ISSN 0254–2757

QI QUÆSTIONES INFORMATICÆ

Volume 6 • Number 1

May 1988

The official journal of the Computer Society of South Africa and of the South African Institute of Computer Scientists

Die amptelike vaktydskrif van die Rekenaarvereeniging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

QUÆSTIONES INFORMATICÆ

The official journal of the Computer Society of South Africa and of the South African Institute of Computer Scientists

Die amptelike vaktydskrif van die Rekenaarvereniging van Suid-Africa en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Production

Mr Q H Gee
Department of Computer Science
University of the Witwatersrand
Johannesburg
Wits
2050

Subscriptions

The annual subscription is

	SA	US	UK
Individuals	R20	$ 7	£ 5
Institutions	R30	$14	£10

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Quæstiones Informaticæ is prepared by the Computer Science Department of the University of the Witwatersrand and printed by Printed Matter, for the Computer Society of South Africa and the South African Institute of Computer Scientists.

Editorial

Volume six of QI heralds several changes. The most visible is the change in format. The black on red cover has been changed to a more readable blue on white, but we have retained the style of the old cover, for the sake of continuity. The papers are now set in a tighter format, using double columns, which will enable more papers to be published for the same cost.

For authors, the most significant change is that as from Volume 6 Number 2 (the next issue), a charge will be made for typesetting. The charge is quite modest – R20 per page – and will enable us to keep up the high standards that we have become used to with QI. It is worth recording that the alternative to this suggestion was that authors should present camera-ready typescript, as is done for *Quæstiones Mathematicæ*. Given that document preparation and electronic typesetting is one of the areas of computer science that we can feel proud of, it seemed right that our journal should use the most modern techniques available. Fortunately, the two controlling bodies, the CSSA and SAICS, eventually agreed to our proposal and the result is the professional journal you have in front of you now.

Supporters of QI may be interested in a few statistics that I compiled when I took over the editorship from Gerrit Wiechers in April this year. In the past two years (June 1985 to June 1988), 73 papers have been received. Of these 39 (53%) have appeared, 19 have been rejected or withdrawn (26%) and 15 (21%) are either with authors for changes or with referees. If we look at the complete picture for Volumes 4 and 5, we find the following:

Volume	Issues	Papers	Pages	Ave. pages per paper
5	3	27*	220	7.7
4	3	21	136	6.4

Although this issue contains one very long paper of 18 pages, the future policy of QI will be to restrict papers to 6 or 7 printed pages, and prospective authors are asked to bear this in mind when submitting papers.

For the future, we are hoping to move towards more special issues. Many of the papers being published at the moment were presented at the 4th SA Computer Symposium in 1987. Instead of continuing the policy of allowing such papers to be accepted by QI without further refereeing, we are hoping to negotiate with Conference organisers to produce special issues of QI. Thus the proceedings would *ab initio* be typeset by QI and all the papers would be in a single issue. Given the competitive charges of QI, there will be financial gains for both parties in such an arrangement.

As this is my first editorial, it is fitting that it should close with a tribute to the previous QI team. My predecessor as editor was Gerrit Wiechers. Gerrit took over the editorship in 1980 and served the journal well over the years. With his leadership, the number and quality of the papers increased to its present healthy state. I must also extend a big thank you to Conrad Mueller and the University of the Witwatersrand who pioneered desk top publishing of QI in August 1985, using the IBM mainframe and its laser writer. Without Conrad's diligence and the excellent facilities provided by the Wits Computer Centre and subsequently the Computer Science Department, QI would easily have degenerated into a second-rate magazine. Quintin Gee, also of the Wits Computer Science Department, has taken over from Conrad and has raised the production quality of QI to new heights, as this issue testifies.

I look forward to your help and support in the future. Long live QI!

Judy M Bishop
Editor
June 1988

A New General-Purpose Operating System

B H Venter
Department of Computer Science, University of Fort Hare, Private Bag X1314, Alice, Republic of Ciskei

Abstract

The current generation of widely-used, multi-user, general-purpose operating systems have evolved from versions that were designed when many of the issues that are important today were unimportant or not even thought of. This evolution has not been totally successful. In particular, the current generation is ill suited for implementation on loosely-coupled multi-processors. A new operating system, designed with current requirements in mind, and flexible enough to adapt successfully to likely future requirements, has been developed as part of a project to build a loosely-coupled multi-processor system that should have the performance and functionality of a 'super main-frame' computer. This paper concentrates on describing the fundamental mechanisms of the operating system: processes, inter-process communication, and servers. It also briefly outlines the support provided for data security, database applications, and real-time applications.
***Keywords**: operating systems, system calls, inter-process communication, distributed systems, operating system security, operating system database support, real-time.*
***Computing Review Category**: D.4 OPERATING SYSTEMS*

Received June 1987, Accepted July 1987

1. Introduction and Motivation

A modern general-purpose operating system can be expected to provide a high level of data security, to provide appropriate support for a comprehensive database management system, and to support real-time applications. Furthermore, such a system can be expected to isolate applications from the underlying hardware. That is, the operating system should be implementable on most current and future hardware systems, and an application written in a standard high level language should be able to run on any hardware system running under the control of such an operating system.

Moreover, a computer controlled by a modern operating system should be able to form part of a network of distributed computers, and provide users with efficient, transparent access to the resources available via the network. In particular, a modern operating system should be able to make effective use of the loosely-coupled multi-processor systems that are beginning to appear.

The current generation of widely-used, multi-user, general-purpose operating systems have gradually evolved from versions designed in the 1970's and 1960's. Then, many of the issues that are of considerable importance today, were unimportant or not even thought of. In particular, loosely-coupled multi-processor systems were unforeseen, and operating systems were designed with a single-processor mind set, making it difficult to port these systems to loosely-coupled multi-processors

For example, current implementations of UNIX cannot even be ported to tightly-coupled multi-processors without major changes and preferably a complete rewrite [2]. The situation is even worse for loosely-coupled multi-processors: the latest 'standard', UNIX System V [4], allows arbitrary processes to share memory which cannot be done efficiently on a loosely-coupled multi-processor. Furthermore, the message-passing mechanism which is of critical importance to distributed applications - is far too complicated and cumbersome to be implemented efficiently. In fact, the designers of this message-passing mechanism have apparently assumed that the sender and receiver processes share access to a common physical memory. There are also other aspects of UNIX which, upon close examination, prove difficult or impossible to implement effectively on a loosely-coupled multi-processor.

The situation is not much better when one considers other widely-used current generation operating systems. Thus, if one aims to build a new computer system based on a loosely-coupled multi-processor, a substantial operating system development effort must be undertaken.

If compatibility with existing software is one's principal concern, one should aim to implement either a UNIX 'look-alike', or an MVS 'look-alike'. However, as pointed out above, UNIX is not well suited to the role. The same is true for MVS.

If, on the other hand, making do with limited resources is one's principal concern, then it makes sense to develop a new operating system, with full use being made of what has been learnt about

operating systems since the 1970's.

The author is currently involved in the development of a loosely-coupled multi-microprocessor system. The aim is that this system should provide 'super main-frame' functionality and performance. The project is being undertaken with relatively limited resources, and building a working system with the available resources is considered more important than providing compatibility with some existing software base. Consequently, a new operating system has been designed for this computer, and a fairly complete 'quick-and-dirty' prototype has already been implemented. A full-scale implementation, using the prototype operating system as the development system, is currently under way.

The rest of this paper is a brief survey of the main features of the operating system design. The intention is not rigorously to justify design decisions, nor to point out what contribution the work makes, but rather to provide the reader with an overall description and enough detail to compare the new operating system to any existing operating system. A more detailed description can be found in [3].

2. Processes

A new process can be created either by forking an existing process, in the style of UNIX, via the following two system calls: ('->' means 'returns')

```
fork_process (process_num, monitoring_io_port)
-> process_num replace_image (file_num,
entry_point)
```

or by loading the process' starting image directly from a file:

```
load_new_process (file_num, monitoring_io_port)
-> process_num
```

The latter method is more appropriate for a loosely-coupled multi-processor. It makes little sense to copy the current memory image of the parent process over the network to the processor that is to execute the new subprocess, only to replace it soon afterwards with a new image obtained from a file, as is usually the case. Forking is only provided to facilitate UNIX compatibility.

Note that, unlike UNIX, a process can fork not only itself, but also one of its subprocesses. Furthermore, a parent process can be informed of all state changes in its subprocesses, and can exert complete control over them, with the ability to terminate, suspend, or restart a particular subprocess.

Furthermore, unlike UNIX, a new process is created in a suspended state, and execution must be started explicitly by its parent. This allows a process to load several subprocesses and to set up communication links between them while they are in a known state.

A parent process may also suspend an executing subprocess and then restart execution at a different instruction. The suspend/restart mechanism is formulated such that the restarted subprocess can execute either an interrupt handler (eventually returning to the point of interruption), or an exception handler (never returning).

As is to be expected, a process can suspend or terminate itself and supply a reason code that will be received by its parent. A process can also dynamically obtain additional memory from the operating system, and release unused memory for use by other processes.

The operating system does not normally allow processes to share access to the same area of memory, since, in general, it is not possible or desirable to ensure that processes execute on physical processors that have access to a common physical memory bank. Memory sharing is therefore discouraged by limiting it to specially privileged 'server' processes. (Server processes are discussed in Section 4.)

It is, however, possible to simulate a form of memory sharing between different 'light-weight' processes executing on the same processor by incorporating a scheduler into the code of a single 'heavy-weight' operating system process. Such 'light-weight' schedulers can ignore most of the fairness, synchronisation, and security issues that an operating system must address, and can thus provide a much cheaper form of single-processor concurrency than the operating system. A typical user of 'light-weight' processes would be a server process that must serve several clients concurrently.

3. Inter-Process Communication

As Hoare pointed out [1], transferring information from a process to a process does not differ from transferring information from a device to a process, or from a process to a device. In fact, in a modern operating system implementation, device drivers are likely to be implemented by processes.

Consequently, the operating system provides a generalised I/O call as the principal means whereby processes must interact with their environment. This system call corresponds to the classical I/O call of current generation operating systems in most respects, generalising it only in so far as to allow I/O operations to be performed on processes as well as on files and devices, and by allowing a single operation to perform both output and input.

The I/O call is invoked as follows:

```
io (io_port, operation, address, out_len, out_buf,
in_len, in_buf)
```

Note that all the parameters, except out_buf, may be updated by the call.

An I/O port corresponds to a UNIX file descriptor, but may represent another process, as well as a file or device. Furthermore, up to sixteen different I/O

operations may be carried out on an I/O port. For example, when an I/O port is linked to a file, the following operations are supported:

0 = read next string (length given in in_len)
1 = write next string (length given in out_len)
2 = read string at offset from file start (given in address)
3 = write string at offset from file start
4 = append string at end of file
5 = get current length of file (result in in_len)
6 = set current length of file to length in out_len
7 = flush updates to non-volatile storage

As can be seen, some operations are input operations, others perform output, and still others do neither. It is also possible to have operations that perform output as well as input. In general, the subset of operations that can be carried out on an I/O port and the effect of the operations depend on the kind of object to which the I/O port is linked. When an I/O port is used as an inter-process communication link, the programmer has full control over the subset of allowable operations and the interpretation given to them.

An I/O port is linked to a file, device, or server via:
open (process_num, io_port, object_num, desired_ops) -> available

Note that a process can open not only its own I/O ports, but also those of subprocesses; object_num identifies the file/device/server; desired_ops is a bit map indicating the subset of the sixteen possible operations that the caller wishes to carry out on the I/O port. (For example, to open a file for sequential read-only access, desired_ops must have a value of 0000000000000001.)

desired_ops) -> available

An I/O port is linked to a process via:
connect (process_a, io_port_a, bitmaps_a, time_out_a, process_b, io_port_b, bitmaps_b, time_out_b)

A process may connect the I/O ports of its subprocesses to each other, or may connect its own I/O ports to the I/O ports of subprocesses. Thus a process can communicate with its subprocesses, siblings, parent, and files/devices/servers.

Note that each I/O port has a bit map associated with it, indicating which I/O operations may be carried out on it, as well as a bit map indicating which I/O operations perform output. Additionally, a time-out is associated with each port. These values are explicitly specified for ports opened with connect, but are determined by the object to which the port is linked for ports opened with open.

The general progression of an I/O call is as follows:

- send operation,address,out_len,in_len to the other process (append contents of out_buf if the operation calls for it)
- wait for a response from the other process (return with an error code if no response within time_out)
- return operation,address,out_len,in_len sent by the other process, and store rest of its response in in_buf (sender out_len = receiver in_len, and sender in_len = receiver out_len)

When an I/O port has just been connected to another, the first operation that outputs the contents of out_buf is delayed until the other process performs an input operation on its corresponding I/O port. The I/O call of the process that performed the input operation then completes, returning the parameters and buffer contents supplied by the process that performed the output operation (resulting in a data transfer taking place; see also figure 1). Meanwhile, the outputting I/O call is suspended while awaiting a response. This response is received when the process that performed the initial input operation performs its next I/O operation. The response consists of the parameters and possibly the contents of out_buf, supplied to the second I/O operation.

Thus, two communicating processes are always synchronised so that the initiation of an operation by one, causes the completion of an operation by the other, as well as a data transfer. The result is that communication takes on a 'hand-shaked' request-response nature, and that when one process sends output to another, the other has already set up an input buffer to the receive the transferred data. This eliminates the need for a system buffer pool and takes care of flow-control and synchronisation.

When a port is opened to a file/device/server, the operating system engages in a dialogue with a corresponding file-driver-process/device-driver-process /server-process, which results in an inter-process communication link being set up between the client process and a driver/server process. After the open, the driver/server is waiting to complete an I/O operation, and the client process has yet to perform its first operation.

When the client performs its first I/O operation on the port, the incomplete I/O at the driver/server completes, resulting in the driver/server receiving the request made by the client. After serving the request, the driver/server responds by initiating a next I/O operation, which in turn causes the client's incomplete I/O operation to complete, returning the desired results.

Naturally, the I/O call allows a proceed option (indicated by setting a modifier bit in operation). A proceed I/O call sends the request to the other process, but does not wait to receive the response. The requesting process can later complete the call and. receive the response by performing another I/O call on the port – setting a modifier bit to indicate whether or not to wait for the response, if this has not yet been received.

It is possible for a process to have a number of I/O ports on which proceed I/O operations were carried

out. Therefore, it is possible to perform an I/O call that will complete any one of these incomplete calls. This facility will typically be used by a server process that serves many clients concurrently, and thus may be waiting for several requests (that is, have several incomplete I/O calls) at the same time.

The I/O call also supports broadcasting/multi-casting, as well as scatter/gather transfers.

4. Servers

Server processes are the principal means by which the operating system provides its services. Furthermore, servers can be used as the basic blocks for building distributed applications.

A server process has a globally visible name, drawn from the same name space used for files and devices. Thus, a prospective client establishes an inter-process communication link with a server by calling open. The operating system carries out a call to open by sending an unsolicited message to the target of the open call. Whether it is a file, device, or explicit server, the target of an open call is always a process; hence the term 'server' will from now on be understood to include files and devices.

When a server starts executing, it sets up a number of 'reconfigurable' I/O ports, using:

make_reconfigurable (io_port, valid_op_bitmap, in_len, in_buf, time_out)

While reconfigurable, an I/O port does not represent a communication link to any particular process, but acts as a receiving port for unsolicited messages. Thus, when the operating system sends an unsolicited message to a server, one of the server's reconfigurable I/O ports is selected to hold the message, and the server will receive the message when it tries to complete an I/O call on that port (it will usually try to complete an I/O call on any port). Note that make_reconfigurable sets up the buffer to hold the unsolicited message.

The unsolicited message received by the server

a) can be trusted because it comes from the operating system
b) fully identifies the prospective client, its privileges, the type of access required, and so on.

After receiving such a message, the server must decide whether or not to accept the client and indicate this by outputting an appropriate message through the I/O port that received the request. If the client is accepted, the I/O call indicating the acceptance remains incomplete until the client performs its first I/O operation on the port it has just opened successfully. Thus, when a client is accepted via a reconfigurable I/O port, the I/O port is configured as a dedicated inter-process communication link between client and server. Note that, among other things, the server's acceptance message supplies information to be associated with the client's I/O port: the valid operations, the operations that output out_buf, and the time-out to be used when waiting for a response from the server.

A server process can be tied to a particular processor of a multi-processor system, in which case it is loaded when that processor bootstraps. Alternatively, a server can be 'untied', in which case it is loaded into any available processor when first referenced by an open call. Untied servers usually terminate when they have no more clients.

Designating a process as a server, thus giving it global visibility, is a privileged operation since a server is trusted to take part in the 'new client' protocol. Servers are also the only class of processes that can be allowed to perform certain privileged operations.

For instance, a server that is tied to a particular processor can be used as a device driver by allowing it the privilege to access I/O space and to install interrupt handlers. A tied server can also be allowed to share memory with other servers tied to the same processor.

It follows that an implementation of the operating system will itself largely consist of a set of server processes distributed among the various physical processors. Adding new servers to the collection making up the basic operating system will be straightforward, resulting in an 'open', extensible, adaptable operating system. Furthermore, structuring the operating system as a set of servers communicating via messages greatly facilitates the transparent integration of local resources (services) with resources available, via a network, from other systems.

The server mechanism also allows a single service (that is, a single object in the name space) to be implemented by several co-operating processes. One way to implement such a multi-process server is to designate several server processes as the members of a single server group, identified by a single 'group object' number. When a client performs an I/O operation on an I/O port linked to a group, the request message is broadcast to all members of the group. It is possible for the structure of the group to be invisible to the client, in which case the members must use a protocol to ensure that only one response will be generated. Alternatively, a client may be required to be aware of the group structure, in which case the client must set up additional I/O ports to receive the multiple responses (an I/O port can receive at most one response for each request).

It is also possible to implement a server group, without using broadcasting, by means of a co-ordinator process. In this case, client requests go to the co-ordinator, and the co-ordinator then 'subcontracts' them to the other processes co-operating to provide the service. A co-ordinator can subcontract all client interaction, in which case the 'request to become a client' message goes to the co-

ordinator and is subcontracted, following which all further interaction is between client and subcontractor (unbeknown to the client). Alternatively, a co-ordinator can subcontract individual requests, inwhich case all requests first go to the co-ordinator and then to the subcontractors.

Note that subcontracting and broadcasting can be combined. It is thus possible to exploit multiple processors to achieve both speed and fault-tolerance, while providing clients with the illusion of dealing with a simple 'single' server.

5. The File System

The file system is intended to facilitate the storage and retrieval of arbitrary strings of bytes, such as text files, object files, and process images. The file system is not intended as an efficient or convenient store for records since it is more reasonable to use a database management system to store and retrieve such data.

The file system is implemented as a set of communicating servers, with each open file having a corresponding 'driver' process that actually receives and acts upon the I/O operations that client processes perform on I/O ports linked 'directly' to files. The file system servers are structured as a hierarchy and in such a way that the database management system can exist 'alongside' the file system, by making use of the lower level 'disk block' servers instead of the file system visible at system call level.

Files are identified by numbers drawn from a global name space that includes devices and servers. This allows user interfaces to use arbitrary symbolic name-to-file number mappings, further adding to the 'open', extensible, adaptable nature of the operating system.

Files are allocated in two steps:

create_temporary_file (logical_vol_num, size_hint) -> file_num make_file_permanent (file_num, future_expansion-hint)

Temporary files are automatically reclaimed when the process that allocated them terminates. By converting these temporary files into permanent files, rather than directly allocating permanent files, it is possible to follow a protocol that results in the allocation of a file and the recording of a symbolic name-to-file number mapping as an atomic operation.

The size hints that may be given when a file is allocated allow the operating system to pre-allocate space for a file so that sequential access is optimised. The file system need not heed the hints and will never refuse to allocate or grow a file because a suitable run of disk blocks is not available.

The design of the file system attempts to minimise the visibility of physical disk volumes to application programmers in order to promote program transportability: All permanently on-line storage media are consolidated into logical volume zero and files from all volumes are identified from a single name space.

However, it is necessary to introduce the concept of distinct volumes to cope with removable disk packs. Clearly, the operating system cannot just incorportate removable disk packs into a global pool and allow arbitrary files to be allocated on removable disk packs. Consequently, removable disk packs are grouped into one or more logical volumes, any of which can be off-line. File allocations on these volumes must be indicated explicitly and can only be performed by suitably authorised users.

6. Security

Users gain access to the operating system by interacting with a user authentication server. It is possible to associate an arbitrary authentication server with a given user access device, and to restrict a given server to admitting only a subset of users. Thus, it is possible to make it reasonably easy to gain entry as a casual user, while making it arbitrarily difficult to gain entry as a privileged user.

When a user gains entry into the system, the corresponding user access device is assigned to an appropriate user interface process, which is 'executing on behalf of' the user. The 'executing on behalf of' property of a process is inherited by all subprocesses, and can only be changed if a process is a specially privileged server process (for example, an authentication server). Thus, all actions initiated by a user are carried out by processes authentically executing on behalf of the user.

Files, devices, and servers all have owners, and are accessible only to processes executing on behalf of the owner or a user explicitly mentioned in an access list - a file listing all the users that may access the protected object, with each entry indicating an individual set of privileges for the user. (Note that a server may reject a client even if the client is listed in its access list.)

Additionally, files, devices, servers, and processes are associated with 'information categories'. An arbitrary number of information categories can be established, and the operating system enforces a set of rules that ensure that information cannot 'flow' from one category to another, unless specifically permitted. This is basically achieved by limiting a process to accessing only objects associated with the same information category, while also allowing a process to have read-only access to objects associated with categories that have flow paths to the category of the process. A process may only change its category if its user is permitted to operate in the new category and if the change cannot result in an illegal flow of information.

The operating system also includes mechanisms for limiting software piracy, denial of resources, Trojan Horse attacks, and the use of covert channels to subvert information flow controls.

7. Database Support

Database managers are principally supported by the server mechanism. Furthermore, the file system is structured in such a way that a database manager can access disk blocks directly.

The only additional support provided by the operating system is in the form of a transaction co-ordinating server. This server is accessed by client processes via the system calls:

```
start_transaction
commit_transaction -> success_indicator
abandon_transaction
```

These calls keep the transaction co-ordinator up to date on the transaction status of processes, and it, in turn, keeps 'transaction supporting' servers up to date on the transaction status of their clients.

The transaction co-ordinator co-ordinates a two-phase commit, provides a 'centralised' deadlock detection service, and allows transaction supporting servers to use a wide range of synchronisation strategies, including optimistic strategies.

While it is logically a centralised service, the transaction co-ordinator can be implemented as a distributed set of co-operating processes, and thus does not preclude the implementation of a distributed database manager.

8. Real-Time Support

The operating system allows the minimum rate of execution of a process to be specified directly rather than be influenced indirectly by means of priorities. Furthermore, processes are classed as 'real-time' or 'ordinary', with real-time processes pre-empting ordinary processes. All operating system server processes execute as ordinary processes, thus giving real-time application programmers complete control over processor allocation.

Note also that server processes can be granted the privilege of handling interrupts directly, as well as to share memory with similar servers on the same processor. Furthermore, the operating system can be configured with a separate 'real-time' file system that serves only real-time processes and thus isolates real-time processes from interference by non-real-time processes in this aspect as well.

9. Conclusion

This paper has briefly outlined the design of an operating system that is hardware independent, provides a high level of data security, and supports distributed applications, database applications, as well as real-time applications.

This has not been achieved by introducing radically new concepts, but by carefully reformulating, generalising, and integrating the basic concepts that any operating system must support.

Firstly, the process concept has been stripped of restrictions and assumptions, leaving a straightforward mechanism that can be utilised efficiently for a wide range of applications. Secondly, the input/output mechanism has been generalised to provide a straightforward, efficient form of inter-process communication. Thirdly, the concept of a device has been generalised into the concept of a server, on which the operating system itself is largely based. Servers make the operating system flexible and extendible, and provide a suitable mechanism for distributed implementations.

References

[1] C.A.R. Hoare, [1978], Communicating Sequential Processes, *Communications of the ACM*, **21** (8), 666-677.

[2] M.D. Jannsens, J.K. Annot and A.J. Van de Goor, [1986], Adapting UNIX for a multiprocessor environment, *Communications of the ACM*, **29** (9), 895-901.

[3] B.H. Venter, [1987], The Design of a new general-purpose operating system, Ph.D. Thesis, University of Port Elizabeth.

[4] XELOS Programmer Reference Manual, Perkin-Elmer Corporation, New Jersey, 1984.

This paper first appeared in the Proceedings of the 4th South African Computer Symposium.

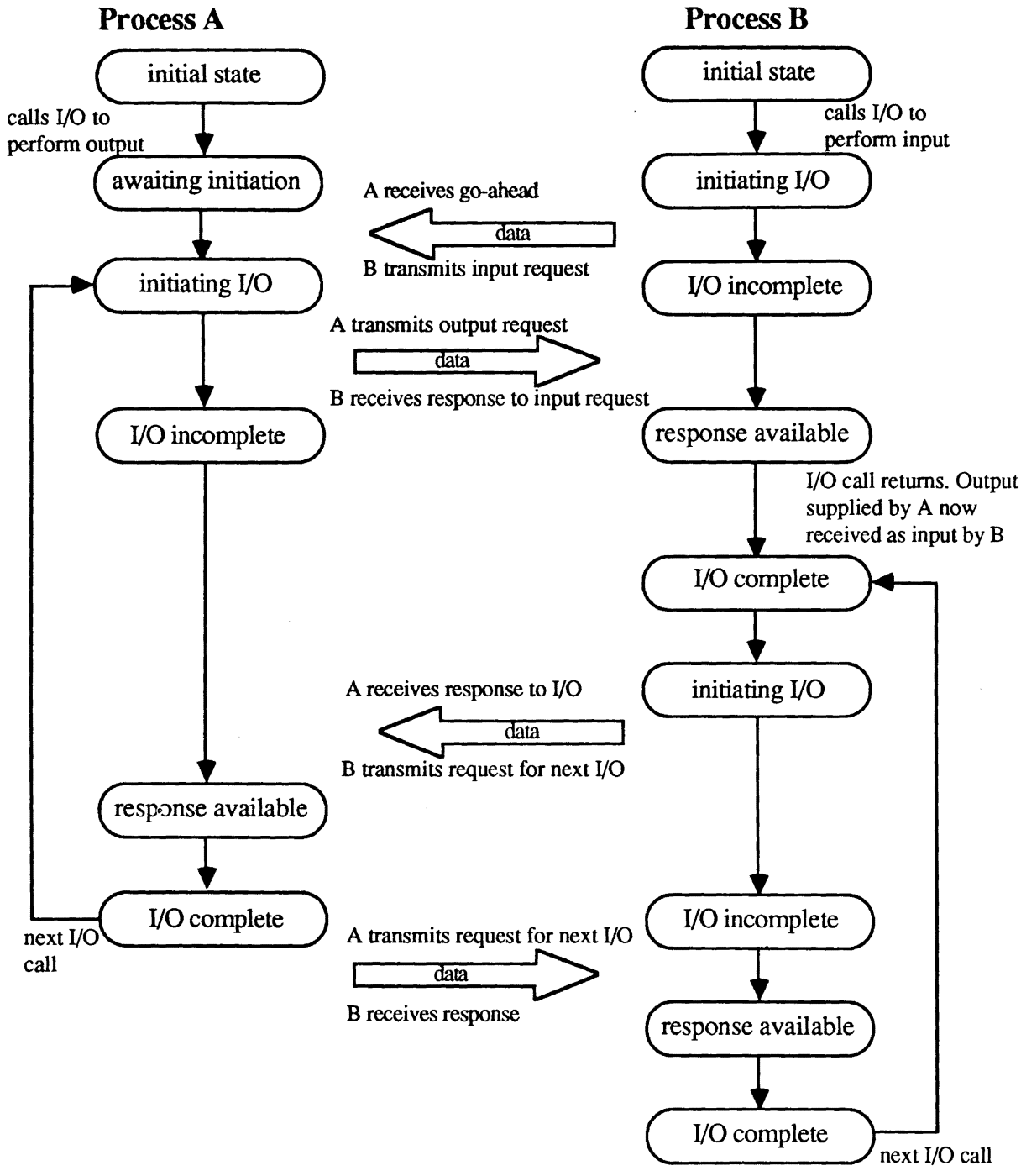

Figure 1 I/O state transition diagram

NOTES FOR CONTRIBUTORS

The purpose of the journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles and exploratory articles of general interest to readers of the journal. The preferred languages of the journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to:

Professor J M Bishop
Department of Computer Science
University of the Witwatersrand
Johannesburg
Wits
2050

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins.

The first page should include the article title (which should be brief), the author's name and affiliation and address. Each paper must be accompanied by an abstract less than 200 words which will be printed at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review Categories.

Manuscripts may be provided on disc using any Apple Macintosh package or in ASCII format.

For authors wishing to provide camera-ready copy, a page specification is freely available on request from the Editor.

Tables and figures

Tables and figures should not be included in the text, although tables and figures should be referred to in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Figures should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the authors name and figure number. Original line drawings (not photocopies) should be submitted and should include all the relevant details. Photographs as illustrations should be avoided if possible. If this cannot be avoided, glossy bromide prints are required.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters; between the letter O and zero; between the letter I, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetic order of the author's name, and cited in the text in square brackets. Journal references should be arranged thus:

[1] E. Ashcroft and Z. Manna, [1972], The Translation of 'GOTO' Programs to 'WHILE' programs, *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.

[2] C. Bohm and G. Jacopini, [1966], Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.

[3] S. Ginsburg, [1966], *Mathematical Theory of Context-free Languages,* McGraw Hill, New York.

Proofs

Proofs will be sent to the author to ensure that the papers have been correctly typeset and *not* for the addition of new material or major amendment to the texts. Excessive alterations may be disallowed. Corrected proofs must be returned to the production manager within three days to minimise the risk of the author's contribution having to be held over to a later issue.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.