Solaris 10 Workshop
DTrace

Bob Netherton
Solaris Adoption, US Client Solutions
Sun Microsystems

You gain proficiency with Dtrace as you would a musical instrument.

Practice, practice, and more practice

Agenda

- Safe Dynamic Instrumentation
- DTrace concepts
  - > Probes
  - > Providers
  - > Actions (and the D language)
  - > Scripts
- General Approaches to using DTrace
- Common Scenarios

Learning Goals

- Understand the motivation for DTrace
- Learn the basic Dtrace concepts and terminology
- Apply these concepts by writing your first DTrace scripts through simple exercises
- Be able to thoroughly digest the Solaris Dynamic Tracing Guide

Why Dynamic Tracing?

- Well-defined techniques exist for debugging *fatal, non-reproducible* failures:
  - > Obtain core file or crash dump
  - > Debug problem *postmortem* using mdb(1), dbx(1)
- Debugging transient failures is more difficult.
  - > Typical techniques push traditional tools (e.g. truss (1), mdb(1)) beyond their design centers
  - > Many transient problems cannot be debugged at all using existing techniques
  - > Where to go after truss ?

Debugging transient failure

- Historically, we have debugged transient failures using process-centric tools:
  - truss(1), pstack(1), prstat(1), etc.
- These tools were not designed to debug systemic problems
- But the tools designed for systemic problems, (i.e., mdb(1)) are designed for postmortem analysis...

Postmortem techniques

- One technique is to use postmortem analysis to debug transient problems by inducing fatal failure during period of transient failure.
- Better than nothing, but not by much:
  - > Requires inducing fatal failure, which nearly always results in more downtime than the transient failure.
  - > Requires a keen intuition to be able to identify a dynamic problem from a static snapshot of state

Invasive techniques

- If existing tools cannot root-cause transient failure, more invasive techniques must be used
- Typically, custom instrumentation is developed for the failing program and/or the kernel (i.e. LD_PRELOAD)
- The customer reproduces the problem using the instrumented binaries

Invasive techniques, continued

- Requires either:
  - > running instrumented binaries in production

  or

  - > reproducing a transient problem in a development environment
- Neither of these is desirable!
- Invasive techniques are slow, error prone, and often ineffective. We must develop a better way!

Dynamic Instrumentation

- Static probes (such as TNF) are designed to collect a specific set of data
  - The questions are asked in advance of the problem
  - Extending the data collection would require a rebuild of the system
- Want to be able to dynamically modify the system to record arbitrary data.
- Must be able to do this on a production system.
- Must be completely safe - there should be no way to induce fatal failure

Introducing DTrace

- Solaris Dynamic Tracing (DTrace) framework introduced in Build 43 of Solaris 10 (Software Express).
- A typical system has more than 30,000 probes.
- Dynamically interpreted language allows for arbitrary actions and predicates.
- Can instrument at both user- and kernel-level.
- Powerful data management primitives eliminate need for most post-processing.
- Unwanted data is pruned as close to the source as possible.

This workshop

- An introduction to the basics of DTrace, with exercises.
- Not designed to be a comprehensive tutorial or to act as reference material.
- After this introduction, you should feel comfortable diving straight into the Solaris Dynamic Tracing Guide.

DTrace Privileges

- dtrace_proc
  > Permits the use of pid and fasttrap providers
  > No visibility into kernel or processes they don't own
  > proc_owner allows probes in all processes
- dtrace_user
  > Permits the use of profile and syscall providers
  > No visibility into kernel or processes they don't own
- dtrace_kernel
  > Superset of dtrace_user
  > Full visibility to kernel and all processes
  > Cannot issue destructive commands (panic, etc)
- dtrace_proc, dtrace_user, dtrace_kernel and [e]uid 0
  > Can issue destructive operations

Create a DTrace role

- Create dtrace role in /etc/passwd

  dtrace:x:6000:10:DTrace User:/export/home/dtrace:/bin/pfksh

- Configure role properties for /usr/sbin/dtrace
  - /etc/security/exec_attr
    - dtrace:suser:cmd:::/usr/sbin/dtrace:uid=0
  - /etc/security/prof_attr
    - dtrace:::DTrace entry:
  - /etc/user_attr
    - 1. To log in directly
      - dtrace::::type=normal;profiles=dtrace
    - 2. To assume dtrace role
      - dtrace::::type=role;profiles=dtrace
      - bobn::::type=normal;roles=dtrace

DTrace Probes

Probes

- A probe is a point of instrumentation.
- A probe:
  - > Is made available by a provider.
  - > Identifies the *module* and *function* that it instruments.
  - > Has a *name*.
  - > Is assigned a integer identifier.
- A probe is uniquely identified by its provider:module:function:name

Providers

- A *provider* represents a methodology for instrumenting the system.
- Providers make probes available to the DTrace framework.
- DTrace informs providers when a probe is to be enabled.
- Providers transfer control to DTrace when an enabled probe is hit (fired).

Providers, continued

- The function boundary tracing (FBT) provider instruments every function entry and return in the kernel.
- The syscall provider can dynamically instrument the system call table
- The lockstat provider can dynamically instrument the kernel synchronization primitives (lockstat)
- The profile provider dynamically interrupts the system at a user-configurable rate

Providers, continued

- The vminfo provider can dynamically instrument the kernel “vm” statistics (vmstat)
- The sysinfo provider can dynamically instrument the kernel “sys” statistics (mpstat)
- The pid provider can dynamically instrument application code
  - Function entry and return
  - Instruction by instruction
- The io provider can dynamically instrument I/O events
- And more!

Listing probes

- Probes can be listed with the “-l” option to dtrace (1M)
- Can list probes
  > from a specific provider with “-P provider”
  > in a specific module with “-m module”
  > in a specific function with “-f function”
  > with a specific name with “-n name”
- A probe is defined as follows: provider:module:function:name

Lab #1: Listing probes

- Use dtrace(1M) to list all available probes
  - How many probes are available?
  - What are the different providers?
  - Which provider provides the greatest number of probes (extra credit for doing it as a one-liner) ?
- List probes:
  - in the “read” function
  - in the “ufs” module
  - with the “xcalls” name
  - List probes from the “sysinfo” provider

Lab #1: Listing probes

- Use dtrace(1M) to list all available probes
  - > How many probes are available?

    # dtrace -l | grep -v PROVIDER | wc -l

    40430

  - > What are the different providers

    # dtrace -l | awk '{print $2}' | grep -v PROVIDER | sort | uniq

    dtrace fasttrap fbt io lockstat mib

    proc profile sched sdt syscall sysinfo

    vminfo

Lab #1: Listing probes

- List probes:
  - > in the “read” function
    dtrace -l -f read
  - > in the “ufs” module.
    dtrace -l -m ufs
  - > with the “xcalls” name
    dtrace -l -n xcalls
  - > List probes from the “sysinfo” provider
    dtrace -l -P sysinfo

Fully specifying probes

- To specify multiple components of a probe, separate the components with a colon.
- Empty components match anything.
- For example, “syscall::open:entry”   specifies a probe:
  - > from the “syscall” provider.
  - > in any module.
  - > In the “open” function.
  - > named “entry”.

Enabling probes

- Enabled a probe by specifying it without the "-l" option.
- When enabled in this way, probes are enabled with the *default* action.
  - > The default action will print the CPU, probe number, and probe name. No other action will occur.
- For example, "dtrace -m nfs" enables every probe in the "nfs" module

Lab #2: enabling probes

- Enable probes in the “random” module.
- Enable probes provided by the “syscall” provider.
- Enable probes named “zfod”.
- Enable probes provided by the “syscall” provider in the “open” function.
- Enable the entry probe in the “clock” function.

Lab #2: enabling probes

- Enable probes in the “random” module.
  dtrace -m random

- Enable probes provided by the “syscall” provider.
  dtrace -P syscall

- Enable probes named “zfod”.
  dtrace -n zfod

Lab #2: enabling probes

- Enable probes provided by the “syscall” provider in the “open” function.

  dtrace -n 'syscall::open:'

- Enable the entry probe in the “clock” function.

  dtrace -n '::clock:entry

Actions

- *Actions* are taken when a probe fires.
- Actions are completely programmable.
- Most actions *record* some specified state in the system.
- Some actions *change* the state of the system system in a well-defined way.
  - These are called destructive actions.
  - Destructive actions are disabled by default.

The D language

- D is a C-like language specific to DTrace, with some constructs similar to awk(1)
- Complete access to kernel C types, complete support for ANSI-C operators
- Global, thread local and clause local variables
- External variables (such as kernel symbols)
- Rich set of built-in variables
- Arrays, associative arrays, and aggregations
- Support for strings as first-class citizen

Built-in D variables

- Available to all probes
- Example of built-in variables:

  pid is the current process ID.

  tid is the current thread ID.

  execname is the current executable name.

  timestamp is the time since boot, in nanoseconds.

  arg[], arg[1], arg[n] are arguments passed to the probe (varies by provider)

  probeprov, probemod, probefunc and probename are the current probe's provider, module, function, and name.

- See Table 3-1 in the Dynamic Tracing Guide for a complete list

Actions: “*trace()*”

- trace() records the result of a “D” expression to the trace buffer.

- Examples:

  trace(pid) traces the current process ID.

  trace(execname) traces the name of the current executable.

  trace(probefunc) traces the function name of the probe.

Actions: continued

- Actions are indicated by following a probe specification with { action }
- Example:

```
dtrace -n 'readch{trace(pid)}'
dtrace -m 'ufs{trace(execname)}'
dtrace -n 'syscall:::entry {trace(probefunc)}'
```

- Multiple actions are separated by semicolons

```
dtrace -n 'xcalls{trace(pid); trace(execname)}'
```

Lab #3: Actions

- Trace the name of every executable that calls the poll (pollsys in Solaris 10) system call.
- Trace the PID in every entry to the pagefault function.
- Trace the timestamp in every entry to the clock function.

Lab #3: Actions

- Trace the executable name in every poll(2) system call.

```
dtrace -n 'syscall::pollsys: {trace(execname)}'
```

- Trace the PID in every entry to the pagefault function.

```
dtrace -n '::pagefault:entry {trace(pid)}'
```

Lab #3: Actions

- Trace the timestamp in every entry to the clock function.

```
dtrace -n '::clock:entry {trace (timestamp}'
```

D scripts

- Complicated DTrace enablings become difficult to manage on the command line.
- dtrace -s will read commands from a script rather than stdin
- Alternatively, executable DTrace interpreter files may be created.
- Interpreter files always begin with:

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

D scripts, continued

- Basic structure of a D script:

```
probe description (provider:module:function:name)
/ predicate /
{
        action statements
}
```

- The following script will trace the executable name upon entry into any system call:

```
#!/usr/sbin/dtrace -s
syscall:::entry
{
        trace(execname);
}
```

Predicates (D conditionals)

- Predicates allow actions to only be taken when certain conditions are met.
- A predicate is a D expression.
- Actions will only be taken if the predicate expression evaluates to true.
- A predicate takes the form “/expression/” and is placed between the probe description and the action.

Predicates, continued

- Example: Trace the pid of every process named "date" that performs an open(2):

```
#!/usr/sbin/dtrace -s
syscall::open:entry
/execname == "date"/
{
        trace(pid);
}
```

Lab #4: Predicates

- Trace the timestamp in every ioctl(2) from processes named dtrace.
- Use the *arg0* variable to trace the executable name of every process read(2)'ing from file descriptor 0.

  ssize_t read(int fildes, void *buf, size_t nbyte);

- Use the *arg2* variable to trace the executable name of every processing write(2)'ing more than 100 bytes
  - ssize_t write(int fildes, const void *buf, size_t nbyte);

Lab #4: Predicates

- Trace the timestamp in every ioctl(2) from processes named dtrace.

```
dtrace -n 'syscall::ioctl:entry     \
    /execname=="dtrace"/ {trace(timestamp)}'
```

- Use the *arg0* variable to trace the executable name of every process read(2)'ing from file descriptor 0.

```
dtrace -n 'syscall::read:entry /arg0==0/   \
        {trace(execname)}'
```

Lab #4: Predicates

- Use the *arg2* variable to trace the executable name of every processing write(2)'ing more than 100 bytes

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

Note: arg[2] represents the requested write size. To find out how much was written, use the write:return probe and consult arg[0] or arg[1].

Actions: more Actions

- *tracemem()* records memory at a specified location for a specified length.
- *stack()* records the current kernel stack trace.
- *ustack()* records the current user stack trace.
  - Java stack displayed when process is a 1.5+ jvm.
- *exit()* tells the DTrace consumer to exit with the specified status.

Actions: Destructive actions

- Must specify "-w" option to DTrace.

  stop() stops the current process

  Use prun(1) to resume the stopped process

  raise() sends a specified signal to the current process

  breakpoint() triggers a kernel breakpoint and transfers control to the kernel debugger (kdb)

  panic() induces a kernel panic

  chill() spins for a specified number of nanoseconds

Lab #5: Actions

- Use dtrace(1M) to record a kernel stack in the "zfod" probe.
- Modify the above to record a user-stack.
- Write a D script to stop any process named "xcalc" that performs an ioctl(2).
- Modify the above to record the process ID of the process being stopped.

Lab #5: Actions

- Use dtrace(1M) to record a kernel stack in the "zfod" probe.

```
#!/usr/sbin/dtrace -s
:::zfod
{
    stack()
}
```

Lab #5: Actions

- Modify the above to record a user-stack.

```
#!/usr/sbin/dtrace -s
:::zfod
{
    ustack()
}
```

Lab #5: Actions

- Write a D script to stop any process named "xcalc" that performs an ioctl(2).

```
#!/usr/sbin/dtrace -ws
syscall::ioctl:entry
/execname == "xcalc"/
{
    stop();
    exit(0);
}
```

- To resume xcalc: prun `pgrep xcalc`

Lab #5: Actions

- Modify the above to record the process ID of the process being stopped.

```
#!/usr/sbin/dtrace -ws
syscall::ioctl:entry
/execname == "xcalc"/
{
stop();
trace(pid);
exit(0);
}
```

Output formatting

- The *printf()* function combines the *trace* action with the ability to precisely control output.
- *printf* takes a printf(3C)-like format string as an argument, followed by corresponding arguments to print.
- Examples:

```
printf("%d was here\n", pid);
printf("I am %s\n", execname);
```

Output formatting, continued

- Normally, dtrace(1M) provides details on the firing probe, plus any explicitly traced data.
- Use the quiet option ("-q") to dtrace(1M) to suppress the probe details.
- The quiet option may also be set in a D script by embedding:

```
#pragma D option quiet
```

Global D variables

- D allows you to define your own variables that are global to your D program.
- Like awk(1), D tries to infer variable type upon instantiation, obviating an explicit variable declaration.

Global D variables, continued

- Example:

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

sysinfo:::zfod
{
        zfods++;
}

profile:::tick-1sec
{
        printf(“%d zfods\n”, zfods);
        zfods = 0;
}
```

Thread-local D variables

- D allows for *thread-local* variables.
- A *thread-local* variable has the same name - but separate data storage for each thread.
- *Thread-local* variables prevent race conditions associated with global variables.
- *self-> variable* denotes a thread-local variable.
- Thread-local variables that have never been assigned in the current thread have the value zero.
- Underlying storage for a thread-local variable is deallocated when assigned a zero value.

Thread-local D variables, cont.

- Example 1:

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

syscall::pollsys:entry
{
        self->ts = timestamp;
}
syscall::pollsys:return
/self->ts && (timestamp - self->ts) > 1000000000/
{
        printf("%s polled for %d seconds\n",  execname,(timestamp - self->ts) / 1000000000);
        self->ts = 0;
}
```

Thread-local D variables, cont.

- Example 2:

```
syscall::ioctl:entry
/execname ==  “date” /
{
        self->follow = 1;
}
fbt:::
/self->follow/
{}
syscall::ioctl:return
/self->follow/
{
        self->follow = 0;
}
```

Lab #6: D variables

- Write a D script to trace the executable name and amount of time spent in every open(2).
- Write a D script to follow a brk(2) system call through the kernel when called from the date(1) command.
- Add "*#pragma D option flowindent*" to the above and observe the change in output.

Lab #6: D variables

- Write a D script to trace the executable name and amount of time spent in every open(2).

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

syscall::open:entry
{
    self->ts = timestamp;
}
syscall::open:return
/self->ts/
{
    printf("open took %d nanosecs\n",timestamp - self->ts);
    self->ts = 0;
}
```

Lab #6: D variables

- Write a D script to follow a brk(2) system call through the kernel when called by a date(1) command.

```
#!/usr/sbin/dtrace -s
syscall::brk:entry
/execname == "date"/
{   self->follow = 1;  }

fbt:::
/self->follow/
{}

syscall::brk:return
/self->follow/
{ self->follow = 0; }
```

Lab #6: D variables

- Add "#pragma D option flowindent" to the above and observe the change in output.

```
#!/usr/sbin/dtrace -s
#pragma D option flowindent
syscall::brk:entry
/execname == "date"/
{   self->follow = 1;  }

fbt:::
/self->follow/
{}

syscall::brk:return
/self->follow/
{ self->follow = 0; }
```

Aggregations

- Often a pattern in the data values is more useful than individual values themselves.
- Aggregation functions allow the observation of the trends and patterns in data values.
- Traditionally, one has had to use conventional tools (e.g. awk(1), perl(1)) and possibly create tables do be displayed by spreadsheet programs.
- DTrace provides a rich set of aggregation functions.

Aggregations, continued

- DTrace supports the aggregation of data as a first class operation.
- An aggregating function is a function f(x), where x is a set of data, such that:

  $$f(f(x0) \cup f(x1) \cup \ldots \cup f(xn)) = f(x0 \cup x1 \cup \ldots \cup xn)$$

  In other words, you can partition the data and still yield the same results.
- e.g. COUNT, SUM, MAXIMUM, and MINIMUM are aggregating functions; MEDIAN and MODE are not

Aggregations, continued

- Aggregations are stored in aggregation arrays
  > Similar to associative arrays
  > Denoted by @
  > Can be anonymous (ie, @[probefunc] = count())
- Example: Count all system calls organized by system call name:

```
dtrace -n 'syscall:::entry \
{ @syscalls[probefunc] = count(); }'
```

- By default, aggregation results are printed when dtrace(1M) exits.

Aggregations, continued

- Aggregations need not be named.
- Aggregations can be keyed by more than one expression.
- Example: Count all ioctl system calls by both executable name and associated file descriptor

```
#!/usr/sbin/dtrace -s
syscall::ioctl:entry
{
        @[execname, arg0] = count()
}
```

Aggregations, continued

- Some other aggregating functions:
  - *avg( )*: the average of specified expressions
  - *min( )*: the minimum of specified expressions
  - *max( )*: the maximum of specified expressions
  - *quantize()*: power-of-two distribution of specified expressions.
- Example: Distribution of write(2) sizes by executable name:

```
dtrace -n 'syscall::write:entry \
{ @[execname] = quantize(arg2); }'
```

Aggregations, continued

```
#   dtrace -n 'syscall::write:entry { @[execname] = quantize(arg2); }'
dtrace: description 'syscall::write:entry ' matched 1 probe
^C
gnome-terminal
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@                                     1
               2 |                                         0
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |@@@@@@@                                  2
              64 |                                         0
             128 |                                         0
             256 |@@@@@@@@@@@@@@@@@@                       5
             512 |                                         0
            1024 |@@@@@@@                                  2
            2048 |@@@@                                     1
            4096 |                                         0
```

Lab #7: Aggregations

- Count the number of system calls by executable name.
- Count the number of write system calls by process ID.
- Get a distribution of read(2) sizes by executable name and file descriptor.
- Get a distribution of time spent in read(2) by executable name and file descriptor.

Lab #7: Aggregations

- Count the number of system calls by executable name.

```
#!/usr/sbin/dtrace -s
syscall:::entry
{    @[execname] = count() }
```

- Count the number of write system calls by process ID.

```
#!/usr/sbin/dtrace -s
syscall::write:entry
{   @[pid] = count() }
```

Lab #7: Aggregations

- Get a distribution of read(2) sizes by executable name and file descriptor.

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
syscall::read:entry
{     self->follow=1;     self->filedes=arg0;   }

syscall::read:return
/self->follow && ((int)arg1 != -1)/
{      @[execname,self->filedes] = quantize(arg1);
       self->follow=0;    }
```

(NOTE: predicate on return probe needed because error return of -1 is treated as large positive number)

Lab #7: Aggregations

- Get a distribution of time spent in read(2) by executable name and file descriptor

```
#!/usr/sbin/dtrace -s
syscall::read:entry
{
        self->ts = timestamp; self-> filedes = arg0;
}
syscall::read:return
/self->ts/
{
        @[execname, self-> filedes] = quantize(timestamp - self->ts);
        self->ts = 0;
}
```

DTrace Providers

DTrace Provider

- Three probes to give awk-like controls
- BEGIN
  - first probe to fire
  - Used for initializations or to print headings
- END
  - Fires as DTrace terminates
  - Used to dump aggregation arrays under user control and formatting
- ERROR
  - Fires when a runtime error occurs in another probe

DTrace Provider

- Example: Hello World

```
#!/usr/sbin/dtrace -s
#pragma D quiet
BEGIN {
                printf("Hello World\n")
           }
END
        {
                printf("Goodbye World\n")
        }
```

Profile Provider

- Allows for a probe to be fired at specific intervals
- Not associated with any other provider
- Relatively low overhead
- Useful to infer system state by time based observations
- profile-<interval> fires on all CPUs
- tick-<interval> fires once on a single CPU
- Arguments
  - arg0 is the program counter in the kernel or 0 if in user
  - arg1 is the program counter in user or 0 if in kernel

Profile Provider

- Example: Sample the system 1,000 times per second and count the top user call frame

```
#!/usr/sbin/dtrace -s
profile-1001
/arg1/
{
    @[ustack(1) = count()
}
```

Function Boundary Trace (FBT)

- Allows instrumentation of most kernel functions
- fbt:::entry
  - Arguments to the probe are the same as the kernel function
  - Use the Open Solaris source browser to follow the code and understand the arguments
- fbt:::return
  - arg0 is the offset of the actual return (many kernel functions have multiple return points)
  - arg1 is the return value, if any.
- Tail-called optimizations may not be completely observable.

Syscall provider

- Instrumentation for the entry and return from all system calls.
- Useful to get a understanding of the interaction between an application and the kernel (system).
- syscall:::entry
  - > Arguments are taken from the system call
  - > See man section 2 for values and types
- syscall:::return
  - > arg0 and arg1 are set to the return value from the system call
  - > Global variable errno may be used to track system call errors.

PID provider

- Allows dynamic instrumentation of applications, regardless of the compiler used or compile flags
- Example, list all function entry probes in 'xclock'

```
# xclock -update 1&
[2] 778
# dtrace -l -n pid778:::entry |wc -l
   8396
```

- Example – print the arguments to the 'DrawSecond' function within 'xclock'

```
#!/usr/sbin/dtrace  -s
#pragma D option quiet
pid$1::DrawSecond:entry
{
printf("len is %d, width = %d, offset = %d, tick_units = %d\n",
arg1, arg2, arg3, arg4);
}
# ./d.d 778
len is 64, width = 3, offset = 53, tick_units = 252
...
```

IO provider

- Allows dynamic instrumentation of physical disk I/O's. Can Show how file system buffering and read-ahead/write-behind is working
- Example, show the device, program, I/O size, I/O type, and file name of all disk I/O, and a summary

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
BEGIN{printf("%-10s %10s %10s %3s %s\n","Device", "Program","I/O Size","R/W", "Path");}
io:::start {
        printf("%-10s %10s %10d %3s %s\n", args[1]->dev_statname, execname,
            args[0]->b_bcount, args[0]->b_flags & B_READ? "R" : "W" , args[2]->fi_pathname);
        @[execname, pid, args[2]->fi_pathname] = sum(args[0]->b_bcount);
}
END {   printf("%-10s %8s %10s %s\n","Program", "PID", "Total", "Path");
        printa("%-10s %8d %10@d %s\n",@);
}
Device      Program   I/O Size R/W Path
cmdk0       mkfile      8192   W /export/home/foo
cmdk0       mkfile     49152   W /export/home/foo
...
Program     PID     Total     Path
mkfile      813   10493952 /export/home/foo
```

MIB provider

- Allows dynamic instrumentation of all MIB counters such as those reported by 'netstat -s' and 'kstat'
- Example, show TCP connections per second

```
# !/usr/sbin/dtrace -s
#pragma D option quiet
mib:::tcpActiveOpens,mib:::tcpPassiveOpens
{
@[execname, probefunc, probename] = sum(arg0);
event = 1;
}
tick-1sec
/event/
{
printa("%20s %20s %20s %@d\n",@);
trunc(@);
event = 0;
}
```

- Warning – user context not available on many probes

General Approaches

- Start with existing tools
  - > `truss`
  - > `vmstat/mpstat/iostat/lockstat`
  - > `pfiles/pmap/pldd/pstack/ptree`
  - > `prstat -mL`
- Understand the new tools in Solaris 10:
  - > `intrstat`
  - > `plockstat` (DTrace consumer)
  - > `pstack` (for Java)
  - > `pfiles` (now shows file names)

Solaris Performance and Tracing Tools

Process stats

- cputrack - per-processor hw counters
- pargs – process arguments
- pflags – process flags
- pcred – process credentials
- pldd – process's library dependencies
- psig – process signal disposition
- pstack – process stack dump
- pmap – process memory map
- pfiles – open files and names
- prstat – process statistics
- ptree – process tree
- ptime – process microstate times
- pwdx – process working directory

Process control

- pgrep – grep for processes
- pkill – kill processes list
- pstop – stop processes
- prun – start processes
- prctl – view/set process resources
- pwait – wait for process
- preap – reap a zombie process

Process Tracing/ debugging

- abitrace – trace ABI interfaces
- dtrace – trace the world
- mdb – debug/control processes
- truss – trace functions and system calls

Kernel Tracing/ debugging

- dtrace – trace and monitor kernel
- lockstat – monitor locking statistics
- lockstat -k – profile kernel
- mdb – debug live and kernel cores

System Stats

- acctcom – process accounting
- busstat – Bus hardware counters
- cpustat – CPU hardware counters
- iostat – IO & NFS statistics
- kstat – display kernel statistics
- mpstat – processor statistics
- netstat – network statistics
- nfsstat – nfs server stats
- sar – kitchen sink utility
- vmstat – virtual memory stats

General Approaches

- You will type "dtrace" thousands of times
- Often you will hit a dead-end
- Think like Edison...
- Use the manual!
  - Install it locally!
  - And the examples in /usr/demo/dtrace

General Approaches

- Use aggregations to make sense of large amounts of data
- look for "outliers" and "holes"
- Use `quantize()/lquantize()`
  - `min()/max()/avg()` can hide important data
- User these providers
  - profile, sched, pid, proc, and io, (less of fbt)
- Be aware of probe effect on pid

General Approaches (cont.)

- A good starting place is `mpstat`
  - > User/system time ratios?
  - > context switching?
  - > lock contention?
  - > cross calls?
  - > faults?
  - > system call levels?
  - > Interrupts?

Scenario 1
High user time

Scenario: High User Time
Code: multi.c (interest rate model)

```
# mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    0   342  240  146   60    0    0    0   229  100   0   0   0
  0    0   0    0   335  233  144   60    0    0    0   181  100   0   0   0
```

- Where is this utilization coming from?
- Use the profile provider. (small probe effect)
- Examples:

```
# dtrace -n 'profile-1001{ @[arg1] = count()}'
           or
# dtrace -n 'profile-1001/pid == 1234/{ @[arg1] = count()}'
           or
# dtrace -n 'profile-1001/execname == "multi"/{ @[arg1] = count()}
           or
# dtrace -n 'profile-1001{@[execname, ustack(1)] = count()}'

# dtrace -n 'profile-1001{@[execname] = count()}'
```

Scenario: High User Time

Profile a user application: look at ustack.

```
# dtrace -n 'profile-1001/execname == “multi”/{@[ustack(1)] = count()} \
                        END{trunc(@,10)}'
^C
CPU     ID                    FUNCTION:NAME
  0      2                             :END
libm.so.2`exp+0x4
 94
libm.so.2`exp+0x54
 99
libm.so.2`exp+0x67
111
libm.so.2`exp+0x98
116
multi`intio_calc_n_month_rate+0x8f
117
multi`intio_calc_n_month_rate+0x186
138
libm.so.2`exp+0x10d
163
libm.so.2`exp+0x90
184
libm.so.2`exp+0x61
617
libm.so.2`log+0x1a
854
```

ustack(1) does not add much overhead

Scenario: High User Time

- Clearly, this application is sitting in math libraries
- Non-Production Probing:
  - > We'd like a function count, and the time spent in each function
  - > (Warning - can be a significant probe effect!!!)

```
# cat quantify.d
#!/usr/sbin/dtrace -s
pid$1:::entry
{
        self->ts[self->stack++] = timestamp;
}

pid$1:::return
/ self->ts[self->stack - 1] /
{
        this->elapsed = timestamp - self->ts[--self->stack];
        @[probefunc] = count();
        @a[probefunc] = quantize(this->elapsed);
        self->ts[self->stack] = 0;
}
```

Scenario: High User Time

```
# ./quantify.d
  intio_calc_pmms_rate                                                16
  intio_calc_adj_rate                                                 81
  intio_calc_adjustment                                               82
  intio_calc_n_month_rate                                          34655
  log                                                              39380
  exp                                                              78763
  intio_calc_adjustment

           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 82
            8192 |                                         0

  log
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 39337
            8192 |                                         19
           16384 |                                         12
           32768 |                                         9
           65536 |                                         0
          131072 |                                         2
          262144 |                                         0
          524288 |                                         1
         1048576 |                                         0
```

Scenario: High User Time

```
exp
           value  ------------- Distribution ------------- count
        >   2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 78685
            8192 |                                         37
           16384 |                                         26
           32768 |                                         9
           65536 |                                         2
          131072 |                                         3
          262144 |                                         0
          524288 |                                         1
         1048576 |                                         0

intio_calc_pmms_rate
           value  ------------- Distribution ------------- count
        16777216 |                                         0
        33554432 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 16
        67108864 |                                         0

intio_calc_adj_rate
           value  ------------- Distribution ------------- count
         2097152 |                                         0
         4194304 |@@@@@@@@@@@@@@@@                         33
         8388608 |                                         0
        16777216 |                                         0
        33554432 |@@@@@@@@@@@@@@@@@@@@@@@@                 48
        67108864 |                                         0
```

Scenario: High User Time

- Another view of the application....

```
# cat watchpid.d
#!/usr/sbin/dtrace -Fs
pid$1:::entry{}
pid$1:::return{}

# ./watchpid.d `pgrep multi`
  0    -> intio_calc_n_month_rate
  0      -> log
  0      <- log
  0      -> log
  0      <- log
  0      -> exp
  0      <- exp
  0      -> exp
  0      <- exp
  0      -> exp
  0      <- exp
  0      -> exp
  0      <- exp
  0    <- intio_calc_n_month_rate
  0    -> intio_calc_n_month_rate
  0      -> log
  0      <- log
...
```

Look for best `exp()`, `log()` , Possibly inline

Scenario: High User Time

- Other things to watch for:
  - > Calls to .mul(), .div(), etc for SPARC
    - > compiled for SPARC V7 - recompile!
  - > Excessive calls to libraries such as:
    - > getenv(3C),getrusage(3C),getrlimit(2)
    - > time(2),gettimeofday(3C)
      - – possibly replace with gethrtime()
      - – time(2) on SPARC is very expensive! (7X)
      - – time(2) on Intel is somewhat expensive! (1.5X)
      - – gettimeofday(3C) slightly more expensive than gethrtime(3C) (10%)
  - > Watch for memmove(3C) vs memcpy(3C)
    - > Use memcpy() if regions do not overlap

Scenario 2
System time being consumed

Scenario: System time being consumed
(>10%, or when usr/sys ratio approaches 2-to-1)

```
# vmstat 1
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd s0 s1 --   in   sy   cs us sy id
 0 0 0 1157600 751464 3188 24781 3937 0 0 0 0 433 0 0 0 1236 15801 1543 32 41 28
 0 0 0 1157292 752688 2738 17618 4578 0 0 0 0 309 0 0 0 995 11657 1289  29 29 43
 0 0 0 1157156 752476 4450 40268 2116 0 0 0 0 306 0 0 0 987 24698 1608  26 63 11
```

- Profile the system, see who is asking for resources:

```
# dtrace -n 'profile-1001 {@[execname,uid,pid,tid] = count()}'
  grep                          0    11239     1            2
  grep                          0    11101     1            2
  grep                          0    12155     1            2
  grep                          0    11617     1            2
  grep                          0    12457     1            3
  grep                          0    11916     1            3
  dtrace                        0    10453     1            3
  grep                          0    11506     1            4
  grep                          0    11488     1            4
  Xorg                       5667      408     1            5
  gnome-terminal             5667      660     1            7
  sched                         0        0     1           19
  find                          0      804     1          509
  sched                         0        0     0         1573
```

Scenario: System time being consumed

- Profile the kernel, see what kernel functions are hit:

```
# dtrace -n 'profile-1001 /arg0/{@[arg0] = count()}END \
          {trunc(@,10);printa("%a %@u\n", @)}'
^C
CPU     ID                    FUNCTION:NAME
  0      2                             :END unix`hwblkpagecopy+0xca 27
unix`page_numtopp_nolock+0x29 29
unix`ddi_get8+0x13 36
unix`cas64+0x1a 46
unix`fakesoftint_return+0x2 59
unix`page_exists+0x3f 64
unix`page_lookup_create+0x5a 66
unix`page_lookup_nowait+0x45 81
unix`mutex_enter+0x11 241
unix`cpu_halt+0x9d 660
```

Scenario: System time being consumed

- What are the most frequently called kernel functions?

```
# dtrace -n 'fbt:::entry {@[probefunc] = count()}END{trunc(@,10)}'
        (can be expensive)

CPU     ID                    FUNCTION:NAME
  0      2                             :END

  htable_e2va                                                     30397
  page_pptonum                                                    40784
  x86_hm_exit                                                     50531
  x86_hm_enter                                                    50531
  htable_va2entry                                                 53327
  x86pte_access_pagetable                                         53706
  x86pte_release_pagetable                                        53706
  apic_setspl                                                     54169
  psm_get_cpu_id                                                  55707
  page_next_scan_large                                            67220
```

- Demand paging related work.

Scenario 3
High System Calls

Scenario: System calls (>100 per second)

```
% vmstat 1
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd s0 s1 --   in   sy   cs us sy id
 0 0 0 1158040 759184 950 3939 1480 0 1 0 55 84 0 0  0  549 3143 577  6  9 85
 0 0 0 1102740 720324 7   42 25  0  0  0  0  5  0  0  0  403  327 244  2  0 98
 0 0 0 1102740 720324 0    0  0  0  0  0  0  0  0  0  0  370  261 202  1  1 98
```

- System calls show interface between application and OS

```
# dtrace -n 'syscall:::entry/pid != $pid/{@[execname,probefunc] = count()}'
...
  gnome-terminal      pollsys                    44
  Xorg                read                       46
  mixer_applet2       ioctl                      54
  acroread            pollsys                   423
  acroread            ioctl                     840
```

Scenario: System calls

- Aggregate on syscall

```
dtrace -n 'syscall:::entry   { @[probefunc] = count()}'
dtrace -n 'syscall:::entry   { @[probefunc,execname] = count()}'
dtrace -n 'syscall:::entry   { @[probefunc,pid] = count()}'
```

- If top syscalls are `read()/write()/poll()`
  - aggregate on `arg0` (fd #)
  - look up file descriptor using `pfiles`

Scenario: System calls

- Use `argN` to drill down into system calls.
- For example, what fd's are being used for ioctl()?:

```
# dtrace -n 'syscall::ioctl:entry/execname == "acroread"/{ @[arg0] = count()}'
          3                 352
```

- Look that up via `pfiles`:

```
# pfiles `pgrep acroread`
654:    /export/home/jdf/Acrobat4/Reader/intelsolaris/bin/acroread ../../dtrac
  Current rlimit: 256 file descriptors
   0: S_IFCHR mode:0666 dev:270,0 ino:6815752 uid:0 gid:3 rdev:13,2
      O_RDONLY
      /devices/pseudo/mm@0:null
   1: S_IFCHR mode:0666 dev:270,0 ino:6815752 uid:0 gid:3 rdev:13,2
      O_WRONLY|O_LARGEFILE
      /devices/pseudo/mm@0:null
   2: S_IFCHR mode:0666 dev:270,0 ino:6815752 uid:0 gid:3 rdev:13,2
      O_WRONLY|O_LARGEFILE
      /devices/pseudo/mm@0:null
   3: S_IFSOCK mode:0666 dev:276,0 ino:33309 uid:0 gid:0 size:0
      O_RDWR|O_NONBLOCK FD_CLOEXEC
        SOCK_STREAM
        SO_SNDBUF(16384),SO_RCVBUF(5120)
        sockname: AF_UNIX
        peername: AF_UNIX /tmp/.X11-unix/X0
```

Scenario: System calls

- Use quantize() on read()/write(). Look for 1-byte I/O's

```
# dtrace -n 'syscall::write:entry {@ = quantize(arg2)}'
   value  ------------- Distribution ------------- count
       0 |                                         0
       1 |@                                        48
       2 |                                         1
       4 |@                                        26
       8 |@                                        43
      16 |@@@                                      115
      32 |@@                                       82
      64 |                                         13
     128 |                                         7
     256 |@@                                       73
     512 |                                         5
    1024 |                                         16
    2048 |                                         7
    4096 |                                         0
    8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@           1280
   16384 |                                         0
```

```
# dtrace -n 'syscall::write:entry /arg2 == 1/ {trace(execname);ustack()}'
  0     13                      write:entry   gnome-terminal
          libc.so.1`_write+0x15
          libvte.so.4.4.0`vte_terminal_io_write+0x73
          libglib-2.0.so.0.400.1`g_source_callback_funcs
          libvte.so.4.4.0`vte_terminal_io_write
```

Scenario: System calls

- Looking at write size again

```
# dtrace -n 'syscall::write:return {@[execname, arg0] = count()}'
xscreensaver                                                    8            2
gnome-netstatus-                                              208            2
gnome-terminal                                                264            4
gnome-terminal                                                300            6
soffice.bin                                                     8           14
soffice.bin                                                     1           26
```

Scenario: System calls

- Look for system call errors:

```
# dtrace -n 'syscall:::return /errno/ {trace(execname);trace(pid);trace(errno)}'
  0     318     pollsys:return   Xorg                                    408        4
  0      12     read:return    gnome-terminal                            660       11
  0      12     read:return    Xorg                                      408       11
  0      12     read:return    dsdm                                      570       11
  0      12     read:return    nautilus                                  650       11
  0      12     read:return    Xorg                                      408       11
```

- Why not `truss`?
  > `truss` is much easier to use, provides better information
  > But watch production use!
  > truss only looks at one process
  > DTrace is better at looking for system-wide events

Scenario 4
High Context Switching

Scenario: Context Switches (>500 per cpu) Code: what.c

```
# mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0   87   261  112 16908   73  939 9144    0 27571   59  40   0   1
  1    0   0   68    70    3 12237   53  659 10460    0 27079   62  38   0   0
  2    0   0   60    66    2 15987   53  622 10071    0 29405   59  41   0   0
  3    0   0   48    93    8 21788   77  932 8587    0 30584   55  45   0   0
```

- Determine what processes are being switched off:

```
# dtrace -n 'off-cpu{@[execname] = count()}'
dtrace: description 'off-cpu' matched 2 probes
^C

  pageout                                                          1
  automountd                                                       2
  sendmail                                                         3
  fsflush                                                          4
  nscd                                                             4
  svc.startd                                                      15
  dtrace                                                          55
  sched                                                        17543
  what                                                         19442
```

Scenario: Context Switches (>500 per cpu) Code: what.c

- From `prstat` we see 'what' is threaded.

```
# prstat -L
   PID USERNAME  SIZE   RSS STATE   PRI NICE      TIME  CPU PROCESS/LWPID
 25774 tgendron 1684K 1312K run      10    0   0:00:32  76% what/34
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/16
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/15
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/14
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/12
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/11
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/10
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/9
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/8
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/6
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/5
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/4
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/3
 25774 tgendron 1684K 1312K sleep    59    0   0:00:00 0.0% what/2
```

Scenario: Context Switches (>500 per cpu)

- From prstat we see 'what' is threaded, so aggregate on tid

```
# dtrace -n 'off-cpu /execname == "what"/{@[tid] = count()}'
^C

       34                 34
       25                501
       23                517
       15                520
       13                522
       27                522
...
        7                551
       31                551
       20                552
       10                555
       26                559
       17                561
       19                564
        2                567
       22                569
       29                571
        9                571
       14                574
       11                579
       32                629
```

  - Looks like all but thread 34 are switching (he's the producer)

Scenario: Context Switches (>500 per cpu)

- Next, get a stack trace when the context switch occurs

```
# dtrace -n 'off-cpu /execname == "what"/{@[ustack()] = count()}END{trunc(@,5)}'
dtrace: description 'off-cpu ' matched 3 probes
^C
CPU     ID                    FUNCTION:NAME
  1      2                             :END
              libc.so.1`__lwp_park+0x10
              what`producer+0xb4
              libc.so.1`_lwp_start
               40
              libc.so.1`__lwp_park+0x10
              libc.so.1`lmutex_lock+0xe0
              libc.so.1`free+0x24
              what`consumer+0x6c
              libc.so.1`_lwp_start
               78
              libc.so.1`__lwp_park+0x10
              libc.so.1`cond_wait_queue+0x28
              libc.so.1`cond_wait+0x10
              what`consumer+0x44
              libc.so.1`_lwp_start
              885
              libc.so.1`__lwp_park+0x10
              what`consumer+0xa8
              libc.so.1`_lwp_start
             8158
              libc.so.1`__lwp_park+0x10
              libc.so.1`cond_wait_queue+0x84
              libc.so.1`cond_wait+0x10
              what`consumer+0x44
              libc.so.1`_lwp_start
            11948
```

Scenario: Context Switches (>500 per cpu)

- The `cond_wait()` is where we're context switching mostly. Also in a `mutex_lock()`
- Examining code, we see the producer is calling `cond_broadcast()`
- Generally a bad idea - use `cond_signal()` instead
- We also see the consumer waiting on a condition variable after each unit of work
- Also bad - why not check for more work to do before waiting?
- Made minor changes (what_new.c)

Scenario: Context Switches

- `mpstat` and `ptime` *before* change:

```
# mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0   90   0    0   266  166 1517   26    0    1    0  2579   30   3   0  67
  0    3   0    0   250  150 2190   69    0    2    0  3645   96   4   0   0
  0    0   0    0   244  145 1976   65    0    0    0  3317   96   4   0   0
  0    0   0    0   256  155 1972   69    0    0    0  3322   97   3   0   0

# ptime ./what 128
real         9.464
user         8.111
sys          0.208
```

- `mpstat` and `ptime` *after* change:

```
# mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0  376   4    0   384  285  480   41    0    0    0 23595    8   5   0  87
  0    9   0    0   332  231  767   13    0    0    0  2053    5   2   0  93
  0  481   0    0   288  188  450   10    0    1    0  1393    4   3   0  93

# ptime ./what_new 128

real         1.189
user         0.161
sys          0.016
```

Scenario 5
Threaded applications

Scenario: Threaded application

- Use prstat to see time waiting for locks:

```
# prstat -mL 1
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
 29621 root      69  31 0.0 0.0 0.0 0.3 0.0 0.3  89  21 25K   0 what/34
 29621 root     5.6 3.7 0.0 0.0 0.0  49 0.0  42  2K   6  3K   0 what/33
 29621 root     5.7 3.4 0.0 0.0 0.0  44 0.0  46  2K   8  3K   0 what/15
 29621 root     5.6 3.5 0.0 0.0 0.0  48 0.0  43  2K   6  3K   0 what/16
 29621 root     5.5 3.5 0.0 0.0 0.0  43 0.0  48  2K   1  3K   0 what/31
 29621 root     5.4 3.5 0.0 0.0 0.0  49 0.0  42  2K   9  3K   0 what/3
 29621 root     5.5 3.4 0.0 0.0 0.0  47 0.0  45  2K   9  3K   0 what/29
```

- Use Dtrace consumer plockstat:

```
# plockstat -p 29618
^C
Mutex block

Count     nsec Lock                          Caller
-------------------------------------------------------------------------------
 3051   636785 what`thelock                  what`consumer+0x44
 2017   465852 what`thelock                  what`consumer+0xa8
  689   205946 what`thelock                  what`consumer+0x44
 2243   162300 what`thelock                  what`producer+0xb4
 1506   115574 what`thelock                  what`consumer+0xa8
  429    50190 what`thelock                  what`producer+0xb4
    3    49706 libc.so.1`libc_malloc_lock    what`consumer+0x6c
    2    14600 libc.so.1`libc_malloc_lock    what`consumer+0x6c
```

Scenario: Threaded application

```
(plockstat output cont.)
Mutex spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
 7080    17731 what`thelock                 what`producer+0xb4
 8722    15680 what`thelock                 what`consumer+0x44
11613    13897 what`thelock                 what`consumer+0xa8
  406    11684 libc.so.1`libc_malloc_lock   what`producer+0x64
  335    11651 libc.so.1`libc_malloc_lock   what`consumer+0x78
  672    10091 libc.so.1`libc_malloc_lock   what`producer+0x20
 4451     8016 libc.so.1`libc_malloc_lock   what`consumer+0x6c

Mutex unsuccessful spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
 3746    30148 what`thelock                 what`consumer+0x44
 2673    27705 what`thelock                 what`producer+0xb4
 3525    27424 what`thelock                 what`consumer+0xa8
    5    27056 libc.so.1`libc_malloc_lock   what`consumer+0x6c
```

Scenario: Threaded application

- We see our hot lock (`thelock`). But also notice `libc_malloc_lock`!
- Replace with `libmtmalloc`, or `libumem`:

```
# export LD_PRELOAD=libumem.so.1
(Run the app, then:)
# plockstat -p `pgrep what`
^C
Mutex block

Count     nsec Lock                          Caller
-------------------------------------------------------------------------------
   45 38589443 what`thelock                  what`consumer+0x44
  442 23234077 what`thelock                  what`consumer+0xa8
    4   304380 what`thelock                  what`consumer+0x44
    9   240817 what`thelock                  what`producer+0xb4
    6    33386 what`thelock                  what`consumer+0xa8
   12    15773 what`thelock                  what`producer+0xb4

Mutex spin

Count     nsec Lock                          Caller
-------------------------------------------------------------------------------
   15    15610 what`thelock                  what`consumer+0xa8
   18    15591 what`thelock                  what`consumer+0x44
  525    12021 what`thelock                  what`producer+0xb4
   41     6353 0x2d000                       libumem.so.1`umem_cache_free+0x6c
    1     6160 0x2d380                       libumem.so.1`umem_cache_free+0x6c
```

Scenario: Threaded application

- Watch for `mutex_lock()` calls:

```
# dtrace -n 'pid29639::*mutex_lock:entry {@[ustack()] = count()}'
...
```

- Then look for cases in the code like this:

```
mutex_lock (&thelock);
foo++;
mutex_unlock(&thelock);
```

- Replace this with new atomic lock API `(atomic_ops(3c))`!
- Example - 32 threads, each incrementing an integer until it hits 10M

```
% cc atom.c -o atom
% ptime ./atom
real        2.010
user        1.887
sys         0.008
% cc -DATOM atom.c -o atom
% ptime ./atom
real        0.396
user        0.376
sys         0.004
```

Scenario: Threaded application

- Watch for robust locks – very inefficient

```
USYNC_PROCESS_ROBUST (passed in mutex_init() call)
PTHREAD_MUTEX_ROBUST_NP
           (pthread_mutex_setrobust_np())
```

- Ask whether truly needed

IO provider

- Allows dynamic instrumentation of physical disk I/O's. Can Show how file system buffering and read-ahead/write-behind is working
- Example, show the device, program, I/O size, I/O type, and file name of all disk I/O, and a summary

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
BEGIN{printf("%-10s %10s %10s %3s %s\n","Device", "Program","I/O Size","R/W", "Path");}
io:::start {
        printf("%-10s %10s %10d %3s %s\n", args[1]->dev_statname, execname,
             args[0]->b_bcount, args[0]->b_flags & B_READ? "R" : "W" , args[2]->fi_pathname);
           @[execname, pid, args[2]->fi_pathname] = sum(args[0]->b_bcount);
}
END {    printf("%-10s %8s %10s %s\n","Program", "PID", "Total", "Path");
         printa("%-10s %8d %10@d %s\n",@);
}
Device      Program   I/O Size R/W Path
cmdk0        mkfile      8192   W /export/home/foo
cmdk0        mkfile     49152   W /export/home/foo
...
Program       PID     Total     Path
mkfile         813   10493952 /export/home/foo
```

Wrap-up - The DTrace revolution

- DTrace tightens the diagnosis loop: *hypothesis->instrumentation->data gathering->analysis->hypothesis*
- Tightened loop effects a revolution in the way we diagnose transient failure.
- Focus can shift from *instrumentation* stage to *hypothesis* stage:
  - Much *less* labor intensive, less error prone
  - Much *more* brain intensive
  - *Much* more effective! (And a *lot* more fun)

For more information on Dtrace

- Solaris 10 Dynamic Tracing Guide

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

- open source™ OpenSolaris Dtrace community
  - http://opensolaris.org/os/community/dtrace

- Sample scripts

  /usr/demo/dtrace

- Brendan Gregg's Dtrace toolkit

  http://users.tpg.com.au/adsln4yb/dtrace.html

- http://blogs.sun.com

Questions ?

Thank you!

Solaris 10 Workshop DTrace

Bob Netherton
bob.netherton@sun.com
http://blogs.sun.com/bobn