Evaluation of Inter-Process Communication Mechanisms

Aditya Venkataraman
University of Wisconsin-Madison
adityav@cs.wisc.edu

Kishore Kumar Jagadeesha
University of Wisconsin-Madison
kishore@cs.wisc.edu

ABSTRACT

The abstraction of a process enables certain primitive forms of communication during process creation and destruction such as wait(). However, the operating system provides more general mechanisms for flexible inter-process communication. In this paper, we have studied and evaluated three commonly-used inter-process communication devices - pipes, sockets and shared memory. We have identified the various factors that could affect their performance such as message size, hardware caches and process scheduling, and constructed experiments to reliably measure the latency and transfer rate of each device. We identified the most reliable timer APIs available for our measurements. Our experiments reveal that shared memory provides the lowest latency and highest throughput, followed by kernel pipes and lastly, TCP/IP sockets. However, the latency trends provide interesting insights into the construction of each mechanism. We also make certain observations on the pros and cons of each mechanism, highlighting its usefulness for different kinds of applications.

1. INTRODUCTION

Inter-process communication mechanisms are used for transferring data between processes. These mechanisms can be broadly classified based on the following criteria [4, 5]:

- Whether the communication is restricted to related processes.
- Whether the communication is write-only, read-only or read/write.
- Whether the communication is between two processes or more.
- Whether the communication is synchronous, i.e. the reading process blocks on a read.

In this paper, we will study and evaluate three popular and powerful inter-process communication mechanisms - pipes, sockets and shared memory.

1.1 Pipes

A pipe is a unidirectional communication device that permits serial transfer of bytes from the writing process to the reading process [7]. Typically, a pipe is used for communication between two threads of a process or between a parent and child process. A pipe is created using the **pipe** system call, which creates file-descriptors for writing and reading from the pipe. A pipe's data capacity is limited; if the writer writes faster than a reader consumes data and if the pipe buffer is full, the writer will block until more capacity becomes available. Similarly, if the reader reads when the pipe is empty, the reader will block. Thus, the pipe provides automatic synchronization between the processes.

1.2 Sockets

A socket is a bidirectional communication mechanism that can be used to communicate with another process on the same machine or on a different machine. Particularly, Internet programs such as World Wide Web and Telnet employ sockets as their communication device. Data sent over a socket is split into chunks called packets. A protocol, such as TCP/IP, specifies how these packets are transmitted over the socket [6]. A socket is uniquely addressed using a combination of the machine IP address as well as a port number. In this paper we will evaluate local-machine sockets only, as we expect unpredictable network latencies to distort the latency for remote-machine sockets.

1.3 Shared Memory

Shared memory allows two or more processes to access the same memory region, which is mapped onto the address space of all participating processes. Since this communication is similar to any other memory reference, it does not involve any system calls or protocol-induced overheads. Hence, one can expect shared memory to offer very low latencies. However, the system does not provide any synchronization for accesses to shared memory; it ought to implemented by user programs using primitives such as semaphores.

In this paper we will construct experiments to eval-

uate the performance - latency and throughput - of these mechanisms. We will also qualitatively reason about their pros and cons to understand the relative importance of each mechanism.

The rest of the paper is organized as follows: Section 2 describes the evaluation methodology, including our measurement methods and experiments. Section 3 presents our results and analysis. Section 4 concludes.

2. EVALUATION

To compare the performance of different communication mechanisms, we used the evaluation environment given in Table 1.

2.1 Evalution of timers

The goal of the study is to empirically compute the latency and throughput of popular IPC mechanisms. This requires a reliable and accurate way of measuring the communication latency; clearly, an inaccurate timer will render the entire evaluation suspect. Hence, we evaluated the accuracy and resolution of three popular Linux timer APIs - RDTSC, clock_gettime() and gettimeofday(). The accuracy was measured by putting the program to sleep for a known time-period in-between two calls to the timer API. The time-period measured through the timer was then compared against the expected period. To identify the smallest, reliable resolution, we successively reduced the number of instructions in between two calls to the timer API until the timer could not identify the time interval.

RDTSC is a 64-bit register in x86 processors that counts the number of cycles since reset [2]. RDTSC used to be an excellent, high-resolution way of measuring CPU timing information, however with the introduction of technologies such as multi-core, hyper-threaded and dynamic frequency scaling, RDTSC cannot be natively relied on to provide accurate results [9]. For this evaluation, we circumvented these limitations as follows. Firstly, we ran all CPUs at their peak frequencies. Secondly, each process was pinned onto a single CPU core using the taskset API. Thirdly, we constructed our experiments such that all measurements were made on a single process, avoiding any synchronization-related issues among TSC counters of different cores.

clock_gettime() retrieves the time of the real-time system clock in the order of nano-seconds. The readings from this timer are guaranteed to be monotonic. In a multi-core system, readings from this API on different cores are expected to have a small offset like RDTSC. [8]

gettimeofday() can be used to retrieve the time since Epoch in the order of micro-seconds. However, the time returned by this API can be affected by discontinuous jumps in system time, and cannot be assumed to be monotonically increasing like the previous two APIs. [3]

Table 1: Evaluation Environment

Architecture	x86_64
CPU mode	64-bit
Byte Order	Little Endian
No. of CPUs	4
Thread(s) per core	1
CPU Frequency	3192.770 MHz
L1d, L1i cache	32KB
L2 cache	256KB
L3 cache	6144KB
Block size	64B

As given in Table 2, our experiments reveal that all three APIs are quite accurate for measurements in the order of thousands of micro-seconds. However, we find that RDTSC and clock_gettime() have a higher resolution than gettimeofday(), as the latter cannot differentiate within a micro-second. We conclude this section of our study by choosing RDTSC and clock_gettime() as suitable choices for measuring IPC latencies. All our remaining experiments were performed with RDTSC, with the results compared against clock_gettime(). We found both timers to corroborate the same trends.

2.2 Experiments

Communication latency is defined as the time between initiation of transmission at the sender and completion of reception at the receiver. However, due to aforementioned offsets in timer readings on different cores, we compute the communication latency for a round-trip communication and then halve it. While the latency is not expected to be exactly symmetrical along each route, we temper the error over a large number of round-trip messages. The transfer rate of a communication mechanism is the amount of data that can be sent per unit time. In our experiments, we send a large message to the receiver, who then acknowledges through a single Byte message. Provided the initial message is much larger than the acknowledgement, one can approximate the measured round-trip latency to be the forward-trip latency. Synchronization is enforced either implicitly by the IPC mechanism (for pipes and sockets) or explicitly using UNIX semaphores for shared memory experiments.

2.2.1 Variables

We ought to be cognizant of the different variables that could affect our experiments. Some variables are relatively obvious, such as the message size; while others are tougher to understand such as cache effects including block sizes and cache coherence, as well as CPU scheduling decisions. Naively, one would expect the latency to be linearly related to the size of the message; to confirm this notion, we consider a wide range of message sizes from 4B to 512KB.

Table 2: Evaluation of timer APIs

API	Average Error (us)	Std. Dev (us)	Resolution (ns)
RDTSC	328	9.41	62
clock_gettime()	81	11.74	30
gettimeofday()	74	9.59	1000

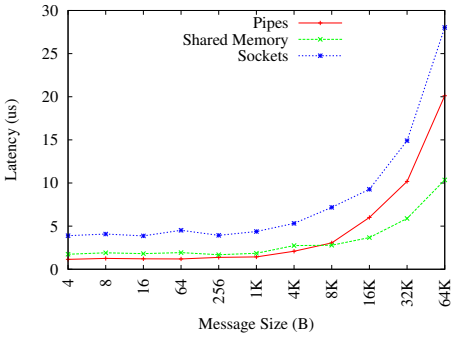
Processor caches play a significant role in reducing memory access latency by fetching an entire block of addresses for each cache miss. Since each communication mechanism uses an internal memory buffer for transmission, these addresses are expected to be cached. In any IPC paradigm, there is an inherent producer-consumer scenario, where the sender writes certain addresses which are expected to be cached in the sender's private cache. [1] The receiver will then read these addresses which will cause the cache block to be transferred from the sender's cache to the receiver's cache. In this way, cache coherence leads to a 'ping-pong' of the cache block during subsequent transfers. In our experiments, we are primarily interested in an 'equilibrium' state, where the communication buffer is largely present in on-chip caches. So, we average our measurements over a large number of iterations to arrive at an equilibrium latency. However, we have considered the effects of the cache in determining the maximum latency as well as effect of cache for very large message sizes.

The effect of the OS scheduler is particularly tricky to anticipate. We simplified the problem by breaking it into two cases - first, with sender and receiver process on separate, fixed cores and second, with sender and receiver on the same core. All experiments were conducted for both cases. While one can expect a realistic scheduler to be more vagrant in its decisions over the course of an application's lifetime, we feel these approximations are reasonable for the lifetime of a single inter-process communication.

3. RESULTS

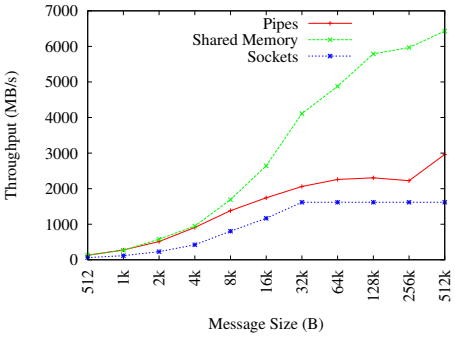
Let us consider the variation in latency as a function of message size, when the sender and receiver processes are fixed on different cores. As Figure 1 suggests the latencies for sockets are consistently higher than the latencies for shared memory and pipes. In each mechanism, the latency is roughly uniform for message sizes up to 4KB. This goes against our naive expectation of a linear relationship between message size and latency. This suggests that each mechanism uses a single-page transfer buffer for all messages up to 4KB. Beyond 4KB, the latencies start to rise as a function of message size. This is more pronounced for pipes and sockets compared to shared memory. In general, one can note that for small messages, pipes and shared memory offer similar latencies; for larger messages shared memory offers vastly better latencies compared to pipes and sockets.

Figure 1: Latency vs Message Size



We see a similar trend for throughput in Figure 2 - pipes and shared memory offer roughly similar throughput for small message sizes, but beyond 16KB, shared memory offers a significantly higher transfer rate. Sockets offer the lowest average transfer rate relative to the other two mechanisms. Since TCP/IP limits the maximum packet size to 64KB, we have not considered higher message sizes for our single-packet experiments; hence the curve is shown to saturate beyond 64KB.

Figure 2: Throughput vs Message Size



Next let us consider the effect of the cache. As explained in Section 2.2, the initial access to an address is expected to miss in the cache and is likely to have high latency. Our experiments confirm this expectation. Figure 3 gives the maximum latency of access for various message sizes. The maximum latencies are higher for larger messages, most likely due to higher number of cache misses. There does not appear to be a deterministic relationship between maximum latency and message size. However, the considerably lower average latencies seen in Figure 1 reveals the advantages of 'hot' caches.

Figure 3: Max Latency vs Message Size

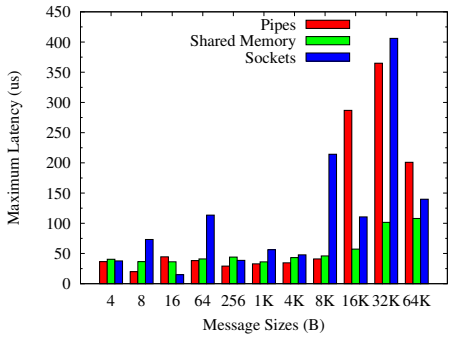
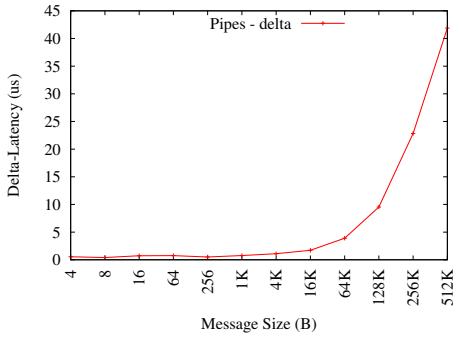


Figure 4: Pipes - Latency Difference between Single and Different Cores



Lastly, we consider the effect of the scheduler. All results presented so far have been obtained from pinning the sender and receiver to different cores. If we pin them to the same core, one can expect the cache coherence-induced 'ping-pong' overhead to reduce. However, the IPC will now involve a context switch overhead between the sender and the receiver. These two effects could dynamically influence the IPC latency. Our experiments on pipes reveal that for larger messages, the IPC latency on a single core is significantly lower than that on different cores as seen in Figure 4.

4. SUMMARY AND CONCLUSIONS

In this work, we have studied and evaluated the performance of three popular inter-process communication mechanisms. We examined the accuracy and resolution of various timer APIs, settling on RDTSC and clock_gettime() as the most suitable choices for our study. We identified the variables that can influence the IPC latency and systematically constructed experiments to study their effects.

We have identified shared memory to be the fastest form of IPC because all processes share the same piece of memory, hence we avoid copying data between process memory and kernel memory. Access to this memory is similar to any memory reference and does not need special system support like system calls. It also permits multiple (>2) processes to perform bi-directional communication. However, the biggest caveat of shared memory is that the system offers no synchronization guarantees. The user processes need to coordinate their accesses into the shared region. This is a potential security threat as well.

Pipes offer a flexible and fast way for communication between two processes. Each pipe is uni-directional, hence we need two pipes for bidirectional communication. The UNIX implementation of pipes represents them as a file, which can easily substitute the stdin or stdout for a process, offering great programming convenience. However, the maximum number of pipes per process is limited to 1024, which is the size of the file-descriptor table. Pipes offer automatic synchronization through a reserved kernel buffer.

Sockets allow bidirectional communication, but their performance for IPC on the same machine is quite poor. Being a packet-based mechanism, sockets may require breaking down large messages into multiple packets, increasing communication latency. However, sockets offer some unique advantages for IPC - sockets can be used for IPC between processes on different machines, allowing scalability for SW growth. Being a standardized mechanism governed by a recognized protocol, sockets are very useful for connecting with and testing unknown or opaque processes (black boxes). Hence, one can recognize that each IPC mechanism offers unique advantages which render them useful for different use-cases.

Acknowledgements

We would like to thank our instructor, Prof. Michael Swift for his helpful guidance in class for the completion of this project. We also acknowledge the contribution of our many class mates for insightful discussions on tackling this problem.

5. REFERENCES

[1] Russell M Clapp, Trevor N Mudge, and Donald C Winsor. Cache coherence requirements for interprocess rendezvous. *International Journal of Parallel Programming*, 19(1):31–51, 1990.

[2] Intel Coorporation. Using the rdtsc instruction for performance monitoring. *Techn. Ber., tech. rep., Intel Coorporation*, page 22, 1997.

[3] Steven A Finney. Real-time data collection in linux: A case study. *Behavior Research Methods, Instruments, & Computers*, 33(2):167–173, 2001.

[4] William Fornaciari. Inter-process communication. 2002.

[5] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced linux programming*. New Riders, 2001.

[6] Jon Postel. Transmission control protocol. 1981.

[7] OM Ritchie and Ken Thompson. The unix time-sharing system. *Bell System Technical Journal,*

The, 57(6):1905–1929, 1978.

[8] Guy Rutenberg. Profiling code using code_gettime. *Retrieved from Guy Rutenberg Website: http://www.guyrutenberg.com/2007/09/22/profiling-code-using-clock_gettime/*, 2007.

[9] Chuck Walbourn. Game timing and multicore processors. *XNA Developer Connection (XDC), Dec*, 2005.