Benchmarking of *nix IPC Mechanisms

Rohit Damkondwar
damkondwar@wisc.edu

Om Jadhav
ojadhav@wisc.edu

University of Wisconsin Madison

Abstract

Unix/Linux Operating System provides different mechanisms for inter-process communication (IPC). The IPC is extensively used to build large scale systems. As a result, the performance of IPC mechanisms are very crucial for applications. The popular IPC mechanisms are Shared Memory, Pipe and Sockets. Each mechanism has its advantages and use cases. We aim to study the performance impacts of these mechanisms provided by the kernel.

1. Background

The major properties of IPC mechanisms are: support for bi-directional communication, performance, platform independence, scalability. The following section describes each mechanism in reference to these properties.

1.1 Pipe

Pipe is a unidirectional communication mechanism between two threads of a process or a process and its child. pipe() system call creates a pipe and allocates a pair of file descriptors. Pipes provide automatic synchronization between two processes. In default behaviour of pipe, process trying to read from an empty pipe is blocked until data is written in the pipe. If a process attempts to write to a full pipe, then it will be blocked until sufficient data has been read from the pipe to allow the write to complete. A parent process has to create a pipe and pass it on its child processes which can communicate using that pipe. Thus, PIPE can only be used if the common parent has created it for child processes.

1.2 Shared Memory

Shared memory is a way to let multiple processes attach a segment of physical memory to their virtual address, which allows two or more processes to access the same memory region. In this method, there is no synchronisation provided by OS, so this is overhead user ought to handle. This can be done using primitives like semaphores/condition variables. In this experiment we use **mmap** system call to map files into memory.

1.3 Sockets

A socket is just a logical endpoint for communication. Unix sockets are just like two-way FIFOs. However, all data communication will be taking place through the sockets interface, instead of through the file interface. TCP and UDP are major know protocols to co-ordinate bi-directional data transfer via Unix sockets.

1.3.1 TCP

TCP is connection oriented protocol that handles reliability and congestion control. TCP guarantees that receiver receives the packet with the help of acknowledgements. TCP protocol can be tweaked to buffer smaller packets and send them in a batch via Nagle's Algorithm [4]. We have disabled this batching in our experiments.

1.3.2 UDP

UDP is connection less protocol with no underlying guarantees. As a result, it offers more performance but needs application to handle errors. Since, our experiments do not cross a host, UDP can be used for Inter process communication without complex bookkeeping algorithms for reliability.

2. Benchmark Design

The Linux host used to run evaluation programs had the specs as described in Table 1. The specs were obtained using **‘lscpu’** command on Linux. We are using a 64 bit Linux machine with 4 CPU cores. The machine has 15GiB RAM which is sufficient for our experiments. The experiments do not measure the overhead caused by storage IO operations and hence, hard disk details can be ignored.

Architecture	x86_64
CPU op-mode(s)	64-bit
Byte Order	LittleEndian
CPU(s)	4
Thread(s) per core	1
Core(s) per socket	4
CPU MHz	800
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	6144K

Table 1

2.1 Variables

IPC mechanisms transfer messages among processes on a single host. The benchmark requires comparing performance on variable parameters. In our experiments, we have used different message sizes. The message sizes include 4B, 16B, 64B, 256B, 1K, 4K, 16K, 64K, 256K and 512K. The metrics used for measurement are Latency and Throughput. Another variable is the packet size in case of UDP Protocol. For evaluation we have fixed the packet size to 6500 bytes to get best performance. Decreasing the packet size further reduces the throughput.

2.2 Clock Evaluation

There are various APIs to measure time required for performance comparisons. The The measurements are recorded via two clocking APIs – clock_gettime () and RDTSC. We have also ensured that two processes are pinned on specific cores throughout experiment to ensure fair comparisons. In a multi-core system, both MONOTONIC and RDTSC can have different readings on different cores taken at same time instant. Hence, pinning of processes reduces this error. We could not disable CPU Frequency scaling on the experiment host because of lack of root privileges. In this experiment we first evaluate two timer APIs, clock_gettime and RDTSC. We first calculated error in their output for various time duration. We noticed that, the error of the RDTSC is started low but as time increased it has increased significantly, whereas errors in case of clock_gettime, started with value more than the starting error value of RDTSC, but that remained constant as time interval increased. This shows that clock_gettime is more reliable. We can also conclude this considering the fact that the CPU frequency is not fixed and hence there is possibility that cycles taken by CPU for a particular time duration may vary a lot. We find clock_gettime to be more suitable for our experiment. All the measurements and comparisons are done using the MONOTONIC clock_gettime API. Since, we have pinned each process on a particular processor, page faults should be reduced significantly because of processor cache. This further increases the accuracy of the comparison results.

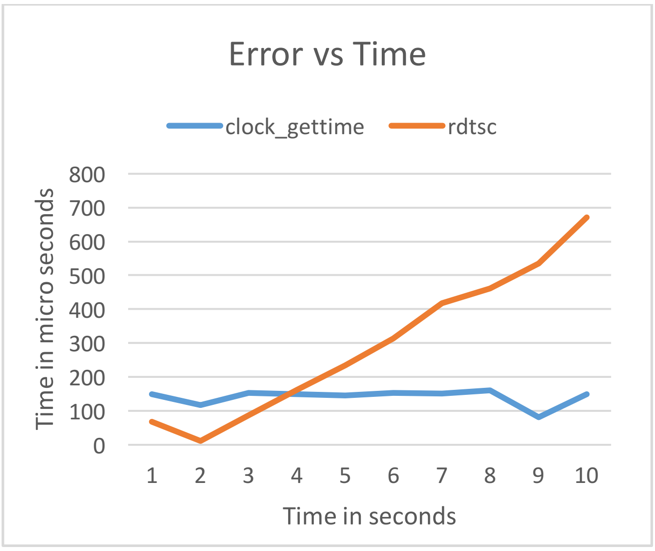

Figure 1: Error vs Time

2.3 Experiment Architecture

IPC mechanisms involve two processes (Sender and Receiver) communicating with each other. Two metrics Throughput and Latency are useful to determine the performance. In this experiment, Sender sends the message to receiver using the mechanism. Receiver replies back with the same message to sender. Sender records time required for message transfer back and forth. Thus, we get the latency of using a mechanism. The Latency is recorded after taking large number of iterations of the above transfer and choosing the minimum latency. Dividing the latency by two gives minimum latency for one-way transfer via the mechanism. Throughput is calculated by sending large data (128MB in our case) in the chunks of variable message size via the mechanism and recording the average latency for all the chunks.

3. Results

In this experiment, we evaluated all four IPC mechanisms on Latency Vs Message Size and Throughput Vs Message size.

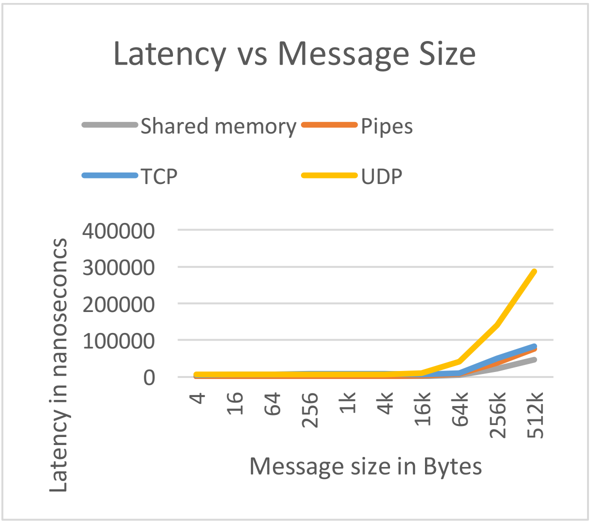

Figure 2: Latency vs Message Size

The Figure2 shows the trend of all the four IPC mechanisms for message sizes of 4, 16, 64, 256, 1K, 4K, 16K, 64K, 256K, and 512K bytes. We can see the trend from the graph that the latency for Shared Memory is minimum and for UDP it is maximum. For the message size of less than 4KBs, all of the IPC mechanisms perform almost similar with not much difference in latency. Beyond 4KB message size, the increase in latency for UDP is much more significant than other three. It is similar with Pipes and TCP when compared to Shared Memory, where the slope is not very steep. The graph clearly shows that Shared Memory has lowest latency. We believe that the major factor in the latency gap is due to number of memory copies of data to be sent. PIPE and Shared Memory require just 1 extra copy of the data. Sockets (TCP and UDP) need at least 2 copies of data. As a result, Shared Memory and PIPE perform better than Sockets. The latency of Shared memory and PIPE being close, we used '**strace'** utility to compare overhead of data copying into Shared memory and PIPE. The figure 4 shows the overhead of "**memset**" (Copying into/from shared memory) and write/read (Writing/reading into pipe).

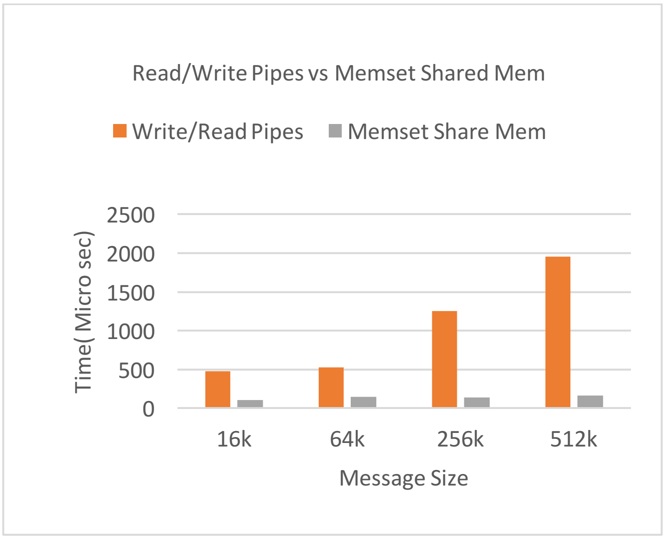

Fig3: Read/Write Pipes vs Memset Sh-Mem

The figure 3 clearly states that reading or writing (1951micro sec) into Pipes is costlier than copying(166 micro sec) from/to Shared Memory. We believe this is the main reason for lower latency in Shared Memory Mechanism.

Throughput vs Message size graph [Figure 4], shows similar trends as Latency Vs Message size. For TCP and UDP throughput is low and it doesn't rise much as a function of message size. The maximum throughput for TCP with message size 512KBs is around 2098MB/s and for UDP it is 833MB/s. However, the rise in the throughput in case of shared memory and pipes are much higher, increases rapidly as function of message size (after 4KB).

So our observation tells that all the mechanisms perform similar when the message sizes is small. Shared memory and pipe performs much better than TCP and UDP when message sizes are significantly bigger. Shared Memory mechanism beats other mechanism in throughput.

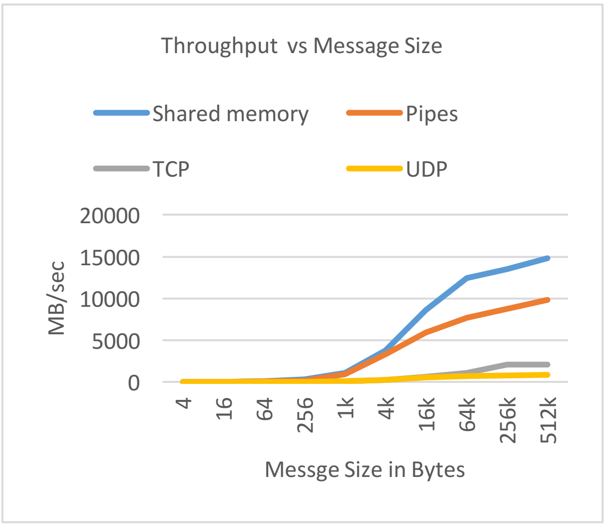

Figure 4: Throughput vs Message Size

Conclusion

The main objective of this evaluation was to compare the performance of IPC mechanisms and describe their usability for applications. As per Throughput and Latency, Shared Memory beats others. But, shared memory requires explicit synchronization by the communicating processes. This adds to the complexity of the application design. PIPE performs better than Socket but it cannot be used by any two random processes. PIPE needs to be created by parent process so that its child processes can communicate with each other. This limitation needs to be considered before choosing this mechanism. Sockets have poor performance because of at least 2 memory copies required, but offer most flexibility. TCP with NO_DELAY flag seems to be a good choice considering performance and reliability. Sockets can be used by processes on different hosts on the network as well. UDP protocol needs explicit book-keeping overhead to ensure reliability and hence, TCP is more favoured choice. We also compared the read/write latency in PIPE against the copying into/from Shared Memory. Shared Memory seems to offer better performance in this operation.

References

1. RDTSC: https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf
2. Clock comparisons: https://github.com/dterei/Scraps/blob/master/intel_tsc/eval_clocks.c
3. Strace man page: http://man7.org/linux/man-pages/man1/strace.1.html
4. https://en.wikipedia.org/wiki/Nagle's_algorithm