Abstract:

A macro notation incorporating path notation is introduced. The notation was designed to permit the specification of the behaviour of a distributed system in terms of resources characterized by associated synchronization statements. The proper co-ordination of processes using the resources can thus be stated as the permissible order of execution of operations on shared resources as part of the resource definition. This contrasts with the usual "process oriented" approach which associates synchronization statements with the processes using the resource. This makes proper co-ordination of processes part of the individual task of each process. We argue that the process oriented approach is not suited for specifying highly parallel systems with distributed decision capabilities. The notions of adequacy, starvation, priority and interrupts are discussed in the context of our "resource oriented" approach and compared with their counterparts in a "process oriented" approach. Semi formal demonstrations of the adequacy of simple but realistic example systems are sketched.

Abstract specification of resource accessing disciplines: adequacy, starvation, priority and interrupts

By

P.E. Lauer and M.W. Shields

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 117
December, 1977

© 1977 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
LAUER, Peter Ernst.


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1977.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 117.)

12"

Added entries SHIELDS, Michael William Enduan
UNIVERSITY OF NEWCASTLE UPON TYNE, Computing Laboratory.

Suggested classmarks (primary classmark underlined)

Library of congress:
Dewey (17th): 001.64404 001.6424
U.D.C. 681.322.06

Suggested keywords ADEQUACY VERIFICATION
CONCURRENT SYSTEMS
DEADLOCK PREVENTION AND DETECTION
PARALLEL PROGRAMMING
PATH NOTATION
PETRI-NETS
PRIORITY
INTERRUPTS

Abstract

A macro notation incorporating path notation is introduced. The notation was designed to permit the specification of the behaviour of a distributed systems in terms of resources characterized by associated synchronization statements. The proper co-ordination of processes using the resources can thus be stated as the permissible order of execution of operations on shared resources as part of the resource definition. This contrasts with the usual "process oriented" approach which associates synchronization statements with the processes using the resource. This makes proper co-ordination of processes part of the individual task of each process. We argue that the process oriented approach is not suited for specifying a highly parallel systems with distributed decision capabilities. The notions of adequacy, starvation, priority and interrupts are discussed in the context of our "resource oriented" approach and compared with their counterparts in a "process oriented" approach. Semi formal demonstrations of the adequacy of simple but realistic example systems are sketched.

About the author

Dr. P.E. Lauer has been a lecturer in the Computing Laboratory of the University of Newcastle upon Tyne since 1972, before then he was with IBM Laboratory, Vienna.

Dr. M.W. Shields has been a research associate in the Computing Laboratory since December 1975. Prior to that he was a research student at Leeds University.
Introduction

A macro notation [Lauer, Torrigiani 1976,77] incorporating path notation [Lauer, Campbell 1974, Campbell 1976, Lauer, Best, Shields 1977] is introduced. The macro notation is a systems specification language which describes systems in abstraction from all implementation detail even in terms of a high-level programming language. To be more specific, the programming notation was designed to permit the specification of the behaviour of a distributed system in terms of resources (systems objects) characterised by associated synchronization statements. The proper co-ordination of processes can thus be stated as the permissible order of execution of operations on shared system objects as part of the object definition. Thus our approach is in contrast with a "process-oriented" approach which would involve the specification of a distributed system by associating synchronization statements with the processes using the resource, as is done in systems using semaphores, for example. Our "resource-oriented" approach allows us to treat every resource as a subsystem capable of making its own decisions (concurrently with other resources) concerning the utilisation of its constituent operations by processes; moreover, these decisions are based only on knowledge about the local history of usage of its operations. Such characteristics are clearly desirable for systems which are intended to achieve high levels of parallel activity.

The notation will be introduced by example rather than using the more formal approach of the cited papers. More specifically, we construct several programs, including path solutions to the reader-writer problems of [Courtois, Heymans 1971], gradually introducing each language feature as it is required.

We attempt to evaluate our resource-oriented approach by comparing our solutions to standard solutions belonging to a process oriented approach (semaphores). We show how these solutions may be translated into path programs which allows us to compare the two approaches on the same level of abstraction. Explicitly expressing properties of synchronization primitives used by the process oriented solutions by means of
paths indicates another difference between the two approaches. The process oriented approach conceptually involves an extension of the resources of the system required by the processes to perform their individual tasks by a set of further resources which serve the sole purpose of properly co-ordinating the processes in using shared resources. Furthermore, the individual tasks of the processes sharing resources become more complicated because the task of co-ordinating the system behaviour becomes part of the individual tasks of all the processes involved.

Since synchronization resources are global to a number of processes and in addition (semaphores) do not permit concurrent activation of their operations, the process oriented solutions usually display less concurrency, and decision capabilities are less distributed.

Next we consider certain possible advantages of the resource oriented approach for expressing crucial properties of systems and for rigorously demonstrating whether a given system has such properties.

A resource oriented specification of a simple spooling system is used to illustrate the notion of an adequate system. The formal approach for specifying the meaning of the notation used in the papers cited above has led to techniques which facilitate the rigorous demonstration of the adequacy of the system. We sketch some of these techniques in the informal presentation by demonstrating the adequacy of the example.

We next turn our attention to further aspects of system behaviour such as conflict, starvation, priority and interrupts. A consideration of the notion of conflict, which corresponds to concurrent competition for a shared resource, leads us to a discussion of the starvation of processes, that is, indefinitely delaying progress of processes due either to unfair conflict resolution or a "conspiracy" on the part of other processes in the system by permanently blocking these processes from accessing some resource that the starved processes need to use in order to progress. This leads naturally to a consideration of the second reader-writer problem. We give a path solution to this problem involving the introduction of a precedence construction,
the function of which is to specify fixed priority in cases of conflict. Again, we evaluate our resource-oriented approach by comparing our solution with a standard solution using a process oriented approach. Finally, we illustrate the use of the precedence construction in defining interrupt mechanisms; in particular, we extend our spooling system program to include an interrupt handling process. The adequacy of this extended program is then demonstrated.

1. The path and process notation

1.1 Two approaches to system structuring

For the present we will think of systems as consisting of two basic types of sub-systems called resources and processes. A specific resource is identified with a collection of operations, for example an arithmetic unit involves operations such as addition, multiplication, shifting etc. Processes may be thought of as groupings of operations from (possibly distinct) resources to accomplish particular tasks such as calculating factorial by iterative use of addition and multiplication.

A system is sequential if at most one operation of the system may be active at any one time. We will be interested in non-sequential systems in the sense that more than one of its component operations may be active concurrently. Each individual process will be a sequential subsystem but operations belonging to distinct processes may be concurrently active, i.e. distinct processes may be progressing in the performance of their tasks in parallel.

It may be necessary to prevent concurrent activations, or to order activations of operations in a particular way, to ensure proper functioning of the system; e.g. prevention of simultaneous updates of a memory word, or enforcing that a buffer frame be written to and read from in a strictly alternating sequence of activations of write and read operations to ensure correct transmission of information, respectively.
The responsibility for ensuring proper functioning of the system may be thought of as lying with the resources or alternatively with the processes using the resources. In the latter or process oriented approach, usually some synchronisation operations (primitives like P and V operations) will occur in the processes and thus the attainment of correct behaviour of the system becomes part of the task of the processes. This complicates the task of each process and makes it harder to understand what the system does. In the former or resource oriented approach, resources may be thought of as including mechanisms ensuring that their operations may only be used correctly by processes. If these mechanisms are entirely transparent to the processes all responsibility for correct behaviour of the system is removed from the processes.

Our own approach is resource oriented but both approaches will be contrasted by comparing "equivalent" solutions to classic problems involving the proper co-ordination of concurrent processes. Solutions representing the process oriented approach will be expressed using standard semaphores [Dijkstra 1968a and b] and solutions representing the resource oriented approach will be expressed in the path notation [Campbell, Habermann 1974, Lauer, Campbell 1974, Campbell 1976, Habermann 1975, Lauer, Torrigiani 1976, Torrigiani, Lauer 1977] whose meaning will be explained in the subsequent sections.

1.2 The first reader-writer problem

The first reader-writer problem [Courtois, Heymans, Parnas 1971] may be paraphrased as follows: Consider a system consisting of a single resource involving read and write operations and a set of "reader" processes and "writer" processes which repeatedly use the operations to read from and write to the resource, respectively. It is required that any number of readers may be concurrently using the resource, but each writer must have exclusive use of it. Also, no writer may jointly use the resource with a reader. Furthermore, no reader should be kept waiting unless a writer is using the resource.
In the resource oriented approach using path notation, we can identify the resource with a collection of operations named "write" and "read_i", 1≤i, where for 1≤j and i ≠ j, read_i and read_j may be concurrently active. Since we will only be concerned with systems consisting of a fixed and finite number of sequential processes once the system becomes operative the number (say n) of processes using the read operation will be the fixed and finite number of read operations the resource will need to permit maximal concurrency of activations of the read operation. If we allow for maximal concurrency of activations of read in the resource, it means that i≤n and j≤n, above. Assuming that we have n reader processes and m writer processes we can express the system of cyclic processes as:

\[
\begin{align*}
\text{process read}_1 \text{ end} \\
\vdots \\
\text{process read}_n \text{ end} \\
\text{process write end} \\
\vdots \\
\text{process write end}
\end{align*}
\]

(1)

Due to our correlation of the number of read operations of the resource and the number of reader processes, we have satisfied the requirement that any number of reader processes of the operative system may concurrently be using the resource and hence no reader will be kept waiting just because some other reader is using the resource. We still need to ensure that each writer has exclusive use of the resource and that no writer uses the resource jointly with a reader. To express this constraint we can use the following collection of paths:

\[
\begin{align*}
\text{path write, read}_1 \text{ end} \\
\vdots \\
\text{path wrote, read}_n \text{ end}
\end{align*}
\]

(2)

read-write resource
Calling the collection of paths the read-write resource indicates that we are now identifying the resource with the set of all paths involving operations of the resource. Our explanation of (1) and (2) taken together will illustrate some of the features of the path notation.

1. If an operation appears in at least one path, then there may be no concurrent activations of the operation. Thus, in our example program, the fact that the operation write appears in the set of paths, entails that the writer processes each have exclusive access to the resource. No two of them may be activating write concurrently.

2. If two operations are immediately associated by a comma, then activations of the operations are to be mutually exclusive. Thus in our example the paths taken together imply that the operation write cannot take place concurrently with any read. The read operations are however not related to each other by any path and if no write is taking place, then any number of reader processes may be concurrently active. If a write is taking place, then for each i, the i-th path specifies that reader number i cannot use the resource until the writer is finished.

3. The constraints expressed by paths may be used repeatedly to decide whether a process may use the resource. Thus, for example, the first path permits any arbitrary sequence of activations of write and read.

What we have said should indicate that (1) and (2) taken together constitute a possible solution to the first reader-writer problem.

It is instructive to compare it with the process oriented solution in terms of binary semaphores given in Courtois, Heymans, Parnas 1971:
integer readcount; (initial value = 0)
semaphore mutex, w; (initial value for both = 1)

READER
P(mutex) ;
readcount := readcount + 1;
if readcount = 1 then P(w);
V(mutex) ;

...

writing is performed

...

P(mutex) ;
readcount := readcount - 1;
if readcount = 0 then V(w);
V(mutex) ;

Several points may be observed about this solution, which contrast it with our own.

Firstly, in the semaphore solution, the achievement of correctness of system behaviour is part of the tasks of the processes. Entry to the critical section is controlled by the semaphore w, which is set to zero by the first of a stream of readers which gains permission to use the resource and is only reset to one when the last reader in the stream finishes using the resource.

Secondly, the semaphore solution lacks the symmetry of the path solution. The first reader in a stream to request use of the resource is distinguished from subsequent readers belonging to the same stream in that the first is the only reader of the stream to access the semaphore w. This effect is achieved by a conditional statement involving the integer readcount, and since it is undesirable that this integer be acted upon by several reader processes concurrently, access to it has to be made mutually exclusive for reader processes. Thirdly, the semaphore solution is less concurrent than the path solution; in the semaphore solution, reader processes must queue on mutex in order
to request or release use of the resource. In the path solution, the reader processes may obtain use of the resource concurrently.

A partial translation of this semaphore solution into paths will serve on the one hand to introduce further details of the notation and on the other to further illustrate that in such solutions, the processes must themselves ensure that concurrent activations of requests to use the resource by distinct processes are excluded. The operations to be considered are the following:

Pmutex, Pw, Vmutex, Vw, read, write (the meaning of which should be self evident) contincr1 (incrementing 'readcount' to 1), contincr (other incrementations of 'readcount'), countdecr0 (decrementing 'readcount' to 0 ) and countdecr.

The first new feature of the notation used in the translation is the semicolon.

4. If two operations are immediately associated in a process by a semicolon then, to perform its task, the process requires them to become active sequentially in the order given. Using this we can express the writer processes as:

process Pw; write ; Vw end

If an operation in a path or process is immediately associated with two other operations, the one by a semicolon and the other by a comma, then the comma binds more strongly. Thus a sub-expression

.... contincr, contincr1; Pw ...

states that before the activation of Pw, either the counter is incremented to 1 or it is incremented to some other value.

5. The binding precedence may be overridden by the use of parentheses. Thus the sub-expression

.... contincr, (contincr1; Pw) ...

states that either the counter is incremented to some value other than 1 or first the counter is incremented to 1 and then a P operation is performed on the semaphore w. The reader processes have the form:
process Pmutex; countincr,(countincr1; Pw); Vmutex; read;
(4)   Pmutex; countdecr,(countdecr0; Vw); Vmutex end

Two resources behaving like binary semaphores mutex and w, initialized to 1, may be identified with the following two paths, respectively.

path Pmutex; Vmutex end
(5)   path Pw; Vw end
since:

6. If two operations are immediately associated by semicolon in at least one path, they may only be activated exclusively and in the order given. Note that implicit mutual exclusion entailed by the appearance of an operation in a path ensures that no two processes may be concurrently performing a P or V operation on the same semaphore. Notice also that the conditional statement has been absorbed into the structure of the processes.

Finally, we need to introduce a counter resource to obtain a more detailed translation of the semaphore solution (3). To do this we need to introduce a further feature of the notation.

7. If a group of operations is followed by a star "*", then the group as a whole may be activated zero or more times by repeatedly activating its constituent operations as specified by the structure of the group. The star binds more strongly than either comma or semicolon, and so parentheses are again required to override this binding.

A counter with an upper bound of three on the set of values it may take may be defined by the following path:

path countincr1;(countincr;(countincr;countdecr)*;countdecr)*;
(6)   countdecr0 end

The translation has the form:
process Pmutex; countincr, (countincr1; Pw); Vmutex; read;
Pmutex; countdecr, (countdecr0; Vw); Vmutex end
(one such expression for each reader process)
process Pw; write; Vw end
(One such expression for each writer process)

path Pmutex; Vmutex end
path Pw; Vw end
path countincr1; (countincr; (countincr; (...,...)*) countdecr*) countdecr0 end

Observe that the conditional statements involving readcount have been replaced by expressions involving choice; thus a reader process may either increment the counter to one, after which, in order to progress, it must perform a P operation on the semaphore w, or it is incremented to some other number, thereby bypassing the operation Pw. The structure of the counter ensures that after countincr1 has been activated it cannot be activated again until countdecr0 has been activated. It also ensures that if the activation of countdecr0 has been preceded by a total of n activations of countincr, then it must also be preceded by a total of n activations of countdecr. Thus the path does indeed describe a counter resource. Note that the translation of (3) requires the introduction of three additional resources whose sole function it is to enable the processes to be programmed in such a way as to ensure proper use of the reader-writer resource. Note also how the details of such a specific "implementation" of the constraints complicates the logic of the solution.

A more detailed translation of the semaphore solution would give rise to a yet more complex path program; one in which testing of the counter was made explicit. All the same, a testable counter resource may be described using the notation. It would be identified with the following operations:

a) countincr and countdecr, being the operations of incrementing and decrementing the counter by one.
b) test1 and test0, being the operations of testing the counter and finding it to be one or zero respectively.

c) test, being the operation of testing the counter and finding it to be some number other than one or zero.

The path describing the constraints that the resource must obey in order to function truly as a testable counter is as follows:

\[
\text{path test}0*, (\text{contincr; test}1*, (\text{contincr; test}*, (\ldots, (\text{contincr; test}*, \text{countdecr})*; \ldots)*)*; \text{countdecr})*; \text{countdecr})* \text{ end}
\]

Thus, initially, the counter may be tested zero or more times and found to be zero, after which it may be incremented by one and then tested zero or more times and found to be one. It may then be decremented, after which the looping property of paths ensures that it may then be tested zero or more times and found to be zero. Alternatively, it may be incremented again and then tested and found to be some number other than zero or one. Again, the structure of the path entails that to every increment there is a corresponding decrement.

The reader processes would now have the form:

\[
\text{process Pmutex; countincr; test}0, \text{test}, (\text{test}1; \text{Pw}); \text{Vmutex; read; (9) Pmutex; countdecr; test}1, \text{test}, (\text{test}0; \text{Vw}); \text{Vmutex end}
\]

1.3 Some useful macro notation

In the examples considered so far, we sometimes found it necessary to make use either of ellipsis or a phrase such as 'N such expressions' in order to indicate a collection of paths or processes having the same pattern. There is, however, yet another feature of the notation that permits one to avoid the use of such expedients. This language feature, called the macro notation, introduces an operator called the 

\text{collectivisor} which will permit the grouping of individual operations into multidimensional collections or arrays, which are thereby given a 

\text{collective name}. Names of individual operations belonging to a collection are obtained from the corresponding collective name by
subscripting. But it is also possible to 'distribute' one of the connectives '; ' or ',' over an entire dimension of the collection by means of a so called distributor. Furthermore, the macro notation involves an iterative copy operator called the replicator, which permits the finite representation of program text of finite but indefinite length which otherwise would require ellipsis or recursion to represent it.

To illustrate the use of the macro notation, let us return to the first path solution of the first reader writer problem. Recall that it is as follows:

\[
\text{process write \  end } \ldots \ \text{process write \  end} \\
\text{process read}_1 \ \text{end process read}_2 \ \text{end } \ldots \ \text{process read}_n \ \text{end} \\
\text{path write, read}_1 \ \text{end } \ldots \ \text{path write, read}_n \ \text{end}
\]

Both the list of reader processes and paths representing the decision mechanisms whereby the resource ensures correct usage of its operations have the form of a pattern reproduced several times with changes only to a subscript. The pattern associated with the reader processes is

\[
\text{process \ READ(i) \ end}
\]

and the pattern for the paths is

\[
\text{path \ write, \ READ(i) \ end}
\]

where the \( i \) in the parentheses may be replaced by an index from 1 to \( n \).

The macro notation contains a facility for both "declaring" an array of indexed operations and for replicating a given pattern with integers introduced in place of some indeterminate index.

To declare a set of indexed operations, one writes a \textbf{collectivisor array collectivenname (upperbound)};

where \( \text{upperbound} \) is a positive integer. For instance, the collectivisor

\[
\text{array READ(n)};
\]

declares a set of operations $\text{READ(1)}$, $\text{READ(2)}$, $\ldots$, $\text{READ(n)}$
Given a collectivisor, it is possible to define sets of paths or processes having similar patterns. In the macro notation, the expression:

\[ \text{[path write, READ(i) end } [i | 1,n,1] \]

would expand to

\[ \text{path write, READ(1) end... path write, READ(n)end} \]

and this is an example of a replicator of the following type.

The replicator \([\text{pattern index}] | n,m,k\), where "pattern" is a string, "index" is an integer variable and \(n,m,k\) are integer expressions will be expanded to:

a) The empty string if \((n>m \text{ and } k>0) \text{ or } (n<m \text{ and } k<0) \text{ or } k=0\)

b) Substitute \((\text{pattern, index, n})[\text{pattern index}] | n+k,m,k\) where "Substitute(pattern, index, n)" indicates the result of substituting "n" for all occurrences of "index" throughout pattern.

Obviously, we constrain "pattern" to be such that the result of the expansion is a valid expression in the notation.

There is a second type of replicator, and one that we may use to properly define a counter such as appeared in the path translation of the semaphore solution to the first reader-writer problem (6). Here the general form is:

\[ [\text{pattern1 index} \text{ pattern2 } | n,m,k] \]

Again, if \((n>m \text{ and } k>0) \text{ or } (n<m \text{ and } k<0) \text{ or } k=0\), then this expands to give the empty string. Otherwise it may be expanded to

Substitute(pattern1, index, n)[pattern1 index] pattern2 | n+k,m,k

Substitute(pattern2, index, n).

Again, pattern1 is constrained to give a valid expression on expansion, pattern2 is allowed to be the empty string, so that we may define, for example:

\[ [\text{process READ(i) end } [i | 1,n,1] \]

which expands to

\[ \text{process READ(1) end process READ(2) end... process READ(n) end} \]
A less trivial and more interesting example is that of the finite testable counter that we used in the path translation of the semaphore solution to the first reader writer problem. Path (8) may be expressed in the macro notation using a replicator as follows:

\[
\text{path test}0*, (\text{countincr}; \text{test}1*[], (\text{countincr}; \text{test}2*1); \\
\text{countdecr}*|n, 1, -1]; \text{countdecr}* \text{end}
\]

This particular counter will count up to \( n+1 \).

The example illustrates two aspects of the notation.

a) It is only possible to define \textit{finite} programs consisting of finite, although unbounded, paths or processes. We cannot, for example, define an \textit{infinite} counter although we may define a counter for any given finite capacity.

We do not consider this a major drawback, however, since systems in general once operative are finite and since there is no a priori limit to the size that a program in the notation may have or the number of operations that a path or process may contain. We suggest a possible interpretation of infinite paths in a paper [Shields, Lauer 1977] demonstrating the equivalence of an extended path and process notation and the extended semaphore primitives of Agerwala [Agerwala 1977].

b) Although the macro notation does not extend the expressive power of the notation, it gives the programmer the ability to conceptually decompose a complex system into simpler subsystems, to make repetition explicit and to define resources such as buffers, stacks and counters, clearly and concisely, thereby allowing him a higher-level view of the system.

Consider, for example, an n-frame ring buffer \( B \) of \( B_i \) \( (i = 1, \ldots, n) \) there are two associated operations, which we shall call \text{DEPOSIT}(i) and \text{REMOVE}(i). \text{DEPOSIT}(i) \) is the operation of depositing an item in \( B_i \) and \text{REMOVE}(i) \) is the operation of removing an item from it. To ensure proper transmission of information through the buffer the activations of these operations must obey the following constraints:
a) Every removal of an item from $B_i$ must be preceded by a deposit (no removing from an empty frame).

b) Every pair of deposits into $B_i$ must be separated by a removal; it is not permitted to overwrite the contents of a frame.

c) Deposits into the frames of $B$ must take place in numerical order and cyclically; deposits obey a ring discipline.

d) Removes from the frames of $B$ must take place in numerical order and cyclically.

These constraints may be translated directly into paths. First we need arrays of deposits and removes:

\begin{verbatim}
array DEPOSIT, REMOVE(n);
\end{verbatim}

Now (a) and (b) say that deposits and removes from each frame must take place alternatingly beginning with a deposit, which can be expressed as

\begin{equation}
\text{path DEPOSIT(i); REMOVE(i) end } [1 | 1, n, 1]
\end{equation}

which constrains deposits to take place in sequence and in numerical order. This can be expressed by:

\begin{equation}
\text{path[DEPOSIT(i)@ [1 | 1, n, 1] end}
\end{equation}

where "[pattern @ separator [1 | n, m, k]]" is understood as "Substitute(pattern, i, n) separator[pattern @ separator [1 | n+k, m, k]]" and "separator" is either ",," or ";".

The pattern "collectivename(index)@separator" occurs so frequently that it has been distinguished as a special feature of the macro notation, the distributor. If we have a collectivisor

\begin{verbatim}
array A(n);
\end{verbatim}

then the distributor

\begin{verbatim}
;A
\end{verbatim}

is defined as

\begin{equation}
[A(i)@ [1 | 1, n, 1]
\end{equation}

and similarly for ;(A).
The above path thus becomes:

\[
\text{path} ; (\text{DEPOSIT}) \quad \text{end}
\]

Finally, (d) translates as

\[
\text{path} ; (\text{REMOVE}) \quad \text{end}
\]

The \text{n-frame} ring buffer may therefore be identified with the following segment of program:

\[
\text{array} \quad \text{DEPOSIT}, \text{REMOVE}(n);
\]

\[
[\text{path DEPOSIT}(i) ; \text{REMOVE}(i) \quad \text{end} | i | 1, n, 1]
\]

(14) \text{path} ; (\text{DEPOSIT}) \quad \text{end}

\text{path} ; (\text{REMOVE}) \quad \text{end}

1.4 \text{ A simple spooling system}

Consideration of a simple spooling system serves to further illustrate our resource oriented approach to system specification. Our formulation of the system will also be used to indicate some of our approaches to the problem of demonstrating the adequacy of a system specified in the notation.

The system consists of the following parts:

1) a disk divided into sectors, each capable of holding five lines.
2) a ring buffer B capable of holding 10 lines.
3) a line printer.
4) a line assembler process P1. P1 takes a sector at a time from the disk and deposits the contents of the sector into the line buffer.
5) a line printer process P2. P2 removes a line at a time from B and prints it on the printer.

We already have our ring-buffer. We assume that P1 uses the operation readsector which reads some (externally specified) disk sector and places it in a temporary storage area private to P1. It then writes the five lines contained in the sector into some five frames of B.
The line assembler may be written as

(15) \texttt{process readsector; }[,(DEPOSIT)@; ||1,5,1]\texttt{ end}

The line printer process P2 removes a line from B and prints it and then recommences. Denoting by "printline" the operation of P2 causing the line printer to print out a line, we may describe this process by:

\texttt{process , (REMOVE); printline end}

Observe that the two processes are quite disjoint, in the sense that they have no operation in common. The manner in which they co-operate is determined completely by the structure of the resource they share, namely the line buffer.

The complete program is:

\texttt{array DEPOSIT, REMOVE(10);}
\texttt{process readsector; }[,(DEPOSIT)@; ||1,5,1]\texttt{ end}
\texttt{process , (REMOVE); printline end}

\texttt{(16) [path DEPOSIT(i); REMOVE(i). end ||1,10,1]}
\texttt{path ; (DEPOSIT) end}
\texttt{path ; (REMOVE) end}

It is often important for systems programmers to know whether programs which describe non-sequential systems satisfy certain criteria. These criteria may include the protection of shared resources, absence of deadlock, absence of starvation and observation of capacity bounds. Associated with the path notation is a formal notion of correctness, called adequacy, which corresponds (roughly) to the notion of liveness in transition net theory \cite{Lauer1974, Lauer1977}. A path program is defined to be adequate if and only if the system it describes may never reach a state in which some operation may never in future become active without infringing the constraints entailed by the program. Thus adequacy implies freedom from deadlock in the program.
process read, abort; write end

(17) path read; write end

after activation of the operation abort, no further system activity is possible since the process may only activate write in order to progress, whereas the path requires that the next operation to be activated must be read. The activation of abort brings about a situation of deadlock.

When its subsystems are progressing a system gives rise to partially ordered sets of activations of operations. Each such set, which we may call a trace, would represent a period of system activity; if two activations belong to such a set and are comparable with respect to the partial order, then it may be deduced that one of them has preceded the other. In particular, if for a given system all such sets are totally ordered then the system is sequential. On the other hand if, in a given period of system activity, two activations of operations are concurrent, then these operations will not be comparable in the partially ordered set corresponding to this period. Intuitively, a set of activations together with a partial order is a trace for a given system if and only if for each sequential subsystem of the system the set obtained by removing from the given poset all activations of operations which do not belong to the subsystem is totally ordered, and the sequence of activations obtained by writing the contents of this new set in the resulting total order satisfies the constraints of the subsystem. Here, we are implicitly distinguishing between operations with the same name appearing in either different processes or in some expression, separated by the comma.

Consider, for instance, the spooling system example (16). The following poset, represented as a Hasse diagram, is a trace of the program.
"readsector" may become active followed by "DEPOSIT(1)". Then "DEPOSIT(2)" and "REMOVE(1)" may take place concurrently. "printline" and "DEPOSIT(2)" may also be taking place concurrently but "printline" must be preceded by "REMOVE(1)". This poset is indeed a trace, since, for instance, its "intersection" with the line assembler process is the totally ordered set

\[
\{ \text{readsector} >  \text{DEPOSIT}(1) >  \text{DEPOSIT}(2) >  \text{DEPOSIT}(3) \}
\]

which is a valid sequence of activations for that process.

Any such trace may give rise to a number of totally ordered sets, as its partial order may be extended to a total order. Such totally ordered sets give rise to sequences of activations in the obvious way. Such sequences are called "firing sequences" of the program in question. For instance, the poset of figure (18) has the following related firing sequences:
readsector DEPOSIT(1) DEPOSIT(2) DEPOSIT(3) REMOVE(1) printline REMOVE(2)
readsector DEPOSIT(1) REMOVE(1) DEPOSIT(2) DEPOSIT(3) printline REMOVE(2)
readsector DEPOSIT(1) REMOVE(1) printline DEPOSIT(2) REMOVE(2) DEPOSIT(3)

and so on.

We may now informally define adequacy as follows: a program is adequate if for any trace and any operation, the trace may be extended to another trace in which an activation of the given operation occurs in the extension. An alternative and equivalent definition is that a program is adequate if and only if for any firing sequence and any operation, the firing sequence may be extended to a firing sequence by the concatenation of it with a string containing the operation.

We shall now sketch a proof that the spooling example is adequate. The proof will involve two steps, each one illustrating a different proof technique.

1.4.1 A transformation of the program into another with equivalent adequacy properties

Let $P'$ be the program obtained from the spooling program $P$ by deleting both processes, then $P$ is adequate if $P'$ is adequate. (In fact the converse is true.)

Proof:

Let $S$ be a firing sequence of $P$. We wish to show that there exists a string $S'$ such that $SS'$ is a firing sequence of $P$ and $S'$ contains every operation belonging to $P$. Let $f(S)$ be the string obtained from $S$ by deleting every instance of "readsector" and "printline". $f(S)$ is a firing sequence of $P'$ and so, by adequacy of $P'$, there exists a string $S''$ such that $f(S)S''$ is a firing sequence of $P'$ and $S''$ contains every operation belonging to $P'$. We may now form from $S''$ a new string $S'$ by appropriately inserting instances of "readsector" and "printline". $S'$ will have the desired properties.
An alternative proof which may be informative is: Let $T$ be a trace of $P$ considered as a Hasse diagram. Tick each node labelled "readsector" or "printline". If the ticked nodes are removed, the resulting diagram is a trace of $P'$ and may therefore be extended to another trace of $P'$ where the extension contains every operation of $P'$. If the ticked nodes are now reintroduced and in the new diagram we replace

\begin{align*}
\text{REMOVE}(i) & \quad \text{REMOVE}(i) & \quad \text{DEPOSIT}(5j) & \quad \text{DEPOSIT}(5j) \\
\text{by \ printline \ and} & \quad \text{by \ readsector} & \\
\text{REMOVE}(i+1) & \quad \text{REMOVE}(i+1) & \quad \text{DEPOSIT}(5j+1 \mod 10) & \quad \text{DEPOSIT}(5j+1 \mod 10)
\end{align*}

where $i$ is an integer and $i = i \mod 10 + 1$, and $j$ is 1 or 2; then the result is a trace of $P$ extending $T$ and having every operation of $P$ in the extension.

1.4.2 Application of a known adequacy theorem

We shall now prove that $P'$ is adequate. Recall that $P'$ is merely the 10 frame ring buffer:

\begin{align*}
\text{[path \ DEPOSIT(i); \ REMOVE(i) \ end \ ] i \ |1,10,1]} \\
(19) \text{path ; (DEPOSIT) end} \\
\text{path ; (REMOVE) end}
\end{align*}

Note that this program consists entirely of paths, that no operation appears more than once in any path and that the only separator in the program is the semicolon. Such paths are called $GE_0$-paths in [Lauer, Campbell 1974] and the following theorem, which originally appeared in that paper in a slightly different form, characterises all adequate paths of this class.
Theorem:

Let $P$ be a path $a_1; \ldots; a_{n_1}$ end ... path $a_1; \ldots; a_{n_m}$ end, a $GE_0$-path, then $P$ is adequate if and only if there exists a string $S$ of operations such that the string $S_i$ obtained from $S$ by deleting all operations not appearing in the $i^{th}$ path of $P$ is precisely $a_1^i; \ldots; a_{n_i}^i$.

Returning to $P'$, we see that the string

$$DEPOSIT(1) \ DEPOSIT(2) \ \ldots \ DEPOSIT(10) \ REMOVE(1) \ REMOVE(2) \ \ldots \ REMOVE(10)$$

satisfies the conditions of the theorem, since, making use of the notation of the theorem, $S_i = DEPOSIT(i) \ REMOVE(i)$ for $i \leq 10$ and

$$S_{11} = DEPOSIT(1) \ldots \ DEPOSIT(10)$$

$$S_{12} = REMOVE(1) \ldots \ REMOVE(10)$$

It is interesting to note that the scheme of the proof has involved reducing the problem of proving the "correctness of the whole system" to that of proving the "correctness of the ring buffer resource".

We shall now turn to the questions of starvation and priority, illustrating our discussion by means of the second reader-writer problem and an extension to the spooling system example involving interrupts.
2. **Priority**

In this section we are going to concentrate on certain aspects of system behaviour, such as conflict and starvation and consider how priority affects such behaviour. As in the first section, we introduce these notions with the aid of examples.

Consider the following alternative solution to the first reader-writer problem:

```plaintext
array READ,READREQUEST(n); WRITERREQUEST, WRITE(m);
(process WRITERREQUEST(i); WRITE(i) end [1] |1,m,1]
(20) (process READREQUEST(i); READ(i) end [1] |1,n,1]
    (path ,(WRITE), READ(i) end [1] |1,n,1]
```

Here we have n reader processes and m writer processes. Again, no two processes may be writing concurrently and no reader and writer may be concurrently using the resource; this is indicated by the paths. In this solution, each process first makes a request to use the resource before actually using it. No request operation appears in a path, and thus any number of processes may simultaneously be requesting use of the resource. Let us suppose that READREQUEST(1) and WRITERREQUEST(1) have both become active concurrently. In order to progress, the first writer process must activate WRITE(1) and the first reader process must activate READ(1). However, according to the path

```
    (path ,(WRITE), READ(1) end
```

concurrent activation of these operations is prohibited. In such a situation, the two operations are said to be in conflict. In general, operations are said to be in conflict if they are associated in a path or process expression by commas and if the system is in a situation where any of them could become active.
Note that a program may contain commas and yet conflict may never arise. In the spooling system (16), for example, the operations \texttt{DEPOSIT(i)} are associated by commas in the disk handler and yet they are never in conflict. This is because of the path \texttt{path ;(DEPOSIT) end} belonging to the ring-buffer resource, which after any period of system activity determines a single frame into which the next deposit must be made.

Suppose that a number of operations are in conflict. When one of them becomes active, the conflict will be said to have been resolved. Thus after \texttt{READREQUEST(1)} and \texttt{WRITERREQUEST(1)} have been activated, activation of \texttt{WRITE(1)} resolves the conflict between \texttt{WRITE(1)} and \texttt{READ(1)}.

Now consider a situation where \( m = 1 \) and \( n = 2 \). Suppose that the operations \texttt{READREQUEST(1)}, \texttt{READREQUEST(2)} and \texttt{WRITERREQUEST(1)} become active concurrently. \texttt{READ(1)}, \texttt{READ(2)} and \texttt{WRITE(1)} are now in conflict. Suppose now that the following happens.

a) The conflict is resolved in favour of \texttt{WRITE(1)} which completes one activation.

b) Now \texttt{READ(1)} and \texttt{READ(2)} are in conflict. Suppose the conflict is resolved in favour of \texttt{READ(2)} and that concurrent with that, \texttt{WRITERREQUEST(1)} becomes active.

c) \texttt{WRITE(1)} and \texttt{READ(1)} are now in conflict. Suppose that the conflict is resolved in favour of \texttt{WRITE(1)} and that concurrent to that \texttt{READREQUEST(2)} becomes active.

We may now cycle through (b) and (c) indefinitely. \texttt{READ(1)} never becomes active. In this case, we would say that the first reader process is starving because of unfair conflict resolution.
In general, starvation occurs when some process is constantly blocked from progressing, in our example the first reader process, where this results from conflict resolution never being in favour of the operation it needs to activate in order to progress. It is worth pointing out that a system may contain the possibility of starvation and yet be adequate; indeed (20) is adequate, as may easily be seen.

Starvation may result from causes other than unfair conflict resolution. Consider the following program, which is yet another solution to the first reader-writer problem:

```plaintext
array READBEGIN, READEND (n);
[process write end 3 |1,m,1]
(21) [process READBEGIN(i); READEND(i) end 1 |1,n,1]
[path write,(READBEGIN(i);READEND(i)) end [1 |1,n,1]
```

Here the read operations are treated as groups of operations, a begin operation and an end operation. (21) is merely a slight extension of the first path solution of this problem (1) and (2), but it allows us to witness another type of starvation, starvation due to "conspiracy" of processes.

Consider the following system activity:

a) READBEGIN(1) becomes active.

b) READEND(1) becomes active and concurrent with this, READBEGIN(2) becomes active.

c) READBEGIN(1) becomes active and concurrent with this READEND(2) becomes active.

We may now cycle though (b) and (c) indefinitely. Of course, concurrently with this behaviour, other reader processes may actually be progressing; the point is, however, that every writer process is starved and yet there is never any conflict within the system in the sense that at no time is it possible for a write to become active, in contrast with the case discussed earlier where READ(1) may become active at various points during the period of system activity discussed but
never actually does because of unfair conflict resolution. Here, the reader processes are "conspiring" together to ensure that at no time the resource is free.

2.1 Second reader-writer problem

The second problem requires that writers still have exclusive access to the resource but that any number of readers may use the resource concurrently. In addition we require that once a writer is ready to write, he performs his write as soon as possible. Here we are attempting to exclude the possibility of starvation of writers by readers which we have already noted as a feature of the first problem.

First we introduce an operation "writerrequest" as in (20) a writer activates this operation to indicate that it is ready to write. We would like it to be possible for as many writers as possible to make concurrent writerrequests. For this reason we declare arrays:

array WRITERSQ; WRITE(m);

where m is the maximum number of processes which will be activating write operations once the system becomes operational.

Note that, in contrast with our earlier solution, we are distinguishing between write operations performed by distinct writer processes. The reason for this is that in our first solution it was not necessary for the resource to distinguish between writes; however, in this solution, the resource must

a) distinguish writerrequests made by distinct writer processes,

b) permit a writer use of the resource only if he has made a writerrequest beforehand.

Thus "write"s must be distinguished according to the writerrequest with which they are associated.

Note also that we must now make mutual exclusion of "write"s explicit (by placing ", (WRITE)" inside a path) whereas previously mutual exclusion of "write"s was guaranteed by the inclusion of a single write within a path.
Second, we wish to constrain the system in such a way that as soon as a writer request becomes active, no reader may be permitted to begin reading until the corresponding write has been completed, although reading processes may finish reading. We shall therefore declare arrays:

\[ \text{array READBEGIN, READ}(n); \]

where \( n \) is the maximum number of reading processes that use the resource.

Our second constraint translates as follows; for each pair \( i, j \):

\[ \text{(22) } \text{path } (\text{WRITER}(i); \text{WRITE}(i)), \text{READBEGIN}(j) \text{ end} \]

If there is a conflict between \( \text{WRITER}(i) \) and \( \text{READBEGIN}(j) \), then either it is resolved in favour of \( \text{READBEGIN}(j) \) or in favour of \( \text{WRITER}(i) \), after which \( \text{READBEGIN}(j) \) is prevented from becoming active until \( \text{WRITE}(i) \) has finished.

The complete solution so far is:

\[ \text{array WRITER}(m), \text{WRITE}(m); \text{array READBEGIN, READ}(n); \]
\[ \text{[process WRITER}(i); \text{WRITE}(i) \text{ end } [i] \mid 1, m, 1] \]

(23) \[ \text{[process READBEGIN}(i); \text{READ}(i) \text{ end } [i] \mid 1, n, 1] \]
\[ \text{[path } (\text{WRITER}(i); \text{WRITE}(i)), \text{READBEGIN}(j) \text{ end } [i] \mid 1, m, 1] \mid 1, n, 1] \]
\[ \text{[path } (\text{WRITE}), (\text{READBEGIN}(i); \text{READ}(i)) \text{ end } [i] \mid 1, n, 1] \]

The last set of paths indicates the constraint that each writer must have exclusive use of the resource.

We may analyse the behaviour of this system as follows:

a) Writer requests may be active concurrently.

b) Readers may be active concurrently; they do not mutually starve each other.

c) Because writers are mutually excluded by the comma, they may still starve each other.

d) If a writer request is active concurrently with a read then according to the last collection of paths, writing cannot begin until the read is completed. However, the first collection of paths prevents
further readers from beginning until the write corresponding to writerequest is completed.

e) Conflict between \text{WRITERQ}(i) and \text{READBEGIN}(j) may be resolved unfairly, that is in favour of readers, so that writers may still starve.

f) Conversely, writers may starve readers.

However, the second problem requires that the arrival of a writerequest should block all readbegins until the corresponding write is finished. \((e)\) shows that \((23)\) does not guarantee this. We could invalidate \((e)\) by giving writerequests a fixed priority over readbegins in cases of conflict. To do so, we make use of the \textit{precedence construction} \cite{Campbell1976}, written ":>".

This separator may be used in a path wherever a comma is used. The significance of
\[
\text{path } a > b \text{ end}
\]
is that in case of conflict between operations \(a\) and \(b\), conflict is always resolved in favour of \(a\). We may use this construct in \((23)\) to obtain the following solution:

\[
\text{array WRITERQ, WRITE(m); array READBEGIN, READ(n);}
\]

\[
\text{[process WRITERQ(i);WRITE(i) end])} [1,m,1]
\]

\[
\text{(24) [process READBEGIN(i);READ(i) end])} [1,n,1]
\]

\[
\text{[[path (WRITERQ(i);WRITE(i)) > READBEGIN(j) end])} [1,m,1] [1,n,1]]
\]

\[
\text{[path , (WRITE),(READBEGIN(i);READ(i)) end])} [1,1,n,1]
\]

(a), (b), (c), (d) and (f) still hold for \((24)\) but the introduction of the precedence construct has the following consequence:
e') If some writer process is ready to activate a write request and some reader process is ready to begin reading, then it is always the writer that is permitted to progress by the resource. Note again that this precedence is a property of the resource and that it is the structure of the resource and not of the processes which ensures proper functioning of the system.

Prevention of starvation of writers by other writers is still not assured by this solution. We can eliminate such a possibility by putting constraints on the order in which writers may use the resource. This can be done with the aid of a "buffer with skip", illustrating another use of the precedence construct. To (24) adjoin an array:

```
array SKIP(m);
```

and a path:

```
(25) path [(WRITE(i) > SKIP(i)) @; i ∈ [1,m,1]]
```

Suppose WRITE(i-1) has just been activated. If WRITE(i) has also just been activated, then the ith writing process is ready to write and the operations WRITE(i) and SKIP(i) are in conflict. The presence of the precedence construction determines that WRITE(i) is performed. If, on the other hand, there has been no request from the ith writing process, then WRITE(i) and SKIP(i) are not in conflict and indeed SKIP(i) may become active, after which the i+1th writer process may be allowed to write and so on. The introduction of this "buffer with skip" ensures that once a writer has requested the use of the resource, his request will be serviced in a finite amount of time.

It is instructive to note how by examining all occurrences of commas in a program we can exhaustively consider all the possibilities of starvation due to conflict. This is not so easy in a solution using semaphores, where mutual exclusion is handled implicitly and not explicitly.
We compare our path solution to the semaphore solution given in [Courtois, Heymans, Parnas 1971].

```plaintext
integer readcount, writecount; (initial value = 0)  
semaphore mutex1, mutex2, mutex3, w, r; (initial value = 1)

READER
P(mutex3);  
P(r);  
P(mutex1);  
readcount:=readcount+1;  
if readcount=1 then P(r);  
V(mutex1);  
V(r);  
V(mutex3);

***

writing is performed

***

P(mutex1);  
readcount:=readcount-1;  
if readcount=0 then V(w);  
V(mutex1);  
if writecount=1 then V(r);  
V(mutex2);

***

reading is done

***
```

We may make the following remarks about this solution in contrast with our own:

1) In the semaphore solution, the responsibility of ensuring correct use of the resource lies with the processes.

2) For each type of process, requests and releases are mutually exclusive. This is to ensure proper use of the counters. In the path solution the write requests may take place concurrently.

3) Requests to use the resource take place in a critical region, entry to which requires performing a P operation on the semaphore r. Once inside the region, either type of process may then enter
the region containing the operation relating to the resource. However, a reader must exit from the request region after gaining the use of the resource. In contradistinction, only the first writer in a stream needs to perform P on r; as long as there are a positive number of waiting writers, which is given by the current value of "writecount", then the value of the semaphore r will remain zero, thereby preventing readers from requesting use of the resource. There is thus a possibility of the writers starving the readers.

4) A reader and writer may concurrently arrive at a point where in order to progress they need to perform a P operation on the semaphore r, where the value of r is one. In such a case, the outcome is undefined. The reader may manage to perform a P on r. However, no other reader is permitted to request the resource until he may enter the section enclosed within P(mutex3) and V(mutex3) which may only take place after the first reader has performed a V on mutex3. But in order to do this, he must first perform a V on r, after which any writer waiting on r immediately enters the section controlled by r which then becomes locked against readers until the final writer in a stream has left the section. Thus, once a write request arrives, it is guaranteed that some write will take place within a finite amount of time.

5) There is still the possibility of starvation of some writer by other writers. If we consider the case in which a collection of writers are waiting on w, then it is possible that for some writer W the resolution of the implicit conflict at P(w) is never resolved in its favour. If there is a constant stream of writers queueing on w, then W will starve indefinitely.

This seems to fail to satisfy the requirement that "once a writer is ready to write, he performs his write as soon as possible".

In regard to 4), it is instructive to consider the P and V implementation of the precedence construction given in [Campbell 1976].
path A > B end

is implemented by

\[ \begin{align*}
    &P(s); \quad P(s'); \\
    &P(s'); \\
    &B; \quad A; \\
    &V(s'); \\
    &V(s); \quad V(s'); \\
\end{align*} \]

3. Interrupts

In the preceding section, we introduced a precedence construction which allows us to specify fixed priority between operations in cases of conflict. In this section, we indicate how this construction may be used to describe interrupts. We shall do this by example; the spooling system of section 1.4 (16), will be extended to include an interrupt handling process.

This process does the following:

1) At any time it may receive an error message from some other unspecified part of the system.

2) It then halts spooling.

3) It prints an alarm message on the printer.

4) Finally, it allows spooling to be resumed and returns to a waiting state.

The interrupt handling process thus may be written as:

(27) \textbf{process} getmessage; stopsystem; printalarm; startsystem \textbf{end}

As soon as an alarm message is received no operation of the spooling system must be permitted to be activated until the interrupt handler has allowed spooling to be resumed. One may express this as follows:

(28) \textbf{path} (stopsystem; startsystem), op \textbf{end} for each operation 'op' in the spooling system.
What this path says is that there is a potential conflict between the operation of stopping the system and the activation of any operation in it (excluding those belonging to the interrupt handler, of course). Once stopsystem has been activated, no operation may become active until startsystem has been activated.

However, we wish to specify an explicit priority of the interrupt handler over the other processes. This may be done with the help of the precedence construction as follows:

For each operation 'op' in the spooling system, there is a path:

(29) \text{path (stop}\text{system; start}\text{system)} > \text{op end}

The full solution called $P_{int}$ is thus:

```
array DEPOSIT, REMOVE(10);
process readsector; [(DEPOSIT) @; \text{[1]} | 1,5,1}] end
process ; (REMOVE); printline end
process getmessage; stopsystem; printalarm; startsystem end

(30) \text{[path DEPOSIT(i); REMOVE(i) end [1 | 1,10,1] path ; (DEPOSIT) end path ; (REMOVE) end path (stop}\text{system; start}\text{system)} > readsector end path (stop}\text{system; start}\text{system)} > printline end}

\begin{align*}
\text{[path (stop}\text{system; start}\text{system)} > DEPOSIT(i) end [1 | 1,10,1]} \\
\text{[path (stop}\text{system; start}\text{system)} > REMOVE(i) end [1 | 1,10,1]}
\end{align*}
```

With the addition of the interrupt handler and the paths, one has a system in which: Spooling continues until the operation getmessage is activated by the interrupt handler, signifying that an error message has been received by it, after which stopsystem is in conflict with every operation in the spooling system. The structure of the paths (*) entails that priority will be given to the interrupt handler. "stopsystem" will then take place followed by printalarm which denotes the writing of an alarm message by the interrupt handler on the line printer. Finally the operation startsystem is activated. At no time between the activation of getmessage and startsystem may
any other operation of the system become active without contravening
the constraints expressed by the paths (*).

One consequence of this is that the sequence

\[ \text{INT} = \text{getmessage stopsystem printalarm startsystem} \]

is incapable of being interrupted or atomic. Here we see a way in
which atomicity may be indicated in a path program.

This property also simplifies proof of the adequacy of the
system (refer to section 1.4). Let us suppose \( S \) is a firing sequence
of this program. If the last operation in the string \( S \) is an operation
from the interrupt handler, then clearly, by the atomicity of the
sequence \( \text{INT} \), it must be possible to adjoin the remaining operations
from the interrupt handler so that the resulting string \( S' \) has the
following property:

If \( S' = S_1 \text{getmessage} S_2 \), where \( S_1, S_2 \) are strings, then there is a
string \( S_2' \) such that \( S' = S_1 \text{INT} S_2' \).

We may thus assume without any loss of generality that \( S \) has this
property. To complete the proof of the adequacy of \( P_{\text{int}} \); it is
necessary to prove that there is a string \( T \) such that \( ST \) is a firing
sequence of \( P_{\text{int}} \) and that \( T \) contains every operation of \( P_{\text{int}} \).

Let \( \tilde{S} \) be the string obtained from \( S \) by deleting every occurrence
of the string \( \text{INT} \) from it. Clearly \( \tilde{S} \) is a firing sequence of the
original spooling system \( P \). But \( P \) was proved to be adequate. It
follows that there exists a string \( T' \) such that \( T' \) contains every
operation of \( P \) and \( \tilde{S}T' \) is a firing sequence of \( P \). Now if we replace
the copies of \( \text{INT} \) in \( \tilde{S} \) giving \( S \) again, it should be clear that \( ST' \) is
a firing sequence of \( P_{\text{int}} \). Finally, if \( P \) is any firing sequence of
\( P_{\text{int}} \), then \( P_{\text{int}} \) is also a firing sequence of \( P_{\text{int}} \), since the activa-
tion of getmessage is not constrained by any other part of the system
and once it has become active only the rest of \( \text{INT} \) may become active.
We now have constructed a string \( T = T'\text{INT} \) such that \( ST \) is a firing
sequence of \( P_{\text{int}} \) and such that \( T \) contains every operation from \( P_{\text{int}} \)
We have shown that $P_{int}$ is indeed adequate.

The addition of the interrupt handling process introduces the possibility of starvation into the system. A constant stream of error messages will cause conflict always to be resolved in favour of stopsystem, thereby starving the spooling processes. However, this would be the case only if the system was constantly malfunctioning in some respect, in which case starvation of the spooler would be desirable rather than not. Likewise, if the system never malfunctioned, the interrupt process would starve, but again this is desirable system behaviour.

**Conclusion**

We have compared two approaches to system structuring, a process-oriented and a resource-oriented approach, and have noted that the latter seems to have advantages over the former in permitting a greater degree of distribution of control and of concurrent behaviour. More specifically, each resource in such a system is capable of taking its own decision in regard to its use by processes in such a way that the outcome of the decision will depend solely on local properties of the resource.

In this connection, we considered the path notation (a programming notation which may be used to specify distributed systems), which lends itself to a resource-oriented approach; a resource may be identified with a collection of (synchronization statements) paths involving all the operations associated with the resource, in this way defining decision mechanisms which ensure correct usage of the resource by processes. The notation has the further advantage of permitting abstraction from details of implementation.
We also indicated a manner in which a system may be shown to have desirable behaviour, by proving the adequacy of its related path specification. In our informal proof we made use of two techniques: adequacy-preserving transformations may be used to reduce a complex program to one more tractable; general results also exist which characterise for certain classes of path programs, those that possess the adequacy property. We are confident that these techniques can be used to prove the correct behaviour of entire operating systems specified in the manner we have suggested.

In [Lauer, Best, Shields 1977], it became obvious to us that there were various versions of the notion of adequacy which should be distinguished, both for theoretical and for practical reasons. In [Devillers 1977, Devillers, Lauer 1977a,b] we concentrated on starvation due to unfair conflict resolution and starvation due to co-operation of a set of processes at the expense of another. Our discussion of the spooling system with interrupts shows that not all starvation possibilities are undesirable. The cited series of papers can be seen as a progressive deepening of our understanding of these important types of system behaviour.

Acknowledgements

The authors wish to acknowledge the Science Research Council of Great Britain for their financial support.

Some of the solutions to the reader/writer problem were first formulated by the authors in collaboration with Professor B. Devillers who was visiting our project for a period of two months.

Finally, we would like to express grateful thanks to Miss Julie Lennox for her patience and efficiency in preparing the manuscript.
Bibliography


[Shields 1977] Shields, M.W.: *A simple spooling system described using the path notation, together with a proof of its adequacy*, ASM/38, Computing Laboratory, University of Newcastle upon Tyne, October 1977.
