

L41: Lab 1 - I/O

Dr Robert N. M. Watson and Dr Graeme Jenkinson

1 November 2016

L41: Lab 1 - I/O

- ▶ Introduce the BeagleBone Black and DTrace
- ▶ Introduce our Jupyter-based experimental environment
- ▶ Explore user-kernel interactions via syscalls and traps
- ▶ Learn a bit about POSIX I/O
- ▶ Measure the probe effect

The benchmark

```
[guest@beaglebone ~/io]$ ./io-static
io-static -c|-r|-w [-Bdqs] [-b blocksize] [-t totalsize] path
```

Modes (pick one):

```
-c      'create mode': create benchmark data file
-r      'read mode': read() benchmark
-w      'write mode': write() benchmark
```

Optional flags:

```
-B      Run in bare mode: no preparatory activities
-d      Set O_DIRECT flag to bypass buffer cache
-q      Just run the benchmark, don't print stuff out
-s      Call fsync() on the file descriptor when complete
-v      Provide a verbose benchmark description
-b blocksize  Specify a block size (default: 16384)
-t totalsize  Specify total I/O size (default: 16777216)
```

- ▶ Simple, bespoke I/O benchmark: `read()` or `write()`
- ▶ Statically or dynamically linked
- ▶ Adjust buffer sizes, etc.
- ▶ Various output modes.

The benchmark (2)

- ▶ Three operational modes:

Create (-c) Create a new benchmark data file

Read (-r) Perform `read()`s against data file

Write (-w) Perform `write()`s against data file

The benchmark (2)

- ▶ Three operational modes:

Create (-c) Create a new benchmark data file

Read (-r) Perform `read()`s against data file

Write (-w) Perform `write()`s against data file

- ▶ Adjust I/O parameters:

Block size (-b) Block size used for each I/O

Total size (-t) Total size across all I/Os

Direct (-d) Use direct I/O (bypass buffer cache)

Sync (-s) Perform `fsync()` after I/O loop

Bare (-B) Don't synchronise cache (etc) on start
(whole-program testing)

The benchmark (2)

- ▶ Three operational modes:

Create (-c) Create a new benchmark data file

Read (-r) Perform `read()`s against data file

Write (-w) Perform `write()`s against data file

- ▶ Adjust I/O parameters:

Block size (-b) Block size used for each I/O

Total size (-t) Total size across all I/Os

Direct (-d) Use direct I/O (bypass buffer cache)

Sync (-s) Perform `fsync()` after I/O loop

Bare (-B) Don't synchronise cache (etc) on start
(whole-program testing)

- ▶ Output flags:

Quiet (-q) Suppress all output (whole-program tracing)

Verbose (-v) Verbose output (interactive testing)

The benchmark (3)

```
[guest@beaglebone ~/io]$ ./io-static -v -d -w /data/iofile
Benchmark configuration:
  blocksize: 16384
  totalsize: 16777216
  blockcount: 1024
  operation: write
  path: /data/iofile
  time: 58.502746875
280.06 KBytes/sec
```

- ▶ Use verbose output
- ▶ Bypass the buffer cache
- ▶ Write to the previously created file `/data/iofile`
- ▶ Use default buffer size (16K) and total I/O size (16M)

Exploratory questions

- ▶ Baseline benchmark performance analysis:
 - ▶ How do `read()` and `write()` performance compare?
 - ▶ What is the performance impact of the buffer cache?
Consider both `-d` and `-s`.
 - ▶ What proportion of time is spent in userspace vs. the kernel?
 - ▶ How many times are system calls invoked during the I/O loop?
 - ▶ What is the role of traps in execution of the I/O loop?
 - ▶ How does work performed in just the I/O loop compare with whole-program behaviour?

Exploratory questions

- ▶ **Baseline benchmark performance analysis:**
 - ▶ How do `read()` and `write()` performance compare?
 - ▶ What is the performance impact of the buffer cache?
Consider both `-d` and `-s`.
 - ▶ What proportion of time is spent in userspace vs. the kernel?
 - ▶ How many times are system calls invoked during the I/O loop?
 - ▶ What is the role of traps in execution of the I/O loop?
 - ▶ How does work performed in just the I/O loop compare with whole-program behaviour?

- ▶ **Probe effect and measurement decisions**
 - ▶ How does performance change if you insert system-call or trap probes in the I/O loop?
 - ▶ What sources of variance may be affecting benchmark performance, and how can we measure them?

Experimental questions for the lab report

- ▶ With respect to a configuration reading from a fixed-size file through the buffer cache:
 1. How does changing the I/O buffer size affect I/O-loop performance?
 2. How does static vs. dynamic linking affect whole-program performance?
 3. At what file-size threshold does any performance difference between static and dynamic linking fall below 5%? 1%?

Experimental questions for the lab report

- ▶ With respect to a configuration reading from a fixed-size file through the buffer cache:
 1. How does changing the I/O buffer size affect I/O-loop performance?
 2. How does static vs. dynamic linking affect whole-program performance?
 3. At what file-size threshold does any performance difference between static and dynamic linking fall below 5%? 1%?
- ▶ Run the benchmark to gather initial measurements
- ▶ Explore through system-call/trap tracing and profiling
- ▶ Use various configurations (e.g., I/O on `/dev/zero`) to explore kernel code-path behaviour

DTrace scripts

- ▶ Human-facing C-like language
- ▶ One or more $\{probe\ name, predicate, action\}$ tuples
- ▶ Expression limited to control side effects (e.g., no loops)
- ▶ Specified on command line or via a `.d` file

DTrace scripts

- ▶ Human-facing C-like language
- ▶ One or more $\{probe\ name, predicate, action\}$ tuples
- ▶ Expression limited to control side effects (e.g., no loops)
- ▶ Specified on command line or via a `.d` file

```
fbt::malloc:entry /execname == "csh"/ { trace(arg0); }
```

probe name Identifies the probe(s) to instrument; wildcards allowed;
identifies the *provider* and a provider-specific *probe name*

predicate Filters cases where action will execute

action Describes tracing operations

Some kernel DTrace providers in FreeBSD

Provider	Description
<code>callout_execute</code>	Timer-driven callouts
<code>dtmalloc</code>	Kernel <code>malloc()/free()</code>
dtrace	DTrace script events (BEGIN, END)
fbt	Function Boundary Tracing
io	Block I/O
<code>ip, udp, tcp, sctp</code>	TCP/IP
<code>lockstat</code>	Locking
proc, sched	Kernel process/scheduling
<code>profile</code>	Profiling timers
syscall	System call entry/return
vfs	Virtual filesystem

Aggregations

Aggregation	Description
<code>count()</code>	Number of times called
<code>sum()</code>	Sum of arguments
<code>avg()</code>	Average of arguments
<code>min()</code>	Minimum of arguments
<code>max()</code>	Maximum of arguments
<code>stddev()</code>	Standard deviation of nfts
<code>lquantize()</code>	Linear frequency distribution (histogram)
<code>quantize()</code>	Log frequency distribution (histogram)

- ▶ Often we want summaries of events, not detailed traces
- ▶ DTrace allows early, efficient *reduction* using aggregations
- ▶ Scalable multicore implementations (i.e., commutative)
- ▶ `@variable = function()`
- ▶ `printa()` to print

Counting kernel `read()` system calls

```
[guest@beaglebone ~/io]$ ./io-static -q -r /data/iofile
```

```
root@beaglebone:/data/io # dtrace -n
'syscall::read:entry
/execname=="io-static"/
{@reads = count(); }'
```

Probe Trace the `read` system call

Predicate Limit actions to processes executing `io-static`

Action Count the number of probe fires

Counting kernel `read()` system calls

```
[guest@beaglebone ~/io]$ ./io-static -q -r /data/iofile
```

```
root@beaglebone:/data/io # dtrace -n
'syscall::read:entry
/execname=="io-static"/
{@reads = count(); }'
```

Probe Trace the `read` system call

Predicate Limit actions to processes executing `io-static`

Action Count the number of probe fires

```
dtrace: description 'syscall::read:entry ' matched 1 probe
dtrace: buffer size lowered to 2m
dtrace: aggregation size lowered to 2m
^C
```

1024

Jupyter Notebooks

Unified environment for:

- ▶ Executing benchmarks.
- ▶ Measuring the performance of these benchmarks with DTrace.
- ▶ Post-processing performance measurements.
- ▶ Plotting performance measurements.
- ▶ Performing statistical analysis on performance measurements.

Jupyter Notebook overview

The screenshot shows a Jupyter Notebook interface. The browser address bar displays the IP address 192.168.141.100. The notebook title is "I41_lab1_template (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. The toolbar contains icons for saving, undo, redo, copy, paste, and running the cell. The code cell contains the following text:

```
In [ ]: # Execute the io-static benchmark displaying the command line options
!io/io-static
```

A tooltip above the run button says "run cell, select below".

- ▶ Series of cells containing Python or cell "magics".
- ▶ Cell magics allow inline plotting of graphs or executing shell commands.
- ▶ Raw data and plots can be save to BBB for inclusion in lab reports.
- ▶ Details of environment in lab1 handout.

A few cautions

There are two kinds of people, those that have experienced data loss and those that haven't experienced data loss **YET**.

- ▶ Copy key scripts and data files to/from your workstation.
- ▶ The SD cards seem a bit fragile during poweroff – make sure you shut down safely using the Lab Setup instruction.
- ▶ We have spare imaged SD cards if you need them.
- ▶ We may replace your SD cards for future labs.
- ▶ The Jupyter Notebook environment is new this year - teething troubles are expected!

A few other useful things

- ▶ Work in pairs for the lab (reports must be written separately)
- ▶ You will likely want multiple SSH sessions open
- ▶ The kernel source code is in `github/freebsd/freebsd.git` (branch `release/11.0.0`)
- ▶ Start with something simple – e.g., DTrace `hello world`
- ▶ Do not hesitate to ask for help if you need a hand!