A Look Inside FreeBSD with DTrace

Introduction and Tutorial Overview

George V. Neville-Neil Robert N. M. Watson

June 8, 2016

Objectives

- Understand key kernel concepts
- Become comfortable with DTrace
    - Terminology
    - Basic Usage
    - Advanced Scripting
- Explore on your own

What is an operating system?

Whiteboarding exercise

What is an operating system?

[An OS is] low-level software that supports a computer's basic functions, such as scheduling tasks and controlling peripherals.

- Google hive mind

General-purpose operating systems

... are for general-purpose computers

- Servers, workstations, mobile devices
- Run ‘applications’ – i.e., software unknown at design time
- Abstract the hardware, provide ‘class libraries’
- E.g., Windows, Mac OS X, Android, iOS, Linux, FreeBSD, ...

General-purpose operating systems

... are for general-purpose computers

- Servers, workstations, mobile devices
- Run 'applications' – i.e., software unknown at design time
- Abstract the hardware, provide 'class libraries'
- E.g., Windows, Mac OS X, Android, iOS, Linux, FreeBSD, ...

Userspace Local and remote shells, management tools, daemons

Run-time linker, system libraries, tracing facilities

General-purpose operating systems

... are for general-purpose computers

- Servers, workstations, mobile devices
- Run ‘applications’ – i.e., software unknown at design time
- Abstract the hardware, provide ‘class libraries’
- E.g., Windows, Mac OS X, Android, iOS, Linux, FreeBSD, ...

Userspace Local and remote shells, management tools, daemons

Run-time linker, system libraries, tracing facilities

- - - - system-call interface - - - -

General-purpose operating systems

... are for general-purpose computers

- Servers, workstations, mobile devices
- Run 'applications' – i.e., software unknown at design time
- Abstract the hardware, provide 'class libraries'
- E.g., Windows, Mac OS X, Android, iOS, Linux, FreeBSD, ...

Userspace Local and remote shells, management tools, daemons

Run-time linker, system libraries, tracing facilities

- - - - system-call interface - - - -

Kernel System calls, hypercalls, remote procedure call (RPC)

Processes, filesystems, IPC, sockets,

General-purpose operating systems

... are for general-purpose computers

- Servers, workstations, mobile devices
- Run ‘applications’ – i.e., software unknown at design time
- Abstract the hardware, provide ‘class libraries’
- E.g., Windows, Mac OS X, Android, iOS, Linux, FreeBSD, ...

Userspace Local and remote shells, management tools, daemons

Run-time linker, system libraries, tracing facilities

- - - - *system-call interface* - - - -

Kernel System calls, hypercalls, remote procedure call (RPC)

Processes, filesystems, IPC, sockets,

What does an operating system do?

- Key hardware-software surface (cf. compilers)
- System management: bootstrap, shutdown, watchdogs
- Low-level abstractions and services
  - Programming: processes, threads, IPC, program model
  - Resource sharing: scheduling, multiplexing, virtualisation
  - I/O: device drivers, local/distributed filesystems, network stack
  - Security: authentication, encryption, permissions, labels, audit
  - Local or remote access: console, window system, SSH
- Libraries: math, protocols, RPC, cryptography, UI, multimedia
- Other stuff: system log, debugging, profiling, tracing

Why study operating systems?

The OS plays a central role in **whole-system design** when building efficient, effective, and secure systems:

- Key interface between hardware and software
- Strong influence on whole-system performance
- Critical foundation for computer security
- Exciting programming techniques, algorithms, problems
  - Virtual memory; network stacks; filesystems; runtime linkers; ...
- Co-evolves with platforms, applications, users
- Multiple active research communities
- Reusable techniques for building complex systems
- Boatloads of fun (best text adventure ever)

FreeBSD

- Open Source
- Unix
- Posix
- Complete System
- 20 years of history

Overview

- This Morning
    - Introduction to DTrace
    - Processes and the Process Model
    - Scheduler
    - Locking
- This Afternoon
    - Networking
    - Filesystems

A Look Inside FreeBSD with DTrace

What is DTrace?

George V. Neville-Neil   Robert N. M. Watson

June 8, 2016

What is DTrace?

- A dynamic tracing framework for software
- Low impact on overall system performance
- Does not incur costs when not in use

What can DTrace show me?

- When a function is being called
- A function's arguments
- The frequency of function calls
- A whole lot more...

A Simple Example

```
dtrace -n syscall:::
dtrace: description 'syscall:::' matched 2148 probes
CPU     ID                    FUNCTION:NAME
  1  51079                    ioctl:return
  1  51078                     ioctl:entry
  1  51079                    ioctl:return
  1  51078                     ioctl:entry
  1  51079                    ioctl:return
  1  51632               sigprocmask:entry
  1  51633              sigprocmask:return
  1  51784                 sigaction:entry
```

- Look at all system calls

How does DTrace Work?

- Various probes are added to the system
- The probes are activated using the dtrace program
- A small number of assembly instructions are modified at run-time to get the system to run in the probe

A more complex example

```
dtrace -n 'syscall::write:entry /arg2 != 0/ { printf("write size %d\n", arg2); }'
dtrace: description 'syscall::write:entry ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  50978                    write:entry write size 1
  0  50978                    write:entry write size 55
  0  50978                    write:entry write size 2
```

DTrace Glossary

Probe A way of specifying what to trace

Provider A DTrace defined module that provides information about something in the system

Module A software module, such as `kernel`

Function A function in a module, such as `ether_input`

Predicate A way of filtering DTrace probes

Action A set of D language statements carried out when a probe is matched

Providers

fbt Function Boundary Tracing (50413)
syscall System Calls (2148)
profile Timing source
proc Process Operations
sched Scheduler
io I/O calls
ip Internet Protocol
udp UDP
tcp TCP
vfs Filesystem Routines

Dissecting a Probe

- `syscall::write:entry`

  Provider syscall
  Module None
  Function write
  Name entry

- `fbt:kernel:ether_input:entry`

  Provider fbt
  Module kernel
  Function `ether_input`
  Name entry

DTrace Requirements

- A kernel with DTrace support built in
  - Default on FreeBSD 10 or later
- The ability to sudo or be root
- The complete command syntax is covered in the dtrace manual page

Finding Probes

- Listing all the probes gets you 50000 to choose from
- Judicious use of providers, modules and grep
- e.g. `dtrace -l -P syscall`

Probe Arguments

- Use verbose (-v) mode to find probe arguments
- `sudo dtrace -lv -f syscall:freebsd:read`

```
 ID   PROVIDER              MODULE
57177    syscall             freebsd

Argument Types
args[0]: int
args[1]: void *
args[2]: size_t
```

The D Language

- A powerful subset of C
- Includes features specific to DTrace, such as aggregations
- Anything beyond some simple debugging usually required a D script

DTrace One-Liners

- A set of useful single line scripts

```
# Trace file opens with process and filename:
dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'

# Count system calls by program name:
dtrace -n 'syscall:::entry { @[execname] = count(); }'

# Count system calls by syscall:
dtrace -n 'syscall:::entry { @[probefunc] = count(); }'
```

Count System Calls

```
dtrace -n 'syscall:::entry { @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 1072 probes
^C
  fstat                                                    1
  setitimer                                                1
  getpid                                                   2
  read                                                     2
  sigreturn                                                2
  write                                                    3
  getsockopt                                               4
  select                                                   6
  sigaction                                                6
  _umtx_op                                                 7
  __sysctl                                                 8
  munmap                                                  18
  mmap                                                    19
  sigprocmask                                             23
  clock_gettime                                           42
  ioctl                                                   45
```

Aggregations

- `syscall:::entry { @[probefunc] = count(); }`
- The `@[probefunc]` syntax
- Aggregates data during a run for later output
- Extremely powerful feature of D language

Quantization

```
# Summarize requested write() sizes by program name, as power-of-2 distributions (bytes):
dtrace -n 'syscall::write:entry { @[execname] = quantize(arg2); }'
dtrace: description 'syscall::write:entry ' matched 2 probes
^C
  find
           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |                                         1
               4 |                                         17
               8 |@@                                       841
              16 |@@@@@@@@@@@@@                            6940
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@                13666
              64 |                                         59
             128 |                                         0
```

Probing the stack

- Find out how we got where we are
- The `stack()` routine

Who called malloc()?

```
1  29371                    malloc:entry
            kernel`cloneuio+0x2c
            kernel`vn_io_fault1+0x3b
            kernel`vn_io_fault+0x18b
            kernel`dofileread+0x95
            kernel`kern_readv+0x68
            kernel`sys_read+0x63
            kernel`amd64_syscall+0x351
            kernel`0xffffffff80d0aa6b
```

- Read upwards from the bottom

DTrace Toolkit

- An open source set of tools written to use D scripts
- Currently specific to Solaris
- Exists as a FreeBSD port (thanks to Steve)
- Currently being updated

An example script: hotkernel

```
./hotkernel
Sampling... Hit Ctrl-C to end.
^C
FUNCTION                                    COUNT   PCNT
kernel`lookup                                   1   0.1%
kernel`unlock_mtx                               1   0.1%
kernel`_vm_page_deactivate                      1   0.1%
...
kernel`amd64_syscall                            9   0.5%
kernel`pmap_remove_pages                        9   0.5%
kernel`hpet_get_timecount                      13   0.7%
kernel`pagezero                                15   0.8%
kernel`0xffffffff80                            34   1.9%
kernel`spinlock_exit                          486  27.0%
kernel`acpi_cpu_c1                            965  53.6%
```

Predicates

- Filtering probes based on relevant data
- Useful for excluding common conditions
- `/arg0 != 0/` Ignore a normal return value

Tracking a Specific Process

- `pid` is used to track a Process ID
- Used in predicates
- `/pid == 1234/`

Running a Program Under DTrace

- DTrace is most often used on running systems
- DTrace can be attached at runtime to a program
    - `dtrace -p pid ...`
- Run a program completely under the control of DTrace
    - `dtrace -c cmd ...`

Going too far

- Overly broad probes slow down the system
    - Watching everything in the kernel
    - Registering a probe on a module

The Probe Effect

- Each probe point has a cost
- Every action has a reaction
- Any action code requires time to run
- Impacts system performance

DTrace Lab Exercises

- Bring up OSCourse Virtual Machine
- Find the current list of providers
- Count the probes available
- Trace all the system calls used by sshd
- Summarize requested write() sizes by program name
- Summarize return values from write() by program name
- Find and modify three (3) of the DTrace one-liners

A Look Inside FreeBSD with DTrace

Processes

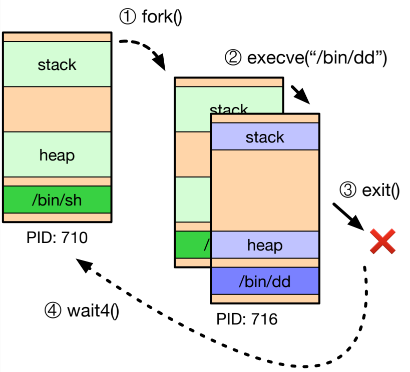
George V. Neville-Neil   Robert N. M. Watson

June 8, 2016

The Process Model

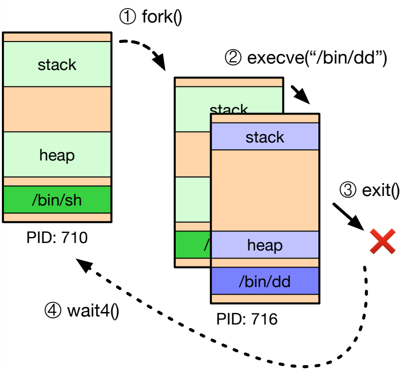
- The most basic container
- All of a program’s resources
- The entity that is scheduled and executed

The UNIX process life cycle



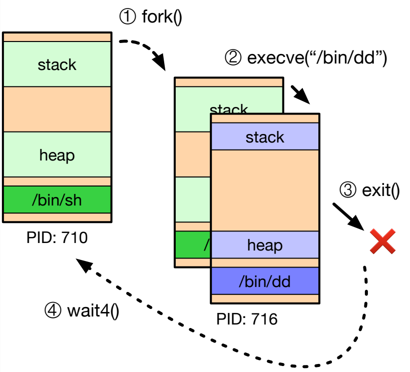
- `fork()`
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., `stdio`)
  - Copy-on-Write (COW)

The UNIX process life cycle



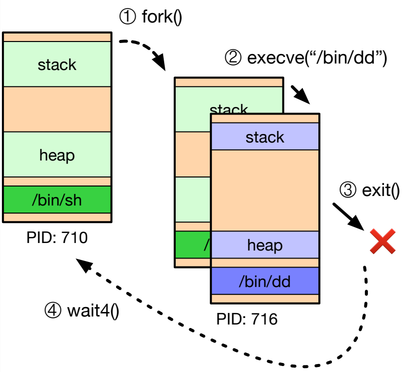
- `fork()`
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., `stdio`)
  - Copy-on-Write (COW)
- `execve()`
  - Kernel replaces address space, loads new binary, starts execution

The UNIX process life cycle



- `fork()`
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., `stdio`)
  - Copy-on-Write (COW)
- `execve()`
  - Kernel replaces address space, loads new binary, starts execution
- `exit()`
  - Process can terminate self (or be terminated)

The UNIX process life cycle



- `fork()`
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., `stdio`)
  - Copy-on-Write (COW)
- `execve()`
  - Kernel replaces address space, loads new binary, starts execution
- `exit()`
  - Process can terminate self (or be terminated)
- `wait4` (et al)

Tracing the Process Lifecycle

fork() Count forks per second

execve() What is being executed?

exit() What programs generate errors?

Who is forking?

```
dtrace -n 'syscall::*fork:entry { @forks[execname] = count();}'
dtrace: description 'syscall::*fork:entry ' matched 8 probes
^C
  csh                                                         7031
```

Fork Discussion

- Why do we use a wild card?
  - `syscall::*fork:entry`

What’s starting on the system?

```
./execsnoop
  UID    PID   PPID ARGS
    0   4661   4398 -csh
    0   4661   4398 ls
    0   4662   4398 -csh
    0   4662   4398 ls
```

A look inside execsnoop

Proc Provider

exec Program execution attempt

exec-failure Program start failed

exec-success Program successfully started

exit Program terminated

signal-send Send a signal

signal-clear Cleared a signal

signal-discard Signal ignored

Process Thrashing

- Process creation is expensive
- Programs that start and fail cause the system to thrash

Tracking Processes

- `newproc.d` track new processes
- `pidspersec.d` processes created per second

Process Termination

- All processes exit
- Return an error status
- May exit due to a fault

Programs that exit with errors

```
dtrace -n 'syscall::exit:entry /arg0 != 0/{ printf("%s %d\n", execname, arg0); }'
```

Signals

- Early form of inter-process communication
- Modeled on hardware interrupts
- Processes can send and receive signals
- Signals can be *caught*
- Uncaught signals often result in program termination
- Kill signal (9) cannot be avoided

Tracking Signals

- `kill.d` displays signals sent and received

Process Lab Exercises

- What happens for each signal sent to `yes`
- Extend newproc script to show program arguments
- Write a script to show the entire process life cycle from creation to exit

A Look Inside FreeBSD with DTrace

The Scheduler

George V. Neville-Neil   Robert N. M. Watson

June 8, 2016

The Scheduler

- Decides which thread gets to run
- The *thread* is the scheduable entity
- Chooses a processor/core
- Can be overridden by `cpuset`

Process States

NEW	Being created
RUNNABLE	Can run
SLEEPING	Awaiting some event
STOPPED	Debugging
ZOMBIE	Process of dying

Scheduling Classes

ITHD interrupt thread

REALTIME real-time user

KERN kernel threads

TIMESHARE normal user programs

IDLE run when nothing else does

Scheduler Framework

- Schedulers have kernel API
- `SCHED_4BSD` and `SCHED_ULE`
- High level scheduler picks the CPU via the runq
- Low level scheduler picks the thread to run
- `sched_pickcpu` selects the CPU
- `mi_switch` Entry to a forced context switch
- `sched_switch` scheduler API

Sched Provider

on-cpu Thread moves on core

off-cpu Thread moves off core

remain-cpu Thread remains on core

change-pri Priority changed

fbt:kernel:cpu_idle:entry Thread went idle

Dummy Probes (Do Not Use)

- Probes purely for D script compatibility
- These never fire
- `cpucaps-sleep`
- `cpucaps-wakeup`
- `schedctl-nopreempt`
- `schedctl-preempt`
- `schedctl-yield`

Idle vs. Running

- `cpudists`

Who's sleeping?

```
dtrace -n 'sched:::sleep { @prog[execname] = count() }
dtrace: description 'sched:::sleep ' matched 1 probe
^C
cron                                        1
devd                                        1
pagezero                                    1
sendmail                                    1
sudo                                        1
nfsd                                            2
```

Idle vs. Active

```
sudo ./cpudist
Ctrl-C
 KERNEL
   value  ---------- Distribution ---------- count
     256 |                                   0
     512 |                                   3
    1024 |@@@@@@@@@                          58
    2048 |@@@@@@@@@@@@@@                     93
    4096 |@@@@@@@@@@@@@@@@@                  120
    8192 |@@                                 17
   16384 |                                   1
   32768 |@                                  4
   65536 |                                   1
  131072 |                                   0
```

A look inside cpudist

Changing Priorities

```
dtrace -n 'sched:::change-pri { printf("%s %d %d", execname, curlwpsinfo->pr_pri, arg2); }' |
dtrace: description 'sched:::change-pri ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1  49443                       :change-pri csh 176 152
  1  49443                       :change-pri ls 176 120
```

A Multi-core World

- All large systems are multi-core
- Scheduling on multi-core is difficult
- Some systems resort to static allocation

Are threads migrating?

- Watching threads with `cpuwalk.d`

Context Switching

- Processes all believe they own the computer
- Context switching maintains this fiction
- Requires saving and restoring state
- Common measure of operating system performance
- `cswstat.d` measures overall context switching

A look inside cswstat.d

Scheduler Lab Exercises

- Write a one-liner to show processes waking up
- Extend wake up one-liner to include stack tracing
- Extend priority one-liner to include stack tracing
- Add periodic output to `cpuwalk.d`
- Track context switching for a single process

A Look Inside FreeBSD with DTrace

Extending DTrace

George V. Neville-Neil   Robert N. M. Watson

June 8, 2016

Death to printf

- Over $10,000$ calls to `device_printf()`
- 75 Separate version of `DEBUG` macro
- `WITNESS` for lock ordering
- `LOCKSTAT` locking statistics
- `KTR` for Kernel Trace
- Enabled at compile time

Statically Defined Tracepoints

- Can appear anywhere in code
  - Not just at entry or return
- Useful for replacing `printf()` and logging and `DEBUG`
- `USDT` vs. `SDT`

The Extension Process

Provider	Add or extend?
Declare	tracepoints in a header
Define	tracepoints in compiled code
Translate	the arguments and structures

Debugger Syntax

Translators

Translators

- Rationalize structures across platforms
- Give convenient names for complex data types
- Do not have a zero cost

Stability

- What makes a provider or probe stable or unstable?

TCP	Stable
UDP	Stable
IP	Stable
mbuf	Unstable

A Look Inside FreeBSD with DTrace

Kernel SDTs

George V. Neville-Neil   Robert N. M. Watson

June 8, 2016

Converting Logging Code

- Most code littered with `printf`
- Many different `DEBUG` options
- Most can be converted

TCPDEBUG Case Study

- TCBDEBUG added in the original BSD releases
- Rarely enabled kernel option that shows:
    - direction
    - state
    - sequence space
    - `rcv_nxt`, `rcv_wnd`, `rcv_up`
    - `snd_una`, `snd_nxt`, `snx_max`
    - `snd_wl1`, `snd_wl2`, `snd_wnd`

TCPDEBUG Before

- 127 lines of code
- 14 calls to printf
- Statically defined ring buffer of 100 entries
- Static log format

TCPDEBUG After

- Four (4) new tracepoints
  - debug-input
  - debug-output
  - debug-user
  - debug-drop
- Access to TCP and socket structures
- Flexible log format

Convenient Macros

- `SDT_PROVIDER_DECLARE` Declare a provider in an include file
- `SDT_PROVIDER_DEFINE` Instantiate a provider in C code
- `SDT_PROBE_DECLARE` Declare a probe in a n include file
- `SDT_PROBE_DEFINEN` Define a probe of X arguments (0-6)
- `SDT_PROBE_DEFINEN_XLATE` Define a probe of N arguments with translation
- Only available for kernel code

TCP Debug Desclarations

```
SDT_PROBE_DECLARE(tcp, , , debug__input);
SDT_PROBE_DECLARE(tcp, , , debug__output);
SDT_PROBE_DECLARE(tcp, , , debug__user);
SDT_PROBE_DECLARE(tcp, , , debug__drop);
```

TCP Debug Call Sites

```
#ifdef TCPDEBUG
        if (tp == NULL || (tp->t_inpcb->inp_socket->so_options & SO_DEBUG))
                tcp_trace(TA_DROP, ostate, tp, (void *)tcp_saveipgen,
                        &tcp_savetcp, 0);
#endif
        TCP_PROBE3(debug__input, tp, th, mtod(m, const char *));
```

TCP Debug Translators

```
SDT_PROBE_DEFINE3_XLATE(tcp, , , debug__input,
    "struct tcpcb *", "tcpsinfo_t *" ,
    "struct tcphdr *", "tcpinfo_t *",
    "uint8_t *", "ipinfo_t *");

SDT_PROBE_DEFINE3_XLATE(tcp, , , debug__output,
    "struct tcpcb *", "tcpsinfo_t *" ,
    "struct tcphdr *", "tcpinfo_t *",
    "uint8_t *", "ipinfo_t *");

SDT_PROBE_DEFINE2_XLATE(tcp, , , debug__user,
    "struct tcpcb *", "tcpsinfo_t *" ,
    "int", "int");

SDT_PROBE_DEFINE3_XLATE(tcp, , , debug__drop,
    "struct tcpcb *", "tcpsinfo_t *" ,
    "struct tcphdr *", "tcpinfo_t *",
    "uint8_t *", "ipinfo_t *");
```

TCP Debug Example Script

```
tcp:kernel::debug-input
/args[0]->tcps_debug/
{
        seq = args[1]->tcp_seq;
        ack = args[1]->tcp_ack;
        len = args[2]->ip_plength - sizeof(struct tcphdr);
        flags = args[1]->tcp_flags;

        printf("%p %s: input [%xu..%xu]", arg0,
               tcp_state_string[args[0]->tcps_state], seq, seq + len);

        printf("@%x, urp=%x", ack, args[1]->tcp_urgent);
```

TCP DEbug Example Script Part 2

```
        printf("%s", flags != 0 ? "<" : "");
        printf("%s", flags & TH_SYN ? "SYN," :"");
        printf("%s", flags & TH_ACK ? "ACK," :"");
        printf("%s", flags & TH_FIN ? "FIN," :"");
        printf("%s", flags & TH_RST ? "RST," :"");
        printf("%s", flags & TH_PUSH ? "PUSH," :"");
        printf("%s", flags & TH_URG ? "URG," :"");
        printf("%s", flags & TH_ECE ? "ECE," :"");
        printf("%s", flags & TH_CWR ? "CWR" :"");
        printf("%s", flags != 0 ? ">" : "");

        printf("\n");
        printf("\trcv_(nxt,wnd,up) (%x,%x,%x) snd_(una,nxt,max) (%x,%x,%x)\n",
                args[0]->tcps_rnxt, args[0]->tcps_rwnd, args[0]->tcps_rup,
                args[0]->tcps_suna, args[0]->tcps_snxt, args[0]->tcps_smax);
        printf("\tsnd_(wl1,wl2,wnd) (%x,%x,%x)\n",
                args[0]->tcps_swl1, args[0]->tcps_swl2, args[0]->tcps_swnd);
```

How Much Work is That?

- 200 line code change
- 167 lines of example code
- A few hours to code
- A day or two to test
- Now we have always on TCP debugging

Lab Exercise: Adding Kernel Tracepoints

Networking and FreeBSD

- Everyone's TCP/IP Stack
- IPv4, IPv6, UDP, TCP, SCTP
- Various drivers
- Multiple firewalls

The User Program View

- User programs use sockets
- Network programs follow UNIX model
- Flexible interfaces for different protocols

Sockets

- Main programmer interface to networking
- Generic API
- Attempts to support read/write semantics

Looking Directly at Sockets

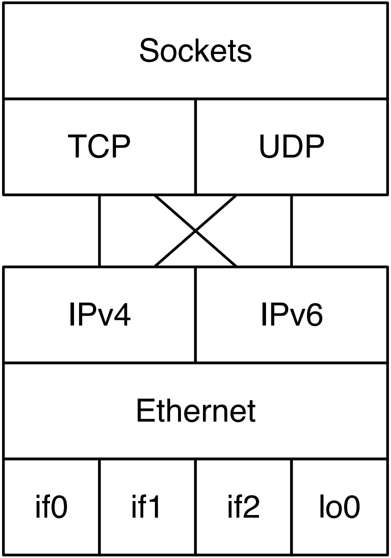
```
# Count sockets by family

# Count sockets by type

# Count sockets by protocol
```

Network Lab (Sockets Exercises)

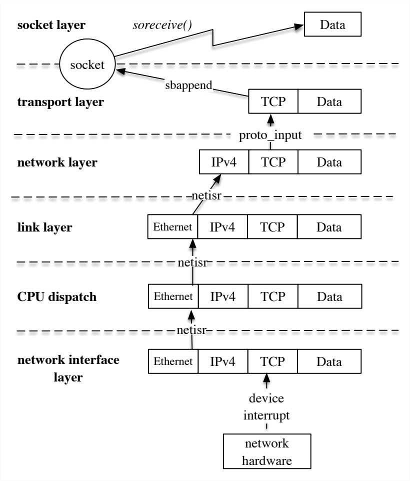
- Count socket calls by domain, type and protocol
- Show programs accepting connections
- Show programs initiating connections
- Write a D script to trace a single socket with the test program

Network Stack

Network Stack Overview



Inbound Layer Transitions



UDP

- Simplest transport protocol
- No states to maintain
- Data is sent immediately
- Supports multicast
- Only probes are `send` and `receive`

UDP Send and Receive

- `udptrack`

TCP

- Transmission Control Protocol
- Stream based
- In order delivery
- Maintains the illusion of a byte stream

TCP Connections

- `tcpconn`

TCP State Machine

- `tcpstate`

Tracking More of TCP

- `tcptrack`

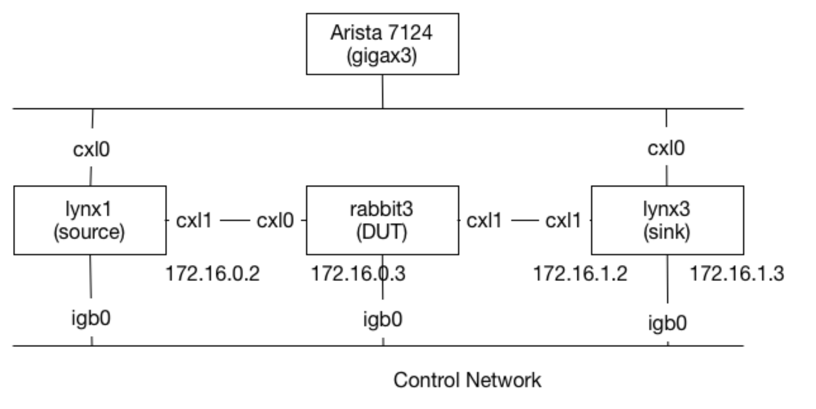
Network Protocol Lab Exercises

- Add IP source and destination information to `tcpstate`
- Add support for `send` and `receive` calls to `tcptrack`
- Show the congestion window for a single connection over time

Packet Forwarding

- System as a router, switch or firewall
- Network Layer Packets only

A Worked Example



Forward vs. Fast Forward

- What difference does this make?
  - `net.inet.ip.fastforward`
- Where do we look?
- What can be known?

```
 value  ------------- Distribution ------------- cou
 512 |                                          0
 1024 |@@@@@@@@@@@@@@@@@@@@@@@  1414505
 2048 |@                                         3547
 4096 |                                          481
 8192 |                                          0

  value  ------------- Distribution ------------- co
 512 |                                          0
 1024 |@@@@@@@@@@@@@@@@@@@@  1721837
 2048 |@                                         4128
 4096 |                                          490
 8192 |                                          0
```

Network Lab (Protocols)

- Show inbound connections to sshd
- What routines are called when a ping packet arrives?
- What routines are called before `tcp_output()`?

A Look Inside FreeBSD with DTrace

Network Memory (mbufs)

George V. Neville-Neil   Robert N. M. Watson

June 8, 2016

What is an mbuf?

- Memory for network data
- Contains meta-data
- Compact and flexible
- Clusters vs. mbufs

mbuf structures

mbuf lifecycle

- Allocation
- Adjustment
- References
- Recycling

mbuf API

- `m_init` Initialize an mbuf
- `m_get` Allocate an mbuf
- `m_gethdr` Allocate a packet header mbuf
- `m_getcl` Allocate an mbuf with a cluster
- `m_free` Free a single mbuf
- `m_freem` Free a chain of mbufs

mbuf tracepoints

- `sdt:::m-init`
- `sdt:::m-gethdr`
- `sdt:::m-get`
- `sdt:::m-getcl`
- `sdt:::m-clget`
- `sdt:::m-cljget`
- `sdt:::m-cljset`
- `sdt:::m-free`
- `sdt:::m-freem`

mbuf translator

mbuf one liners

- Where are clusters allocated?
- `sdt:::m-getcl { @[stack()] = count();}`
- Where do we wait?
- `m-getcl/arg0 == 2/{@[stack()] = count();}`
- Where do we not wait?
- `m-getcl/arg0 == 1/{ @[stack()] = count(); }`

Network Lab (mbufs)

- Write an mbuf one liner to track mbuf frees.
- Write a short script that tracks `m_get` vs. `m_free`

File Systems Overview

naming Translating human names to usable objects

storage Store and retrieve blocks of data

Naming

- Translate a human name to something
- `namei` is the main interface
- All names reside in the name cache

Name Lookup

- What names are being looked up?

```
dtrace -n 'vfs:namei:lookup:entry { printf("%s", stringof(arg1));}'
CPU     ID                    FUNCTION:NAME
  2  27847                     lookup:entry /bin/ls
  2  27847                     lookup:entry /libexec/ld-elf.so.1
  2  27847                     lookup:entry /etc
  2  27847                     lookup:entry /etc/libmap.conf
  2  27847                     lookup:entry /etc/libmap.conf
```

Name Cache

- Speeds up searching
- Maintains positive and negative results
- Invalidation on changes in directories

Who is missing the cache?

```
dtrace -n 'vfs:namecache:lookup:miss { printf("%s", stringof(arg1));}'
```

Name Cache Module

enter Add a positive entry

enter_negative Add a negative entry

lookup:hit Name found in positive cache

lookup:hit-negative Name found in negative cache

lookup:miss Name not found in cache

purge Remove positive entry

purge_negative Remove negative entry

zap Remove positive entry with or without vnode

zap_negative Remove negative entry with or without vnode

Adding negative entries

```
dtrace -n 'vfs:namecache:enter_negative: { printf("%s", stringof(arg1)); }'
```

Name Caching Lab Exercises

- Create a one-liner to count zaps vs. purges
- Write a script to track all namecaching statistics

VNODE Operations

- After a path or name is looked up
- Do something with a vnode
- `open`, `close`, `read`, `write`

VFS Lab Exercises

- Compare VFS reads with the read system call
- Compare VFS writes with the write system call
- Track all VOP operations and count their frequency