# L41: Lab 2 - Kernel Implications of IPC

Dr Robert N. M. Watson Dr Graeme Jenkinson

Michaelmas Term 2017

The goals of this lab are to:

- Continue to gain experience tracing user-kernel interactions via system calls and traps.
- Explore the performance of varying IPC models, buffer sizes, and process models.
- Gather data to support writing your first assessed lab report.

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

## Background: POSIX IPC objects

POSIX defines several types of Inter-Process Communication (IPC) objects, including pipes (created using the `pipe()` system call) and sockets (created using the `socket()` and `socketpair()` system calls).

**Pipes** are used most frequently between pairs of processes in a UNIX *process pipeline*: a chain of processes started by a single command line, whose output and input file descriptors are linked. Although pipes can be set up between unrelated processes, the primary means of acquiring a pipe is through inheritance across `fork()`, meaning that they are used between closely related processes (e.g., with a common parent process).

**Sockets** are used when two processes are created in independent contexts and must later rendezvous – e.g., via the filesystem, but also via TCP/IP. In typical use, each endpoint process creates a socket via the `socket()` system call, which are then interconnected through use of `bind()`, `listen()`, `connect()`, and `accept()`. However, there is also a `socketpair()` system call that returns a pair of interconnected endpoints in the same style as `pipe()` – convenient for us as we wish to compare the two side-by-side.

Both pipes and sockets can be used to transmit ordered byte streams: a sequence of bytes sent via one file descriptor that will be received reliably on the other without loss or reordering. As file I/O, the `read()` and `write()` system calls can be used to read and write data on file descriptors for pipes and sockets. It is useful to know that these system calls are permitted to return *partial reads* and *partial writes*: i.e., a buffer of some size (e.g., 1k) might be passed as an argument, but only a subset of the requested bytes may be received or sent, with the actual size returned via the system call's return value. This may happen if the in-kernel buffers for the IPC object are too small for the full amount, or if *non-blocking I/O* is enabled. When analysing traces of IPC behaviour, it is important to consider both the size of the buffer passed and the number of bytes returned in evaluating the behaviour of the system call.

You may wish to read the FreeBSD `pipe(2)` and `socketpair(2)` manual pages to learn more about these APIs before proceeding with the lab.

## The benchmark

As with our earlier I/O benchmark, the IPC benchmark is straightforward: it sets up a pair of IPC endpoints referencing a shared pipe or socket, and then performs a series of `write()` and `read()` system calls on the file descriptors to send (and then receive) a total number of bytes of data. Data will be sent using a smaller userspace buffer size – although as hinted above, there is no guarantee that a full user buffer will be sent or received in

any individual call. Also as with the I/O benchmark, there are several modes of operation: sending and receiving within a single thread, a pair of threads in the same process, or between two threads in two different processes.

The benchmark will set up any necessary IPC objects, threads, and processes, sample the start time using the `clock_gettime()` system call, perform the IPC loop (perhaps split over two threads), and then sample the finish time using the `clock_gettime()` system call. Optionally, both the average bandwidth across the IPC object, and also more verbose information about the benchmark configuration, may be displayed. Both statically and dynamically linked versions of the binary are provided: `ipc-static` and `ipc-dynamic`.

## Compiling the benchmark

The laboratory IPC benchmark has been preinstalled onto the BeagleBone Black (BBB) SD card image. However, you will need to build it before you can begin work. Once you have configured the BBB so that you can log in (see *L41: Lab Setup*), you can build the benchmark as follows:

```
# cd /data
# make -C io
```

## Running the benchmark

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

```
# ipc/ipc-static
```

or:

```
# ipc/ipc-dynamic
```

If you run the benchmark without arguments, a small usage statement will be printed, which will also identify the default IPC object type, IPC buffer, and total IPC sizes configured for the benchmark. As in the prior lab, you will wish to be careful to hold most variables constant in order to isolate the effects of specific variables. For example, you might wish the vary the IPC object type while holding the total IPC size constant.

## Required operation argument

Specify the mode in which the benchmark should operate:

**1thread** Run the benchmark entirely within one thread; note that, unlike other benchmark configurations, this mode interleaves the IPC calls and must place the file descriptors into non-blocking mode or risk deadlock. This may have observable effects on the behaviour of the system calls with respect to partial reads or writes.

**2thread** Run the benchmark between two threads within one process: one as a 'sender' and the other as a 'receiver', with the sender capturing the first timestamp, and the receiver capturing the second. System calls are blocking, meaning that if the in-kernel buffer fills during a `write()`, then the sender thread will sleep; if the in-kernel buffer empties during a `read()`, then the receiver thread will sleep.

**2proc** As with the `2thread` configuration, run the benchmark in two threads – however, those threads will be in two different processes. The benchmark creates a second process using `fork()` that will run the sender. System calls in this variation are likewise blocking.

## Optional I/O flags

**-b *buffersize*** Specify an alternative userspace IPC buffer size in bytes – the amount of memory allocated to hold to-be-sent or received IPC data. The same buffer size will be used for both sending and receiving. The total IPC size must be a multiple of buffer size.

**-i *ipctype*** Specify the IPC object to use in the benchmark: `pipe`, `local`, or `tcp` (default `pipe`).

**-t *totalsize*** Specify an alternative total IPC size in bytes. The total IPC size must be a multiple of userspace IPC buffer size.

**-B** Run in `bare mode`: disable normal quiescing activities such as using `sync()` to cause the filesystem to synchronise before the IPC loop runs, and using `sleep()` to await terminal-I/O quietude. This will be the more appropriate mode in which to perform whole-program analysis but may lead to greater variance if simply analysing the IPC loop.

**-s** When operating on a socket, explicitly set the in-kernel socket-buffer size to match the userspace IPC buffer size rather than using the kernel default. Note that per-process resource limits will prevent use of very large buffer sizes.

### 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 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:

**-q** *Quiet mode* suppress all terminal output from the benchmark, which is preferred when performing whole-program benchmarking.

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

### Example benchmark commands

This command performs a simple IPC benchmark using a pipe and default userspace IPC buffer and total IPC sizes within a single thread of a single process:

```
# ipc/ipc-static -i pipe 1thread
```

This command performs the same pipe benchmark, but between two threads of the same process:

```
# ipc/ipc-static -i pipe 2thread
```

And this command does so between two processes:

```
# ipc/ipc-static -i pipe 2proc
```

This command performs a socket-pair benchmark, and requests non-default socket-buffer sizes synchronised to a userspace IPC buffer size of 1k:

```
# ipc/ipc-static -i local -s -b 1024 2thread
```

As with the I/O benchmark, additional information can be requested using *verbose mode*:

```
# ipc/ipc-static -v -i pipe 1thread
```

And, likewise, all output can be suppressed, and *bare mode* can be used, for whole-program analysis:

```
# ipc/ipc-static -q -B -i pipe 1thread
```

## Note on kernel configuration

By default, the kernel limits the maximum per-socket socket-buffer size that can be configured, in order to avoid resource starvation. You will need to tune the kernel's default limits using the following command, run as root, prior to running benchmarks. Note that this should be set before any benchmarks are run, whether or not they are explicitly configuring the socket-buffer size, as the limit will also affect socket-buffer auto-sizing.

```
# sysctl kern.ipc.maxsockbuf=33554432
```

## Notes on using DTrace

On the whole, this lab will be concerned with just measuring the IPC loop, rather than whole-program behaviour. As in the last lab, it is useful to know that the system call `clock_gettime` is both run immediately before, and immediately after, the IPC loop. In this benchmark, these events may occur in different threads or processes, as the sender performs the initial timestamp before transmitting the first byte over IPC, and the receiver performs the final timestamp after receiving the last byte over IPC. You may wish to bracket tracing between a return probe for the former, and an entry probe for the latter; see the notes from the last lab for an example.

As with the last lab, you will want to trace the key system calls of the benchmark: `read()` and `write()`. For example, it may be sensible to inspect `quantize()` results for both the execution time distributions of the system calls, and the amount of data returned by each (via `arg0` in the system-call return probe). You will also want to investigate scheduling events using the `sched` provider. This provider instruments a variety of scheduling-related behaviours, but it may be of particular use to instrument its `on-cpu` and `off-cpu` events, which reflect threads starting and stopping execution on a CPU. You can also instrument `sleep` and `wakeup` probes to trace where threads go to sleep waiting for new data in an empty kernel buffer (or for space to place new data in a full buffer). When tracing scheduling, it is useful to inspect both the process ID (`pid`) and thread ID (`tid`) to understand where events are taking place.

By its very nature, the probe effect is hard to investigate, as the probe effect does, of course, affect investigation of the effect itself! However, one simple way to approach the problem is to analyse the results of performance benchmarking with and without DTrace scripts running. When exploring the probe effect, it is important to consider not just the impact on bandwidth average/variance, but also on systemic behaviour: for example, when performing more detailed tracing, causing the runtime of the benchmark to increase, does the number of context switches increase, or the distribution of `read()` return values? In general, our interest will be in the overhead of probes rather than the overhead of terminal I/O from the DTrace process – you may wish to suppress that output during the benchmark run so that you can focus on probe overhead.

## Notes on benchmark

As with the prior lab, it is important to run benchmarks more than once to collect a distribution of values, allowing variance to be analysed. You may wish to discard the first result in a set of benchmark runs as the system will not yet have entered its steady state. Do be sure that terminal I/O from the benchmark is not included in tracing or time measurements (unless that is the intent).

## Experimental questions (part 1/2)

You will receive a separate handout during the next lab describing *Lab Report 2*; however, this description will allow you to begin to prepare for the assignment, which will also depend on the outcome of the next lab. Your lab report will compare several configurations of the IPC benchmark, exploring (and explaining) performance differences between them. Do ensure that your experimental setup suitably quiesces other activity on the system, and also use a suitable number of benchmark runs; you may wish to consult the *FreeBSD Benchmarking Advice* wiki page linked to from the module's reading list for other thoughts on configuring the benchmark setup. The following questions are with respect to a fixed total IPC size with a statically linked version of the benchmark, and refer only to IPC-loop, not whole-program, analysis. Using `2thread` and `2proc` modes, explore how varying IPC model (pipes, sockets, and sockets with `-s`) and IPC buffer size affect performance:

- How does increasing IPC buffer size uniformly change performance across IPC models – and why?
- Is using multiple threads faster or slower than using multiple processes?

Graphs and tables should be used to illustrate your measurement results. Ensure that, for each question, you present not only results, but also a causal explanation of those results – i.e., why the behaviour in question occurs, not just that it does. For the purposes of graphs in this assignment, use achieved bandwidth, rather than total execution time, for the Y axis, in order to allow you to more directly visualise the effects of configuration changes on efficiency.