Advanced Operating Systems:
Lab 2 – IPC
General Information

Prof. Robert N. M. Watson

2021-2022

The goals of this lab are to:

- Continue to gain experience tracing user-kernel interactions via system calls and traps.
- Explore the implementations and performance of UNIX pipe IPC, and its optimisation using VM page borrowing.
- Use DTrace and hardware performance counters (HWPMC) to analyse these properties.
- Generate data to complete the second lab assignment.

You will do this by using DTrace and HWPMC to analyse the behaviour of our kernel-intensive IPC benchmark.

1 Assignment documents

This document provides Lab 2 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 2 – IPC – Part II Assignment*.

L41 students should perform the assignment found in *Advanced Operating Systems: Lab 2 – IPC – L41 Assignment*. Follow the lab-report guidance found in *L41: Lab Reports*, and use the lab-report LaTeX template, `l41-labreport-template.tex`.

2 Background: POSIX IPC objects

Pipes are an IPC primitive most frequently used 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).

Pipes are used to transmit ordered byte streams: a sequence of bytes sent via one file descriptor that will be received reliably on another file descriptor without loss or reordering. As with file I/O, the `read()` and `write()` system calls can be used to read and write data on file descriptors for pipes.

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., 1KiB) 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, due to object-specific implementation choices, 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)` manual page to learn more about these APIs before proceeding with the lab. This is installed on your RPi board, and can be read using the command `man 2 pipe`.

2.1 Virtual-memory optimisations

Pipe IPC normally involves two memory copies: once copying data from the sending process into kernel buffer (using `copyin()`), and a second time copying from the kernel buffer into the receiving process (using `copyout()`). Contemporary UNIX implementations use virtual-memory page-borrowing optimisations to eliminate the sender-side copy by borrowing the sender page. However, there is a trade-off: There are also overheads associated with virtual-memory management, as well as other behavioural changes required to implement this technique. The kernel therefore has a heuristic threshold based on message size, which, by default, enables VM optimisations only when messages are 8KiB or above. You can query and set the threshold using the `sysctl` management tool:

```
# sysctl kern.ipc.pipe_mindirect
kern.ipc.pipe_mindirect: 8192
# sysctl kern.ipc.pipe_mindirect=16384
kern.ipc.pipe_mindirect: 8192 -> 16384
```

This can be done within JupyterLab using the `!` operator in the same way that we execute the benchmark itself.

3 Hypotheses

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

1. *Larger pipe buffer sizes improve IPC performance.*
2. *Page-borrowing virtual-memory optimisations:*
   (a) *Are well tuned to current microarchitectures; and*
   (b) *Always achieve a performance improvement, when enabled at or above the default 8KiB threshold, over the unoptimised baseline.*
3. *The probe effect associated with using HWPMC to explore this workload is negligible.* (L41 only)

We will test these hypotheses by measuring net throughput between two IPC endpoints in two different threads. We will use DTrace and HWPMC to establish the causes of divergence from these hypotheses, and to explore the underlying implementations leading to the observed performance behaviour. For L41 only, we will consider the probe effect of HWPMC in the lab, as we did for DTrace.

4 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, 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 configured 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.

The benchmark will set up IPC objects and processes, sample the start time using `clock_gettime()`, perform the IPC loop, and then sample the finish time using `clock_gettime()`. Optionally, both the average bandwidth across the IPC object, and also more verbose information about the benchmark configuration, may be displayed. A single, dynamically linked binary will be used in this lab: `ipc-benchmark`.

5 Getting started

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.

5.1 Obtaining the benchmark

The laboratory IPC benchmark source code has been preinstalled onto your RPi4 board. Once you have logged into your RPi4 (see *Advanced Operating Systems: Lab Setup*), you can find it in:

`/advopsys/labs/2021-2022-advopsys-lab2.tbz`.

We recommend untarring this file into the `/data` directory on your board:

```
# cd /data ; tar -xzvf /advopsys/labs/2021-2022-advopsys-lab2.tbz
```

5.2 Compiling the benchmark

Once you have obtained the benchmark, you need to compile it before you can begin work. Log into your RPi4 (see *Advanced Operating Systems: Lab Setup*) and build the bundle:

```
# make -C ipc
```

5.3 Benchmark arguments

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:

```
# ipc/ipc-benchmark
ipc-benchmark [-Bgjqsv] [-b buffersize] [-i pipe|local|tcp] [-n iterations]
    [-p tcp_port] [-P arch|dcache|instr|tlbmem] [-t totalsize] mode

Modes (pick one - default 1thread):
    1thread     IPC within a single thread
    2thread     IPC between two threads in one process
    2proc       IPC between two threads in two different processes
    describe    Describe the hardware, OS, and benchmark configurations

Optional flags:
    -B                      Run in bare mode: no preparatory activities
    -g                      Enable getrusage(2) collection
    -i pipe|local|tcp       Select pipe, local sockets, or TCP (default: pipe)
    -j                      Output as JSON
    -p tcp_port             Set TCP port number (default: 10141)
    -P arch|dcache|instr|tlbmem  Enable hardware performance counters
    -q                      Just run the benchmark, don't print stuff out
    -s                      Set send/receive socket-buffer sizes to buffersize
    -v                      Provide a verbose benchmark description
    -b buffersize           Specify the buffer size (default: 131072)
    -n iterations           Specify the number of times to run (default: 1)
    -t totalsize            Specify the total I/O size (default: 16777216)
```

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 will need to vary the `kern.ipc.pipe_mindirect` and buffer size while holding the total size constant.

In addition to a set of arguments specifying parameters for the benchmark itself, which will feel familiar from the prior lab, there is a new argument (`-P`) to request that hardware performance counters be measured around the benchmark run.

5.4 Running the benchmark

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

```
# ipc/ipc-benchmark
```

For this assignment, you will run the benchmark in only one of its operational modes (`2proc`) and IPC types (`pipe`). You will vary buffer size and the virtual-memory optimisation threshold, `kern.ipc.pipe_mindirect`.

5.5 Benchmark mode

While this benchmark supports multiple modes of operation, this lab will use only one mode:

2proc Run the benchmark between two processes: 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 until there is space; if the in-kernel buffer empties during a `read()`, then the receiver thread will sleep until there is data to read.

5.6 Benchmark configuration flags

These flags configure benchmarking and data collection:

-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.

-g Collect `getrusage()` statistics, such as sampled user and system time, as well as message send and receive statistics.

-i *ipctype* Specify the IPC object to use in the benchmark; for the purposes of this lab, specify only `pipe` (the default).

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

-P *mode* Enable performance counters across the IPC loop. See the document, *Advanced Operating System: Hardware Performance Counters (HWPMC)* for information on the available modes and their interpretation.

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

5.7 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 IPC size.

5.8 Example benchmark commands

This command performs a simple IPC benchmark using a pipe, default total IPC size, and 16KiB buffer between two processes:

```
# ipc/ipc-benchmark -i pipe -b 16384 2proc
```

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

```
# ipc/ipc-benchmark -v -i pipe 2proc
```

This command instructs the IPC benchmark to capture information on memory instructions issued when operating on a pipe with a 512-byte buffer from two processes:

```
# ipc/ipc-benchmark -i pipe -b 512 -P tlbmem 2proc
```

This command performs the same benchmark while tracking L1 data-cache and L2 cache hits and refills:

```
# ipc/ipc-benchmark -i pipe -b 512 -P dcache 2proc
```

This command performs the same benchmark while tracking architectural loads, stores, function returns, and exception returns:

```
# ipc/ipc-benchmark -i pipe -b 512 -P arch 2proc
```

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

```
# ipc/ipc-benchmark -g -j -n 2 -i pipe 2proc
{
  "benchmark_samples": [
    {
      "bandwidth": 1007235.62,
      "time": "0.016266303",
      "stime": "0.019953",
      "utime": "0.000000",
      "msgsnd": 128,
      "msgrcv": 256,
      "nvcsw": 262,
      "nivcsw": 256
    },
    {
      "bandwidth": 1013240.88,
      "time": "0.016169897",
      "stime": "0.019269",
      "utime": "0.000000",
      "msgsnd": 128,
      "msgrcv": 256,
      "nvcsw": 262,
      "nivcsw": 256
    }
  ]
}
```

This run of `ipc-benchmark` transfers 16MiB of data between two processes using pipe IPC, running the benchmark loop twice, collecting additional `getrusage` information, and prints the results in JSON for input into Python.

5.9 Assignment parameters

For the purposes of this assignment, please:

- Hold the total IPC size (`totalsize`) constant at 16MiB.
- Measure power-of-two buffer sizes (`buffersize`) values from 32 bytes to 16MiB (inclusive).
- Use only the `2proc` mode.
- Use only the `pipe` IPC type.
- Disregard the `-B`, `-p`, and `-s` arguments.

6 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 may 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.

7 Note on graphs in this lab assignment or lab report

Because of the large amounts of data (and number of data sets) explored in this lab, you will need to pay significant attention in writing your lab assignment or lab report to how you present data visually. Graphs should make visual arguments, and how a set of graphs are plotted can support (or confuse) that argument. Make sure all graphs are clearly presented with labels and textual descriptions helping the reader identify the points you think are important.

When two graphs have the same independent variable (e.g., buffer size), it is important that they use the same X axis in terms of labelling and scale – and ideally also visual layout. Graphs with the same X axis will often benefit from being arranged so that they align vertically stack on the page, such that inflection points can be visually compared. This might useful, for example, if attempting to argue that inflection points in microarchitectural counters (e.g., cache or TLB misses) relate to resulting performance change (e.g., in bandwidth).

The objective is that visual artifacts, such as convergence, divergence, or intersections of lines have meaning when interpreting the graph. We therefore also discourage combining data with multiple Y-axis interpretations in the same plot – instead, use adjacent plots. Be sure to clearly label all lines, utilize shading of regions, point symbols, and colours to ensure that related data is grouped visually, and unrelated data is clearly distinct. If you have trouble distinguishing the different data sets, then there are too many data sets on the graph.