On the safety of the IO primitive in Concurrent PASCAL

A. Silberschatz**

Department of Mathematical Sciences, The University of Texas at Dallas Richardson,
Texas 75080, USA

In Concurrent PASCAL the peripheral device disc is viewed as an array of pages which can only be accessed via the standard procedure IO. One of the input parameters to the procedure is an index I to indicate which page in the array has to be accessed. The IO procedure can be invoked from any system module and the index I can be set arbitrarily by that module. Hence, one system module can jeopardise the integrity of a system written in this language.

This paper proposes an extension to Concurrent PASCAL to resolve this difficulty. In particular, we define a new concept scope which specifies the names of the program components which can declare an instance of a particular type. Given this concept and the program component manager, we devise a mechanism which will be shown to be consistent with the design goals of Concurrent PASCAL and which can be used to enforce processes to use the IO procedure with the index I set to only those pages which they have a legal right to access.

(Received January 1978)

Concurrent PASCAL is a programming language for structured programming of computer operating systems (Brinch Hansen 1975a; 1975b). The language supports the definition of abstract data types by the program component type class. It also supports sharing of abstract data types among processes by implementing the monitor (Brinch Hansen, 1973; Hoare, 1974) concept. In addition to providing a mechanism by which a data segment may be safely shared, the language only allows processes to share access to a data segment if that segment has been declared within a monitor. Moreover, a program component can access a data segment only if it has been granted an access right to that segment at compile time. This is accomplished via the permanent parameter mechanism explained below.

Suppose that program components A and B wish to share an access right to a monitor M of type Mtype. In order to accomplish this they must declare in their headings a permanent parameter of type Mtype. During the initialisation phase of A and B these permanent parameters can be set to point to the program component M. This in fact is the only way components A and B can gain a permanent access to M. Thus a program component can invoke an externally declared monitor only if it had been granted an access right to that monitor at compile time.

In Concurrent PASCAL the peripheral device disc is considered to be a monitor implemented in hardware. It can only be accessed by a single procedure IO which can be invoked by any program component. This violates the underlying principle in Concurrent PASCAL that a program component can access a data segment only if it has an access right to it.

At first glance it appears that this problem could be resolved by interpreting that every program component in the system has in its heading an implicit permanent parameter declaration to the peripheral device disc. This, however, results in systems which are difficult to verify since the program components can access the disc in some arbitrary way as explained below.

The standard procedure IO (Block: integer, P: Param, Device: integer) can be used to transfer a block to or from the disc. The IO Param is a record,

```
var Param: record
  Operation: iooperation;
  Result: ioresult;
  Pageno: integer;
end;
```

that defines an input/output operation, its result, and a page number on the disc.

Since Pageno can be set arbitrarily by the program component invoking the IO operation, it is possible for that component to access disc pages which are associated solely with another process. This is clearly an undesirable situation.

Alternatively, one might permit only a small 'reliable' subset of the program components in a system to be able to invoke the IO operation. This however results in a considerable overhead and loss of parallelism since every IO transfer will require at least one call to a 'reliable' component; in the very least, each of these components must check the identity of the calling process (a mechanism not available in Concurrent PASCAL), decide if it can access a particular page on the disc, etc. Thus this solution is not an attractive one and alternative methods should be considered to resolve this difficulty.

The problem

The problem of securing disc accesses, motivated in the previous section can be viewed as follows: We are given a data abstraction disc with an array A[1 ... N] of pages declared within it. The array can be accessed only via a single procedure IO (x, y, i), where i specifies the i-th element in A to be accessed. The array is partitioned among several program components; that is, an element A[j] may be associated with n > 0 different program components. We wish to devise a mechanism which will guarantee that a program component P can access only those elements in A associated with it. Moreover, the mechanism to control this should not be embedded within the data abstraction disc.

Note

We do not specify how the association between disc pages and the program components is determined. This will be discussed in the next section.

As we have stated before we must disallow the use of the IO operation directly by the program components wishing to access the disc. Instead, we will provide the users with a new operation IODisc to transfer data to and from the disc. The IODisc operation will be an entry procedure to an ordinary class. This class will be the only system type allowed to invoke the IO operation. In the next section a mechanism will be introduced to guarantee that this requirement is

*An early version of this paper was presented in the IEEE Computer Software and Application Conference, Chicago 1977
**This work was partially supported by the National Science Foundation under Grant No. MCS 7702463
observed. The \texttt{IOdisc} parameters include some of the parameters required by the IO operation plus a record type variable defined as,

\begin{verbatim}
type Discbound = record
    Base: integer;
    Limit: integer;
end
\end{verbatim}

The class containing the \texttt{IOdisc} operation is defined as,

\begin{verbatim}
type Discaccess = class (Device: integer);
    Procedure entry IOdisc (D: Discbound; P: Param; var Block: Page);
    begin
        if (P.Pageno < D.Base) or (P.Pageno > D.Limit)
        then P.Result = wrong-page
        else IO (Block, P, Device);
    end;
endclass.
\end{verbatim}

Having provided the \texttt{Discaccess} class to ensure that an IO operation can access only disc pages \texttt{Base}, \texttt{Base + 1}, \ldots, \texttt{Limit}, our next task is to provide a mechanism to control access to \texttt{Discbound} instances. Clearly, if every program component could declare instances of type \texttt{Discbound} then we return to our original problem since these components could initialise the \texttt{Base} and \texttt{Limit} as they please. In order to prevent this, we will use the resource manager concept introduced in an earlier paper (Silberschatz, Kieburtz and Bernstein, 1977a). A resource manager is a program component type designated to distribute resources among customer processes. The resources that a manager distributes are instances of a previously defined type. These instances must be declared within the address space of a manager and thus processes can gain access to an instance only by executing an allocation procedure within the manager. The manager has access to the several instances of the managed resource, as well as to other variables that may be used for bookkeeping.

The syntax of a manager definition is similar to that of a monitor or class definition, except for the heading. The heading has the form

\begin{verbatim}
type \langle id \rangle = manager of \langle id_1 \rangle: \langle type identifier \rangle.
\end{verbatim}

The \texttt{\langle type identifier \rangle} declares the type of the managed resource, and \texttt{\langle id \rangle} declares the name of an implicit parameter, passed in any call to one of the manager procedures. The implicit parameter is either a null reference, or, a reference to one of the instances of type \texttt{\langle type identifier \rangle} declared within the manager.

For example, suppose one defines a manager using the heading:

\begin{verbatim}
type Mtype = manager of Tref: Ttype;
\end{verbatim}

and declared

\begin{verbatim}
var M: Mtype;
\end{verbatim}

Then a number of instances of \texttt{Ttype} would ordinarily be declared within \texttt{M} and one of these instances could be referred to from a different program component if it contained the declaration.

\begin{verbatim}
var T: Ttype from M;
\end{verbatim}

This notation indicates both the type of \texttt{T} and the identity of its manager. The initial binding of \texttt{T} that results from such a declaration is not to any particular instance of an address space for type \texttt{Ttype}, but is a null reference. The name \texttt{T} can be bound to an instance of a dynamically allocatable type, and \texttt{M} is the program component that can accomplish the binding.

In addition to distributing resources to customer processes, the manager can distribute different access privileges (Kieburtz and Silberschatz, 1978). For example, if \texttt{Ttype} defines entry procedures \texttt{P}_1, \texttt{P}_2, and \texttt{P}_3, the manager \texttt{Mtype} can distribute to one process the capability to invoke only procedures \texttt{P}_1 and \texttt{P}_2, and to another process the capability to invoke \texttt{P}_2 and \texttt{P}_3. In case of a record type variable, the access privileges are read

and write. The manager thus can allow customer processes to either read a managed record, write onto a managed record, or both.

Given this mechanism, we will define a manager for distributing discbound instances as follows:

\begin{verbatim}
type Discmanager = manager of Discref: Discbound;
var Discboundarray: array [1 .. N] of Discbound;

procedure entry Create (var B: integer; var L: integer);
begin
    This procedure binds one of the instances Discboundarray[I] to a customer process; initialises Base and Limit of Discboundarray[I] according to some policy; gives the customer process the privilege to only read from the instance it has been allocated; returns in \texttt{B} the value of \texttt{Base}; returns in \texttt{L} the value of \texttt{Limit};
end;
\end{verbatim}

Our task is not done yet; we must provide a mechanism to guarantee that \texttt{Discbound} instances cannot be declared by arbitrary program components. Such a mechanism must be provided since otherwise every program component could define a manager for distributing \texttt{Discbound} instances. These components then could use their own declared manager to initialise the base and limit of \texttt{Discbound} instances as they please.

In order to prevent this, we must restrict the class of program components that can declare an instance of a particular type. For example, suppose that we have defined the type \texttt{Mtype} and wanted to ensure that only program components \texttt{X} and \texttt{Y} can declare an instance of \texttt{Mtype}. This will be accomplished by declaring within the definition of \texttt{Mtype} the following:

\texttt{scope X(1), Y(2);}

This notation specifies that only components \texttt{X} and \texttt{Y} can declare an instance of type \texttt{Mtype} further, \texttt{X} can declare one instance of this type while \texttt{Y} can declare two instances.

In order to ensure that the \texttt{Discmanager} is the only component which can declare instances of type \texttt{Discbound}, we will declare

\begin{verbatim}
scope Discmanager (N);
\end{verbatim}

The compiler then can check and verify that \texttt{Discbound} instances are indeed only declared in the type \texttt{Discmanager};

The scope restriction can also be used to allow only one declaration of an instance of type \texttt{Discmanager} in the initial process (e.g. declaring in the type \texttt{Discmanager} scope initial (1)). This will ensure that the policy as to how the initialisation of \texttt{Base} and \texttt{Limit} of each instance of type \texttt{Discbound} is controlled in one and only one program component.

\section*{Enforcing the use of the IOdisc operation

In order for the above described scheme to work, we must ensure that the \texttt{Discaccess} is the only type that can invoke the IO operation. This can be achieved in two different ways:

(a) Using the \texttt{scope} restriction described in previous section.

(b) Using compile time environments (Silberschatz and Madison, 1977).

\section*{Scope restriction

In order to use the \texttt{scope} attribute, we must first describe how the IO operation is handled in the system. The peripheral devices are represented in the kernel by classes of the form:

\begin{verbatim}
type Peripheral = class (Device: integer);
var User: Process;
procedure Start (Operation: T);
begin
    pre-empt the calling process;
end;
\end{verbatim}

\end{verbatim}
put appropriate information in
the IO device registers;
end;
procedure Interrupt;
begin . . . end;
begin (initialisation)
User: = nil;
end.

An IO statement in Concurrent PASCAL is translated into a
call to procedure Start of the appropriate device. One can
therefore view the IO operation as an entry procedure to this
class. We will adopt this view and thus require that a program
component can invoke the IO operation only if it has an access
right to the appropriate peripheral device.

In order to guarantee that processes can access disc pages
only through the IODisc operation, we will declare within the
type Peripheral:

\[\text{scope Discaccess} (\cdot)\];

Hence, a process cannot invoke the IO operation directly;
instead it has to call upon the IODisc procedure in order to
access disc pages.

Compile time environments

In an earlier paper (Silberschatz and Madison, 1977), we have
introduced a mechanism through which high level languages
(e.g. Concurrent PASCAL) can provide machine dependent
features and yet remain safe. The safety is achieved by com-
piling program modules in a restricted compile time environ-
ment in which only certain features of the language are
granted to the module. Each module thus has all the power it
needs and yet may not abuse the language by utilising other
features. Providing these checks at compile time means run
time overhead is unnecessary. Such a mechanism allows all
aspects of an operating system to be written in a single high
level language easily and efficiently with no loss in security.

Using this mechanism one can ensure that no module in a
system, except instances of type Discaccess can use the IO
feature. Thus, a process can access the disc only through the
IODisc operation.

Example

Let us illustrate how the above described mechanism works.
We will rewrite the types Disbuffer and Virtualdisc described
in Brinch Hansen (1975a) to conform to our specifications.

\[\text{type Disbuffer} = \text{monitor} (Z: \text{Discmanager}; \ldots);\]
\[\text{var} \ X: \ \text{Discbound from} \ Z;\]
\[\text{Vdisc: Virtualdisc;}\]
\[\text{Sender, Receiver: queue;}\]
\[\text{Head, Tail, Length, Base, Limit, Diff: integer};\]

\[\text{procedure entry Send (Block: Page); begin}\]
\[\text{if Length = Diff then delay (Sender)};\]
\[\text{Vdisc.Write (X, Base + Tail, Block);}\]
\[\text{Tail: = (Tail + 1) mod Diff;}\]
\[\text{Length: = Length + 1};\]
\[\text{continue (Receiver)};\]
\[\text{end};\]

\[\text{procedure entry Receive (var Block: Page); begin}\]
\[\text{. . .}\]
\[\text{end};\]

\[\text{begin (initialisation)}\]
\[\text{X.Create (Base, Limit)};\]
\[\text{Diff: = Limit - Base;}\]
\[\text{Head: = 0};\]
\[\text{Tail: = 0};\]

\[\text{\footnotesize \*For the purpose of this paper, we ignore the details of how other peripheral devices are handled in the system.}\]

\[\text{Length: = 0};\]
\[\text{init Vdisc ( )};\]
\[\text{end.}\]

\[\text{type Virtualdisc = class (Discscheduler: Resource; . . . );}\]
\[\text{var Realdisc : Disc;}\]

\[\text{procedure entry Read (X: Disbound;read); Pageno: integer;}\]
\[\text{var Block: Page};\]

\[\text{begin}\]
\[\text{repeat}\]
\[\text{Discscheduler.Request;}\]
\[\text{Realdisc.Read (X, Pageno, Block, Error);}\]
\[\text{Discscheduler.Release;}\]
\[\text{if Error then Terminal.Write ('disc failure');}\]
\[\text{until not Error};\]
\[\text{end};\]

\[\text{endclass};\]

The real disc which is controlled by the class type disc (as
explained in Brinch Hansen (1975a)) will use the procedure
IODisc to access disc pages. The type disc can use the IODisc
procedure since it can declare an instance of type Discaccess
and it is passed an instance of type Disbound every time it is
called upon to either read or write on to the disc.

Please note that in the type Virtualdisc, the formal parameter
X of type Disbound has been restricted to the read only
operation. This is required because of the restriction placed
upon allocated Disbound instances by the Discmanager. That
is, only the Discmanager can change the values of Base and
Limit of instances of type Disbound.

Discussion

In the previous section we illustrated that disc accesses can be
controlled by appropriately defining the program components
Discaccess and Discmanager. The integrity of this scheme
clearly rests upon the correctness of the Discmanager since it is
the only program component that can initialise the Base and
Limit variables of instances of type Disbound.

The Discmanager acts as the owner of the disc; it determines
the policy as to how the association between disc pages and
program components should be done. Once such an association
has been determined, the manager is not directly involved in
enforcing this policy; this is carried out by the Discaccess.

The mechanism we have proposed can be implemented using
Concurrent PASCAL itself. This has the advantage that policies
can be changed without affecting the rest of the system. This
mechanism can also be used to protect other peripheral
devices and program components which possess the property
of having an array of elements partitioned among different
processes.

In the scheme we have proposed, peripheral devices were
treated as ordinary shared program components; they are of
type monitor and can be accessed only by program compo-
nents that hold access rights to them. Moreover, an access
ing to a peripheral device does not imply that it can be
exercised freely; only a subset of the peripheral pages can be
accessed by the program component holding that access right.

In the example of the Discmanager, we have chosen to confine
the algorithm to static allocation of Disbound instances. This
need not be the case in general, and Disbound instances can be
allocated and deallocated as needed. Such a scheme can con-
veniently be used to implement the concept of sequential or
random files.

Acknowledgements

I wish to thank my colleagues Brian Johnson and Richard
Kieburz for reviewing an early draft of the manuscript.
References

Book reviews


I found this short (only 120 effective pages) book easy to read but surprisingly difficult to review. It attempts to cover a large theme and I attribute some of the repetition to the trinity of authors whose fondness for twenty words where two would suffice is sustained throughout.

The book appears under the auspices of Unesco and its purpose is defined as being to present practical guidelines... for the planning design and management of... processing, storage, retrieval and dissemination of information and knowledge. Interestingly the book is also meant to serve as an advisory tool for education and training in developing countries (LDCs). Many readers will be familiar with the practical problems of co-ordinating suppliers, PTTs and expectant users when attempting to implement even modest systems in so-called advanced countries. I would hope LDCs could avoid the fastest fringes of information networks for a few decades yet, at least until they have had a chance to organise a comprehensive education and training programme. Certainly all readers will proceed with caution having met, on p. 8 in regard to teamwork and multi-disciplinary orientation (sic), "the methodology should be understood by each of the (team) members, and all should operate on the basis of reasonable but demanding mutual expectations". The early chapters are a useful list of do's and don'ts for any ambitious systems project and there are helpful hints that data and methodology (that word again) are no substitute for judgement. What is missing, as so often in this type of book, is any nod towards the sheer bloodmindedness, obstinacy and unhelpfulness of the majority of homo sapiens with whom one is likely to be dealing on any major project involving change and uncertainty.

Regarding preliminary tasks I have found the review of past literature a fixation of the civil service mentality and of strictly limited value. Past skeletons emit the odour of decay and it is best to attempt to be a new and energetic brompt. It is true that the involvement of top management is important but unfortunately it cannot be guaranteed—there may be valid political reasons for favouring an arm's length approach.

The section on analytical techniques is rather sketchy; it is not possible for analysts to ask "are things being done in the right amount?". The design of alternative solutions follows logically from the analysis and it is correctly stated that today's new system soon becomes subject to examination and updating.

Moving on to "Operations and Systems Implications" the book edges a little closer to the people problem. There is an important statement that the technical man (e.g. project manager) may not realise (as would apparently an experienced manager) that the members of an organisation will feel less threatened by change if they have some control over their own destiny. Albert Booth has made the same point recently—industrial democracy is the price we will have to pay for the rapid adjustment to "chip technology".

Management information systems for administrative decision making are well covered through time "communication techniques that effectively annul space and time between people" is an absurd overstatement of current offerings. There is an interesting attempt to dissect types of information and relate these to operational levels in management hierarchies. It should be more clearly said that several previous attempts at "full utilisation of mind augmenting resources" have been costly disasters and that many practitioners now opt for the piecemeal approach which has implications for the scope of trials and pay-back.

I turned to "Networking and Perspectives" in the hope that the whole exercise would be put in context; however I would be wary of a "Master of Systems, Cybernetics and Informatics"—he it would be who could write, without tongue in cheek, "these hybrids (networks) are "constitutional", i.e. their characteristics are not "summative" but amount to much higher magnitudes than the sum of the isolated parts". Never fear, gentle reader, that "through combinatory thinking we can advance to hitherto unforeseen levels of abstraction and new extensions of reasoning or augmenting of intellect", come back de Bono all is forgiven! And what are we to make of "symbiotic cross fertilisation"? Well may the perplexed systems analyst regret his Aunty paying for the correspondence course and search the small ads for a quiet life in insurance with real people with real problems. "Formalised concepts" are to be our watchword—what a hope in today's chaotic business world trying to adjust to the sickly seventies. The key element will certainly be man himself and good luck to him as he fights to keep his nose above the "constellation of proliferating star-nets". I will not be alone in my sneaking sympathy for "the guy who took the ball and ran"—at least he was shocked and frightened and aware, unlike the remote authors of this curious book. They really do seem to dream of a "World brain". Information may be power, but there is an argument that more means worse.

JOHN A. S. WEIR (Slough)


This book provides a clear description of the theory and application of precedence networks. Explanations are clear and the worked examples are realistic although, as is often the case with this subject, small examples tend to obscure problems encountered with much larger networks. The general presentation of the book is satisfactory, with half of each page left blank for the reader to make notes, although I would hesitate to write all over a book as expensive as this.

In addition to constructing precedence networks and performing time analysis, information is also given on resource allocation and costs, including development of cost curves for cash flow projections and the monitoring of actual project costs. The uses of computers are discussed with examples of computer output. A separate appendix describes several computer packages although their cost and availability is presumably based on the American market.

In conclusion, this book would be a useful addition to any construction engineer's bookshelf, although the material and techniques described could equally well be applied to the fields of research and development or maintenance.

R. A. HUNTY (London)