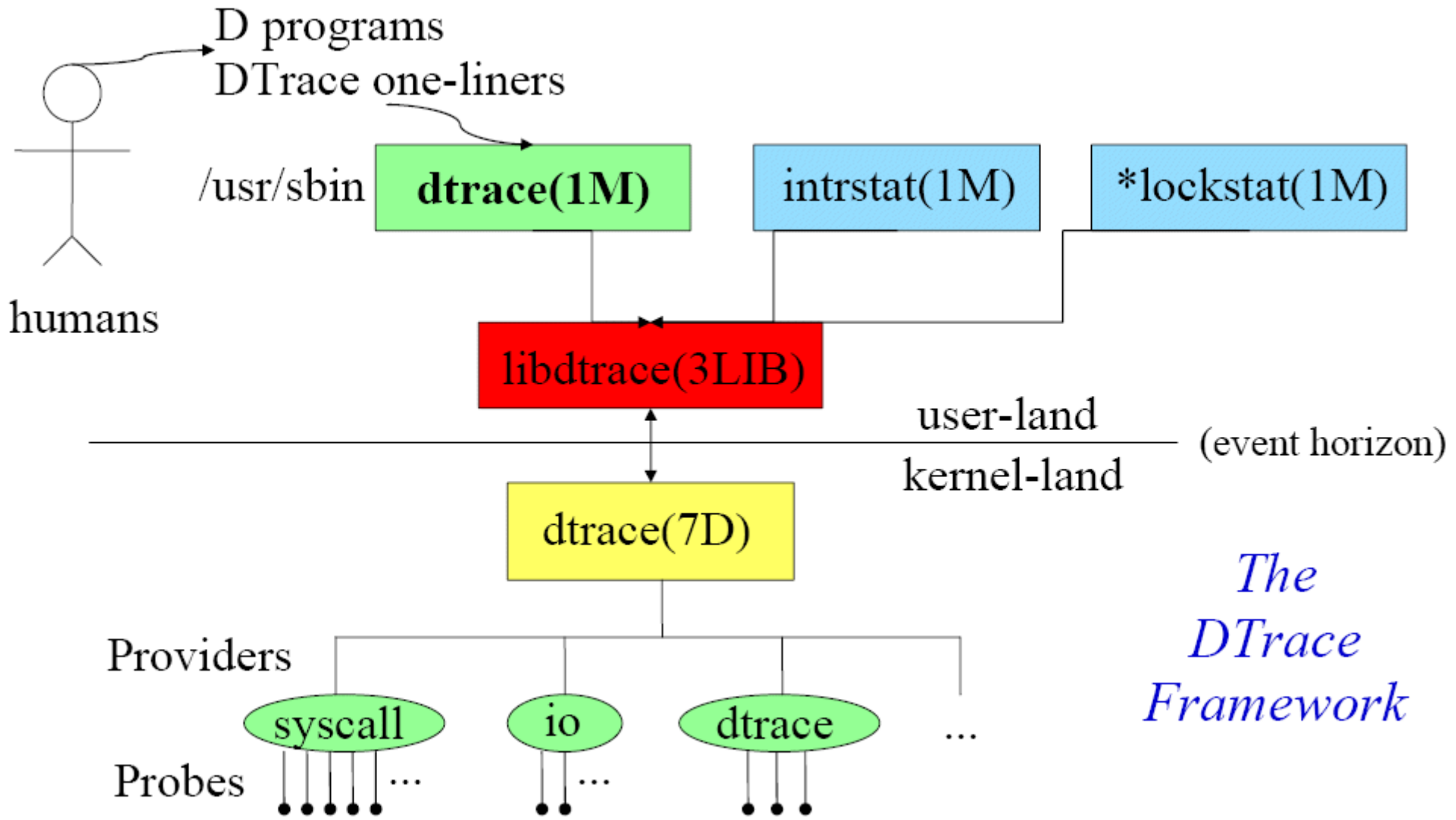
DTrace Features

- 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*
- ... *and there is more*

What is "DTrace"



*The
DTrace
Framework*

When to use DTrace

- System Administrators can use DTrace for performance analysis and troubleshooting ...

1. Monitoring

- *SNMP, sar, SunMC, SWAT ...*

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 elements and terminology
- The aim is to explain the following:

```
# 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  
0     5992          exece:return   expr  
...
```

Listing Probes #1

```
# dtrace -l
ID    PROVIDER      MODULE          FUNCTION        NAME
1     dtrace           
2     dtrace           
3     dtrace           
4     fbt            pool            pool_info      entry
5     fbt            pool            pool_info      return
6     fbt            pool            pool_detach    entry
...
# dtrace -l | wc -l
44797
```

- `dtrace -l` lists the current probes
- For this demonstration, there were 44797 probes. The number varies depending on the OS build and which providers have been recently used.

Listing Probes #2

```
# dtrace -lv
  ID      PROVIDER      MODULE      FUNCTION NAME
...
   8      fbt            pool        pool_open entry
```

Probe Description Attributes

```
  Identifier Names: Private
  Data Semantics:   Private
  Dependency Class: Unknown
```

Argument Attributes

```
  Identifier Names: Private
  Data Semantics:   Private
  Dependency Class: ISA
```

Argument Types

```
  args[0]: dev_t *
  args[1]: int
  args[2]: int
  args[3]: cred_t *
```

...

- `dtrace -lv` lists the probes with attribute and argument details - *Solaris 10 Update 3 & later*

Listing Probes #3

- `-P provider`
lists the current probes for the named provider

```
# dtrace -lP sched
  ID    PROVIDER      MODULE      FUNCTION NAME
1426    sched            unix        resume_from_zombie off-cpu
1427    sched            unix        resume            off-cpu
1433    sched            unix        preempt           preempt
1437    sched            unix        resume            on-cpu
1442    sched            unix        setkpdq           enqueue
1443    sched            unix        setfrontdq       enqueue
1444    sched            unix        setbackdq        enqueue
1445    sched            unix        dispdeq          dequeue
1446    sched            unix        disp             dequeue
1451    sched            unix        swtch            remain-cpu
1454    sched            unix        cpu_surrender    surrender
1455    sched            genunix       turnstile_block  sleep
...
```

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.*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

```
# dtrace -ln 'syscall::exec:entry'
ID      PROVIDER      MODULE              FUNCTION  NAME
1089    syscall           syscall             exec      entry
# dtrace -ln 'syscall::exec*:'
ID      PROVIDER      MODULE              FUNCTION  NAME
1089    syscall           syscall             exec      entry
1090    syscall           syscall             exec      return
5991    syscall           syscall             exece     entry
5992    syscall           syscall             exece     return
```

- 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 ' : : '

[Remember to protect meta-characters with the ' quote]

Specifying Probes #2

- Short cuts exist – dropping left-hand fields is the same as using 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 – setting variables and printing headers*).

Specifying Probes #3

- The following are not the same:

```
syscall::read:entry
```

```
read:entry
```

```
# dtrace -ln 'read:entry'
```

ID	PROVIDER	MODULE	FUNCTION	NAME
1073	syscall		read	entry
12978	fbt	genunix	read	entry
49474	pid581	libc.so.1	read	entry

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

**Brendan's Programming Style Hint:
Use fully-qualified Probe names - They are safest!**

Probe Names

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

Provider

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

Module

Function

Name

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

ID	PROVIDER
1089	syscall

MODULE

FUNCTION	NAME
exec	entry

Tracing

- After finding the desired probes (-l), you are ready to trace activity:

```
# dtrace -n 'syscall::exec*:'  
dtrace: description 'syscall::exec*:' matched 4 probes  
CPU      ID                FUNCTION:NAME  
0        5991                exece:entry  
0        5992                exece:return  
0        5991                exece:entry  
0        5992                exece:return  
0        5991                exece:entry  
...
```

- This is the default output
- Each output line represents a probe firing

Default Output

```
# dtrace -n 'syscall::exec*:'  
dtrace: description 'syscall::exec*:' matched 4 probes  
CPU      ID  
  0      5991  
  0      5992  
...
```

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

DTrace
Probe-id.
not so
interesting

FUNCTION:NAME
exece:entry
exece:return
Last 2 fields of
probe that fired

How many probes
your probe description
actually matched

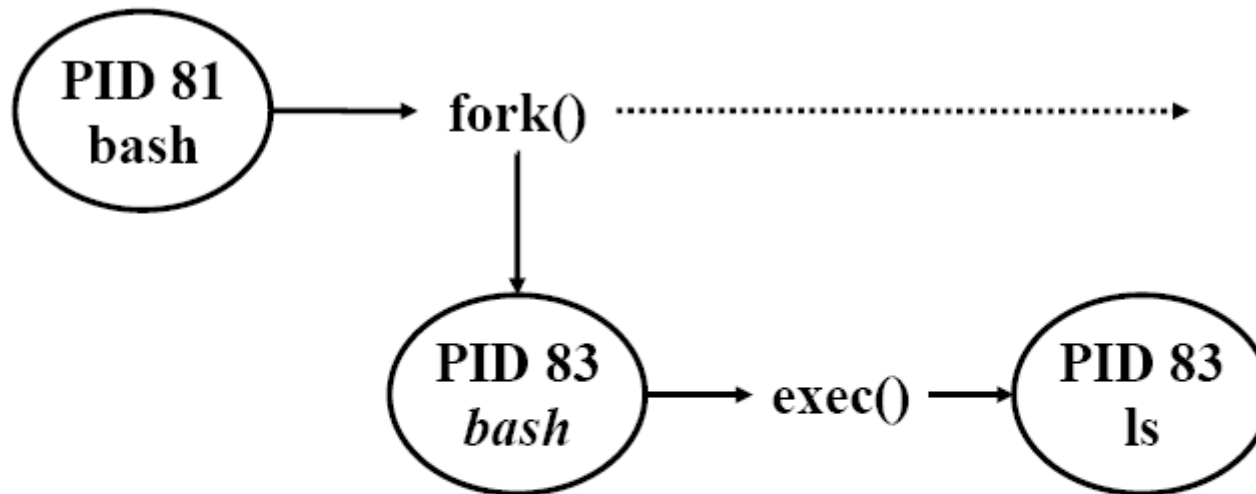
Actions

- A custom action can be performed when a probe fires:

```
# 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  
...
```

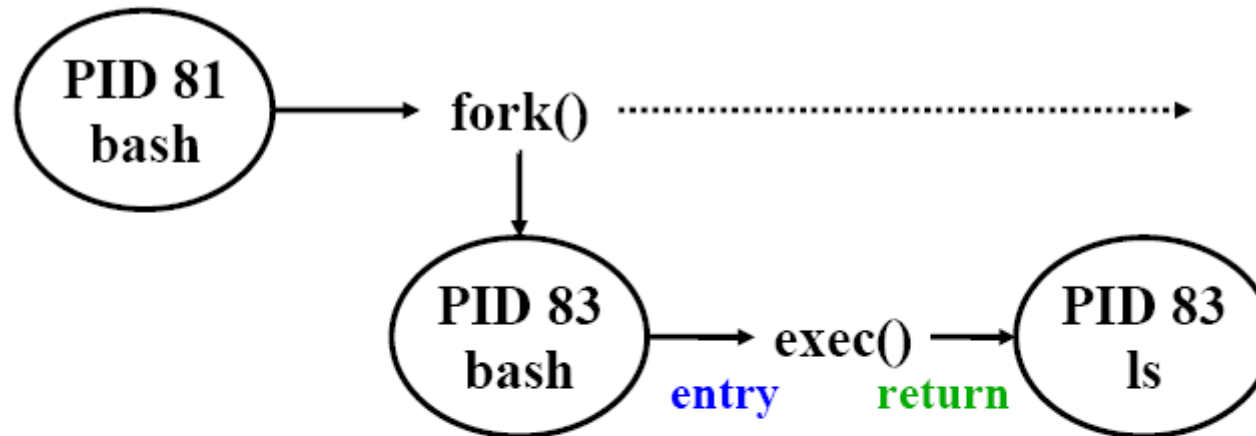
- `trace()` prints one argument
- `execname` is the in-built 'D' variable relating to the process name

Process Creation



- *fork variants: fork1(2), forkall(2), vfork(2)*
- *exec variants: exec(2), exece(2)*

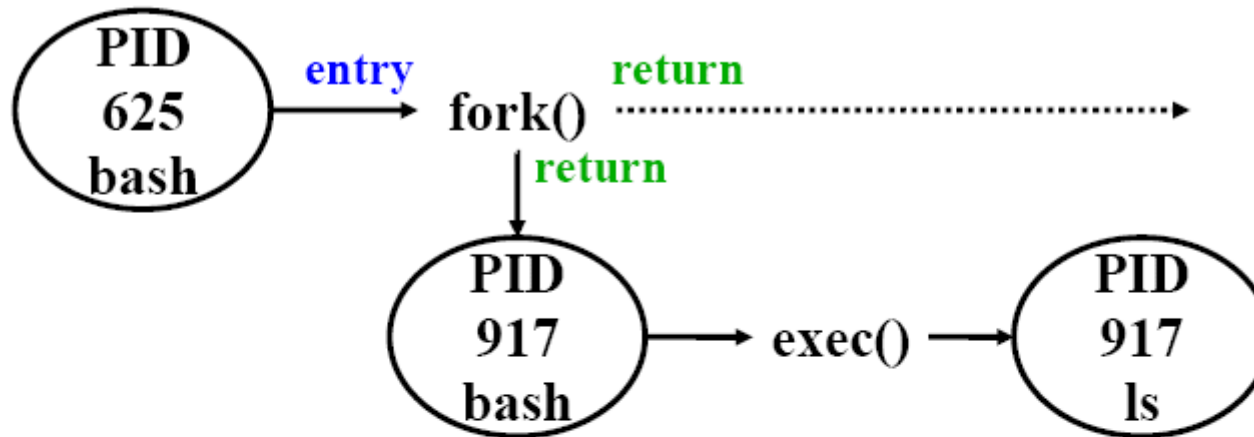
Syscall Entry and return: exec()



```
# 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  ls  
...
```

- Both syscall entry and return can be traced

Syscall Entry and return: fork()



```
# dtrace -n 'syscall::fork*: { trace(pid); }'  
dtrace: description 'syscall::fork*: ' matched 4 probes  
CPU      ID          FUNCTION:NAME      625  
0        6127        fork1:entry  
0        6128        fork1:return      917  
0        6128        fork1:return      625
```

- Two returns! And the child returned first.

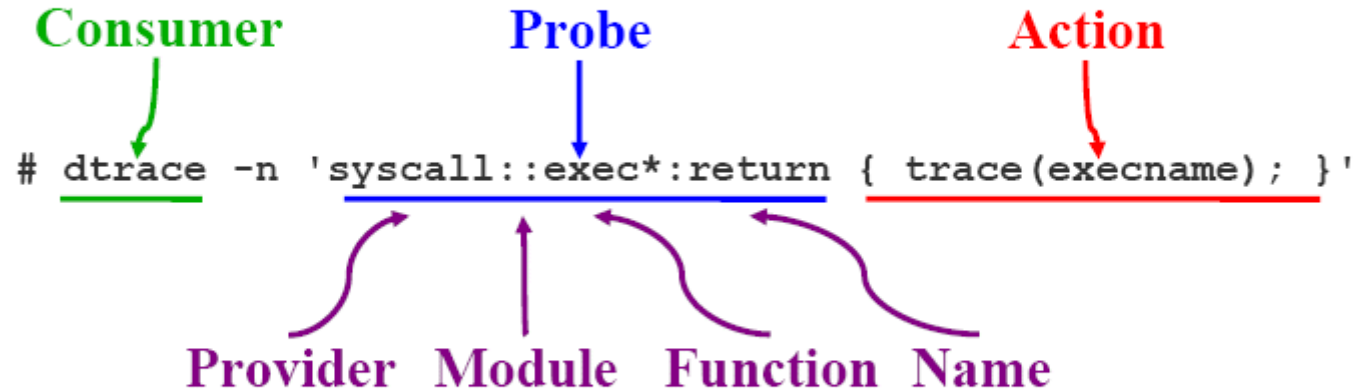
Tracing Process Creation

```
# 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  
...
```

- 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,



It's important to be able to talk-the-talk...

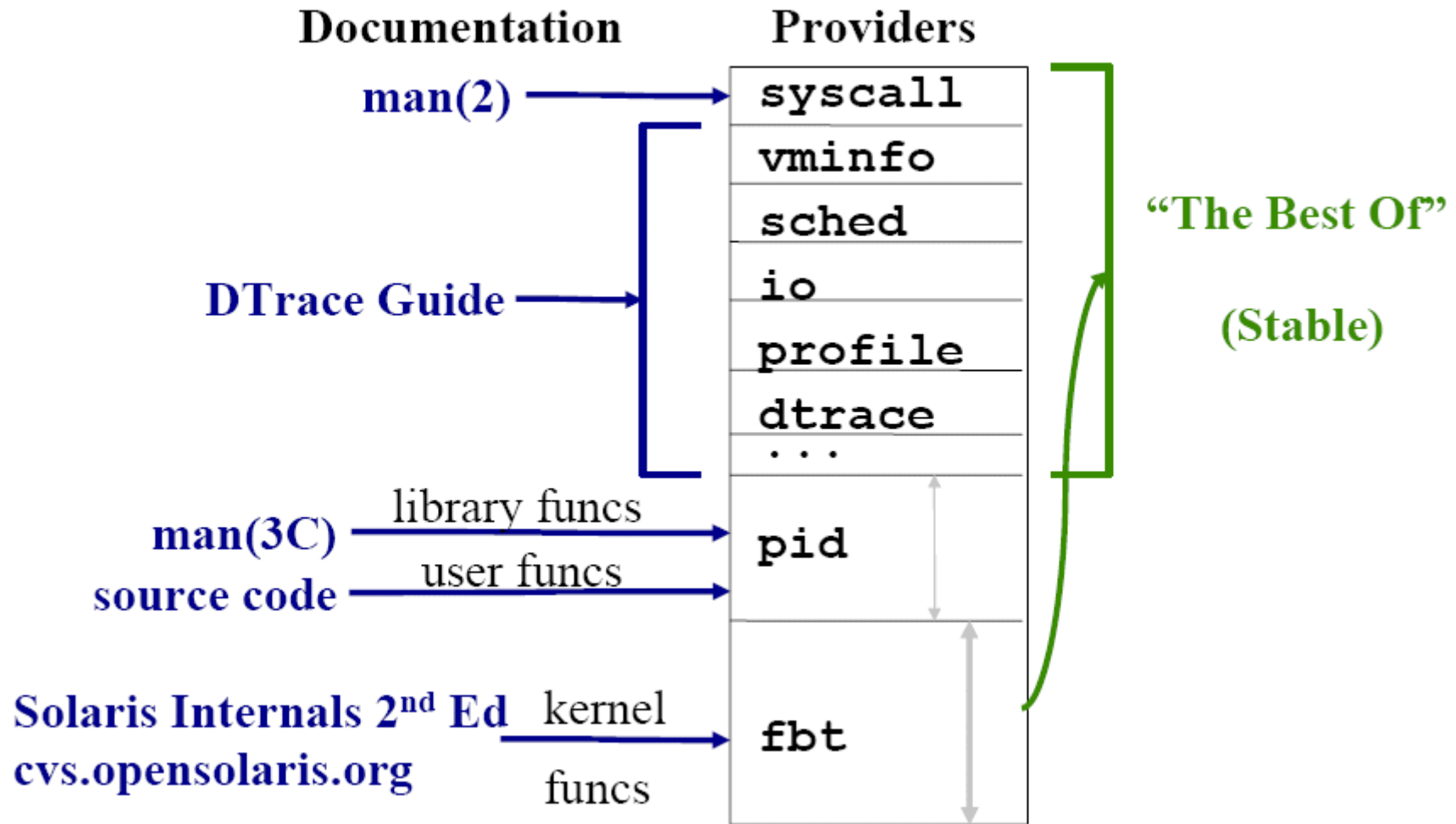
Provider Info

- There are many providers,
 - `syscall` System call entry/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

- *Where are syscalls documented?*
- The pid provider traces user-level activity:
Where are library functions documented?
Where are user functions documented?
- The fbt provider traces kernel-level activity:
Where are kernel functions documented?
- DTrace provides its own custom providers:
Where would they be documented?

Documentation



Action Intro

- `trace()` prints 1 argument
- `printf()` may also be used
- **DTrace built-in variables 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,

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

- format characters include:

%d integers

%f floats

%s strings

%S safe strings – *escaped binary characters*

%Y text formatted time (*use with walltimestamp*)

Style Hint:
Use spaces after commas

walltimestamp

```
# dtrace -n 'syscall::exec*:return { printf("%-20Y %6d %s", walltimestamp, pid,
execname); }'
dtrace: description 'syscall::exec*:return ' matched 2 probesCPU ID FUNCTION:NAME
0 5992      exece:return      2006 May 22 02:12:36  28762 staroffice
0 5992      exece:return      2006 May 22 02:12:36  28763 grep
0 5992      exece:return      2006 May 22 02:12:36  28765 grep
0 5992      exece:return      2006 May 22 02:12:36  28766 soffice
0 5992      exece:return      2006 May 22 02:12:36  28767 dirname
0 5992      exece:return      2006 May 22 02:12:36  28764 expr
0 5992      exece:return      2006 May 22 02:12:36  28768 basename
...
```

- 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\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 u+x 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
...
```

Style Hint:
Indent actions

- So far, so good. The script file is easier to edit.
- Some improvements can be made...

DTrace Scripting

```
# 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
TIME                PID  CMD
2006 May 22 02:24:17 28801 staroffice
2006 May 22 02:24:17 28802 grep
2006 May 22 02:24:17 28804 grep
2006 May 22 02:24:17 28805 soffice
...
```

DTrace Scripting

- Seems like awk!
- `#pragma D option quiet` is the same as `-q`, however is more obvious (*especially when more options are being used in the script*).
- Printing a heading serves to both label the output and to indicate when DTrace has began tracing

`dtrace:::BEGIN` probe fires at start (*heading*)

`dtrace:::END` probe fires at end (*reports*)

curpsinfo->pr_psargs

- To print more than just the execname, try:

```
# 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/staro  
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/ :\(/\)/*[^]*//*$ | /u  
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/Gregg/Mauro ISBN: 0-13-156819-1

End of Module 2

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

Module 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 system call entry and returns
- *syscalls* are the interface between user-land and the kernel, and reflect much of an application's 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

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  
...
```

- Integer arguments are easy to use.
- Here `read()`'s `arg2` is printed – the requested read size. Most are over 1 Kbyte.

entry Arguments: Strings

```
# dtrace -n 'syscall::open:entry { trace(copyinstr(arg0)); }'  
dtrace: description 'syscall::open:entry ' matched 1 probe  
CPU ID          FUNCTION:NAME  
0  1077          open:entry    /var/ld/ld.config  
0  1077          open:entry    /usr/lib/libc.so.1  
0  1077          open:entry    /var/ld/ld.config  
0  1077          open:entry    /usr/lib/libgen.so.1  
0  1077          open:entry    /usr/lib/libc.so.1  
0  1077          open:entry    /var/ld/ld.config  
0  1077          open:entry    /usr/lib/libgen.so.1  
0  1077          open:entry    /usr/lib/libc.so.1  
0  1077          open:entry    /var/ld/ld.config  
0  1077          open:entry    /usr/lib/libc.so.1  
...
```

- `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

```
# dtrace -n 'syscall::read:return { trace(arg0); }'  
dtrace: description 'syscall::read:return ' matched 1 probe  
CPU  ID                                FUNCTION:NAME  
0  1074                                read:return    80  
0  1074                                read:return    45  
0  1074                                read:return    63  
0  1074                                read:return    63  
0  1074                                read:return    80  
0  1074                                read:return   315  
0  1074                                read:return    80  
...
```

- Most actual bytes read are < 1 Kbyte.

return Code Hints

- The `return` code is often typed as an *int*, but sometimes (**x86**) isn't:

```
# 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:*

```
# 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:

```
probe-description
```

```
/predicate/
```

```
{
```

```
    action;action; ...
```

```
}
```

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

Style Hint:
Place the probe,
predicate and actions
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

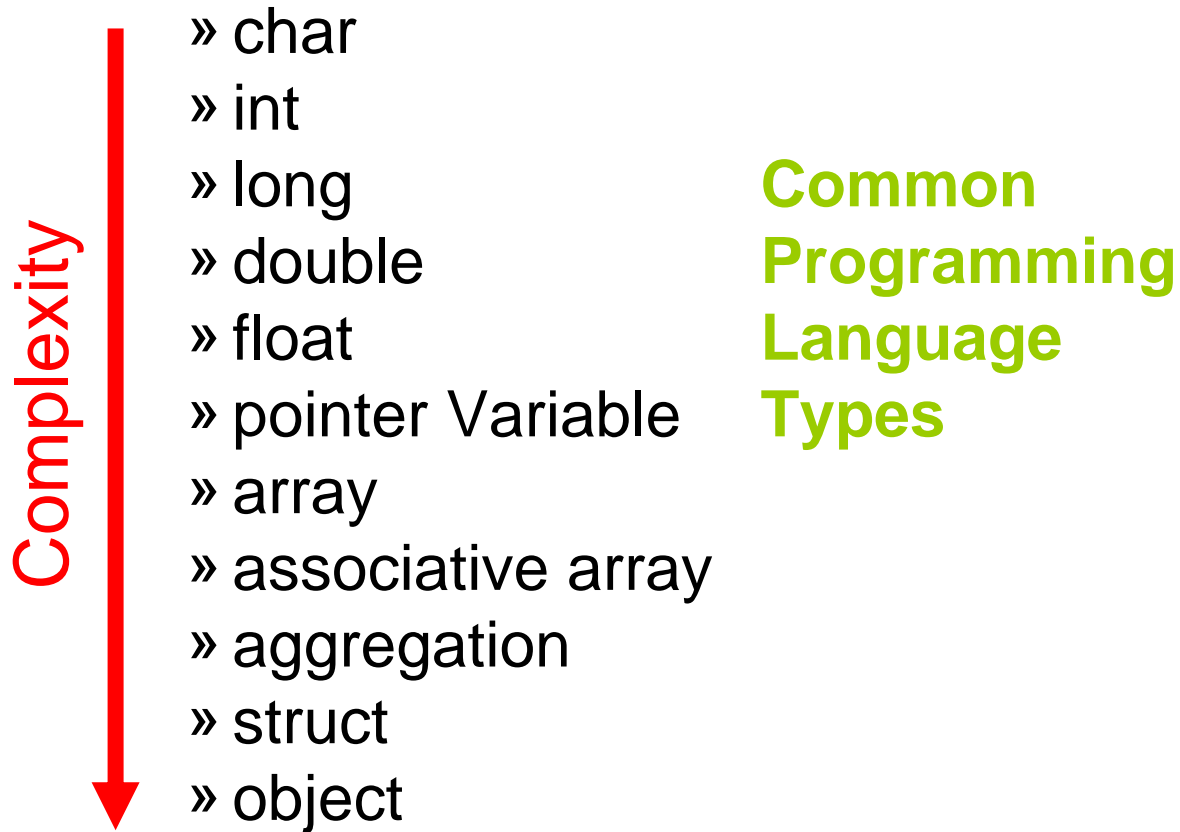
truss-like DTrace

```
# dtrace -n 'syscall::entry /execname == "ls"/ { printf("%x %x %x",
arg0, arg1, arg2); }'
dtrace: description 'syscall::entry ' matched 231 probes
CPU  ID                                FUNCTION:NAME
0   6247                                resolvepath:entry 8047fed 804738c 3ff
0   6119                                sysconfig:entry  6 468d fec1d444
0   6247                                resolvepath:entry d27fd9dc 804738c 3ff
0   6093                                xstat:entry      2 8047fed 80477b8
0   1077                                open:entry       d27f7a24 0 0
0   6093                                xstat:entry      2 d27fbf38 8047070
0   6247                                resolvepath:entry d27fbf38 80470f8 3ff
0   1077                                open:entry       d27fbf38 0 0
...
```

- 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,



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

Aggregation: count()

```
# dtrace -n 'syscall:::entry { @fred = count(); }'  
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 typed, 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[key] = function(args)
```

- The name and the key are optional

- Examples,

Brendan's Style Hint:
I like to name my aggregates

```
- @num[execname] = count();
```

```
- @[execname] = count();
```

```
- @total = count();
```

- Multiple keys can be used. Eg,

```
@num[pid, execname] = count();
```

Multiple Keys

```
# 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 1064644 bytes (1040 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
```

– 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*:

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

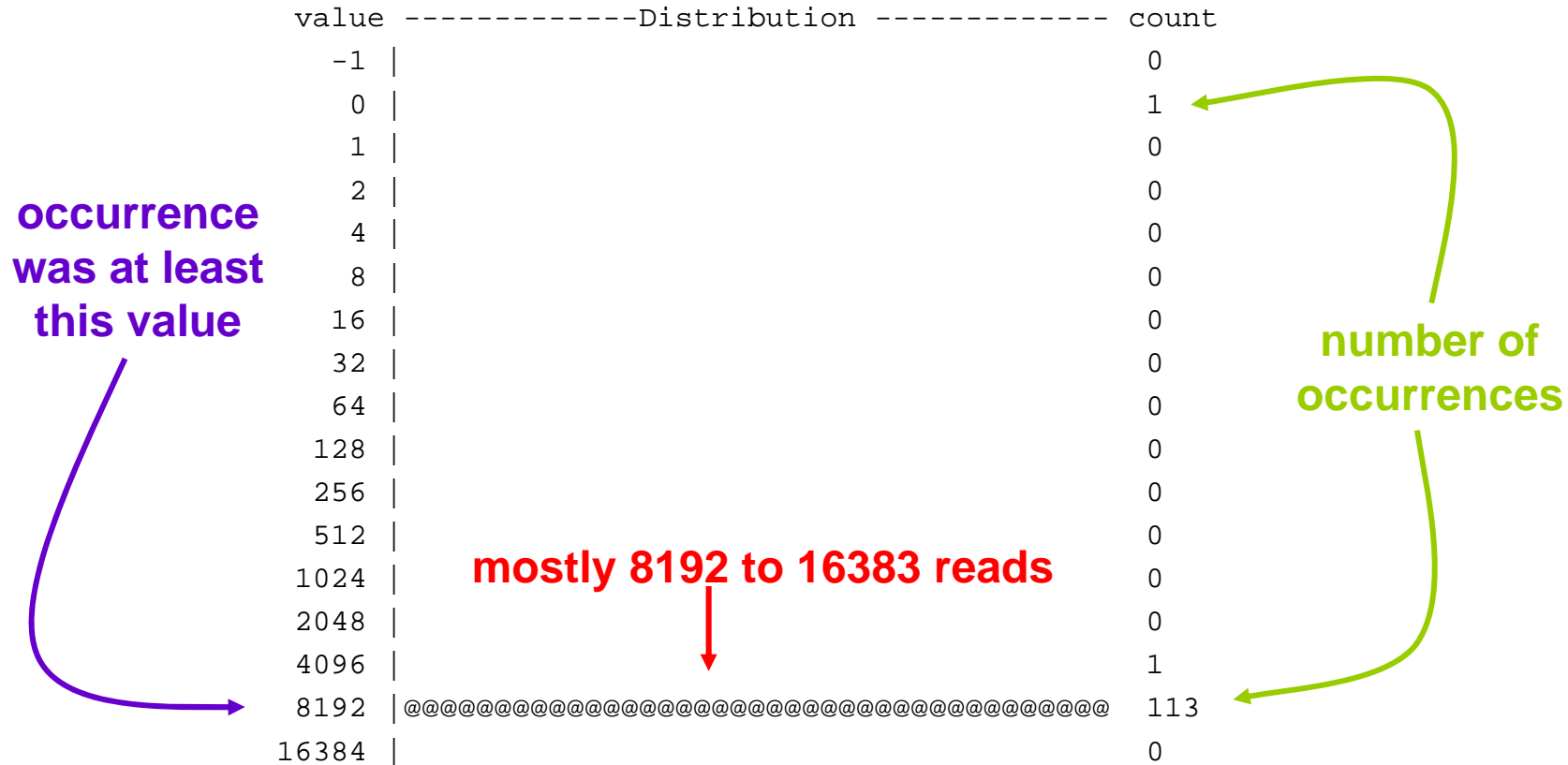
```
^C
```

```
...
```

```
wc
```

value	-----Distribution-----	count
-1		0
0		1
1		0
2		0
4		0
8		0
16		0
32		0
64		0
128		0
256		0
512		0
1024		0
2048		0
4096		1
8192	@@@@	113
16384		0

Distribution plots



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

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()

```
# 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 shows that `wc` used File Descriptor 3 for `read()`s, while `bash` used File Descriptor 0 (*stdin*).

Raw Data

- Remember that raw data can always be dumped, for other post processing:

```
# 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   0  
^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(); }'
```


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

```
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
```

ustack() Short Lived Processes

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

```
# 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 disappears 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` (*another uses DTrace*):

```
$ 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                               read:return
                                           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.

```
# 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`zsh_main+0xe  
zsh`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`ssh_main+0x1d0
zsh`main+0xe
zsh`0x8059bc
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 are:**
 - `write()`, `writew()`, `pwrite()`, `pwrite64()`
- **when tracing reads – be aware that there are:**
 - `read()`, `readv()`, `pread()`, `pread64()`
 - There is also `readlink()` -for symlinks
- **Other variants exist:** `open()`, `open64()`...
- `readv()` and `writew()` 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 writetch`
- 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

```
# 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()

```
# 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
```

Style Hint:
The `hz` isn't necessary
but it helps readability

- This can help identify where a process is spending its time (*extremely useful!*)

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 trace on an event-by-event basis.*

End of Module 3

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

Module 4

DTrace one-liners

- List useful one-liners
- Discuss when to use them

One-Liners

- 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 Brendan Gregg 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 Brendan writing >100 scripts!)

Useful One-Liners

- The DTrace one-liners list is kept in the DTraceToolkit under Docs/one liners.txt
<http://www.opensolaris.org/os/community/dtrace/dtracetoolkit>
or
<http://www.brendangregg.com/dtrace.html#DTraceToolkit>
- So far we have covered enough DTrace syntax to understand most of them.
- Many probes these one-liners use will be new, but readily understandable in such a practical context.
- *We will demonstrate many of these now ...*

One-Liners #1

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

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\n", 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 Module 4

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

Module 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 @

Snoop

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

- 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.

Summary

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

```
dtrace: description 'syscall::exece:return ' matched 1 probe
```

```
^C
```

soffice.bin	1
pagein	1
javaldx	1
basename	1
sopatchlevel.sh	1
soffice	1
staroffice	1
dirname	2
grep	2
expr	3
uname	4

- The most frequent occurring item is often an issue, but not always.

Strategy #: 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  96
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   279 1285    269  1   4  95
0  0  51  707120   75408     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   276 1154    234  1   3  96
^C
```

- `vmstat` provides a general system-wide view - a good starting point. Module 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:

```
# 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
acoread	659
top	4615

Drill down analysis

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

```
# dtrace -n 'syscall:::entry /execname == "top"/ { @num[probfunc] = 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:

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

```
lib.so.1`__open+0x15  
lib.so.1`open+0x77  
top`getptable+0xe5  
top`get_process_info+0x14  
top`main+0x695  
top`_start+0x80  
2424
```

Drill down analysis

```
libc.so.1`__open+0x15  
libc.so.1`open+0x77  
top`getptable+0xe5  
top`get_process_info+0x14  
top`main+0x695top`_start+0x80  
2424
```

the culprit

- 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.

```
# dtrace -n 'sysinfo::: /execname == "top"/ { @num[probename] = count(); }'
```

```
dtrace: description 'sysinfo::: ' matched 59 probes
```

```
^C
```

trap	2
writecth	9
syswrite	9
outch	9
inv_swch	48
pswitch	68
readch	2727
sysread	2727
namei	2727

- Be aware that this is counting the occurrences that a probe was fired - ie, the `readch` statistic probe was called 2727 times.

Don't infer too much...

Frequency Count

```
# dtrace -n 'sysinfo::: /execname == "top"/ { @num[probename] = sum(arg0); }'  
dtrace: description 'sysinfo::: ' matched 59 probes  
^C  
  syswrite                8  
  inv_swch                38  
  pswch                   56  
  sysread                 2424  
  namei                   2424  
  writech                 3135  
  outch                   3158  
  readch                  814464
```

- 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:

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

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

```
^C
```

```
getpid  
stat64  
ioctl  
read  
write
```

All multiples of 17!

17
17
34
51
102

Yellow Pig

- Brendan likes 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 (*who else would?*)
(*This isn't made up - try Google, Yahoo, Ask!*)

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

Event Probes with Stacks

- Lets see why Xvnc calls so many `read()`s:

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

```
^C
```

```
...
```

```
libc.so.1`_read+0x15  
Xvnc`0x8094a5b  
Xvnc`_XSERVTransRead+0x15  
Xvnc`ReadRequestFromClient+0x11e  
Xvnc`Dispatch+0xdb  
Xvnc`main+0x479  
Xvnc`_start+0x5d  
309
```

```
libc.so.1`_read+0x15  
Xvnc`ReadExact+0x4b  
Xvnc`rfbProcessClientMessage+0x73  
Xvnc`rfbCheckFds+0x109  
Xvnc`ProcessInputEvents+0xb  
Xvnc`Dispatch+0x65  
Xvnc`main+0x479  
Xvnc`_start+0x5d
```

Profile Sampling with Stack

- Lets sample to estimate why Xvnc is on-CPU:

```
# dtrace -n 'profile:::profile-1000hz /execname == "Xvnc"/ {  
@num[ustack()] = count(); }'  
dtrace: description 'profile:::profile-1000hz ' matched 1 probe  
^C  
...
```

```
Xvnc`cfb32FillRectSolidCopy+0x9c
```

```
200
```

That looks interesting



```
libc.so.1`_writev+0x15
```

```
Xvnc`0x8094aaf
```

```
Xvnc`_XSERVTransWritev+0x15
```

```
Xvnc`FlushClient+0x72
```

```
Xvnc`FlushAllOutput+0xeb
```

```
Xvnc`Dispatch+0x11d
```

```
Xvnc`main+0x479
```

```
Xvnc`_start+0x5d
```

```
370
```

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 (Module 1)*
 - *know the OS well (Solaris Internals 2nd edition)*
 - *study source code, and how applications are constructed and operate*
 - *be willing to think outside the box!*

End of Module 5

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

Module 6

Providers

- Provider Guide
- Using *fbt*
- Using *pid*

Providers

- *syscall* System call entry and return probes
- *sysinfo* Classic sysinfo statistic probes
- *vminfo* Virtual Memory statistic probes
- *io* Disk and NFS events
- *proc* Process events such as creation
- *sched* System scheduling events
- *lockstat* Kernel synchronisation lock events
- *plockstat* User synchronisation lock events
- *mib* MIB statistic updates
- *fasttrap* User location tracing
- *fpuinfo* SPARC FPU simulation events

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 an application's behaviour
- *syscalls* are documented in section 2 of the man pages: `man -s 2 intro`

sysinfo

- Example one-liner:

```
# 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`

io

- Example one-liner:

```
# 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](#))
- *Note - There is also a provider specific to ZFS!*

proc

- 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 signals
- Provides high level probes so that process analysis can be performed easily
- Documented in the DTrace Guide ([chap 25](#))

sched

- Example one-liner:

```
# 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

fbt

- Example one-liner:

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

- Function Boundary Tracing – traces raw kernel function events
- These probes are regarded as **unstable** - their names and arguments can and do change between minor releases of Solaris (*which is why we have the higher level providers*)
- *fbt* makes up the bulk of the probes: 30,000+

fbt

- 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

pid

- Example one-liner,

```
# 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

pid:::

- The provider name is,

`pid`*PID* Where *PID* is the target to trace

`pid81` Trace PID *81*

`pid$target` Trace the target of *-c* or *-p*

- The full syntax is:

pid*PID*:**segment**:**function**:**name**

segment is the name of the mapping. eg,

`libc.so.1` (or “libc”) a library

`a.out` the binary

function is the function name. eg, “main”

name is *entry*, *return*, or instruction *address*

pid demonstrations

- user functions,

```
# 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,

```
# 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
CPU    ID                                FUNCTION:NAME
0     39188                             set_IPv4_options:entry
0     39189                             set_IPv4_options:0
0     39190                             set_IPv4_options:1
0     39191                             set_IPv4_options:3
0     39192                             set_IPv4_options:6
0     39193                             set_IPv4_options:7
0     39194                             set_IPv4_options:8
0     39195                             set_IPv4_options:9
0     39196                             set_IPv4_options:10
0     39197                             set_IPv4_options:12
0     39198                             set_IPv4_options:19
...
0     39187                             set_IPv4_options:return
```

pid demonstrations

- library functions,

```
# 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	39596	munmap:entry
0	39296	rt_bind_clear:entry
0	39287	_rt_bind_clear:entry
0	39294	rt_mutex_unlock: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,

```
# 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
```

CPU	ID	FUNCTION:NAME
0	39188	getpid:entry
0	39189	getpid:0
0	39190	getpid:5
0	39191	getpid:6
0	39192	getpid:b
0	39193	getpid:d
0	39194	getpid:13
0	39195	getpid:15
0	39187	getpid:return
0	39188	getpid:entry
0	39189	getpid:0
0	39190	getpid:5
0	39191	getpid:6
...		

End of Module 6

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

Module 7

The D Language

- Comments
- Variable Types
- Macros
- Script Options
- Pragmas

Comments

- 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 its first declaration
- Variable types can be pre-declared
- Types include:
 - integer
 - string
 - pointer
 - associative array
 - aggregate
 - thread-local *(linked specifically to an event thread!)*
 - clause-local *(local to a DTrace program clause)*

Integer Variables

- 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. *Such declarations are usually placed at the top of the program.*

String Variables

- 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 (eg, `vnode_t`, `proc_t`, ...)
- Special builtins,
 - `curthread` pointer to the *on-cpu thread*
 - `curpsinfo` pointer to a *psinfo* like structure
- Eg,
 - `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:

<code>string name[int];</code>	declaration
<code>name[3] = "Fred";</code>	declaration / setting
<code>/name[3] == "Fred" /</code>	predicate testing
<code>name[3] = 0;</code>	memory cleanup
<code>name["sd0", 1] = 4;</code>	two key array
- Try to use aggregates instead

Aggregates

- Discussed in Module 3

- Using Aggregates:

<code>@name[keys] = func();</code>	setting
<code>printa("%s... %@d", @name);</code>	printing
<code>trunc(@name);</code>	clearing
<code>trunc(@name, size);</code>	truncating to size
<code>clear(@name);</code>	clearing values
<code>normalize(@name, n);</code>	dividing by n

- Aggregates perform well, and produce great reports

Thread-Local

- A variable that is stored private to each thread
(linked to a probe firing on an event)

- Using thread-local:

<code>self->start = timestamp;</code>	setting
<code>self int start;</code>	declaration
<code>/self->start > 0/</code>	predicates
<code>self->start = 0;</code>	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 (*and available only in that clause!*)
- Using clause-local:

<code>this->delta = 5;</code>	setting
<code>this int delta;</code>	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:

<code>\$1, \$2, ...</code>	arguments to script as <i>ints</i>
<code>\$\$1, \$\$2, ...</code>	arguments to script as <i>strings</i>
<code>\$pid</code>	<i>PID</i> of dtrace(1M)
<code>\$uid</code>	real <i>UID</i> of dtrace(1M)
<code>\$target</code>	<i>PID target</i> of dtrace (<i>-c</i> or <i>-p</i>)

- For the argument variables to assume default values (0 or ""), use:

```
#pragma D option defaultargs
```

Script Options

- When writing complex scripts, it can be useful for them to process options. Eg:

```
# opt/DTT/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 inside a shell script, and using the shell's *getopts* function. Be careful to write neatly and follow programming best practices – *such scripts can get messy very quickly*.

Pragmas

- These change the behaviour of DTrace, and often an alternative to command line options
- Useful pragmas include:
 - `#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!*

- *Restart a stopped process:*

```
system("/usr/bin/prun %d", pid);
```

Destructive Actions: An example

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

```
#pragma D option quiet
```

```
#pragma D option destructive
```

Destructive action is applied

```
syscall::chmod:entry  
{
```

```
    stop() ;
```

```
    self->start = pid ;
```

```
    system("/usr/bin/ptree %d", self->start) ;
```

```
    printf("The chmod command was: %d %s\n", self->start, curpsinfo->pr_psargs) ;
```

```
    printf("Executed by the userID: %d \n", uid) ;
```

```
    system("/usr/bin/prun %d", self->start) ;
```

```
}
```

The process is stopped.

```
syscall::chmod:return
```

```
/self->start/
```

```
{
```

```
    self->start = 0
```

```
}
```

**The process is re-started.
In the interim, we can trace
process details.**

End of Module 7

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

Module 8

Advanced Scripting

- Step through an example script
- Discuss script internals

Example: shortlived.d

- The following is from the DTraceToolkit:

```
# ./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
    sopathlevel.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
```

Style Hint:
Say Sampling, if
sampling and
Tracing, if tracing

Style Hint:
Show units on output,
if possible
(such as secs, ms, ns)

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.

shortlived.d

```
#!/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.
```

**Style Hint:
Comment & Document**

shortlived.d

```
* 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.  
*/
```

Style Hint:
Histories are crucial

shortlived.d

```
/*
 * Start
 */

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

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

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

Style Hint:
Comment often. You'll forget why you did something when you come back to the script later!

shortlived.d

```
/*  
 * 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  
 */
```

Style Hint:
arg0 != 0 is verbose
to aid readability

shortlived.d

```
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;
}
```

shortlived.d

```
/*
 * 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);
}
```

Style Hint:
80 char width,
4 char continuation

End of Module 8

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

Module 9

The DTrace Toolkit

- Introduce the DTrace Toolkit
- Explain the layout
- A recipe for getting started

DTrace Toolkit

<http://www.opensolaris.org/os/community/dtrace/dtracetoolkit>

<http://www.brendangregg.com/dtrace.html>

- Version 0.99 contains 230 scripts
- Freeware (CDDL)
- Serves to:
 - Provide tools to analyse common problems
 - Provide numerous demonstrations of programming in D
 - Help promote the use of DTrace
- Originally written for Solaris 10 3/05
- Newer tools written for Solaris 10 8/07

Main Parts

- The DTraceToolkit is currently 3 main parts:
 - The Scripts *
 - Man Pages Man
 - Examples (found in the Docs/Examples directory)
- There is a man page for every script, and an example file for every script.
- It took considerable time to write 200+ man pages – *so please read them!*
- The *Example* files are often more useful than the man pages

Installation

- Isn't actually required – the scripts can be run immediately after extraction into a directory
- 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 0.99, 30-Sep-2007
```

```
hit Ctrl-C any time you wish to quit.
```

```
Enter target directory for installation [/opt/DTT]:
```

Script Layout

- The scripts are positioned in a hierarchy - the most useful are in the top-level directory; the others are in meaningful sub-directories
- The `Bin` directory contains *symlinks* to all the scripts (*in their respective sub-class directories*):

```
# ls -F /opt/DTT
Apps/      Examples/  Java/      opensnoop*  Shell/
Bin/       execsnoop* JavaScript/ Perl/        Snippets/
Code/      FS/        Kernel/    Php/         statsnoop*
Cpu/       Guide     License@   Proc/        System/
dexplorer* hotkernel* Locks/     procsystime* Tcl/
Disk/      hotuser*  Man/       Python/      User/
Docs/      Include/  Mem/       README@     Version
dtruss*    iopattern* Misc/      Ruby/        Zones/
dvmstat*   iosnoop*  Net/       rwsnoop*
errinfo*   iotop*    Notes /    rwtop*
```

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 or try running them with the `-h` option.
- 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

Man Pages

- Man Pages can be read using,

```
# 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
```

```
iosnoop [-a|-A|-Deghinostv] [-d device] [-f filename] [-mmount_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.

The Example Files

- Read using:

```
# more Docs/Examples/bitesize_example.txt
```

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.

```
# bitesize.d
```

```
Tracing... Hit Ctrl-C to end.
```

```
^C
```

PID	CMD	value	-----Distribution	count
7110	-bash\0	512		0
		1024	@@@@@@@@@@@@@@@@@@@@@@@@@@@@	2

The Toolkit Script Directories

- There are, amongst the directories:

Bin/	Symlinks to the scripts
Apps/	Application specific scripts
Cpu/	Scripts for CPU analysis
Disk/	Scripts for disk I/O analysis
Extra/	Misc scripts
Kernel/	Scripts for kernel analysis
Locks/	Scripts for lock analysis
Mem/	Scripts for memory analysis
Net/	Scripts for network analysis
Proc/	Scripts for process analysis
System/	Scripts for system analysis
User/	Scripts for user based activity analysis
Zones/	Scripts for analysis by zone

- 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`

dexplorer

- 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
- An HTML-iser for `dexplorer` has yet to be written (*this is a work-in-progress*)

Behind the DTrace Toolkit

- 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**?*

Testing

- *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 so long as we are aware that they are estimates.

End of Module 9

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

Module 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 which it 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 the 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, ...

Elapsed vs On-CPU

```
# more Docs/Examples/dtruss_example.txt
```

```
...
```

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,

```
# dtruss -eon bash
```

PID/LWP	ELAPSD	CPU SYSCALL(args)	= return
3911/1:	41	26 write(0x2, "l\0", 0x1)	= 1 0
3911/1:	1001579	43 read(0x0, "s\0", 0x1)	= 1 0
3911/1:	38	26 write(0x2, "s\0", 0x1)	= 1 0
3911/1:	1019129	43 read(0x0, " \001\0", 0x1)	= 1 0
3911/1:	38	26 write(0x2, " \0", 0x1)	= 1 0
3911/1:	998533	43 read(0x0, "-\0", 0x1)	= 1 0
3911/1:	38	26 write(0x2, "-\001\0", 0x1)	= 1 0
3911/1:	1094323	42 read(0x0, "l\0", 0x1)	= 1 0
3911/1:	39	27 write(0x2, "l\001\0", 0x1)	= 1 0
3911/1:	1210496	44 read(0x0, "\r\0", 0x1)	= 1 0
3911/1:	40	28 write(0x2, "\n\001\0", 0x1)	= 1 0
3911/1:	9	1 lwp_sigmask(0x3, 0x2, 0x0)	= 0xFFBFFEFF 0
3911/1:	70	63 ioctl(0x0, 0x540F, 0x80F6D00)	= 0 0

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
 - ustack() when solaris statistics are triggered
 - engaging in “known count” strategy
 - dereferencing random strings

End of Module 10

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

DTrace Workshop Exercises

- Now it's time for you to get your hands dirty solving a variety of problems (*guidance will be available*)
- If relevant, problems can be created 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 and practice with DTrace coding*
- **Good Luck! Enjoy 😊**

DTrace Workshop Exercises

- A range of exercises will be provided by the instructor running the workshop
- The exercises will include accessing the documentation and examples in the DTrace Toolkit - You will, therefore, be expected to install the most recent DTrace Toolkit onto your systems.