Evaluation of Interprocess Communication Mechanisms

Mohammed Danish Shaikh & Krati Agrawal
{mshaikh4, kagrawal6}@wisc.edu
University of Wisconsin, Madison

Abstract— **The operating system provides general mechanisms for flexible interprocess communication (IPC). In this paper, we have studied and evaluated three popular IPC techniques in Linux - pipes, sockets and shared memory. We have carefully constructed the experiments to measure the latency and throughput of each IPC. An accurate timer is the most basic necessity for this evaluation. Therefore, we have compared three timer APIs and picked the one with most reliable results to conduct our experiments. Our observations show that the best performance is given by shared memory, followed by TCP/IP sockets and the slowest is pipes. However, it is highly dependent on two important factors - message sizes and hardware cache. We have studied the effects of these factors to draw some insightful conclusions. We also discuss the usefulness of the three IPCs for different applications.**

I. INTRODUCTION

Interprocess communications are a set of programming interfaces that allow multiple processes to communicate with each other. All IPCs involve synchronization and management of shared data, either by the kernel or by the application. In this paper, we will study and evaluate three commonly-used and powerful IPC mechanisms - pipes, sockets and shared memory.

A. Pipes

Pipe is a one-way communication medium only i.e we can use a pipe such that one process writes to the pipe, and the other process reads from it. Pipe can be used by the creating process, as well as all its child processes, for reading and writing. It is created using a **pipe** system call. The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a *fork* and data will be passed using *read* and *write*. If a process tries to read before something is written to the pipe, the process is suspended until something is written. Similarly, if the writing process exceeds the buffer size, it is blocked until space becomes available. Thus, pipe ensures synchronization between the processes.

B. Sockets

Sockets provide point-to-point, two-way communication between two processes. Sockets are created and used with a set of programming requests or *function calls* sometimes called the socket API. When a socket is created, the program has to specify the address domain and the socket type. Two processes can communicate with each other only if their sockets are of the same type and in the same domain. There are two widely used address domains, the unix domain, in which two processes which share a common file system communicate, and the Internet domain, in which two processes running on any two hosts on the Internet communicate. Each of these has its own address format. In addition, each socket needs a port number on that host. There are two widely used socket types, stream sockets, and datagram sockets. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Each uses its own communciations protocol. Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream oriented protocol, and datagram sockets use UDP (Unix Datagram Protocol), which is unreliable and message oriented.

C. Shared Memory

A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. Intuitively, the fastest way to communicate is to bypass the kernel altogether and use shared memory. This mechanism uses a simple bounded buffer to communicate in each direction. The shared memory can be created with the mmap system call, and synchronization can be done with pthread mutexes or spinning.

In this paper, we have constructed experiments to evaluate performance, i.e. latency and throughput of these mechanisms. The rest of the paper is organized as follows : Section II describes the evaluation methodology, including our measurement methods and experiments. Section III presents our results and analysis. Section IV concludes.

II. Evaluation

Latency is the time for a given activity to complete, from beginning to end. For message passing, it is the time from the start of a send to the completion of a receive. Since the clocks on two different cores (for some timer APIs) may not be sufficiently aligned, the easiest way to measure message latency is to measure the time it takes to complete a round-trip communication (and divide by two). Stated below are our hypotheses of the variables that we expect the latency to depend on:

- **Message Size:** Intuitively, one would expect the latency to be an increasing function of message size. We perform the experiment over message sizes ranging from 4B to 512KB in order to confirm this notion.
- **Processor Caching:** If the message buffer is stored in cache, it would be easier for the reading process to retrieve it and the latency would be reduced considerably. Our experiment with different message sizes will test this hypothesis too.

The evaluation environment shown in Table 1 was used in our experiments.

TABLE I
Evaluation environment

Architecture	x86_64
CPU mode	64-bit
Byte order	Little Endian
No. of CPUs	4
Threads per core	1
Cores per socket	1
Sockets	1
CPU Max Frequency	3600Mhz
CPU Min Frequency	800MHz
L1d, L1i cache	32K
L2 cache	256K
L3 cache	6144K
Block size	64B

A. Selection of Timer

The accuracy and precision of the timer used will often have a large impact on the measurements. Therefore, the best timer available should be used. Out of the several APIs available for this purpose, we chose *gettimeofday* and *clock_gettime* as candidate timers for our experiments. In addition, on x86 platforms a highly accurate cycle counter is available. The instruction to use it is known as *rdtsc*, it returns a 64-bit cycle count. By knowing the cycle time, one can easily convert the result of rdtsc into a useful time. However, it has the following caveats:

- If the processor can automatically vary the clock speed, the timestamp counter may not reflect real time. One way to avoid this could be to set the CPU frequency using **cpupower** before conducting the experiments. However, we assume that this variation in CPU speed is uniform across all our experiments. Due to lack of root access on our evaluation environment, we proceed with the stated assumption.
- On a multicore system, different processor cores may have different values for the timestamp; you can only compare values on a single core and not across cores. Our experiments were pinned to a single CPU core using **taskset** in order to avoid this situation.

The following method was used for calibrating the rdtsc clock:

- Measure the difference in the timestamp counter value for performing *sleep* over 1000 iterations.
- Find the time equivalent to one timestamp counter using simple algebra.

The following method was used to calculate precision for each timer API:

- Calculate the timer value for performing a simple operation (For example : setting a memory location to a constant value) over several iterations.
- Reduce the number of iterations until the timer can't calculate time difference of the operation, i.e. it starts showing 0.

The aforementioned precision calculation was repeated several times for each timer API thereby retrieving a series of precision values for each of them. The statistics observed have been summarized in Table II.

TABLE II
Evaluation of timer APIs

API	Avg. Err.(ns)	Std. Dev.(ns)	Precision(ns)
rdtsc	3.89	4.66	7.5
clock_gettime()	28.25	34.14	63
gettimeofday()	0.17	0.29	1000

Since rdtsc presents the highest precision and low average error, we further use this timer API in our experiments throughout the paper.

B. *Evaluation of IPCs*

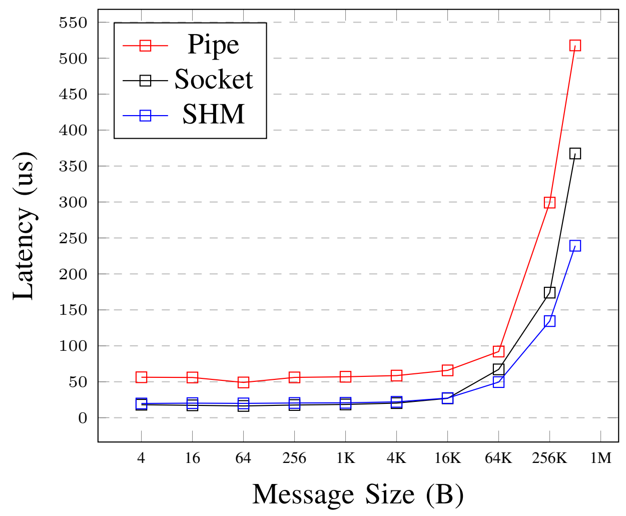
In this section, we describe the implementation and execution of all three interprocess communication mechanisms discussed so far.

1) Implementation:
   - **Pipes:** Pipe is created from a parent process using the *pipe* system call. The parent then forks into a child process which shares the pipe created by parent. This pipe can now be used to communicate between the parent and child.
   - **Sockets:** We have two processes for socket implementation - a server and a client. The server creates a socket and listens for a process to connect to it, while the client connects to server's socket using its port number. After the connection is set up, the server and client can send and receive messages through the socket. We have implemented internet domain streaming sockets in our experiments since in practical scenarios it's difficult to find out if server and client processes are running on the same machine or not.
   - **Shared Memory:** For shared memory, the following two approaches were considered:
     - *Using threads:* We created two threads, one for each side of the communication. We created shared memory using *mmap* system call and exchanged messages between the threads by using **pthread mutexes** for synchronization. Note that creating shared memory using mmap could have been avoided by declaring a character array in heap of the parent process. However, we didn't perform our experiments this way since we wanted to test IPC using shared memory.
     - *Using fork:* We created shared memory using mmap system call before doing a fork. Subsequently, both parent and child processes would have read/write access to the shared memory. We used shared memory to exchange messages between parent and child processes in a synchronized manner using pthread mutexes.

     All our experimental data have been collected by exchanging messages using shared memory between threads.
2) Calculation of latencies: In our experiments, we took a series of readings for each IPC and calculated the average of last-N readings after the readings seemed to have stabilized. The reason behind doing this was to eliminate any overheads associated with inconsistent cache behavior during the start of our experiments.
3) Calculation of throughput: We calculated the difference in timer values before and after writing the data for each IPC. We then obtained the throughput value by dividing the message size by the timer value.

III. Results

Figure 1. Latency vs message size

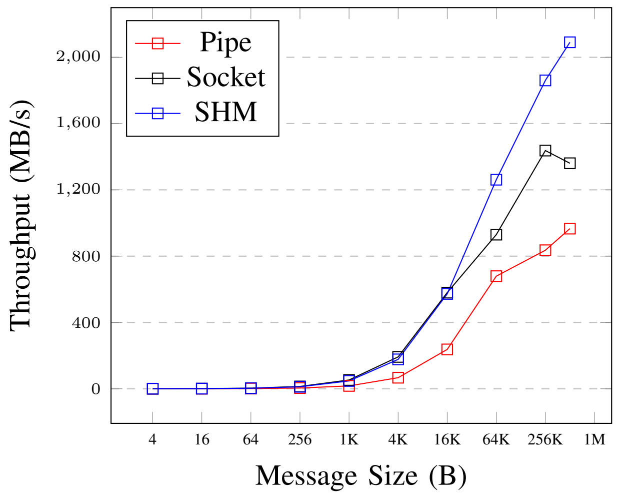


Let us consider the effect of message size on latency. From Fig. 1 we observe that shared memory provides the minimum latency, followed by sockets and lastly pipes. In shared memory, the processes simply share a memory region thereby allowing them to read and write to that region without the overhead of performing system calls. The higher latency in sockets can be attributed to high network congestion which makes data transfers slow. An interesting point to note here is that latency within each mechanism is almost constant for message size upto 4KB. This is due to the fact that the page size is 4KB. Therefore, for messages as small as 4B, atleast one page has to be accessed which adds to the latency. The maximum message size used for measuring pipe latency is 64KB. This is because the pipe buffer size is set to 64KB by default. For message sizes larger than 64KB, we send multiple

64KB messages and measure the latency. Due to the overhead of sending multiple messages, pipes has the highest latency.

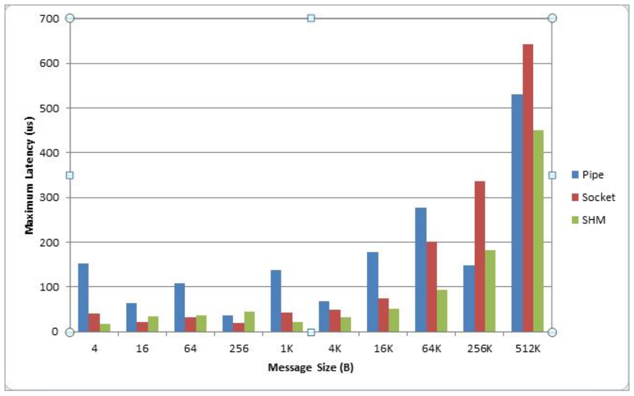
Fig. 2 shows the variation of throughput with message size. We see similar throughput in all three IPCs for message size less than 1KB. Beyond 16KB, shared memory has a significant increase in data transfer rate, whereas, pipes offer the worst throughput among all three mechanisms. We observe that the throughput generally increases with message size.

Figure 2. Throughput vs message size



Another key factor in determining latency is the cache. As explained earlier in Section II B, the initial access to message will result in cache miss and is likely to increase the latency considerably. Our experiments corroborate this hypothesis as can be seen in Fig. 3. As expected, larger message size results in higher latency for the first access. However, there is no deterministic relationship between message size and maximum latency.

Figure 3. Maximum Latency vs message size



We analyzed the **strace** output for each of our implementations. We posit, on the basis of above results that the *read* and *write* system calls take more time for pipes than for sockets. We also note that the strace output for shared memory does not use read and write system calls, thereby confirming that this mode of IPC bypasses the kernel. However, the *futex* system call is used for synchronization. We state, on the basis of our experiments that the overheads for performing read and write system calls in case of sockets and pipes is more than that of performing futex system call in case of shared memory.

IV. CONCLUSIONS

We have studied and evaluated the performance of three interprocess communication mechanisms. We did experiments to find precision of three timer APIs and selected rdtsc to use in our evaluations. We make the following assumptions before presenting our final conclusions:

- We did not conduct the experiments in an isolated computing environment, hence, workload of other user processes might have had some effect on our latency calculations. However, since all experiments were conducted on the same core of the same machine independently, we assume the effect of other workload to be uniform across all our experiments.
- We didn't set the CPU frequency before performing our experiments. This might have had some effect on our latency calculations since the CPU frequency keeps on fluctuating between a minimum and maximum value. Since the experiments were conducted on the same core of the same machine independently, we assume that the effect of CPU frequency fluctuations is uniform across all our experiments.
- We didn't change the maximum transmission unit (MTU) size throughout our experiments. This might have had an effect on our IPC experiments using TCP Sockets. However, since the client and server resided on the same machine, we assume this effect to be negligible.

Shared Memory has the best performance in terms of latency since it eliminates the overheads of doing system calls through kernel. However, it's harder to implement since synchronization has to be implemented by the application. Shared Memory is usually suitable to use in scenarios wherein applications of TypeA want to broadcast data to be read by applications of TypeB or vice versa (classical producer-consumer problem) and when we want more control over memory allocation decisions.

TCP Sockets are the second best among all three IPCs. TCP sockets are widely used for communication between processes over a network. No synchronization mechanisms is required from the application end. Note that the results might have been entirely different had the experiment been performed with client and server processes residing on separate machines.

Pipes have the worst performance for all message size. Pipes are unidirectional way of communication and are usually used as a simple message passing mechanism between processes. They are extensively used in the UNIX filter programs like *cat, grep, sort and uniq* for efficiently sharing data between each other for processing. Synchronization is simple with pipes and is built into the pipe mechanism itself - the reads (writes) will freeze and unfreeze the application based on the data (space) availability in pipe buffer.

ACKNOWLEDGMENT

We would like to thank our professor Michael Swift for giving us the opportunity to work on this project. His advice during office hours and classroom lectures helped us in successfully completing this project. Also, thanks to our batchmates who helped us in various aspects by providing key experimentation ideas.

REFERENCES

[1] https://queue.acm.org/detail.cfm?id=2878574
[2] https://www.sharelatex.com/learn/Pgfplots_package
[3] https://www.mcs.anl.gov/ kazutomo/rdtsc.html
[4] http://btorpey.github.io/blog/2014/02/18/clock-sources-in-linux/
[5] http://faculty.kfupm.edu.sa/ics/garout/Teaching/ICS431/Lab14.pdf
[6] http://www.cs.rpi.edu/m̃oorthy/Courses/os98/Pgms/socket.html
[7] http://man7.org/linux/man-pages/man2/pipe.2.html
[8] http://man7.org/linux/man-pages/man2/socket.2.html
[9] https://www.tenouk.com/cnlinuxsockettutorials.html
[10] https://linux.die.net/man/1/taskset
[11] http://man7.org/linux/man-pages/man2/gettimeofday.2.html
[12] https://en.wikipedia.org/wiki/Producer-consumer_problem
[13] https://linux.die.net/man/3/clock_gettime