OpenWorld 2018

ORACLE OPEN WORLD

HOL6339
Observing and Optimizing Your Application on Oracle Linux with DTrace

Jeff Savit
Director
Oracle Linux & Oracle VM
Oracle Infrastructure Technologies

October 25, 2018

ORACLE

Program Agenda

1. **Introduction**
2. D Language
3. DTrace for User Space
4. DTrace on Oracle Linux
5. Conclusion

First, if you're doing the lab…

- Log into your lab host
- Start the lab VM
- At the Grub prompt, use up-arrow to select UEK, then press ENTER
- Open two terminal windows, and become 'root'
- Look for the HOL lab folder, and open the examples text file
- Follow the examples step-by-step

What is DTrace?

- A Brief history
  - Dynamic Tracing framework originally introduced in Solaris 10 in 2005
  - First Oracle Linux DTrace release in 2011, currently available in UEK4, UEK5
  - Also ported to MacOS X and FreeBSD
- High level D language
- No overhead when not enabled
- Can be used on production binaries, does not require rebuilding instrumented software
- Can trace kernel and user applications
- Observability across entire stack

DTrace vs Other Linux Tracing Tools

- DTrace vs perf
  - perf was born as a profiler, DTrace provides more comprehensive debugging capabilities.
  - You can do profiling with D, but perf is missing the debugging capabilities of DTrace.
- DTrace vs BPF & BCC
  - BPF (Berkeley Packet Filter) provides in-kernel VM, and JIT (Just In Time) compiler to native code
  - BCC (BPF Compiler Collection) provides framework/language support to write your own tools
  - DTrace makes it easier to quickly write scripts and modify to zero-in on issue
  - No speculations in BPF
- DTrace vs SystemTap
  - SystemTap is a DTrace clone, Linux only vs DTrace which is supported on multiple OS
  - SystemTap relies on kprobes, and compiles scripts into binary kernel modules which are loaded into the kernel
    - SystemTap relies on tool chain vs DTrace which is ready to use once installed
    - DTrace uses in-kernel VM with strict checks to execute probe actions

Example with Oracle VM Manager

- **Analyzing system behavior while Oracle VM Manager is starting in a Xen guest**
- mpstat does not report number syscalls per second as it does on Solaris, so use DTrace to quickly gather that info.
- Each system call issued on the system is counted and printed every second

```
# dtrace -q -n 'syscall:::entry { syscalls++; }
tick-1s { printf("%d\n", syscalls); syscalls =
0; }'
234
267
1818
455
9825
398
3928    <= service ovmm start
50748
102282
68670
65105
93341
91780
145055
```

Example with Oracle VM Manager – Drill Down

- **More system calls than expected. Which processes issue the most system calls?**
- The notation @[execname,pid] = count(); counts the number of system calls per execname/pid. The probe syscall:::entry is invoked for each system call invoked on the system, except dtrace itself.
- execname and pid are built-in variables containing the name of the process executable and its pid at the time the probe fired.
- The predicate guards the action so syscalls from DTrace are ignored (to reduce noise on terminal).
- Java processes running Oracle VM Manager are issuing most of system calls, which is expected

```
# dtrace -n 'syscall:::entry
/execname!="dtrace"/ { @[execname,pid] =
count(); }' -x aggsortrev
dtrace: description 'syscall:::entry '
matched 310 probes
^C

  java          26523             473316
  java          26416             205199
  java          26240               3583
  pgrep         26467               1696
  wlst.sh       26477               1204
  mysqld        25559                815
  python        24731                545
```

Example with Oracle VM Manager – Drill Further Down

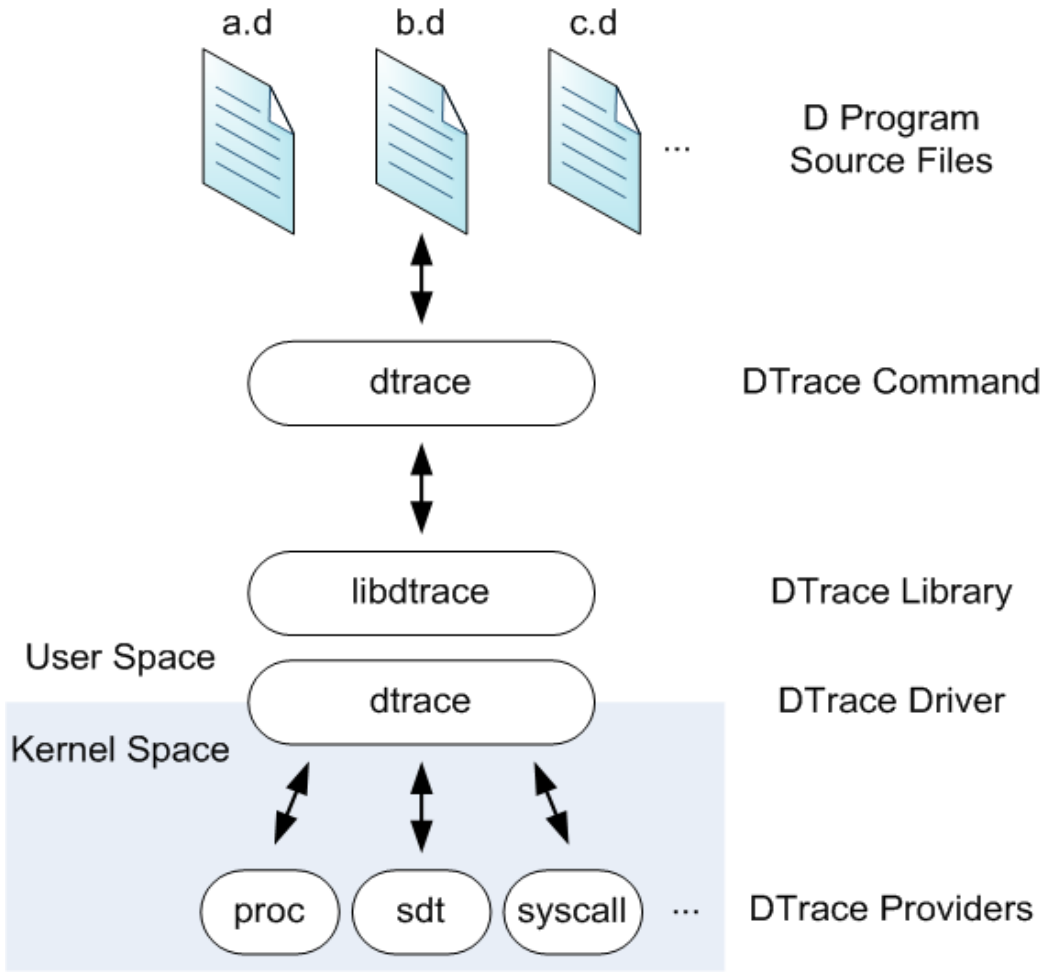
- **Which system calls are invoked by Oracle VM Manager java processes?**
- probefunc is a built-in variable containing the name of the probe fired, in this case the name of the system call
- gettimeofday() system call is the winner by a wide margin
- Shouldn't these calls use vDSO (virtual dynamic shared object)?
- Not with the Xen clock source, they fall back to slower full system calls
- Impacts not only OVMM but every other process running in Xen guest that relies on these calls.

```
# dtrace -n 'syscall:::entry
/execname == "java"/
{ @[probefunc] = count(); }' -x
aggsortrev
dtrace: description 'syscall:::entry '
matched 310 probes
^C

  gettimeofday                  373559
  read                           81569
  lseek                          78504
  write                          50803
  clock_gettime                  31289
  newstat                        11599
  futex                           8535
  newlstat                        7504
  close                           4106
  newfstat                        3635
```

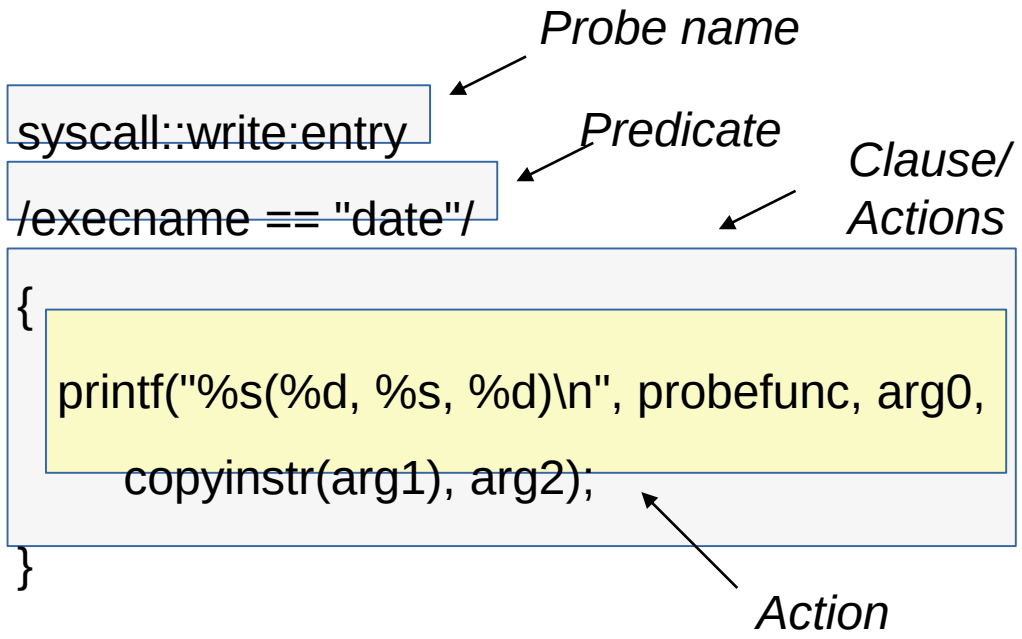
DTrace Components

- D Language
  - To enable probes and describe what to do when probes fire.
- Consumers
  - A process that interacts with DTrace through libdtrace
  - Typically the dtrace command
  - On other platforms, some observability commands are DTrace consumers, for example lockstat on Solaris
- Providers
  - Set of probes and types which share a method of probing (e.g. static probes vs timer events) or are specific to a subsystem
  - Hide implementation details from consumer



DTrace Language Structure

- **Probe**: point in software where instrumentation can occur
  - Probes can be enabled dynamically
- **Predicate**: control whether a probe action is executed
- **Clause/Actions**: series of statements to perform when a probe fires
- **Action**: operation executed as part of a clause.



DTrace Probes

- All available probes can be listed with the “dtrace -l” command

```
# dtrace -l
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4        fbt            vmlinux                 native_read_tscp entry
    5        fbt            vmlinux                 native_read_tscp return
    ...
```

- Can also list probes for a specific provider (-P), module (-m), function (-f)
- Probe names are of the form ***provider:module:function:name***
  - Empty fields match everything and can use wildcards, for example: “syscall::read*:entry”
  - Left-most blank fields can be omitted, for example: “:::probe-name” -> “probe-name”
- A probe fires when it is enabled, and the code it is associated with is executed
  - When a probe fires, it is passed arguments. The types and semantics of arguments are defined by the providers.

Actions

- Series of actions (also called clauses) are executed when a probe fires
  - { action1; action2; action3; }
  - fall-through code only – no loops or branches available (no risk of hangs induced by probe actions)
- Different types of actions:
  - Calls which record or report state to DTrace consumers: trace(), printf(), stack(), tracemem(), etc
  - Calls which change the DTrace state (also called subroutines): copyin(), cleanpath(), etc.
  - Variable assignments
  - Destructive: change the state of the system (must be enabled with -w option): chill(), panic()

Other Actions

- Default action: if no action is specified for a probe, the default action will print the CPU, probe id, function, and probe name
- Data recording: printf(), printa(), trace(), tracemem(), freopen(), ftruncate(), func(), mod(), stack(), ustack(), sym(), usym()
- Destructive actions (enabled with -w option): chill(), panic(), system(), stop(), raise(), copyout(), copyoutstr()
- Special actions: exit(), setopt(), speculate(), discard(), commit()
- Routines: alloca(), basename(), bcopy(), cleanpath(), copyin(), copyinstr(), copyinto(), d_path(), dirname(), getminor(), getmajor(), ntohl(), …, htonl(), …, str*(), etc

Predicates

- A predicate is a D expression of the form “/<expression>/” specified between the probe name and action
- An action is executed only if its predicate evaluates to non-zero
- Example: trace the processes issuing write() calls greater than 100 bytes. The predicate uses arg2 which is the second argument value of the write system call, and contains the length to be written.

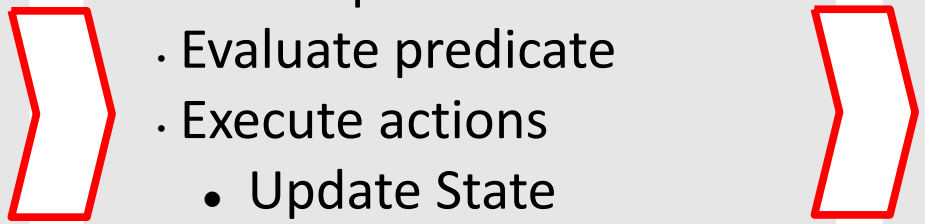
```
ssize_t write(int filedes, const void *buf, size_t nbytes);

# dtrace -n 'syscall::write:entry /arg2 > 100/ {trace(execname);}'
```

- To find out how much was actually written, use the syscall::write:return probe and consult arg0

What Happens When A Probe Fires

Kernel	DTrace core	Consumer
Probe is NOP if disabled Probe reached: · Direct call · Trap Timer-based: · Direct call	Process probe: · Evaluate predicate · Execute actions • Update State • Push data on buffer	Loop: · Retrieve buffer · Process events · Show output



Supported Providers

- Currently supported on Oracle Linux
  - dtrace: DTrace start/end, error handling
  - sdt: static probes
  - profile: profiling timers
  - fbt: function boundary tracing
  - io: disk IO
  - syscall: system calls
  - proc: process life cycle, signals processing
  - sched: CPU scheduling
  - ip: IP processing
  - perf: static probes for each perf tracepoint
  - lockstat: kernel synchronization
  - udp, tcp, netdev

- Also available on Solaris
  - plockstat: user-level synchronization
  - sysinfo: core kernel events
  - vminfo: VM-related events
  - pid: user process function boundary and instructions tracing
  - cpc: fires probes when hardware counters are incremented
  - mib: fire probes wn MIB counters (used by SNMP) are incremented
  - fpuinfo: invoked when floating ops are simulated
  - fsinfo: file system operations
  - iSCSI: iSCSI target
  - nfsv{2,3,4}: NFS server
  - smb: SMB server
  - udp, tcp
  - user-space: ssh, python, Xserver

Loading Provider Kernel Modules

- The provider kernel modules must be loaded with “modprobe <module>”
  - syscall: load module “systrace”
  - fbt: load module “fbt”
  - profile: load module “profile”
  - other providers: load module “sdt”
  - dtrace provider: loaded automatically when loading providers above
- Good news: most are done automatically. To load DTrace providers modules by default, list them in /etc/dtrace-modules
  - Default automatically loaded: sdt, systrace, profile, fasttrap, and syscall

Invoking DTrace

- DTrace probes can be enabled directly with the dtrace -n option

```
# dtrace -n 'syscall::exec*:'
dtrace: description 'syscall::exec*:' matched 4 probes
CPU     ID                    FUNCTION:NAME
  0   5192                     execve:entry
  0   5193                    execve:return
...
```

- When a probe is defined without an action, the default action is to print the probe ID, function, name, and CPU which triggered the probe.
- Default output can be removed with -q option
- Can also enable probes for a specific module (-m) or provider (-P)

DTrace Scripts

- **Example: print every program executed**
- Probes can be stored in scripts written in D language, which can be started with dtrace -s
- The BEGIN probe is part of the dtrace provider, and fires when the script starts. The dtrace provider also implements an END probe (fires when dtrace terminates) and an ERROR probe (fires when a run-time error occurs)

```
# cat simple.d
BEGIN
{
        trace("started");
}


syscall::exec*:
{
        trace(execname);
}

# dtrace -s simple.d
dtrace: script 'simple.d' matched 5 probes
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN  started
  0   5192                 execve:entry  ksmtuned
  0   5193                execve:return  awk
  0   5192                 execve:entry  ksmtuned
  0   5193               execve:return   pgrep
...
```

Program Agenda

1. Introduction
2. **D Language**
3. DTrace for User Space
4. DTrace on Oracle Linux
5. Conclusion

Variables

- Variables can be used in DTrace actions (different from variables of the code being traced)
- Automatically instantiated on first assignment
- Multiple scopes: global, thread-local, clause-local
- Scalars, associative arrays, aggregations
- C-like types
- Support for structures, unions, pointers
- Set of predefined built-ins variables
- Can access kernel variables (`symbol or <module-name>`<symbol>), including structures (e.g. `current->nr_cpus_allowed)

Variables: Global-scope Scalars

- The BEGIN and END probes are part of the DTrace provider, fired at the beginning and end of execution of the script
- The tick-1s is part of the profile provider and fires every second
- The tick-<n> probes fire on one CPU, while profile-<n> probes fire on every CPU
- Recall: blank fields to the left in a probe name can be left out, i.e. tick-1s is equivalent to :::tick-1s, which corresponds to profile:::tick-1s
- dtrace -q option sets quiet mode (suppresses # of probes matched, columns headings, names of probes fired, etc)

```
BEGIN
{
        ticks = 0;
}

tick-1s
{
        ticks++;
        printf("ping\n");
}

END
{
        printf("%d ticks\n", ticks);
}
```

```
# dtrace -q -s global-tick.d
ping
ping
ping
^C
ping
4 ticks
```

Thread-local Variables

- **Example: for every read system call, print the pid and program name of caller, and time taken by system call**
- self-> variables are local to the thread from which probes are fired.
- Preferable to globals and associative arrays (more efficient and avoid possibility of concurrent access by multiple CPUs)
- Setting a self variable to zero reclaims its storage
- The predicate in this example is used to avoid tracing the return probe without the entry probe, for system calls in progress when DTrace is started

```
syscall::read:entry
{
    self->start = timestamp;
}

syscall::read:return
/self->start != 0/
{
    printf("%d/%s spent %d nsecs in read()\n",
        pid, execname,
        timestamp - self->start);

    self->start = 0;
}
```

```
# dtrace -q -s self.d
1168/master spent 532 nsecs in read()
4835/sshd spent 7164 nsecs in read()
4838/tcsh spent 1383 nsecs in read()
4838/tcsh spent 383 nsecs in read()
4835/sshd spent 677 nsecs in read()
```

Thread-local Variables

- **Example: for every unlink() system call, print the removed file and executable name/pid of calling process.**
- copyinstr() copies-in the specified string for access by probe action, which runs in kernel space
- cleanpath() cleans path, removes “/./”, “/../”, etc
- There is a risk that the file name argument (arg1) is not yet paged in when system call is entered, so save address in self->filename, and print path on return

```
syscall::unlinkat:entry
{
    self->filename = arg1;
}

syscall::unlinkat:return
{
    printf("%s %s[%d] \n",
        cleanpath(copyinstr(self->filename)),
        execname, pid);

    self->filename = 0;
}
```

```
# dtrace -q -s unlink.d
include/generated/uapi/linux/version.h.tmp rm[24711]
.24713.tmp rm[24719]
.24713.o rm[24719]
.24727.tmp rm[24733]
.24727.o rm[24733]
.24743.tmp rm[24749]
.24743.o rm[24749]
```

Clause-local Variables

- **this->** variables are local to the clauses started by the same probe.
- More efficient than global or thread-local variables
- Typically used as temporary variable
- Note that more than one action can fire for the same probe, with potentially different predicates

```
BEGIN
{
        ticks = 0;
}

profile:::tick-1s
{
        ticks++;
        this->local_tick = ticks * 10;
        printf("ping %d\n", this->local_tick);
}

profile:::tick-1s
{
    this->local_tick++;
    printf("pong %d\n", this->local_tick);
}

# dtrace -q -s this.d
ping 10
pong 11
ping 20
pong 21
^C
ping 30
pong 31
```

Built-in Variables

- Note: The following is a subset, see DTrace User Guide for full list
- args[] - Typed probe arguments
- arg0-arg9 (uint64_t) - First ten probe arguments
- cpu (int) and curcpu (cpuinfo_t *) - current CPU id and info
- curthread – address of current task (struct task_struct *)
- tid – task id of current thread
- uid – user id of current process
- execname – name of current process
- probefunc/probename/probemod/probeprov – info about current probe
- timestamp/walltimestamp/vtimestamp - timestamps

Associative Arrays

- **Example: measure time taken by read() and write() calls**
- Collection of data indexed by one or more key values
- Keys can be of any type
- As shown by example, arguments can be specified when invoking dtrace and accessed from the script with $1, $2, etc.
- Here, $1 contains the pid of the process to be observed
- /ts[probefunc] != 0/ ensures we ignore system calls that were in progress when the script was started

```
syscall::read:entry
syscall::write:entry
/pid == $1/
{
  self->ts[probefunc] = timestamp;
}

syscall::read:return,
syscall::write:return
/pid == $1 && self->ts[probefunc] != 0/
{
  printf("%8s %8d nsecs\n", probefunc,
    timestamp - self->ts[probefunc]);
}

# dtrace -q -s array.d `pgrep -n sshd`
    read    44327 nsecs
   write    35933 nsecs
    read     1782 nsecs
   write    62185 nsecs
    read     3897 nsecs
   write    36391 nsecs
    read     4407 nsecs
   write    32299 nsecs
^C
```

Aggregations

- **Distribution of time taken by read() calls**
- Data collected into aggregation (@ notation). Apply function on that data.
- Aggregations printed by default when DTrace exists, or can be formatted with printa()
- quantize() aggregation function provides a distribution of the data (power of 2 scale)
  - buckets values in the “value” column
  - count per bucket in the “count” column
- lquantize() for linear scale
- Other aggregation functions: min(), max(), count(), sum(), avg(), stddev()

```
syscall::read:entry
/pid == $1/
{
        self->start = timestamp;
}

syscall::read:return
/pid == $1 && self->start != 0/
{
        this->time = timestamp - self->start;
        @readsyscall = quantize(this->time);
}

# dtrace -s aggr.d `pgrep -n ssh`
dtrace: script 'aggr.d' matched 2 probes
^C

     value  ------------- Distribution ------------- count
       128 |                                         0
       256 |@@                                       3
       512 |@@@@@@@@@@@@                             19
      1024 |@@@@@@@@@@                               15
      2048 |                                         0
      4096 |@@@@@@@@@@@@@@@@@@@@                     31
      8192 |                                         0
```

Stack traces

- **Code paths sending an IP packet**
- The stack() action can be used to retrieve a kernel stack trace from the current probe.

```
# dtrace -n 'ip:::send {stack();}'
dtrace: description 'ip:::send ' matched 8 probes
CPU     ID                    FUNCTION:NAME
  0   6984             __ip_local_out_sk:send
              sdt`sdt_invop+0xb0
              vmlinux`dtrace_die_notifier+0x9f
              vmlinux`notifier_call_chain+0x4e
              vmlinux`notify_die+0x45
              vmlinux`do_error_trap+0x67
              vmlinux`do_invalid_op+0x20
              vmlinux`invalid_op+0x1e
              vmlinux`ip_local_out_sk+0x1b
              vmlinux`ip_queue_xmit+0x14e
              vmlinux`tcp_transmit_skb+0x4c8
              vmlinux`tcp_write_xmit+0x115
              vmlinux`__tcp_push_pending_frames+0x32
              vmlinux`tcp_push+0xef
              vmlinux`tcp_sendmsg+0xd0
              vmlinux`inet_sendmsg+0x64
              vmlinux`sock_sendmsg+0x3d
              vmlinux`sock_write_iter+0x85
              vmlinux`__vfs_write+0xfb
              vmlinux`vfs_write+0xa9
```

Stack Traces as Keys and Aggregations

- Distribution of stack traces sending an IP packet.
- Combine aggregations and associative arrays, using stack trace as key

```
# dtrace -n 'ip:::send {@[stack()] = count(); }'
dtrace: description 'ip:::send ' matched 8 probes
^C
…
              sdt`sdt_invop+0xb0
              vmlinux`dtrace_die_notifier+0x9f
              vmlinux`notifier_call_chain+0x4e
              vmlinux`notify_die+0x45
              vmlinux`do_error_trap+0x67
              vmlinux`do_invalid_op+0x20
              vmlinux`invalid_op+0x1e
              vmlinux`ip_local_out_sk+0x1b
              vmlinux`ip_send_skb+0x1f
              vmlinux`udp_send_skb+0x173
              vmlinux`udp_sendmsg+0x2d3
              vmlinux`inet_sendmsg+0x64
              vmlinux`sock_sendmsg+0x3d
              vmlinux`SYSC_sendto+0x102
              vmlinux`SyS_sendto+0xe
              vmlinux`system_call_fastpath+0x12
               11
```

```
sdt`sdt_invop+0xb0
vmlinux`dtrace_die_notifier+0x9f
vmlinux`notifier_call_chain+0x4e
vmlinux`notify_die+0x45
vmlinux`do_error_trap+0x67
vmlinux`do_invalid_op+0x20
vmlinux`invalid_op+0x1e
vmlinux`ip_local_out_sk+0x1b
vmlinux`ip_queue_xmit+0x14e
vmlinux`tcp_transmit_skb+0x4c8
vmlinux`tcp_write_xmit+0x115
vmlinux`__tcp_push_pending_frames+0x32
vmlinux`tcp_push+0xef
vmlinux`tcp_sendmsg+0xd0
vmlinux`inet_sendmsg+0x64
vmlinux`sock_sendmsg+0x3d
vmlinux`sock_write_iter+0x85
vmlinux`__vfs_write+0xfb
vmlinux`vfs_write+0xa9
 20
```

Speculative Tracing

- **For all open system call failures, list the process, file, and error code**
- Tentatively trace data and later decide to commit or discard it
- Everything in action after speculate() is traced to speculative buffer

```
# dtrace -q -s spec.d
systemd-udevd: /run/udev/data/+power_supply:AC (2)
systemd-udevd: /run/udev/data/+acpi:ACPI0003:00 (2)
upowerd: /sys/devices/LNXSYSTM:00/.../voltage_max_design (2)
upowerd: /sys/devices/.../power_supply/BAT0/temp (2)
```

```
syscall::open:entry
{
    self->spec = speculation();
    self->filename = arg0;
    speculate(self->spec);
    printf("%s: ", execname);
}

syscall::open:return
/self->spec/
{
    speculate(self->spec);
    printf("%s (%d)\n",
        copyinstr(self->filename),
        errno);
    self->filename = 0;
}

syscall::open:return
/self->spec && errno != 0/
{
    commit(self->spec);
    self->spec = 0;
}

syscall::open:return
/self->spec && errno == 0/
{
    discard(self->spec);
    self->spec = 0;
}
```

Program Agenda

1. Introduction
2. D Language
3. **DTrace for User Space**
4. DTrace on Oracle Linux
5. Conclusion

DTrace for User-space - Overview

- syscall provider: entry and return calls for every system call
- ustack(): process stack when probe fired
- User-space static probes
- [Future] pid provider
  - trace user-space function entry and return for any process
  - does not require restarting the process to be observed
- Helper actions
  - uaddr()/usym(): symbol corresponding to a user address

ustack()

- ustack() returns the top user-space frames at the time the probe was triggered.
- The number of frames can be specified optionally.
- In this example, ustack() is combined with the profile provider to find the functions most executed and what called them.
- The profile probe arg1 argument contains the program counter (PC) of user process when the probe fired, or zero if executing in the kernel, in which case arg0 will contain the PC in the kernel

```
profile-1001
/arg1/
{
        @[execname, ustack(4)] = count();
}


# while true; do echo "" > /dev/null; done

# dtrace -s ustack.d
dtrace: script 'ustack.d' matched 1 probe
^C
...
  bash
              libc.so.6`close+0x10
              bash`0x4666b0
              bash`sh_chkwrite+0x1b
              bash`echo_builtin
               12
  bash
              libc.so.6`write+0x10
              libc.so.6`_IO_file_write+0x43
              libc.so.6`_IO_do_write+0x7c
              libc.so.6`_IO_file_overflow+0xf3
               17
  bash
              libc.so.6`mbrtoc32+0x9e
              bash`xmalloc+0x1b
              bash`mbschr+0x9e
              bash`0x451e40
               19
```

Program Agenda

1. Introduction
2. D Language
3. DTrace for User Space
4. **DTrace on Oracle Linux**
5. Conclusion

Installing DTrace

- Starting with Oracle Linux 6 or 7 running UEK4 or later
- Configure machine to have access to ULN (Unbreakable Linux Network)
- Do a “# yum update”
- Install DTrace user package: “# yum install dtrace-utils”
- Run “# dtrace -l” (as root) and you’ll see the probes. No reboot needed
- Load provider modules if not in default set

Program Agenda

1. Introduction
2. D Language
3. DTrace for User Space
4. DTrace on Oracle Linux
5. **Conclusion**

Conclusion

- DTrace is a powerful tool on Oracle Linux, and is continuously being enhanced.
- Recommendation: use DTrace for introspection when “stock” tools like vmstat, perf, and iostat don’t provide necessary information.
- Recommendation: try it out! It works whether on bare metal or in a VM.

Resources

- Oracle Linux DTrace Tutorial: http://docs.oracle.com/cd/E52668_01/E50705/html/index.html
- Oracle Linux DTrace Guide: https://docs.oracle.com/cd/E52668_01/E38608/html/index.html