Inter-process communication seen as Input/Output

B H Venter
Department of Computer Science, University of Port Elizabeth
P O Box 1600
6000 Port Elizabeth
e-mail: csabhv@upe.ac.za

Abstract

Inter-process communication (IPC) and Input/Output (I/O) are closely related. If the classical operating system calls for Input/Output are suitably generalized, they can be used as an IPC mechanism. Not much needs to be done, apart from allowing process to "open" another process for I/O operations. Initiating an I/O operation corresponds to sending a request message to device driver, file system, or another process. The completion of an I/O operation corresponds to receiving a reply message. A particular set of generalized operation system calls for I/O+IPC is presented in the paper. The semantics of these calls, and their implications, are discussed in some detail, with particular reference to IPC issues. The paper argues that a generalized mechanism is useful enough in the "raw" form, and efficient enough, for an operating system designer to consider replacing existing I/O-only and IPC-only sets of calls with a single, general set.

Categories and Subject Descriptors: D4.4 [**Operating Systems**]: Communications Management – *buffering, input/output, message sending.*
General Terms: Design
Additional Key words and Phrases: operating systems design, inter-process communication, input/output.

1 Introduction

The aim of this paper is to show that a straightforward generalization of the classical system calls for dealing with Input/Output (I/O) from files and devices can also be used as an efficient and general inter-process communication (IPC) mechanism. Operating system designers therefore need not add special system calls dedicated to IPC, but need only generalize existing I/O mechanisms.

A great many papers dealing with IPC have already been published, and many have influenced the ideas expounded here. This paper, however, is not a report on IPC research, but an explanation and defense of the above claim only. Consequently no references will be made to the literature and no attempt will be made to sort out which researcher should get credit for proposing what. References to quite a few applicable papers can be found in [1].

2 Generalized Input/Output

Many classical operating system calls for I/O can readily be generalized into a single call, say IO, with the following **in out** parameters:

io-port The internal name (number or handle) of the file or device that is the target of the operation.

operation A code specifying the operation to be carried out (read, write, etc.).

address The offset from the start of the file, of the byte string affected by the operation. Or some device address.

out-buf A pointer to a buffer containing data to be output.

out-len The number of bytes to be output.

in-buf A pointer to the buffer into which data must be input.

in-len The maximum number of bytes that may be input into the buffer.

Making the operation a parameter, rather than hard-wiring it into the system call name, adds important flexibility. For example, one can add support for new devices with new operations without having to add new system calls to the operating system.

A particular operating system will accept a particular set of operations when IO operates on files, and a particular subset of the other parameters will be interpreted in a particular way for each of these particular operations. Likewise, IO can be constrained and interpreted in particular ways for particular devices.

But this is not what this article is about. Below it will be shown that I/O operations, using IO, can sensibly be carried out on processes, and in such cases amount to an efficient but general form of IPC.

To perform an I/O operation on a process, one has to be able to "open" the process. That is, one must be able to assign an internal name to another process much like one assigns internal names to files and devices when opening them for use. How this can be achieved will be considered in a later section. For now, assume that when processes A and B are communicating then X is A's internal name for B, and Y is B's internal name for A. Also, the term "port" will henceforth be used instead of "internal name".

3 Communicating requests and replies

The IO call can be used to simulate various styles of IPC. However, communication between client and server can be regarded as the native style, since that is essentially what classical I/O amounts to.

In (simple) client-server interaction, the server waits until it receives a request message. It then carries out the request, sends a reply message and waits for the next request. The client, on the other hand, is active until it sends a request after which it waits until it receives a reply.

Seen in terms of I/O, "waiting" is what happens in between initiating the I/O operation and completing it. Supposing that process A is the client and process B the server, we thus must have an *initial* state where B is already in the midst of an I/O operation using port Y, while A is free to execute.

When A finally needs to make a request to B, it performs an approriate I/O operation using port X. The IO call now packages the *operation, address, out-len* and *in-len* parameters supplied by A, as well as the byte string pointed to by *out-buf* into a message which is sent off to B. The receipt of the message now causes B to complete the I/O operation that it has been waiting to complete.

Moreover, the **out** values of the parameters B supplied when calling IO, is derived from the **in** values that A supplied when making *its* call to IO. Which is to say that the message received from A is unpacked into B's **out** parameters. The unpacking proceeds as follows:

- B.*operation*.**out** := A.*operation*.**in**
- B.*address*.**out** := A.*address*.**in**
- B.*out-len*.**out** := A.*in-len*.**in**
- B.*in-len*.**out** := A.*out-len*.**in**
- The storage pointed to by B.*in-buf* is overwritten with the first A.*out-len*.**in** bytes pointed to by A.*out-buf*.**in**.

Thus, B knows as much about A's "I/O request" as a file system would need to know, had A's request involved I/O from a file. B can thus serve A's request much like a file system would, and send its reply by performing another I/O operation using port Y. The initiation of this I/O operation by B, causes A to complete the I/O operation that it has been waiting for while B was executing. Furthermore, the **out** values of the parameters that A used for its I/O operation have been unpacked from the reply received from B. A can now carry on with execution while B waits for A's next request.

At a first reading, the above description may make IO sound too complicated for serious attention. However, it is worth noting that what complexity there is, lies mainly in writing servers. The programmer of a client process need not even be aware of using IO for IPC. To this programmer, the server process is a "device" that can be manipulated by means of classical I/O operations.

For example, a client process may open an IPC link to a name server and treat it as just another I/O device, except that this device has operations like *store-name, retrieve-first-name, retrieve-next-name,* etc., instead of operations like *rewind, write-file-mark, skip-file-mark,* etc. which a tape drive might offer.

This is certainly just as simple as any other way of interacting with a name server.

In order to write server processes, one needs a more explicit understanding of how to use IO for IPC. Principally, one needs to become comfortable with the fact that the *initiation* of an I/O operation by one process can cause the *completion* of an in-progress I/O operation in another process. One also needs to become used to thinking of the **in** values of IO's parameters as a message sent to another process, which will receive it in the form of the **out** values of IO's parameters. Once this is understood, using IO as an IPC mechanism is not more difficult than using any other IPC mechanism.

4 Asynchronous Input/Output

No realistic operating system can afford to omit provision for non-blocking forms of I/O. Consequently there is the need to provide three variant forms of IO, namely Initiate-IO, Complete-IO and Try-to-complete-IO.

Initiate-IO takes the I/O operation up to the point where the parameters and output have been sent off to the other process (or device driver or file system), but does not wait for a reply. Complete-IO goes on where Initiate-IO left off. If no reply has been received in the mean time, Complete-IO waits for one to arrive. Try-to-complete-IO is a version of Complete-IO that will return (setting a status code) rather than wait for a reply.

When used for IPC, asynchronous I/O makes it possible to write server processes that serve several client processes, since it allows a server to have *several* incomplete I/O operations. Of course, the server will have to use a different I/O port for each client. And, since an I/O operation will complete on a particular port only if the corresponding client makes a request, a server waiting to serve the next request from any client should never wait on a particular client port. Nor is it acceptable for the server to sit in a busy loop, polling its I/O ports with Try-to-complete-IO.

The solution is to introduce an "any port" value that can be supplied as *io-port*.**in** to Complete-IO and Try-to-complete-IO. In such cases the caller is suspended until an I/O operation completes on any of its ports. The I/O port selected is returned as *io-port*.**out**. The value of *in-buf*.**out** is taken from the value of *in-buf*.**in** supplied to the call to Initiate-IO that started the I/O operation on the port concerned.

5 Message buffers

One of the advantages of using IO in the manner described so far, is that there is no need for the operating system to manage a pool of message buffers. Communication proceeds on a "hand-shaked" request-reply basis. And, whenever a process outputs a buffer, its communicating partner has already provided a corresponding input buffer. Moreover, the process doing the output knows the exact length of this buffer (*out-len*.**out**). So too does the operating system, and hence the operating system can ensure that a receiving process need not contend with more input than was asked for.

Starting up a communication link so that there is always an input buffer available to hold an incoming message, is somewhat tricky:

The solution is to treat the first I/O call on an I/O port differently from subsequent calls: Should the first I/O call not do any output ($out\text{-}len.\mathbf{in} = 0$), it proceeds normally. That is, the parameters are strung into a message which is sent off to the communicating partner. However, should the first I/O call involve output ($out\text{-}len.\mathbf{in} \neq 0$), its initiation is delayed until the communicating partner successfully initiates its first I/O operation. That is, the parameters are sent off to the communicating partner, only *after* a message is received from the partner. Moreover, this synchronization message is *not* returned to the caller in the form of the **out** parameters of IO, but is effectively discarded.

The implication of this strategy is that a server process, which normally speaks only when spoken to, must initialize a port linked to a client by performing an input operation on it. The message resulting from this operation will allow the first output operation by the client to proceed. And since the server's first message is discarded, the client need not deal with an unexpected message from the server.

Unfortunately, things are more complicated when the client chooses to start communicating with an input operation.[1] If the message resulting from the first input operation on a port is always discarded, then the server will not receive the first request made by the client. But if the message is discarded only when received at a port whose first operation is an output operation, then the client will have to deal with an unsolicited message from the server.

The only way out of this is to insist that all communication links be initialized by one process performing an output operation and the other an input operation, thus avoiding the complications. In other words, a client must *always* perform an initial output operation.

If need be, the server will have to provide an additional dummy output operation to be used by the client for initializing the link. Note that a suitably defined subroutine library can make the whole procedure transparent to client programmers.

It should also be noted that this difficulty arises chiefly because of the need to have a single set of rules apply to all styles of communication, and to all I/O operations performed by user processes. This simplifies things once communication is initialized.

[1] On a first reading of this paper it might be better to skip the rest of this section

6 Producer-consumer pipe-lines

One of the most universally praised features of the UNIX operating system, is its ability to "pipe" the sequential output of one process into the sequential input of another.

Similar producer-consumer pipe-lines can readily be formed using IO, simply by connecting the appropriate I/O ports of the producer and consumer. Neither producer, nor consumer need necessarily be aware that their I/O operations do not end up in files. As already described, the buffering strategy automatically ensures that the producer will be delayed until the consumer has made a buffer available to receive the producer's output. The **out** values of IO's parameters can simply be ignored.

However, this form of pipe-lining is not as transparent as the form provided by UNIX. In fact, it is only completely transparent when the producer and consumer write and read byte strings of exactly the same length. If the consumer tries to read 256 bytes from its input "file", but the producer writes only 128 bytes, the consumer's read returns only 128 bytes. Furthermore, had the consumer requested 64 bytes, the producer's write would fail.

This is not so much a mistake as a feature. UNIX-style pipe-lining is too transparent to use as a general-purpose IPC mechanism. Furthermore, the implicit buffer accessible to both producer and consumer is not very desirable in distributed systems. Moreover, UNIX-style pipe-lining can be provided, using IO, simply by inserting a buffer process between the producer and consumer.

It should also be noted that a process consuming a text stream from file or pipe, by reading a line at a time, in any case has to operate on a basis of "ask for a long line, but be prepared for a short one". Generalizing the I/O subroutine library will therefore make reading the "hand-shaked" pipe offered by IO, indistiguishable from reading a text file or a UNIX-style pipe as text.

7 Connectivety

An I/O port "opened" for operations on a process, rather than on a file or device, forms part of a point-to-point IPC link. Such a link can be fully specified as a pair of pairs

((process A, port X), (process B, port Y))

In other words, an IPC link is established by "connecting" two I/O ports. Consequently, an operating system need only add a system call for connecting I/O ports, in order to include IPC into the realm of classical I/O:

Connect (*ProcessA, PortX, ProcessB, PortY*)

A security conscious operating system can make this call safe by placing restrictions on allowable values for *ProcessA* and *ProcessB*. For example, if *ProcessA* or *ProcessB* is not the process number of the process executing Connect, then the operating system may require that it should be the process number of a child process. In other words, a process will be restricted to communicate only with its siblings and children.

This is not as severe a restriction as it may sound, since an arbitrarily interconnected network of processes can readily be set up, simply by loading them all from a common parent. And it does have the significant advantage that processes need never deal with stray or malicious messages.

Of course, such a restriction will make it impossible for a user process to use Connect to set up IPC links to multi-user servers that should already be part of the environment when the process starts up.

Fortunately, such server processes can readily be viewed as abstract devices. Consequently, I/O ports used for IPC between "user" process and "server" process would be opened for operations by using the classical system call:

Open (*process-num, io-port, object, desired-ops*)

where *process-num* denotes the process whose port is to be opened (either the calling process or one of its child processes), and *object* denotes the file, device, or server that is the target of the operations. *Desired-ops* denotes the set of operations the caller intends to be carried out. For example, *desired-ops* = {Read}, indicates that the object is to be opened in read-only mode.

This convention enables server processes to be integrated into the name space used to identify files and devices, and allows the operating system to regulate the creation, visibility and accessibility of servers in exactly the same way as for files. The conceptual simplication brought about by this strategy should not be underestimated.

8 More about servers

Server processes that are included into the file/device/server name space are "system objects" in the sense that they are not subprocesses of any of any "user" process. Consequently, a server cannot link one of its I/O ports to a new client process by calling Connect, or by having its parent process call Connect. Nor can a server sensibly call Open.

Consequently, an additional "non-classical" system call is needed:

Make-available (*io-port, in-len, in-buf*)

This call opens an I/O port without actually establishing a complete point-to-point IPC link. Once made "available", an I/O port is regarded as in "midst I/O" and an I/O operation can be completed on it by calling Complete-IO (either directly, or by specifying the "any port" value).

The completion of an I/O operation on an available port happens only when a server becomes the target of a call to Open by a prospective client. When this happens, the operating system chooses one of the server's available I/O ports and sends an "application to become a client" message to the port.

Receipt of an "application" message at an available port will make it possible for the server to complete an I/O operation on the port, and to receive the application in the process. The message (which can be trusted, since it comes from the OS and not from the client) fully identifies the client, the current privileges of the client, and the operations requested by the client. The server can then decide whether or not to accept the client, and indicate this by sending a specially formatted message via the port from which the application was received. The server's reply is intercepted by the operating system and used to decide whether or not the client's call to Open should be successful.

Note that, following an acceptance of the client by the server, the server is in "midst I/O" with a receive buffer set up by the call to IO that sent the acceptance message. Thus, no "initializing" I/O operation by the client is needed when the server is a system object rather than a "user" process. In other words, much of the agonizing done in section 5 need never concern the average programmer.

9 Multi-casting

Although IO has thus far been solely described as a point-to-point IPC mechanism, it is quite possible to use IO for multi-casting.

Of course, the operating system has to know about it. And the simplest way to add appropriate OS support is to generalize the concept of system servers into a system *groups* of servers. In other words, the OS should allow one to define a single name for a group of servers, and to add this name to the file/device/server name space.

When a prospective client opens an I/O port to a group of servers, the "request to become a client" message is sent to every server in the group, as are all subsequent messages sent via the port. In other words, the client automatically multi-casts to the server groups. Of course, the multiple replies received by the client present something of a problem.

One solution is to discard all but one of the replies (treating the others as though they are duplicates introduced by the network). For many multi-casting applications, this may be perfectly acceptable, and one would expect the OS to allow this approach, particularly since it makes multi-casting transparent to the client.

However, it should also be possible to use multi-casting in applications where the client needs to receive and process *all* the replies. This can be made possible, simply by adding an option to IO that would allow the client to set up a *list* of receive buffers when making the request. The client would then use multiple calls to Complete-IO or Try-to-complete-IO in order to receive the multiple replies.

Note that the semantics outlined here for the purpose of multi-casting can also be used to implement streaming (non-handshaked) protocols on top of IO: Several messages can be sent from a port without needing a "handshake" reply for each, and several messages can be received at a port without needing a "handshake" request for each. "Handshaked" request-reply interaction is merely the natural way of using IO, not the only way.

10 Reliability vs. Efficiency

Since the generalized I/O call should be the only explicit support given to IPC by the operating system, the implementation of IO should be as efficient as is possible. This rules out the use of a "heavy weight" protocol that ensures correct operation despite network failures.

When used for IPC, Initiate-IO should do little more than:

- check that the I/O port is open
- retrieve the destination process and port
- construct a message from the **in** parameters
- send the message

Furthermore, Complete-IO should do little more than:

- wait until a message is received
- unpack the message into the **out** parameters

However, protocol/hardware specific details, such as having to "packetize" messages, should naturally not be the concern of the programmer using IO. Which is to say that IO should support the exchange of arbitrarily sized messages over fallible virtual circuits.

Reliable IPC can readily be constructed on top of IO, provided that the semantics presented thus far are extended as follows:

- I/O ports must have time-outs associated with them, and an I/O must complete on a port (with an error code set) if no reply message is received within the time-out period.
- A new IO variant, Retry-IO must be introduced.

Retry-IO should be the same as Initiate-IO, except that the resulting message should be sent with the same sequence number used for the message which received no reply. This will enable the message to be discarded as a duplicate if the original message actually was delivered.

Apart from returning with an error code when a time-out occurs, Complete-IO should also return an error code when the operating system discovers that the request message is non-deliverable. A distributed OS can ensure that remote crashes are detected by requiring the remote kernel to send a "reassuring" reply to the local kernel, in answer to any message sent by Retry-IO. Failure to receive reassurance after a fixed number of retries could then be interpreted as a remote crash. Furthermore, a negative reply by the remote kernel can be used to indicate that the recipient process no longer exists.

Despite the fact that IO does not make network errors transparent to the programmer, it should not be assumed that IO should therefore not be used "as is" for IPC. Most uses of IPC occur in systems where the communications hardware (if any) are inherently reliable anyway. Network errors are therefore likely to be rare (much like classical I/O errors) and it will often be quite acceptable to simply abort a process encountering an error.

If aborting on occurence of a network error is not acceptable, then aborting on the occurence of a remote node/process failure is not likely to be acceptable. Since these errors are all detected in the same way (Complete-IO returns an error code), a fault-tolerant distributed software system may as well use a single error-recovery mechanism to recover from both communication failures and remote failures.

11 Adding some kitchen sinks

IO can be further extended in a variety of ways. For example, one can release a server process from the duty of supplying each client port with an input buffer capable of holding the largest possible request that can be made by a client. This can be done by allowing the server to specify an "installments" option when setting up an input buffer. If client attempts to send a request that is larger than the buffer supplied by a server accepting installments, its request will not fail. Instead, the request will (transparently to the client) be sent over installment by installment, with the server sending a reply to each installment. The details are left to the reader's imagination.

A similar scheme can be devised to allow a server to meet a large client output request, installment by installment – without the client being aware of it.

Put together, these schemes can also be viewed as support for scatter/gather operations – provided that system calls are relatively cheap to make.

It is even possible to extend IO so that it allows "no-copy" message transfers between processes with access to shared memory areas. This would be done by providing a sending process with a way to indicate that a buffer is sharable and need not be copied. In addition, a receiver must be able to indicate that it expects a sharable buffer, so that a pointer to the sharable buffer is returned in *in-buf*.**out** rather than placing a copy of the message in the buffer pointed to by *in-buf*.**in**.

"No-copy" message transfers can even be made portable to machines without shared memory. Allocating "sharable" buffers would simply allocate non-sharable buffers. Sending a "sharable" buffer would send a copy and de-allocate the buffer. A "sharable" buffer is received by implicitly allocating a receive buffer when the caller specifies that a sharable buffer is expected. This can all be hidden from the programmer by means of suitable subroutine libraries.

12 Conclusion

The essential equivalence between I/O and IPC has been recognized by many and is perhaps obvious to most. Nevertheless, a great many different sets of IPC-only primitives have appeared and continue to appear. Furthermore, operating system designers tend to extend existing operating sytems by adding IPC-only system calls, rather than by means of generalizing the I/O-only system calls.

Given the pletora of different and not-so-different sets of IPC primitives, and the relative consensus on I/O primitives, the prevailing approach is probably not the best one.

References

[1] B. H. Venter, *The design of a new general-purpose operating system.* PhD thesis, University of Port Elizabeth, 1987.