L41: Lab 1- I/O

Lecturelet 1

Dr Robert Watson / Dr Graeme Jenkinson

2019-2020

L41: Lab 1 – I/O

- Introduce our experimental environment:
  - BeagleBone Black
  - FreeBSD operating system + DTrace
  - I/O benchmark
  - Jupyter notebooks
- Explore user-kernel interactions via syscalls and traps
- Engage with POSIX I/O and its implications
- Measure the probe effect

The platform

<u>TI BeagleBone Black</u>

- 1GHz ARM Cortex-A8 32-bit CPU
- Superscalar pipeline, MMU, L1/L2 caches
- FreeBSD operating system + DTrace
- Bespoke “potted benchmarks”
- Jupyter notebook measurement and analysis environment

DTrace scripts

- Human-facing (C/AWK inspired) language
- One or more {**probe name**, **predicate**, **action**} tuples
- Expression limited to control side effects (e.g. no loops)
- Specified on the 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
`callout_execute`	Timer-driven callouts
`dtmalloc`	Kernel `malloc()`/`free()`
`dtrace`	DTrace script events (`BEGIN`, `END`)
`fbt`	Function Boundary Tracing
`io`	Block I/O
`ip, udp, tcp, sctp`	TCP/IP
`lockstat`	Locking
`proc, sched`	Kernel process/scheduling
`profile`	Profiling timers
`syscall`	System call entry/return
`vfs`	Virtual filesystem

Aggregations

Aggregation	Description
`count()`	Number of times called
`sum()`	Sum of arguments
`avg()`	Average of arguments
`min()`	Minimum of arguments
`max()`	Maximum of arguments
`stddev()`	Standard deviation of arguments
`lquantize()`	Linear frequency distribution (histogram)
`llquantize()`	Log-linear frequency distribution (histogram)
`quantize()`	Log frequency distribution (histogram)

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

Counting kernel `read()` system calls

```
$ ./io-static -q -r /data/iofile
```

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

Probe name	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
```

The benchmark

```
$ ./io-static
io-static -c|-r|-w [-Bdqsv] [-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 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 `writes()`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)

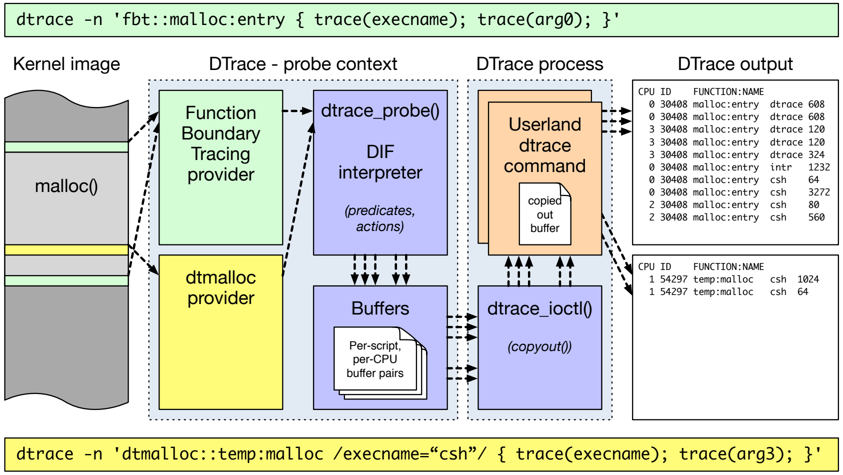
```
$ ./io-static -v -d -w /data/iofile
Benchmark configuration:
  blocksize: 16384
  totalsize: 16777216
  blockcount: 1024
  operation: write
  path: /data/iofile
  time: 58.502746875
280.06 KBytes/sec
```

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

Probe effect

- Probe effect - act of measuring disturbs system
  - Electronics - probes introduce additional capacitance, resistance or inductance
- Software tracing - probes take time to execute
  - Don't benchmark while running DTrace ...
  - ... unless **measuring probe effect**
  - Be aware that traced applications may behave differently
  - E.g., more timer ticks will fire, I/O will "seem faster"

Zero when disabled



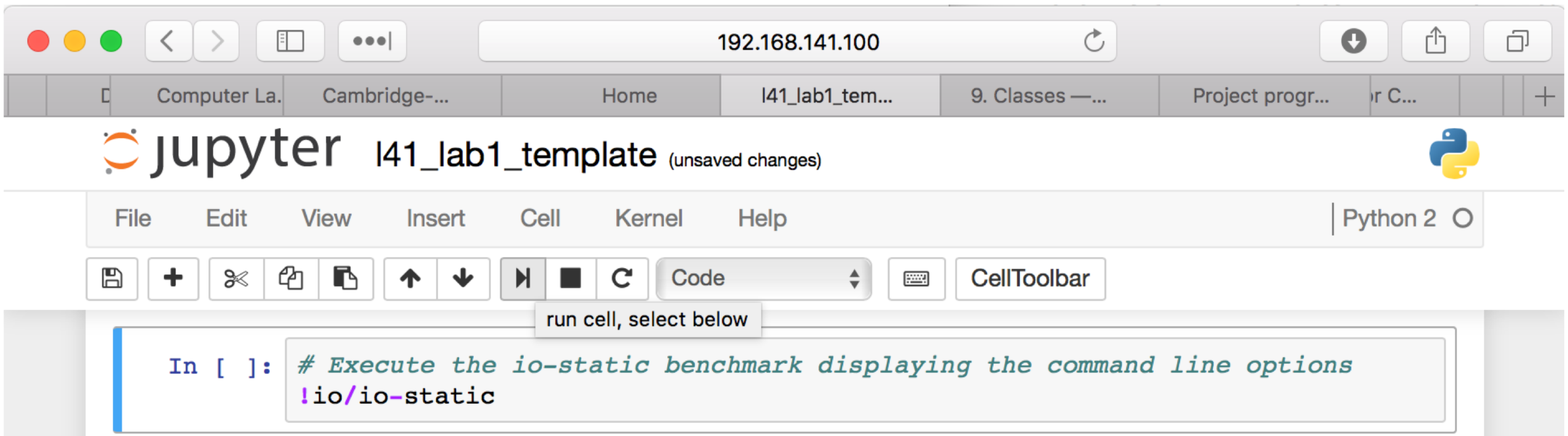
Definitely not zero

Jupyter notebooks

Unified environment for:

- Executing benchmarks.
- Instrumenting the behaviours and performance of benchmarks using DTrace.
- Post-processing performance measurements.
- Plotting performance measurements.
- Performing statistically analysis on performance measurements.

Jupyter notebooks (2)



- Series of cells containing Python or cell "magics".
- Cell magics allow, for example, inline plotting of graphs or executing shell commands.
- Raw data and plots can be saved to the BBB for inclusion in laboratory reports.
- Details of experimental environment in lab setup handout.

Hypotheses

- Larger I/O and IPC buffer sizes amortize system-call overheads
- A purely architectural (SW) view dominates
  - HW platform is irrelevant
- The DTrace probe effect is insignificant in real workloads

Experimental questions for the lab report

- With respect to a configuration **reading** from a fixed-size file through the buffer cache:
  - How does changing the I/O buffer size affect I/O-loop performance?
- 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
- Ensure that you directly consider the impact of the **probe effect** on your causal investigation

A few cautions

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

- The SD cards seem a bit fragile during power off – make sure that you shut down safely using the laboratory setup instructions.
  - We have spare imaged SD cards if you need them.
- Backup key scripts and data files on your workstation
  - We may replace your SD cards for future labs.

A few other useful things

- Feel free to work in pairs or groups in the lab:
  - Laboratory 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`).
- Experiment on the command line:
  - Start with something simple – e.g., DTrace `hello world`.
- Do not hesitate to ask for help.