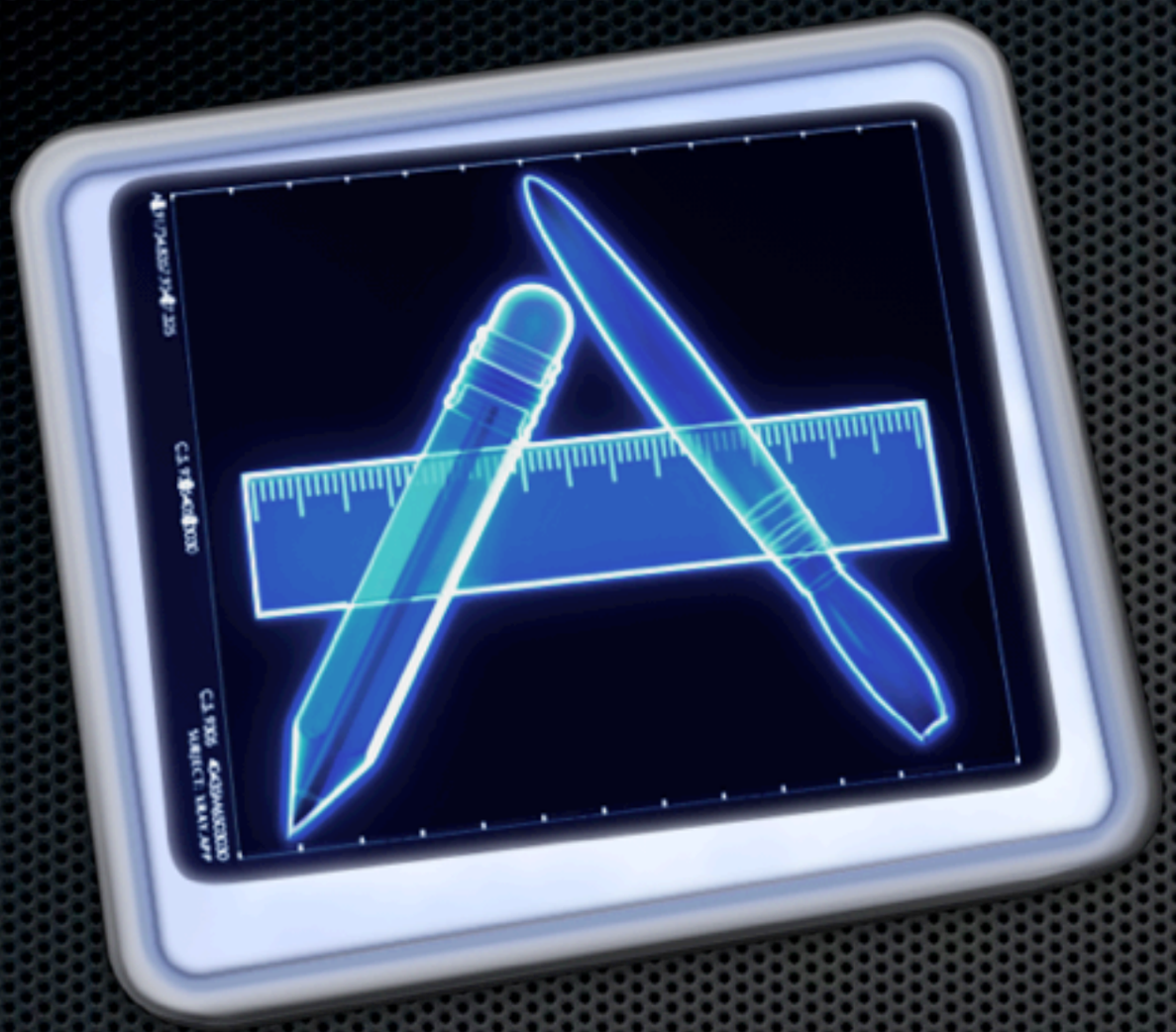
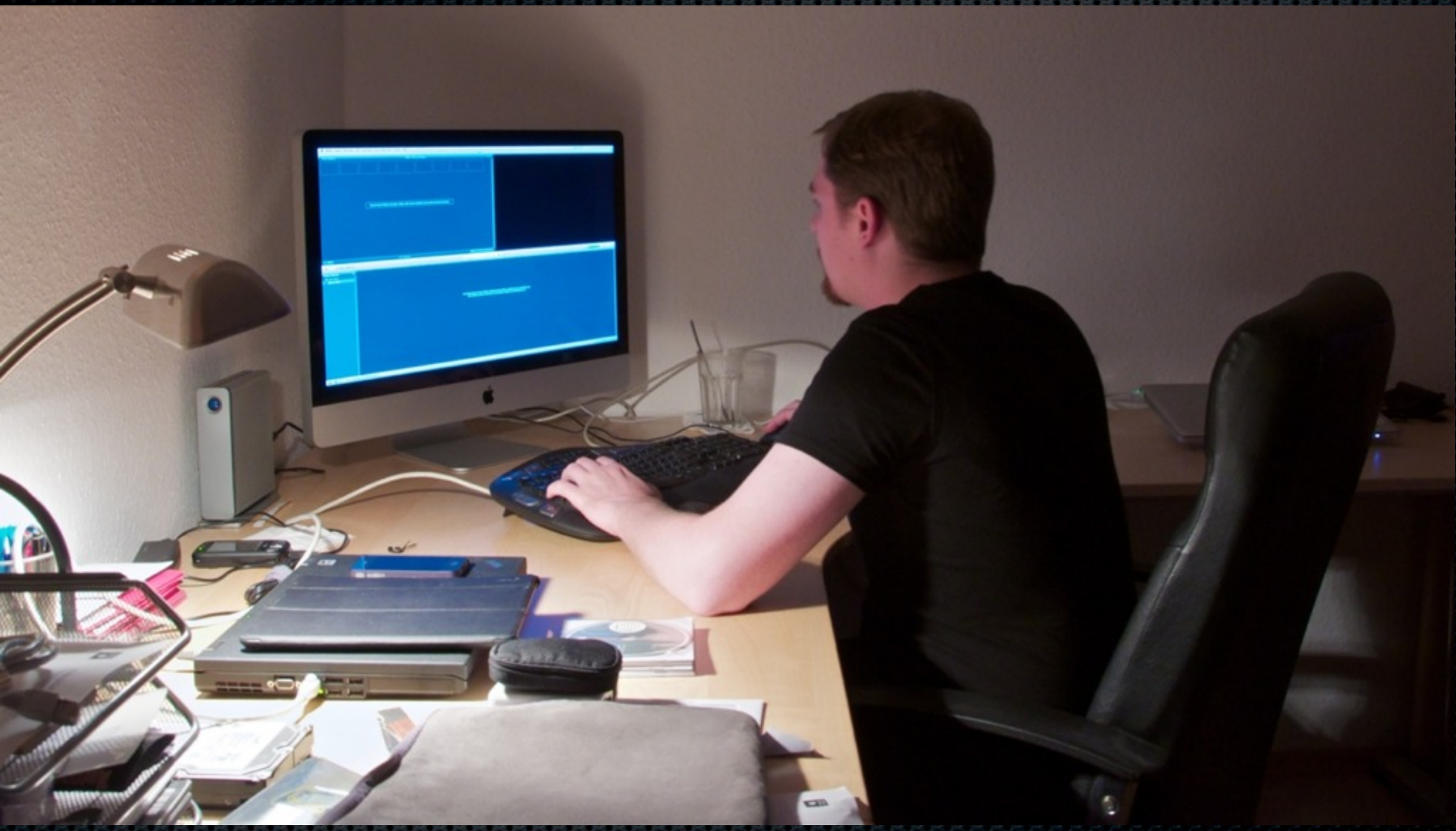


DTrace & Instruments



http://www.flickr.com/photos/rob-wei/6156468402/

When suddenly…

Why?

(h)top

Acitivity Monitor

strace

truss

printf

ps

gdb

vm_stat

DTrace!

“DTrace allows you to ask arbitrary questions about what the system is doing, and get answers.”

~ Bryan Cantrill

Developed @ Sun
First Release 2005

Solaris, Mac OS X, FreeBSD, NetBSD, Oracle Linux

OHAI!

Solaris, Mac OS X, FreeBSD, NetBSD, Oracle Linux

(since 10.5)

What can it trace?

Trace all the programs in user space

...and kernel functions

as well as kernel functions.

Detour: P_LNOATTACH

On OS X, processes can set this flag to prevent DTrace from attaching to them

Bascially, because of iTunes

Hello World

```
dtrace -n ':::BEGIN
{
  printf("Hello, world.");
  exit(0);
}'
```


Run with sudo

```
dtrace: description ':::BEGIN ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN Hello, world.
```


Output

A more useful example

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


Run with sudo, go make a cup of coffee, and hit Ctrl-C when you come back

How it works

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


So what does this do?

enable specified probes

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


`-n` tells dtrace we're specifying our stuff directly on the command line. You can use a file as well, in fact, that's what you'll probably prefer to do.

probe description

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


This describes the probe we want to match. We don’t have to provide the exact name of the probe, we can, in fact, match several probes with one description. More on this later.

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

here be actions

`{` lets DTrace know that between here and `}` are all the actions we want to take when probes fire.

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

associative array

`@num` is an associative array called `num`. Note we didn't have to declare it before we used it. Associative arrays can have multiple keys.

There are also scalar variables that simply don't have a prefix.

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

key for array is the executable name

`execname` is a (reserved) DTrace variable that contains the name of the firing process

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

value is the number of times it has been called

`count()` is one of the aggregation functions provided by DTrace

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

‘tis be all the action

`}` lets DTrace know that this is where the fun ends

Syntax

But that is not all

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

/* This is a comment */
provider : module : function : name
/ predicate /
{
  actions;
}
```


This is how the general syntax basically looks like

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

/* This is a comment */
provider : module : function : name
/ predicate /
{
  actions;
}
```

C-style comments

C-style comments, no //

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

/* This is a comment */
provider : module : function : name
/ predicate /
{
  actions;
}
```

predicate

filters

Predicates allow us to specify conditions on when the actions are executed

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

/* This is a comment */
provider : module : function : name
/ predicate /
{
  actions;
}
```


Also note that you can have more than one probe definition in your file.

$macro
scalar_variable
@aggregate_array
self->thread_local
this->clause_local

These are the kinds of variables and their scope you have available in DTrace.

self-> are thread-local variables, they are local to the firing probe
this-> are local to a clause

Variables can be declared as a type, but don't have to be.

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

syscall::read:entry
/execname != "dtrace"/
{
  @reads[execname,
         fds[arg0].fi_pathname] =
    count();
}
```


This is a more complicated usage of associative arrays.

This records how often executables access a certain file path. “fds” is an array of structs that contain information about file descriptors, like “fi_pathname”.

```
LaunchBar           ??/English.lproj/InfoPlist.strings                  1
LaunchBar           ??/Resources/Exceptions.plist                       1
SyncServer          ??/Local/cleanup.time                               1
Twitter             ??/Contents/Info.plist                              1
Twitter             ??/Resources/Exceptions.plist                       1
less                <unknown (not a vnode)>                             1
less                ??/78/xterm-256color                                1
less                ??/<unknown (NULL v_name)>/tty                      1
mdworker            ??/Cookies/Cookies.binarycookies                    1
mdworker            ??/History/http:%2F%2Ft.co%2FgaGL6M0.webhistory     1
mtmfs               ??/Caches/com.apple.FindSystemFiles.plist           1

[…]

bash                ??/libexec/rbenv-version-file                     216
git-remote-http     <unknown (not a vnode)>                           253
trustevaluation     ??/system/mdsDirectory.db                         260
Propane             <unknown (not a vnode)>                           273
bash                ??/libexec/rbenv-version-name                     288
git                 ??/.git/HEAD                                     330
nvALT               <unknown (not a vnode)>                           460
FindSystemFiles     ??/Contents/Info.plist                            492
WebProcess          <unknown (not a vnode)>                           787
sh                  <unknown (not a vnode)>                           951
bash                <unknown (not a vnode)>                          2704
```

pid	PID of firing process
ppid	PPID of firing process
errno	Last errno encountered by firing process
execname	Name of process firing the probe
arg0 … argX	64bit integers value returned from probe
args[0]…args[X]	Typed values returned from a probe

A selection of variables that are provided by DTrace

$pid	PID of dtrace process
$ppid	PPID of dtrace process
$cwd	Current working directory
$0…$9	Arguments to DTrace ($0 is script name)
$target	This will resolve to a PID of a process using the -p or -c switch
$[e](g|u)id	effective and real group or user id

Macros are expanded by DTrace when it processes the input file. You can explicitly convert a macro to a string by adding an additional “$” in front, e.g. $$pid

count()	Returns the number of times called
avg()	computes arithmetic average
sum()	adds the input to the aggregation
max()/min()	Returns max/min of the inputs provided
quantize()	generate a power of 2 distribution of the input
printf()	It's printf as we know and love/hate it

Selection of functions available.

"printa" is also very useful to print out an aggregation.

copyin(pointer, size)
copyinstr(pointer_to_str)

Important: When probes pass you addresses to pointers, they are relative to that process, not dtrace. `copyin` copies whatever the pointers point to, and make it available in your dtrace process. `copyinstr` is specifically for NULL-terminated strings.

DTrace Probes

Many, many probes

If you have DTrace, then you'll have a lot of probes to choose from

```
dtrace -l
```


Pipe this to `less` or a file, it’s a lot of output.

~300.000 probes available

Right now, on my system, there are about 300k probes available for me to, well, probe.

dtrace	Pseudo-probes like BEGIN, END, …
syscall	Entry and return points for every syscall
profile	Time-based probes that fire on ticks, seconds or fractions
io	Disk-I/O related probes
proc	process creation, forking and signaling
objc	Information about ObjC messages
ruby	Ruby memory/GC information
pid<pid>	Allows you to trace userland functions

Selection of probe providers.

syscall entry args are the same as the syscall gets. syscall return arg0 are the return value of the syscall.

http://wikis.sun.com/display/DTrace/Providers

More info on (most of) the providers here

Embedding your own

```
provider primes {
  /* Start of the prime calculation */
  probe primecalc__start(long prime);

  /* End of the prime calculation */
   probe primecalc__done(long prime, int
isprime);

  /* Exposes the size of the table of existing
primes */
   probe primecalc__tablesize(long tablesize);
};
```


Simple example: We have a program to calculate primes, and we want three DTrace probes.

First, we write a prime_probes.d file like this.

```
dtrace -o prime_probes.h -h -s prime_probes.d
```

```
#include <stdio.h>


long primes[1000000] = { 3 };
long primecount = 1;

int main(int argc, char **argv)
{
  long divisor = 0; long currentprime = 5; long isprime = 1;

  while (currentprime < 1000000)
  {
     isprime = 1;

     for(divisor=0;divisor<primecount;divisor++)
     {
        if (currentprime % primes[divisor] == 0) { isprime = 0; }
     }
     if (isprime)
     {
        primes[primecount++] = currentprime;

        printf("%ld is a prime\n",currentprime);
     }
     currentprime = currentprime + 2;
   }
}
```

```c
#include <stdio.h>
#include "prime_probes.h"

long primes[1000000] = { 3 };
long primecount = 1;

int main(int argc, char **argv)
{
  long divisor = 0; long currentprime = 5; long isprime = 1;

  while (currentprime < 1000000)
  {
     isprime = 1;
     PRIMES_PRIMECALC_START(currentprime);
     for(divisor=0;divisor<primecount;divisor++)
     {
        if (currentprime % primes[divisor] == 0) { isprime = 0; }
     }
     PRIMES_PRIMECALC_DONE(currentprime, isprime);
     if (isprime)
     {
        primes[primecount++] = currentprime;
        PRIMES_PRIMECALC_TABLESIZE(primecount);
        printf("%ld is a prime\n",currentprime);
     }
     currentprime = currentprime + 2;
  }
}
```

```
clang -c primes.c
clang -o primes primes.o
```


Compile the file(s).

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

#pragma D option quiet

primes*:::primecalc-start { self->start = timestamp; }

primes*:::primecalc-done
/arg1 == 1/
{
        @times["prime"] = sum(timestamp - self->start);
}

primes*:::primecalc-done
/arg1 == 0/
{
        @times["nonprime"] = sum(timestamp - self->start);
}

END
{
        normalize(@times,1000000);
        printa(@times);
}
```


Compile the file(s).

```
» ./primes

(in another shell)

» sudo dtrace -s timing.d
^C

  prime                                             3344
  nonprime                                     385675795
```


Compile the file(s).

```
if (PRIMES_PRIMECALC_START_ENABLED()) {
  PRIMES_PRIMECALC_START(currentprime);
}
```

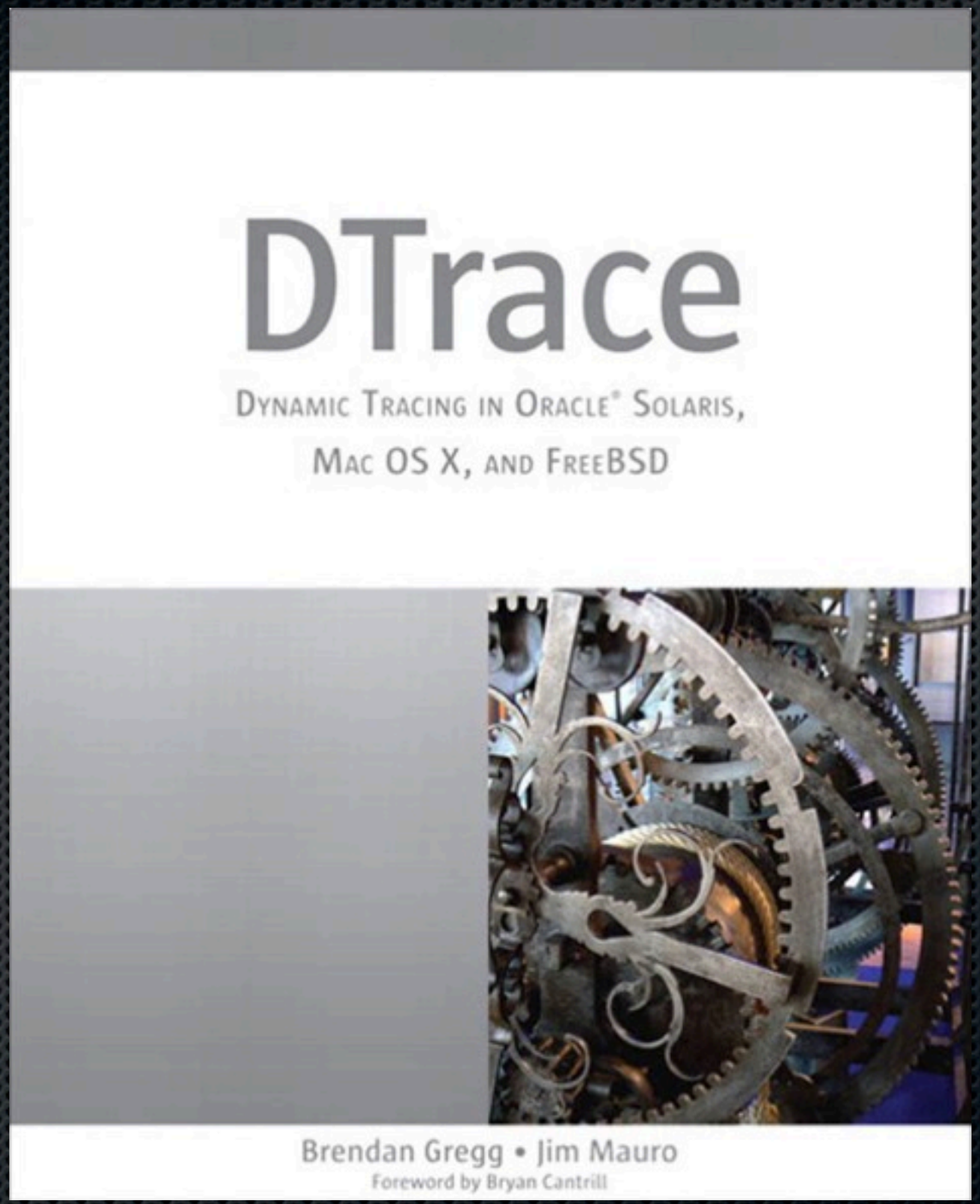

If you want to be a good citizen, you use this. It also allows you to calculate additional information to pass to the probe.

Instruments

Demo

apropos dtrace

See all the scripts that use DTrace to provide you with information


DTrace
Dynamic Tracing in Oracle® Solaris,
Mac OS X, and FreeBSD
Brendan Gregg • Jim Mauro
Foreword by Bryan Cantrill