Advanced Operating Systems: Lab 1 – Getting Started with Kernel Tracing **Part II and ACS/Part III Assignment**

Prof. Robert N. M. Watson

2023-2024

This first lab introduces you to DTrace, asking you to take a series of kernel measurements and present the results in a JupyterLab Notebook. You will step through a set of example DTrace and Python scripts collecting and illustrating results from a simple `dd(1)` workload. To do this:

- Complete a set of short exercises set out in a Jupyter Lab Notebook, building on those examples.
- Capture data using several different DTrace *providers* including *systrace* (system-call tracing), and *profile* (timer-based profiling).
- Present this data using standard Python plotting mechanisms as well as Flame Graphs.
- Submit a generated PDF of these results for assessment and comments; to do this, use Jupyter's File → Export Notebook As → PDF. **This takes quite a long time on a Raspberry Pi – 30 seconds or more – please be patient while you wait for the PDF to download**.

This will teach you skills that will support your future labwork.

1 Assignment documents

All students should read *Advanced Operating Systems: Lab Setup*, which provides information on the lab platform and how to get started.

2 Background: DTrace

DTrace, or Dynamic Tracing, is an OS feature to dynamically instrument software behaviour on operating systems such as FreeBSD, macOS, Windows, and some versions of Linux. As explored in Lecture 2, DTrace is highly programmable, with its D programming language used to specify instrumentation points, conditions for data capture, and specific actions to take when those conditions hold. You can refer to *Advanced Operating Systems: DTrace Quick Start* for a quick tutorial to get you started, lecture slides and recordings, and the original research paper by Cantrill et al.

3 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 as the root user. Example command lines are prefixed with the `#` symbol signifying the shell prompt.

4 The JupyterLab notebook template

We have populated the directory /advopsys-packages/labs with the Lab 1 JupyterLab tarball, which you will unpack into the /data directory. You will also use /data as the working directory for JupyterLab.

```
# cd /data
# tar -xzf /advopsys-packages/labs/2023-2024-advopsys-lab1.tbz
```

This will extract the file 2023-2024-advopsys-lab1.ipynb, which you will use to explore example tracing and plotting scripts that we have prepared for you.

5 The assignment

In this exercise, we use the command-line tool dd(1), as also shown in Lecture 2, to explore some of the tracing techniques available via DTrace. We will then take those DTrace results and analyse them in a JupyterLab Notebook to build skills with that framework and its integration with DTrace, as well as Python plotting tools. The specific dd(1) command line we will use in this lab is:

```
# dd if=/dev/zero of=/dev/null bs=8k count=5000
```

This command line asks dd(1) to transfer 5,000 blocks of size 8KiB from /dev/zero (and synthetic kernel device node providing an endless source of zeroes) to /dev/null (another synthetic kernel device node that discards any data written to it).

In later labs, we will use bespoke microbenchmarks as the targets of analysis; these also contain internal instrumentation to report performance data as well as collect statistics on their operation. This will provide more precise bracketing of the benchmark behaviour, as well as easy access to I/O or IPC statistics already captured by the kernel, and with lower probe effect than with DTrace.

5.1 Example executions, tracing, and plotting

First, step through the examples in the Lab 1 JupyterLab notebook that we provide. These explore how to execute the benchmark from within JupyterLab, using DTrace from both the command line and within Python, how to plot with Matplotlib and our own Flamegraph module, and several potential uses of DTrace to collect data about a running program including system-call tracing and timer-based profiling.

5.2 Exercises

Start by creating a new JupyterLab notebook clearly labeled with the current exercise number, name, your own name, and the date. Now complete the following exercises, including your D script with each piece of generated output, allowing us to see how you collected the data that you are presenting. Alongside your code blocks tracing the workload and presenting your results, use markdown blocks to both label which exercise is being addressed, and also to describe your work and conclusions. Ensure that plots are suitably labeled with units / scale, dataset labels, and indications of key thresholds or inflection points. Marks are directly awarded for clear, succinct presentation of your work.

1. Plot two histograms, using matplotlib, on a single plot that compares the distribution of system-call wall-clock execution times for the read(2) and write(2) system calls for our dd(1) workload. To do this, instrument the system-call entry and return probes for the system call; use the timestamp DTrace variable[1] to capture time stamps. [2] Describe the data sets, and explain the results.

2. Plot two flamegraphs profiling stack traces executed while being in the read system call handler sys_read using the profile-4997 probe, once for our dd(1) workload with the default count of 5,000 blocks, and a second time with a modified count of 500 blocks. Explain how and why the proportional execution time of various kernel functions may differ between these two similar workloads.

[1]Note that DTrace supports multiple clocks; we recommend timestamp as it offers a precise timestamp with good granularity. Other timestamp variables may not offer the precision required for this exercise.

[2]If the two distributions appear identical, double check your code, and that you are using the right timestamp variable.

As mentioned in Lecture 1, Part 3: Kernel dynamics, the source code of the kernel is placed in `/usr/src/sys` on your board. You can use that source code to analyse the stack traces collected with DTrace to understand why they were executed.