Advanced Operating Systems:
Lab 1 – Getting Started with Kernel Tracing / I/O

Dr Robert N. M. Watson

2020-2021

The goals of this lab are to:

- Introduce you to our experimental environment including Jupyter and DTrace.
- Have you explore user-kernel interactions via system calls and traps.
- Gain experience tracing I/O behaviour in UNIX.
- Gain experience with performance analysis.
- Build intuitions about the probe effect.

You will do this by using DTrace to analyse the behaviour of a potted, kernel-intensive block-I/O benchmark.

1 Lab Report or Lab Assignment?

This document is accompanied by two specific lab assignments, one for Part II Advanced Operating Systems, and a second for Part III/ACS L41: Advanced Operating Systems. Please ensure that you are completing the correct assignment for your course, as they differ substantially.

2 Background: POSIX read(2) I/O system call

POSIX defines a number of synchronous I/O APIs, including the `read()` system call, which accept a file descriptor to a file or other storage object, a pointer to a buffer to read to, and a buffer length as arguments. You may wish to read the FreeBSD `read(2)` system-call manual page to learn more about the call before proceeding with this lab. You can do this using the `man` command; you may need to specify the *manual section* to ensure that you get API documentation rather than program documentation; for example, `man 2 read` for the system call to ensure that you do not get the `read(1)` program man page.

3 Hypotheses

In this lab, we provide you with two hypotheses that you will test and explore through benchmarking:

1. *System-call overhead is substantial, and so larger I/O buffer sizes will improve performance by reducing the number of system calls. This continues until we hit the system's peak I/O throughput, at which point performance will stabilise.*
2. *The probe effect associated with DTrace is negligible.*

We will test these hypotheses by measuring `read(2)` performance across a range of buffer size – both without, and with, DTrace instrumentation.

4 The benchmark

Our I/O benchmark is straightforward: it performs a series of `read()` or `write()` I/O system calls in a loop using configurable buffer and total I/O sizes. On our experiments, we will use only the `read()` variant. We will hold the total I/O size constant (i.e., the number of bytes read from disk), but vary the buffer size. While this seems a simple activity, and simplistic experiment, we will discover that it immediately triggers quite a deep investigation of OS behaviour. The benchmark application is able to collect a number of pieces of information about its own behaviour:

- OS and architectural configuration information
- Wall-clock execution time from start to end of the I/O loop, and hence also average I/O bandwidth.
- `rusage` information,

You can further instrument its behaviour using DTrace to collect further information, such as tracing system calls and their arguments, profiling kernel execution, and so on.

The lab bundle will build `io-benchmark`, a dynamically linked version of the benchmark, which you will use for all experiments. The benchmark performs some activities to reduce noise in results, such as initialising itself, pinning its process to a single CPU, and doing a short `sleep(3)` call before commencing measurement. The benchmark can be figured to run multiple loops, reporting on each; you may wish to discard the first result.

5 The UNIX command line

All commands will be run as the root user. Example command lines are prefixed with the `#` symbol signifying the shell prompt; you should type in only text after the prompt.

5.1 Compiling the benchmark

The laboratory I/O benchmark source code has been preinstalled onto your RPi4 board. However, you will need to build it before you can begin work. Once you have logged into your RPi4 (see *Advanced Operating Systems: Lab Setup*), build the bundle:

```
# make -C io
```

5.2 Benchmark arguments

If run without any parameters, the benchmark will list its potential arguments and default settings:

```
usage: io-benchmark -c|-r|-w [-Bdjqsv] [-b buffersize]
    [-n iterations] [-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
    -g              Enable getrusage(2) collection
    -j              Output as JSON
    -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 buffersize   Specify the buffer size (default: 16384)
    -n iterations   Specify the number of times to run (default: 1)
    -t totalsize    Specify the total I/O size (default: 16777216)
```

5.3 Running the benchmark

Once built, you can run the benchmark binary as follows, with command-line arguments specifying various benchmark parameters:

```
# io/io-benchmark
```

The benchmark will be run in one of two operational modes: `create` or `read`, specified by flags. In addition, the target file must be specified. If you run the `io-benchmark` benchmark without arguments, a small usage statement will be printed, which will also identify the default buffer and total I/O sizes configured for the benchmark.

In your experiments, you will need to be careful to hold most variables constant in order to isolate the effects of specific variables; for example, you may wish to hold the total I/O size constant as you vary the buffer size. You may wish to experiment initially using `/dev/zero` – the kernel's special device node providing an unlimited source of zeros, but will also want to run the benchmark against a file in the filesystem.

5.4 Required operation flags

Your command line must specify the mode in which the benchmark should operate:

-c Create the specified data file using the default (or requested) total I/O size – run exactly once, with a filename such as `iofile`.

-r Benchmark `read()` of the target file, which must already have been created.

5.5 Optional I/O flags

-b *buffersize* Specify an alternative buffer size in bytes. The total I/O size must be a multiple of buffer size.

-g Collect `getrusage()` statistics, such as sampled user and system time, as well as block I/O statistics.

-j Generate output as JSON, allowing it to be more easily imported into the Jupyter Lab framework, as well as other data-processing tools.

-n Specify the number of times to run the benchmark loop, reporting on each independently.

-t *totalsize* Specify an alternative total I/O size in bytes. The total I/O size must be a multiple of buffer size.

5.6 Terminal output flags

The following arguments control terminal output from the benchmark; remember that output can substantially change the performance of the system under test, and so you should ensure that output is either entirely suppressed during tracing and benchmarking, or that tracing and benchmarking only occurs during a period of program execution unaffected by terminal I/O:

-v *Verbose mode* causes the benchmark to print additional information, such as the time measurement, buffer size, and total I/O size.

5.7 Example benchmark commands

This command creates a default-sized data file in the `/data` filesystem:

```
# io/io-benchmark -c iofile
```

This command runs a simple `read()` benchmark on the data file, printing additional information about the benchmark run:

```
# io/io-benchmark -v -r iofile
```

To better understand kernel behaviour, you may also wish to run the benchmark against `/dev/zero`, a pseudo-device that returns all zeroes, and discards all writes:

```
# io/io-benchmark -r /dev/zero
```

If you enable `getrusage(2)` collection, the benchmark will report on wall-clock, user, and system time:

```
# io/io-benchmark -r -g /dev/zero
```

During performance analysis, you will primarily want to run the benchmark using a command line such as the following:

```
# io/io-benchmark -r -g -j -n 2 iofile
{
  "benchmark_samples": [
    {
      "bandwidth": 977330.20,
      "time": "0.016764037",
      "utime": "0.000000",
      "stime": "0.016774",
      "inblock": 0,
      "oublock": 0
    },
    {
      "bandwidth": 981186.65,
      "time": "0.016698149",
      "utime": "0.000000",
      "stime": "0.016709",
      "inblock": 0,
      "oublock": 0
    }
  ]
}
```

This run of `io-benchmark` reads its data from `iofile`, runs the benchmark loop twice, captures additional `getrusage` information, and prints the results in JSON. You will notice that the wall-clock execution time (`time`) is slightly more than the sum of user time (`utime`) and system time (`stime`). This imprecision likely occurs for two reasons: (1) wall-clock time is measured using a precise clock, and the `utime` and `stime` metrics are gathered via sampling; and (2) the two sets of data can't be collected atomically as a single system call, so `getrusage` information includes execution time to collect the wall-clock timestamp.

6 Files you can run the benchmark on

You may wish to point the benchmark at one of two objects:

`/dev/zero` The zero device: an infinite source of zeroes, and also an infinite sink for any data you write to it.

`/data/iofile` A writable file in a journalled UFS filesystem on the SD card. You will need to create the file using `-c` before performing `read()` benchmark.

7 Jupyter

JupyterLab is web-based data analysis and presentation tool structured around the idea of a "Notebook", or collection of cells containing text, code, and output such as plots and data. In Advanced Operating Systems, we will be running JupyterLab on our RPi4 boards with the Python programming language to orchestrate benchmark execution, as well as data collection, analysis, and presentation.

This laboratory work requires students to bring together a diverse range of skills and knowledge: from shell commands, scripting languages and DTrace to knowledge of microarchitectural features and statistical analysis. The course's aim is to focus on the intrinsic complexity of the subject matter and not the *extrinsic* complexity resulting from integrating disparate tools and platforms. JupyterLab supports this goal by providing a 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.

Further information about the Jupyter Notebooks can be found at the project's website: `jupyter.org`.

For Part III/ACS L41 students, you will take output from your JuputerLab notebook and incorporate it into your submitted lab report. For Part II students, you will print the JupyterLab notebook to a PDF, which you will submit.

7.1 Template

The RPi4 comes preinstalled with a template Jupyter Notebook `2020-2021-l41-lab1.ipynb`. This template is designed to give working examples of all the features necessary to complete the first laboratory: including measuring the performance of the benchmarks using DTrace, producing simple graphs with `matplotlib` and performing basic statistics on performance measurements using `pandas`. Details of working with the Jupyter Notebook template are given in the *Advanced Operating Systems: Lab Setup* guide.

8 Notes on using DTrace

You will want to use DTrace to analyse just the program I/O loop. It is useful to know that the system call `clock_gettime` is both run immediately before, and immediately after, the I/O loop. If you configure the benchmark for a single execution (`-n 1`), then you can bracket tracing between a return probe for the former, and an entry probe for the latter. For example, you might wish to include the following in your DTrace scripts:

```
syscall::clock_gettime:return
/execname == "io-benchmark" && !self->in_benchmark/
{
    self->in_benchmark = 1;
}

syscall::clock_gettime:entry
/execname == "io-benchmark" && self->in_benchmark/
{
    self->in_benchmark = 0;
}

probe:of:your:choice
/execname == "io-benchmark" && self->in_benchmark/
{
    /* Only perform this data collection if running within the benchmark. */
}

END
{
    /* Print summary statistics here. */
    exit(0);
}
```

Other DTrace predicates can then refer to `self->in_benchmark` to determine whether the probe is occurring during the I/O loop. This bracketing will help prevent the inclusion, for example, of `printf()` execution in your traces. The benchmark is careful to set up the run-time environment suitably before performing its first clock read, and to perform terminal output only after the second clock read, so it is fine to leave benchmark terminal output enabled.