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

Prof. Robert N. M. Watson

2021-2022

The goals of this lab are to:

- Introduce our experimental environment including Jupyter Labs and DTrace.
- Explore user-kernel interactions via system calls and traps.
- Introduce performance analysis methodology.
- Measure and explore the probe effect.

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

1 Assignment documents

This document provides Lab 1 information common to the Part II and L41 variations of this course. All students will also want to read *Advanced Operating Systems: Lab Setup*, which provides information on the lab platform and how to get started.

Part II students should perform the assignment found in *Advanced Operating Systems: Lab 1 – Getting Started with Kernel Tracing / I/O – Part II Assignment*.

L41 students should perform the assignment found in *Advanced Operating Systems: Lab 1 – Getting Started with Kernel Tracing / I/O – L41 Assignment*. They should follow the lab-report guidance found in *L41: Lab Reports*, and use the lab-report LaTeX template, `l41-labreport-template.tex`.

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

POSIX defines several 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. Start by reading the FreeBSD `read(2)` system-call manual page by running `man 2 read` on your Raspberry Pi, or via the FreeBSD web site.

3 Hypotheses

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

1. *System-call overhead is substantial, and so larger system-call buffer sizes will improve performance by amortizing that overhead.*
2. *Perform growth continues until we hit the system's peak I/O throughput, at which point performance will stabilise.*

3. *The probe effect associated with DTrace is negligible.*

We will test these hypotheses by measuring `read(2)` performance using a fixed total data size utilizing 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()` system calls in a loop using configurable buffer and total I/O sizes. 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 simplistic experiment, we will quickly discover that it illustrates initially surprising behaviours, and triggers 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), and
- Various kernel statistics about the process captured via `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. Whenever we use DTrace and other instrumentation tools, we should be aware of the possibility that they may introduce a probe effect.

The lab bundle builds `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.

You may notice that the benchmark supports other modes of operation (e.g., using the `write()` system call); for the purposes of this assignment, please use only the functionality required for the assignment.

5 Getting started

Before you start, please review the *Advanced Operating Systems: Lab Setup* document, including how to log into your RPi4, how to extract the first lab bundle, and how to run JupyterLab.

It is possible to run the following commands from both the UNIX shell prompt, and also from within JupyterLab. For your labs, we generally recommend the latter. Either way, all commands will be run aas the root user. Example command lines are prefixed with the `#` symbol signifying the shell 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 and extracted the bundle, build it from the command line or JupyterLab:

```
# 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 [-Bdjqsv] [-b buffersize] [-n iterations] [-t totalsize]
    create|describe|read|write path

Modes:
    create      Create the specified I/O data file
    describe    Describe the hardware, OS, and benchmark configurations
    read        Perform the read benchmark variant
    write       Perform the write benchmark variant
```

```
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
```

For this assignment, you will run the benchmark in three of its operational modes: `create`, `describe`, and `read`. 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 Benchmark mode

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

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

describe Display information about the current harware, OS, and benchmark configuration.

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

5.5 Benchmark configuration flags

These flags configure benchmarking and data collection:

-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.

-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 Output flags

The following arguments control output from the benchmark:

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

-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 create iofile
```

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

```
# io/io-benchmark -v read 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 -v read /dev/zero
```

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

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

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

```
# io/io-benchmark -g -j -n 2 read iofile
{
  "benchmark_samples": [
    {
      "bandwidth": 955217.85,
      "time": "0.017152108",
      "utime": "0.008584",
      "stime": "0.008584",
      "nvcsw": 1,
      "nivcsw": 0,
      "inblock": 0,
      "oublock": 0
    },
    {
      "bandwidth": 946291.53,
      "time": "0.017313903",
      "utime": "0.001522",
      "stime": "0.015804",
      "nvcsw": 1,
      "nivcsw": 0,
      "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 for input into Python.

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.

5.8 Assignment parameters

For the purposes of this assignment, please:

- Hold the total I/O size (`totalsize`) constant at 16MiB.
- Consider power-of-two buffer sizes (`buffersize`) values from 32 bytes to 16MiB (inclusive).
- Use only the `read` benchmark, and not the `write` benchmark.
- Disregard the `-B` and `-s` arguments.

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.

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

7 Further notes

Quiescing system state Ensure that your experimental setup quiesces other activity on the system, and uses a suitable number of benchmark runs. Drop the first run of each set, which may experience one-time startup expenses, such as loading pages of the benchmark from disk.

Presenting plots Carefully label all axes, lines, etc., in plots, and take care to ensure legibility. Use logarithmic scaling of X axes representing buffer size; do ensure that all plots have the same X axis so that they can be visually compared more easily. When describing plots, consider partitioning them based on key inflection points, and explaining what each region (and transition) represents.

Experiments to run Although the benchmark contains a number of modes and further options, use only the modes specifically identified in the assignment for your work. For example, please do not evaluate `write()` behaviour, as that work will not be marked or assessed, and may distract from the assignment goals.

Benchmark execution time The benchmark can run for a considerable period of time – especially if we are scanning a parameter space, and using multiple runs. You may wish to initially experiment using a smaller number (e.g., 3) and get tea. For the final measurement, a larger number is desirable (e.g., 7). For short runs, plan on a cup of tea. For long runs, plan on having dinner.

8 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 perform 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:

`https://jupyter.org`

For Part III/ACS L41 students, you will take output from your JupyterLab 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.

8.1 Template

The RPi4 comes preinstalled with a template Jupyter Notebook `2021-2022-advopsys-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.

9 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->done/
{
    self->in_benchmark = 1;
}

syscall::clock_gettime:entry
/execname == "io-benchmark" && self->in_benchmark && !self->done/
{
    self->in_benchmark = 0;
    self->done = 1;
}

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.