Advanced Operating Systems:
Lab 2 – IPC
General Information

Prof. Robert N. M. Watson

2026-2027

The goals of this lab are to:

- Introduce performance analysis methodology.
- Explore user-kernel interactions via system calls and traps.
- Explore the implementations of, and tradeoffs between, UNIX pipe and shared-memory IPC.
- Use DTrace and hardware performance counters (HWPMC) to analyse these properties.
- Measure and explore the probe effect for DTrace.

You will analyse the behavior of a "potted" IPC-intensive benchmark with support for multiple IPC types and configurations.

1 Assignment documents

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

Please also be sure to read *Advanced Operating Systems: Lab Exercises*, which describes expectations for the submission as well as our lab assessment criteria / marking guidance.

2 Background: POSIX IPC objects

POSIX defines several types of Inter-Process Communication (IPC) objects, including *pipes* (created using the `pipe()` system call) and shared memory (set up via the `minherit()` system call).

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).

Shared memory allows two processes to exchange data via memory load and store operations against a single physical page mapped into both of their address spaces, avoiding entering the kernel to communicate. A variety of APIs are available to set up shared memory, including `mmap()` to create memory mappings of both swap-backed ("anonymous") and file-backed pages. However, synchronisation (e.g., via locks, condition variables, and so on) is required to manage concurrent accesses, which itself may cause overhead.

2.1 Pipes

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 IPC, 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`. You can find our use of pipes in the `ipc_fd.c` file in the lab benchmark bundle.

2.2 Pipe 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. Due to "copy semantics" for the `write(2)` system call, the write operation blocks until the kernel is done with any borrowed pages – i.e., the data has been received via a corresponding `read(2)` system call.

2.3 Explicit shared memory

UNIX systems are rich in APIs to set up and utilize shared memory, having had multiple generations of successive API designs. For the purposes of this lab, we will use shared memory mappings of anonymous (swap-backed) memory set up via the `mmap()` system call and then shared across fork using the `minherit()` system call. This is a somewhat uncommon way to set up shared memory – more frequently a shared memory object is created and then mapped into two processes – but it maps into the operational model of our benchmark well.

In addition, we make use POSIX threading mutexes and condition variables for concurrency control. The mutex and condition-variable attribute `PTHREAD_PROCESS_SHARED` instructs the implementation to support inter-process, not just inter-thread, synchronization via shared pages.

You can find our use of shared memory in the `ipc_shmem.c` file in the lab benchmark bundle.

3 Hypotheses

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

1. *Due to implementing similar shared memory optimisations, the pipe and explicit shared-memory implementations of this benchmark do roughly the same amount of work, and therefore achieve roughly the same performance.*

2. *The probe effect associated with using DTrace to trace this workload's system calls, or use timer-based profiling, is negligible, and need not affect our interpretation of trace information.* only)

We will test these hypotheses by measuring net throughput between two IPC endpoints in two different processes. We will calculate *performance* as the total amount of data transfered divided by the time between the first byte sent by the sender and the last byte received by the recipient – i.e., a measure of *bandwidth*. In general, we will use the bandwidth measurement provided by the benchmark itself, rather than measuring it with DTrace.

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. We will also consider the probe effect of DTrace in the lab.

Above, we use the term 'work', and the key metrics here relate to architectural and microarchitectural throughput. We recommend that you begin by considering two such metrics (in addition to cycles): instructions retired,

and AXI bus memory accesses performed. You will also need to utilize performance data from other counters (e.g., load vs. store instructions, cache misses) in order to interpret these results. Care should be taken when utilizing counters that measure microarchitectural efficiency (e.g., cycles per instruction (CPI)), as they measure how efficient the microarchitecture is in performing the requested work – and not whether the work performed is the most efficient way to achieve the algorithmic goal.

4 The benchmark

The IPC benchmark is straightforward:

`pipe` This variant 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.

`shmem` This variant creates a shared memory window of the requested userspace buffer size between two processes, and uses POSIX mutexes and condition variables for concurrency control. Data is copied into the shared memory window by the sender, and out by the receiver, in a classic producer-consumer pattern. Complete buffers are processed in each iteration, without the possibility of a partially filled buffer.

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-packages/labs/2026-2027-advopsys-lab2.tbz`.

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

```
# cd /data ; tar -xzvf /advopsys-packages/labs/2026-2027-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|shmem]
    [-n iterations] [-p tcp_port] [-P arch|dcache|instr|tlbmem]
    [-t totalsize] mode
```

```
Modes (pick one - default 2thread):
    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|shmem   Select pipe, local sockets, TCP, or shared memory
                              (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)
```

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 two of its IPC types (`pipe`, `shmem`).

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 operating on pipes 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) or `shmem`.

-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
```

This command uses the same IPC workload but via an explicit shared-memory channel set up by the kernel:

```
# ipc/ipc-benchmark -i shmem -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.

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.

5.9 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 128 bytes to 16MiB (inclusive).
- Use only the `2proc` mode.
- Use only the `pipe` or `shmem` IPC types.
- 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. It is useful to know that the system call `clock_gettime()` is both run immediately before, and immediately after, the IPC loop (in `sender()` and `receiver()` functions from the `ipc.c` file in the lab benchmark bundle):

```
/* Instrument the 'return' probe for this invocation. */
if (__sys_clock_gettime(CLOCK_REALTIME, &sap->sa_starttime) < 0)
        xo_err(EX_OSERR, "FAIL: __sys_clock_gettime");

/* ... benchmark ... */

/* Instrument the 'entry' probe for this invocation. */
if (__sys_clock_gettime(CLOCK_REALTIME, &finishtime) < 0)
        xo_err(EX_OSERR, "FAIL: __sys_clock_gettime");
```

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 below for further details.

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.

6.1 Example D script

Bracketing in our DTrace script will allow us to focus tracing and profiling on the IPC loop itself, and not the other inevitable activities of the program – run-time linking, benchmark setup, etc. If you configure the benchmark for a single execution (`-n 1`), then you can bracket tracing between a return probe of the first call to `clock_gettime()`, and the entry probe of the second call. For example, you might wish to include something like the following in your DTrace scripts:

```
BEGIN {
    /* Initialise scalar (global) variables. */
    in_benchmark = 0;
    done = 0;
}

syscall::clock_gettime:return
/execname == "ipc-benchmark" && !in_benchmark && !done/
{
    in_benchmark = 1;
}

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

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

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

If you use more than one iteration per run of the benchmark program, you will need to extend this script to take that into account. Additionally, remember that:

- In case you would like to trace multiple system calls in a single D script, it might be problematic to parse D script output and distinguish data between system calls. Since D associative arrays can be indexed by a list of values of any type, you might consider adding a key with a string indicating what case your data refer to;
- For `syscall:::entry` probes, `arg0`, ..., `argN` variables refer to system call arguments;
- For `syscall:::return` probes, `arg0`, `arg1` refer to system call return values;

- The `errno` variable is set to `0` if a system call succeeded or a non-zero value if it failed;
- The `pid` variable is set to a process ID of the current process, e.g., the process running the sender and receiver threads in our case.
- The `tid` variable is set to the thread ID of the current thread, e.g., a sender thread or a receiver thread in our case.

The following code snippet presents the above suggestions:

```
syscall::read:entry
/execname == "ipc-benchmark" && in_benchmark/
{
    @["read-request", arg2] = count();
}

syscall::write:return
/execname == "ipc-benchmark" && in_benchmark && errno == 0/
{
    @["write-response", arg0] = count();
}
```

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.

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 be 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.

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 `2thread` 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.