LANGUAGE FEATURES FOR PROCESS INTERACTION*

G.R. Andrews and J.R. McGraw

TR 76-290

September 1976

Computer Science Department
Cornell University
Ithaca, NY 14853

* This work was supported in part by the National Science Foundation under grants GJ41115 and GJ42512.

LANGUAGE FEATURES for PROCESS INTERACTION

Gregory R. Andrews
James R. McGraw

Cornell University
September, 1976

Number LDRS - 158

This work was supported in part by the National Science Foundation under grants GJ41115 and GJ42512.

Abstract

Languages for parallel programming should meet four goals: expressiveness, reliability, security, and verifiability. This paper presents a set of language features for describing processes and process interaction, gives examples of their use, and briefly discusses their relation to the goals. Two constructs, resources and protected variables, are introduced as the mechanisms for describing interaction. Resources are extensions of the monitor concept of Hoare; protected variables are global variables which can only be accessed by one process at a time. Two types of access control are introduced: restrictions on scope rules for static access, and capabilities for dynamic access. Examples include the interface to machine devices, files and virtual devices, device scheduling, device reservation, and buffer allocation.

1.0 INTRODUCTION

In order to increase the reliability of parallel programs, a number of high-level language features have been proposed in recent years. The work of Brinch Hansen [1,2], Hoare [5] and Wirth [13,14] is especially noteworthy and has led to numerous refinements and extensions [for example, 7 and 11]. In this paper, we present a unified set of language features to specify and control processes and their interaction. In particular, we describe language mechanisms for processes, resources, shared variables and protection. We then demonstrate the use of our tools by specifying an interface between machine hardware and high level programs.

The next section of the paper describes a syntax for processes and presents four process control operations. Then we discuss mechanisms for process interaction and describe two new features: resources (an extension of monitors [5]) and protected variables. The fourth and fifth sections describe mechanisms for static and dynamic allocation and access, respectively. A scope restriction mechanism is introduced to limit static access; dynamic access is controlled by resource capabilities and pointers to protected variables. One major difficulty in any high-level language is designing a machine independent interface. This problem is accentuated in systems languages because of the need of systems programs to control machine resources. In the sixth section of the paper, we show how our language-defined resource construct can be used to represent machine resources, such as devices and clocks, as well as logical resources such as files and message buffers. The use of protected variables and pointers to efficiently manage IO buffers is also illustrated. Consequently, our language allows the definition of all process interaction, whether for programmed or machine resources.

Four specific goals have guided the work presented here: expressiveness, reliability, security, and verifiability. They are, we feel, the yardstick against which any systems language should be measured. First, the language must be sufficiently rich to enable a wide variety of policies to be naturally expressed. To use the distinction so aptly drawn by the Hydra group at Carnegie-Mellon [15], language features are the mechanisms by which system policies are described. As such, the features must be devoid of policy decisions which might preclude the efficient implementation of reasonable systems algorithms. Our focus is upon describing systems of interacting processes. Therefore, we are concerned, when implementing a system, with scheduling the activity of processes, defining the means by which they interact, controlling access to shared resources, and allocating resources. To be useful, our language features must make it possible and easy for us to perform these tasks. Related to the richness of expression, is the goal of uniformity of expression. Similar concepts should have a compatible representation and similar operations should have the same interpretation.

The second design goal is to insure that programs are reliable. Reliability means that all time-dependent errors which could result from the activity of parallel processes are detected at compile time. This requires that no two processes ever have access to the same variable(s) at the same time. The most insidious errors in systems programming are those which occur asynchronously. In our opinion, the main argument in favor of using high-level languages for systems programming, as long as they are "good" languages, is to increase the reliability and security of systems. A good language at a minimum is sufficiently structured that all access paths and access overlap can be enumerated at compile time. Our language features meet this requirement.

Our third design goal, security, has two aspects: access restriction and guaranteed service. First, every module (process, procedure, block) in a program should only have access to what it requires to carry out its function. In addition, data should only be accessed by valid operations (data encapsulation). Global variables are therefore bad because they violate this notion. There are also many situations where access should be further restricted to a subset of the valid operations; for example one process can only send messages of a certain type while another can only receive. For efficiency, all static access restrictions should be enforced at compile time. As we argue in section five, some dynamic access control is required to efficiently manage dynamic resource allocation. We introduce capabilities for this purpose. The cost of their use must be and is minimal, however.

The other aspect of security is guaranteeing service. Whereas access control is concerned with preventing invalid operations, guaranteed service is concerned with allowing valid operations, if they are attempted. This requires that no process can become deadlocked [6] and that all scheduling mechanisms are fair. Although absence of deadlock is a property of a program that cannot be guaranteed a priori just by programming in a particular language, the language can provide enough structure to make it possible to decide if a given program contains a deadlock possibility. We comment on the relation between languages and deadlock at the end of the paper.

The fourth and final goal motivating our work is the ability to prove the correctness of programs. Work in program verification affects language design by indicating which language constructs are hard to handle (e.g. procedures as parameters) and which lead to more readily verified programs (e.g. encapsulated data types). The semantics of our language proposals have not yet been formalized, so we are not completely sure of their impact. We have not, however, knowingly made any choices which lead to difficulty. And we have retained the axiomatic structure proposed by Hoare [5] in our resource facility.

The following sections describe our process specification and process interaction proposals. At the end of the paper we comment further upon their relationship to the four goals described above. The language notation is based upon Pascal syntax [12]. As a base, we assume a sequential language like Pascal; our proposals are extensions to the base.

2.0 PROCESSES

Any language for parallel programming must provide a means for describing processes. In order to enhance reliability, parallel activity should be clearly separated from sequential activity. And if the language is to be useful for writing systems programs, it must be possible to control the scheduling and execution status of processes. As stated in the introduction, a parallel language should merely provide the mechanisms for creating and controlling processes; control policies themselves are expressed by algorithms in the program of a system. In this section, we present a process data type and describe the operations on process objects.

Because processes are like procedures in that they contain declarations and code, we choose to represent them syntactically by the following notation:

```
pname: process;
       local declarations;
       body;
       end pname;
```

Processes in Concurrent Pascal [1] and Modula [13,14] are declared similarly.

Processes are manipulated by four operations: setstate, changestate, activate, and suspend. When created, a process cannot yet execute. First it requires a scheduling state and then it must be activated. A process state consists of a priority, time limit, and time quantum. Process scheduling is based on priorities and uses time quantums for splitting time among equally high priority processes. At all times, one of the highest priority, active processes is executing. It executes until it blocks waiting for an event (see next section), is suspended, or exceeds its time quantum. A blocked process waits until awakened by another process, a suspended processes waits for activation, and a process which exceeds its quantum waits its turn for another quantum. If a process executes for a total amount of actual time in excess of its time limit, it is automatically suspended.

In addition to a scheduling state, each process has a status: running, ready-active, ready-suspended, blocked-active, or blocked-suspended. The running process is executing on a processor; ready and blocked indicate whether or not a process is waiting for an event; active and suspended indicate whether or not a process has been allowed by its controller (e.g. its creator) to continue execution. A more detailed discussion of these five execution states can be found in [10].

To manipulate the scheduling state and status of a process, four operations can be used. First, if a process pn has been created, its scheduling state can be set or changed by invoking:

```
pn.setstate(priority,quantum,limit);
```

The process can then be activated by:

```
pn.activate;
```

These operations are always executed in order to start a newly created process. Should it be necessary to ever suspend the process, it can be done by:

```
pn.suspend;
```

Finally, it may be necessary or desirable to examine a process state, perhaps before changing it. This can be done by calling:

```
pn.readstate(priority,quantum,limit,used);
```

which returns four values. The "used" parameter is set to indicate how much time the process pn has currently used. We now describe the mechanisms for process interaction.

3.0 PROCESS INTERACTION

In any system with multiple processes, interaction will occur. Processes compete for access to limited, reusable resources and cooperate by exchanging information and synchronization signals. A programming language can permit processes to interact in at least four different ways:

(1) through a language defined message passing facility;

(2) through encapsulated, synchronized data types such as <u>monitors</u>;

(3) through access to shared reentrant procedures; and

(4) through access to global variables and types, such as buffers.

We propose three facilities for interaction: resources to implement shared data types requiring exclusive access, reentrant shared procedures, and protected variables to allow processes to access global data one at a time. A system wide message passing facility can be designed using our resource type. We do not feel that the language should define the message passing facility, however, because policy decisions would then have to be made. For example, how much information can be passed, what is its type, and is it passed by value or reference?

3.1 Resources

A resource is an extension of a monitor and is used for much the same purposes. The two main differences are that we allow resource operations to execute in parallel when possible and we distinguish between static resources controlled by scope rules and dynamic resources controlled by capabilities. We use the term resource to emphasize that the resource construct is the way in which any machine or logical system resource is represented. They are used in encapsulate each machine device and describe its operational interface to the software system. This is illustrated in section 6.

A resource has the following form:

rname: <attribute> resource;
    entry operation$_1$,...,operation$_m$;
    parallel* procedure$_1$,...,procedure$_p$;
    data declarations
    procedure operation$_1$(parameters);
        end operation$_1$;
    .
    .
    .
    procedure operation$_m$(parameters);
        end operation$_m$;
    local procedures
    initialization code
    end rname;

The *'d component (parallel phrase) is optional. The attribute is either static or dynamic; it governs storage allocation and access control. A static resource is allocated storage when its declaration occurs whereas a dynamic resource is allocated storage via a create operation. These two topics are discussed in detail in the next two sections.

When a resource instance is created, the initialization code is executed. It initializes the local data of the resource. Subsequent access to the local data is provided by calls on the entry operations. A call has the form:

rname.operation(actual parameters);

where rname is a resource name. Within resources, synchronization of operations is provided by wait and signal operations on condition variables just as within monitors.

The normal mode of execution within resources is exclusive access to the local data. When an entry operation calls a local utility procedure or calls outside to another resource or global procedure, it retains exclusive control of the resource. Control in the normal (monitor-like) situation is relinquished when the resource operation returns, a wait is executed on a condition variable, or a signal is executed which awakens another process.

Two situations arise where exclusive access is unnecessary and leads to inefficient or clumsy problem solutions. One occurs when one or more resource operations only examine the local data; the other when one resource is used to schedule access to another, for example a disk or drum scheduler. Whenever it is possible for resource operations to overlap without disrupting integrity, the parallel phrase can be used. It has the form:

$$\underline{\text{parallel}}\ \text{procedure}_1, \ldots, \text{procedure}_p;$$

and names all those procedures defined by or called by the resource which can be executed in parallel. The effect of calling a parallel procedure is to give up exclusive resource access; on return it must be regained, however.

To insure data integrity, only one process at a time must be able to modify the resource data, namely the process which has exclusive resource access. Therefore, parallel procedures must not be able to modify the resource data in any way. To insure this, two constraints are placed on resources: first, parallel procedures inside a resource cannot assign values to resource data; second, no resource data can be passed as a parameter. The reason for the second constraint is that parameters are passed by reference, which could cause implicit assignments. Both of these constraints can easily be checked by a compiler.

An example of a simple resource employing the parallel phrase is shown in Figure 1. It solves the famous reader/writers problem [3] assuming readers have priority over writers. Another example, device scheduling, is shown in Section 6. The reader writer resource, named rw, implements local data, store, of type T. The data is examined by calls to rw.read(d) and updated by calls to rw.write(d) where d is of type T. Within the resource two extra variables, oktowrite and readers, are used for synchronization of read and write. Variable oktowrite is a condition which writers wait for if other processes are reading; readers indicates the number of processes actively reading. Because more than one process can safely read at once, actual reading takes place in a local, parallel procedure doread.

In the solution, read and write are executed as critical sections since they are not in the parallel phrase. Therefore, no two writes can occur at the same time. Simultaneous reading and writing is prevented because writing proceeds only when the value of readers is zero and readers is only set to zero if no process is inside either read or doread. Concurrent reading is permitted by having a separate, parallel doread procedure which is invoked by read. Other variants of the readers/writers

Figure 1

Readers/Writers

```
type T = record  . . .  end;

rw: static resource;
    entry read, write;
    parallel doread;

    var store: T;
        oktowrite : condition;
        readers : integer;

    procedure read(var d:T);
        readers := readers + 1;
        doread(d);
        readers := readers - 1;
        if readers = 0 then oktowrite.signal;
        end read;
    procedure write(var d:T);
        if readers > 0 then oktowrite.wait;
        /* write */
        store := d;
        end write;
    procedure doread(var d:T);
        /* read */
        d := store;
        end doread;
    /* initialize readers */
      readers := 0;
end rw;
```

problem have similar <u>resource</u> solutions.

We feel that our algorithm is superior to that in [5] because we completely capture the solution in one place. Hoare's solution requires that each process call a monitor to request and release permission and then access the store directly. The drawbacks are that (1) processes must be trusted to get permission before attempting access and (2) they require direct access to the global store. Insuring that the store is not erroneously accessed by concurrent processes would be very difficult, if not impossible to insure. The whole advantage of monitors is encapsulating data and enforcing exclusive access constraints. Yet the monitor solution to readers/writers throws both advantages away. We were lead to our <u>parallel</u> construct for these very reasons.

3.2 Shared Procedures

In addition to sharing resources, for efficiency we allow processes to share the code of global procedures. We therefore require that procedures be pure (reentrant). This implies that compilers for a language containing our extensions must allocate separate local storage for each activation of a shared procedure and that compiled code must not modify itself. Both requirements are easily met with known techniques. If a procedure needs local storage (own variables) which is retained from call to call, then the procedure must and should be implemented as an operation in an encapsulated <u>resource</u>.

3.3 Protected Variables

Whenever data is shared, we must insure that it can only be accessed by one process at a time. If multiple access paths exist, then the data must be contained within a resource. There are situations, however, where data is shared but only one process at a time has access to it. For example, buffers used in IO processing are often stored in a global pool used by several input, output and/or user processes. Each individual buffer is used by only one process at a time, however, and is accessed by simple read and write (assignment) operations. If shared variables, such as buffers, are implemented in resources, there is significant execution overhead caused by procedure calls to do reads and writes and by unneeded mutual exclusion. For those situations where simple data types, such as records or arrays, are shared but only one access path at a time ever exists, we propose using protected variables.

A protected variable is any variable declared with a protected attribute. Operations on the variable are exactly those for its type, but a protected variable can only be accessed by using a variable of type pointer. To illustrate the declaration and use of protected variables and pointers, we declare a protected type and two pointers as follows:

type T = protected <basic type>;

P_1, P_2: pointer to T;

Once p_1 is bound to an instance of T (see Section 5), the contents of the instance are referenced by:

$$*p_1.\text{field}$$

where the .field, if present, names a subfield of T. Pointers themselves can be assigned and passed as parameters. To insure that only one pointer is ever bound to the same instance, however, assignment and parameter passing both destroy one pointer when transferring its value to another.

The use of protected variables and pointers to implement a buffer pool manager is shown in Section 6. Their allocation and access control are discussed further in Section 5.

4.0 STATIC ALLOCATION AND ACCESS

The two previous sections defined language facilities for process definition and interaction. In this section and the next, we show how these facilities are created and controlled, and also show how they interact with common sequential language constructs. Each object defined in a program (variables, structures, procedures, resources, and processes) is either statically or dynamically controlled. This section defines the allocation, access, and implementation of static objects.

4.1 Allocation

Every static object obeys the standard ALGOL rules for allocation. When a declaration is encountered (at the beginning of a block) space is immediately allocated and the object comes into existence. The object remains until the block in which its declaration appears is exited. At that time, the space for the object is deallocated.

The syntax for declaring static objects is fairly standard. All of the normal sequential language objects (variables, arrays, records, procedures) are declared with conventional Pascal syntax; the syntax for processes, static resources, and pointers has been illustrated in previous sections. The only other static object type is capability. Although both capabilities and pointers are used to represent dynamic objects, they themselves have static control. The syntax for a capability declaration is:

```
name: capability for <dynamic resource def>;
```

As in PASCAL, new types can be defined. The type definition itself falls under static control even if it defines a dynamic type of object. An example of the declaration and use of a type is:

```
type terminal = process (parameters);
                declarations
                code
                end terminal;

console: terminal (actual parameters);
```

As shown, a type declaration can be parameterized. Type declarations for other objects all have the same form.

4.2 Scope of Access

The maximum scope of access for any static object is the range of the block in which the instance is defined. Algol-like scope rules allow inner blocks to access global objects, unless the inner block re-defines a global name. With the introduction of process and resource blocks, this rule is too strong. Two of our design goals can be achieved by modifying Algol's default scope rules. First, simultaneous access to data objects by two processes can be prevented by only permitting a data object to be accessible in a single process environment. Second, the transmission of rights for a dynamic object from one process environment to another can be limited to parameter passing. These two goals have lead to the default scope rules shown in Figure 2 which indicates the types of objects which are accessible to a block if they are declared global to it.

The only entry which might cause confusion is the starred one. Normally, internal procedures are permitted to access global variables. If the procedure can be called from different processes, however, simultaneous data access is possible. Therefore, the rule (enforceable at compile time) is that a procedure may access global variables only if it will be called solely from the process environment that defined it.

The default scope rules define the maximum allowable access policy of any program. However, many circumstances arise where it would be useful to be more restrictive. This type of con-

Figure 2

Default Scope Rules

Global Object Type	Block Types to which object is accessible+ (if within scope of object)
1. Data declatation (integer, record, etc.)	procedure*
2. Procedure declaration	procedure, process, resource
3. Process	procedure, process, resource
4. Static resource	procedure, process, resource
5. Pointers and capabilities	procedure
6. Type declarations	procedure, process, resource

+ not including the block declaring the object.

* allowed if all calls to the procedure are from the reach of the process in which the procedure is declared.

straint can be enforced through the use of a scope clause which can be attached to the declaration of any type or instance. The general form is demonstrated with a variable declaration:

alpha: integer scope $block_1, \ldots, block_N$;

Only blocks declared at the level where the scope appears can be included in the block list. The effect of the clause is to permit only those blocks listed to access the object being declared. The scope clause cannot be used to relax the default rules, only to further restrict them on an individual object basis.

In addition to direct access allowed by the scope rules, some objects can be passed as parameters. Simple variables and structures (records and arrays) are passed to procedures by reference. They can be passed to type definitions by value only, however, because of the possibility of simultaneous access. Pointer variable and resource capability passing is described in the next section. We do not allow, or see a need which requires procedures, processes or resources to be passed as parameters.

4.3 Implementation

One of the major concerns with any proposed language feature is the efficiency of its implementation. Our constructs are good in this respect. All of the access rules can be enforced at compile time, particularly the scope clause which only involves changes to the symbol table control procedures. Memory management is only complicated slightly. For a particular process the standard stack allocation is possible since object existence is tied to the block structure. The possibility of nested processes creates the only memory difficulty because it involves finding space for a new stack. In.general though, the static objects do not cause any problems.

5.0 DYNAMIC ALLOCATION AND ACCESS

The second method of object control is dynamic allocation and access. Only two types of objects can be controlled in this way: protected data structures (variables and records) and dynamic resources. The major difference between static and dynamic control is that the access rights possessed by a process environment for a dynamic object may vary during execution. All dynamic objects are referenced through a form of pointer mechanism: resources via capabilities, and protected objects via pointers. In this section we describe dynamic allocation, the use of capabilities and pointers, and the implementation of dynamic control.

5.1 Allocation

The lifetime of a dynamic object is different from its static counterpart. A dynamic object may be created at any time during execution, not just on block entry. Deallocation occurs only when no references (pointers) to the object remain instead of on block exit. Creating a dynamic object involves three steps: (1) defining the type of object, (2) defining a capability

(or pointer) to reference the object, and (3) executing the create operation to generate an instance. The creation of a dynamic resource begins with a type definition.
For example:

```
type buffer = dynamic resource (parameters);
     {body}
     end buffer;
```

Step two is a simple capability declaration, in this case:

```
IO_buf: capability for buffer;
```

Then an actual instance of the dynamic object is created by executing:

```
IO_buf := buffer.create (parameters);
```

Notice that both the capability name and the resource type name must be accessed in order to accomplish the creation. This requirement will be useful later. The steps for the creation of a protected data object are almost identical. An example follows.

```
type count = protected integer;
     size : pointer to count;

     size := count.create;
```

In general, integer could be replaced by any data structure definition.

Dynamic objects could be created without using a type definition, but then only one instance exists and, more importantly, only one reference pointer exists. No other block could then refer to the object because they could not declare a name for referencing it. Although legal syntactically, untyped dynamic objects merely act like static objects.

5.2 Access Control

Every dynamic resource is accessed through capabilities [8,15]. A resource capability has two components: (1) a reference to a particular instance, and (2) a set of access rights for the instance. The maximum rights for a resource are call rights for every resource entry plus the language-defined copy right. A capability is used to call an entry by executing a statement of the form:

capability_name.entry (parameters);

The call is permitted only if the capability contains a right for that entry. This check must be made at run-time.

Capabilities are manipulated by assignment statements and parameter passing. The copy right allows the contents of one capability to be copied into another capability of the same type. The form of the copy is:

cap_name_1 := cap_name_2 {rights list};

The bracketed rights list identifies the rights to be given to cap_name_1; obviously all these rights must be possessed by cap_name_2. Rights which may appear in the list are the entry point rights and copy. The rights list may be omitted if all rights are to be copied. The copy operation does not affect the contents of cap_name_2.

Since the default scope rules (Figure 2) do not allow a capability to be shared by two processes, sharing can only occur through parameter passing. When a capability is passed, its contents are transferred to the formal parameter capability and the actual parameter loses all rights. On return from the call, the reverse transfer is performed. The purpose of this approach is to be able to control all capability copying by the copy right yet still permit the acquisition of a resource by passing an empty capability and returning with a full one; or the release of a resource by the reverse action.

A last operation that can be performed on a capability is to empty it. This action requires no right in order to execute: The form is:

cap_name := null;

One use of this tool would be in a resource responsible for transferring capabilities among processes. The resource could receive the capability in a parameter, copy it into its global information, then empty the parameter. When the resource returns, the calling process has lost its access. This is illustrated by an example in Section 6.

The control and use of protected data structures (via pointers) is much the same as with capabilities, but with some important differences. The purpose of the protected attribute is to permit a structure to move among processes. A key difference is that data structures do not have built-in exclusion to avoid simultaneous access by different processes. To solve this problem, each protected object will only be permitted to have one pointer to it at any time (implying only one process can access it at any time); the value of a pointer is simply the "name" of a protected object. This restriction simplifies the access and manipulation mechanisms for pointers. There is no copy concept because copying is forbidden. The object referenced by a pointer can be transferred to another pointer by the assignment statement:

name_1 := name_2;

This assignment both copies the pointer and empties name_2. A null value can also be assigned in order to empty a pointer.

Access to a protected object is accomplished by explicit dereferencing of the pointer. The form is:

*pointer_name

For example, if pointer_1 and pointer_2 are both pointers to protected integers, the value of the second integer can be copied by the assignment:

*pointer_1 := *pointer_2;

The result is that the pointers reference different objects which now have the same value. If the data structure is a record, the sub-fields of the record can be specified using the standard dot notation following the dereferenced pointer. Subscripts are used with pointers to arrays.

As with capabilities, parameter passing is what makes pointers useful; it is the only way for a protected object to be moved through various process environments. When a pointer is passed, the effect is exactly the same as assigning the actual pointer to the formal one; the calling environment loses access. On return, the reverse assignment is made.

5.3 Implementation

There are several points that should be made concerning the implementation of dynamic objects. The first, and probably most troublesome, is that dynamic objects involve dynamic allocation which implies that some form of dynamic memory management must be provided by the compiler or computer system. We no longer have a simple stack allocation system; each object's existence is independent of all others so some form of memory recovery is needed. Another implementation point concerns capabilities. A run-time check of each access is needed. The overhead can be minimized, however, by implementing a capability's rights in a bit vector. An access check then involves examining a bit and allowing access if the bit is set. Since capabilities are only used to reference resources, most capability accesses will already involve the overhead of a procedure call to a resource entry. The extra overhead of capability checking should therefore be small. The last implementation point concerns pointer variables. These objects can be implemented with almost no run-time checking because no rights need to be examined. By difference is that data structures do not have built-in exclusion to avoid simultaneous access by different processes. To solve this problem, each protected object will only be permitted to have one pointer to it at any time (implying only one process can access it at any time); the value of a pointer is simply the "name" of a protected object. This restriction simplifies the access and manipulation mechanisms for pointers. There is no copy concept because copying is forbidden. The object referenced by a pointer can be transferred to another pointer by the assignment statement:

name_1 := name_2;

This assignment both copies the pointer and empties name_2. A null value can also be assigned in order to empty a pointer.

Access to a protected object is accomplished by explicit dereferencing of the pointer. The form is:

*pointer_name

For example, if pointer_1 and pointer_2 are both pointers to protected integers, the value of the second integer can be copied by the assignment:

*pointer_1 := *pointer_2;

The result is that the pointers reference different objects which now have the same value. If the data structure is a record, the sub-fields of the record can be specified using the standard dot notation following the dereferenced pointer. Subscripts are used with pointers to arrays.

As with capabilities, parameter passing is what makes pointers useful; it is the only way for a protected object to be moved through various process environments. When a pointer is passed, the effect is exactly the same as assigning the actual pointer to the formal one; the calling environment loses access. On return, the reverse assignment is made.

5.3 Implementation

There are several points that should be made concerning the implementation of dynamic objects. The first, and probably most troublesome, is that dynamic objects involve dynamic allocation which implies that some form of dynamic memory management must be provided by the compiler or computer system. We no longer have a simple stack allocation system; each object's existence is independent of all others so some form of memory recovery is needed. Another implementation point concerns capabilities. A run-time check of each access is needed. The overhead can be minimized, however, by implementing a capability's rights in a bit vector. An access check then involves examining a bit and allowing access if the bit is set. Since capabilities are only used to reference resources, most capability accesses will already involve the overhead of a procedure call to a resource entry. The extra overhead of capability checking should therefore be small. The last implementation point concerns pointer variables. These objects can be implemented with almost no run-time checking because no rights need to be examined. By having only one pointer to each data object, simultaneous multiple access is avoided without incurring the overhead of some synchronization mechanism. The only question at run-time is whether or not a pointer is empty. By placing an illegal address in any empty pointer, the run-time check is made automatically by existing hardware.

6.0 IO PROCESSING - USING RESOURCES AND PROTECTED VARIABLES

As Hoare has observed [5], the utility of proposed language features cannot be assessed without presenting a convincing set of examples. In this section, we describe five IO processing problems and outline solutions using our language features. Numerous other examples can be found in [9]. First, we use resources to define the interface between program modules and hardware devices. Then we add another resource layer for files to give users a higher level interface to devices and to illustrate general system structure. Two device management problems are then discussed: scheduling of "shared" devices such as disks, and allocation of serial devices such as consoles. Finally we show how protected variables can be used to solve a typical buffer allocation problem.

6.1 Device Interface

One of our main goals has been to develop a single, uniform way to describe all process interaction. Competition for and sharing of physical devices is one of the most basic kinds of interaction. A device has much in common with any abstract data type; it has local storage and it defines operations for accessing the storage. And like shared data types, devices require exclusive access and synchronization. Therefore, we propose that every addressable device (or group of identical devices such as consoles) be represented by a resource. Wirth has recently made a similar proposal [13, 14].

A major complaint about using high level languages for controlling devices is that they can be inefficient compared to machine (assembly) programs. As a result, when high level languages have been used, machine language inserts are often employed to code critical portions. In most cases, machine code is used to interface to machine hardware. We do not reject this approach but feel that it must be used with care. Therefore, we propose that the only place where machine language is used is within resources which interface to devices. In this way, device peculiarities can be handled efficiently but, equally if not more important, the device's interface to the rest of the system is defined by a high level interface.

The general form of a device resource is:

```
device name:  static resource;
              entry operation_1,...,operation_n;

              local data - channel program storage, etc.
              .
              .
              .
              procedure operation_i (parameters);
                    build channel programs          }  machine
                    initiate task                   }  language
                    interrogate completion codes    }
                    end operation_i;                }
              .
              .
              .

           /* initialization */

           end device name;
```

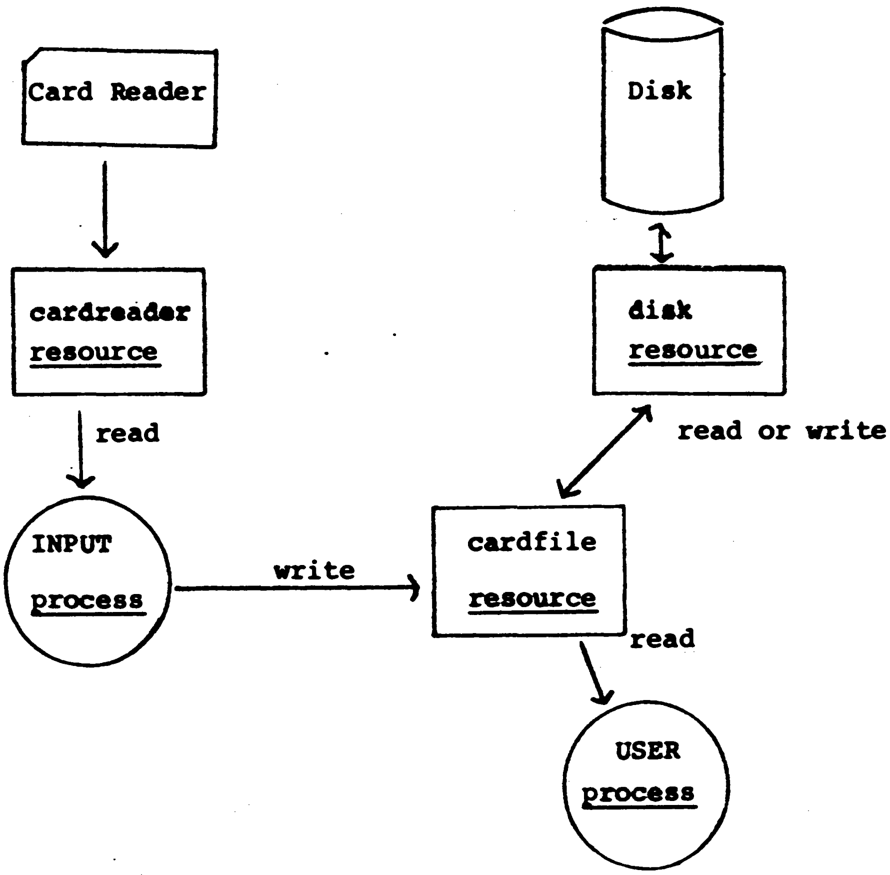
One final point should be made about the role of device resources. We believe that machine dependency occurs in only two places in our language: within the run-time package implementing language features (e.g. processes) and within device resources. It seems appropriate, therefore, for each language implementation to contain a library of pre-defined device resources appropriate for the target machine. Any systems written in our language can then reference, via procedure calls to resources, the machine devices. The same system, if implemented on another machine with the same types of devices, would still function correctly even if the hardware had different characteristics. As long as the name and operations were the same, the resource interface would be the same.

6.2 Files and Virtual Devices

Although a system process may communicate directly with peripheral devices, it is common to provide user (application) processes with a higher level interface implemented by a file system. In order to illustrate general system structure using processes and resources, we consider a simple example of a portion of an input spooler. The system has two processes and three resources as shown in Figure 3. The INPUT process reads cards and writes them into a cardfile resource which in turn stores them on disk. At some later time a USER process reads its input from the cardfile which causes the input to be fetched from disk.

Figure 3

A Simple Input Spooler



Ignoring all details of synchronization and IO buffering, the structure of this system is shown in Figure 4. Cardreader and disk are device resources as discussed in the last section. Each defines procedures to access its associated device. By using the <u>scope</u> phrase, access to cardreader is restricted to the INPUT process and access to the disk is restricted (in this example) to the cardfile resource. Cardfile contains two entries, read and write, used by USER and INPUT. The functions of cardfile are to implement a sequential file organization and to perform IO buffering and synchronization (synchronization is required to prevent a "card" from being read before it has been written).

The INPUT process executes a loop to read physical cards and write card images into the cardfile. The USER process gets its input by reading the cardfile. The remaining code initializes and activates the two processes.

This example is greatly simplified and not representative of a real system. Its intent is merely to illustrate the general structure of a language defined system and give an example of the use of <u>scope</u> restriction. One step toward realism would be to add multiple cardfiles. The INPUT process would then create a new cardfile, fill it, and then pass it on to another process (or resource) such as a job manager. Once a USER process is scheduled, it is given access to the appropriate cardfile instance. This can be achieved in our language in the following manner. First cardfile is declared as a <u>dynamic</u> <u>resource</u> <u>type</u>. INPUT

Figure 4

Outline of Input Spooler Code

```
system: begin
      cardreader: static resource;
                  entry read;
                     :
                  end cardreader
                  scope INPUT;
            disk: static resource;
                  entry read,write;
                     :
                  end disk
                  scope cardfile;
        cardfile: static resource;
                  entry read,write;
                     :
                  end cardfile
                  scope INPUT,USER;
           INPUT: process;
                      repeat
                          cardreader.read (b);
                          cardfile.write (b);
                      until  end of file;
                  end INPUT;
            USER: process;
                     /* fetch next card by calling: */
                        cardfile.read (b);

                  end USER;
        /* start processes */
        INPUT.setstate (   );
        USER.setstate (   );
        INPUT.activate;
        USER.activate;
      end system;
```

creates new instances of the type for each new card stream and passes them (via capability passing) to the job scheduler. USER then references a card file by declaring a capability and acquiring the actual file instance from the scheduler. The changes to cardfile and INPUT are:

```
type cardfile = dynamic resource
                entry read, write;


                end cardfile
                scope INPUT, system;

type cardfile_reference = capability for cardfile
                scope USER;

INPUT:  process;
         c :  capability for cardfile;
         c := cardfile.create;
         repeat

              cardreader.read(b);
              c.write(b);
         until end of file;
         send c to job scheduler
        end INPUT;
```

The USER process references a card file by declaring a variable cf of type cardfile_reference. Note how the scope phrase has been used to let INPUT have access to the definition of cardfile (and hence to the create operation) but only let the USER have access to cardfile_reference. In this way, USER can acquire capabilities for cardfiles but cannot create them himself. This type of access control is common in operating systems.

6.3 Device Scheduling

We now turn to a concrete example of device scheduling, in particular scheduling operations on a moving head disk. The problem involves reordering disk operations in order to improve device utilization. The algorithm we will use, for demonstration purposes, is the "elevator algorithm" in Hoare [5]. The basic idea is to move the disk head in complete sweeps of the cyclinders servicing all requests for the same cylinder at the same time.

Our solution to this problem employs two resources: disk and userdisk. Disk is a machine interface resource as described in Section 6.1. Userdisk is the resource available to users. It sequences the requests, and then calls disk to carry out the request. The code for both resources is outlined in figure 5.

The key to our solution is that disk scheduling and access can execute concurrently. Since the disk resource procedures are listed as <u>parallel</u> procedures, a process which calls disk loses exclusive control of userdisk. This permits other processes to enter userdisk and queue up for eventual access. When the disk operation completes (i.e. returns to userdisk) the next process is selected for continuation. Notice that without the <u>parallel</u> feature, this organization would imply FIFO scheduling.

Hoare's solution to this problem is weaker than ours because monitors cannot do the scheduling and access in parallel. As a result, his users had to take three steps: (1) request device access, (2) make the access, and (3) release the device. There is no mechanism for enforcing this sequence and problems arise if any user forgets or reorders some steps.

Figure 5

Disk Scheduler

```
disk: static resource;
      entry read, write;

      /* machine interface resource */
      /* read and write called only from userdisk */
      end disk
      scope userdisk;

userdisk: static resource;
          entry read, write;
          parallel disk.read, disk.write; /* to exit
                   userdisk when IO is begun */

          request queue /* tasks waiting for disk */
          condition variables /* for waiting processes */
          executing: boolean; /* true if disk in use */

          procedure read (IO parameters);
                 if ¬executing then begin
                            c := next free condition variable
                            save task info in request queue
                            c.wait;
                            remove task info from request queue
                            end;
                 executing := true;
                 disk.read (IO parameters);
                    /* control of userdisk surrendered during call */
                 if request queue is empty
                      then executing := false;
                      else begin
                           find next task on request queue
                           c := condition task is waiting on;
                           c.signal;
                           end;
          end read;
```

```
procedure write (IO parameters);
        /* similar to read -
             calls disk.write        */
        end write;

/* initialize data */
 executing := false;
 request queue empty

end userdisk;
```

6.4 Device Reservation

With random access devices such as disks, many processes can share the device and interleave their read and write requests. For serial devices however, the device must be reserved if more than one record (line) is to be read or written at one time. To allow processes to reserve a device we need a device allocator with two operations: request and release. Implementing the allocator is straightforward. What is not so easy, however, is enforcing the policy that a serial device must be requested by and allocated to a process before the process can use it. One possible approach is to nest two resources; the first does allocation and then calls the other to do access. Once a device has been allocated, the allocator blocks requests from other processes. The trouble with this approach is that every access operation incurs the overhead of going through the allocator. A second approach is to separate the allocator from the device, give all processes access to both and trust that they will request the device before using it. Unfortunately, not all processes (or even most) are trustworthy.

With our language, we can use dynamic resources and capabilities to insure that a device is requested before it is used. Our solution is outlined in Figure 6. The device <u>resource</u> is now declared as a <u>type</u> of <u>dynamic</u> <u>resource</u>. This allows a capability to be used to provide secure access, even if only one instance of the device is generated. The typed device declaration is only accessible to the allocator; this insures that only the allocator can create instances of the device.

Figure 6

Device Allocation

```
type device = dynamic resource;
              entry read, write;
                 :
              end device
              scope allocator;

type device_capability = capability for device;
                         /* scope is global */

allocator: static resource;
           entry request, release;
           available : boolean;
           waiting : condition;
           dc : device_capability;

           procedure request (d: device_capability);
               if ¬ available then waiting.wait;
               available := false;
               d := dc {read,write} ;
                    /* give access to device but not
                       ability to copy  */
               end request;

           procedure release (d: device_capability);
               /* d is returned by calling process */
               d := null;
               available := true;
               waiting.signal; /* awaken someone
                                  if possible  */
               end release;

           /* initialize */
           available := true;
           dc := device.create; /* actually generate
                                   device resource */
           end allocator;
```

Access to the device is through device capabilities which can be declared in any block. The allocator resource is the key part of our solution. It implements two operations, request and release, which enable a process to acquire access to the device and give it up, respectively. When a process calls request it first waits for the device to be available. Once it is, parameter d is assigned the device capability dc, but only with read and write access. This gives the user the ability to access the device but prevents him from copying his one capability. Once the user is through accessing the device, he calls allocator.release(d) where d is the device capability previously granted. The assignment of null to d prevents the capability from returning to the user at the end of the call. The device can then be allocated to another process. The only weakness is the inability to force a user to return his rights.

6.5 Buffer Allocation and Access

As our final example, we consider one further portion of a typical IO system: buffer allocation. IO buffers are often stored in a pool used by multiple processes. When a process requires storage for another data record, it gets an empty buffer from the pool and fills it with information. Once a full buffer is no longer needed, it is returned to the empty pool. Buffers are also used to, for example, implement a message passing system.

Our problem here is to implement a buffer allocator analagous to the device allocator in the previous example. The buffer allocator has two entries, request and release, which are called to fetch and return an empty buffer, respectively. As with device allocation, we want to insure that a buffer can only be accessed after it is allocated. But buffers are not like devices; they are data records instead of resources and they are only accessed by one process at a time. Therefore, we implement buffers by protected variables and access them via pointers. The request procedure of the allocator consequently returns a pointer to the allocated buffer. A call to allocator.release gives back a buffer pointer. The creation and control of buffers is very similar to that used for devices. The complete code for the allocator is shown in Figure 7. The main advantage of using protected variables and pointers instead of resources is that each buffer can be directly and therefore efficiently accessed. No access control or data integrity is sacrificed, however, because the manipulation of pointers is carefully controlled.

7.0 CONCLUSION

In the introduction we enumerated four goals guiding the work presented here: expressiveness, data integrity, security, and program verification. Our focus in this paper has been the first, namely expressing the structure and control of systems of interacting processes. In order to describe processes, we introduced a process type. To specify process interaction, we introduced resources and protected variables. As the examples have attempted to show, our language features lead to simpler problem solutions than is possible

Figure 7

Buffer Allocation and Access

```
type buffer = protected array 1..size of bits
              scope allocator;

type bufptr = pointer to buffer;

allocator: static resource
          entry request, release;

          bufs: array 1..N of bufptr;
                /* stack of empty buffers */
          waiting: condition;
          top: integer;

          procedure request (p: bufptr);
             if top = 0 then waiting.wait;
             p: = bufs(top);
             top: = top-1;
             end request;

          procedure release (p: bufptr);
             top: = top+1;
             bufs(top): = p;
             waiting.signal;
             end release;
    top = N;
    do i = 1 to N;  /* create buffer instances */
        bufs(i): = buffer.create; end;
    end allocator;
```

with existing languages. One specific application of resources is to describe the interface to machine devices. This is, we feel, a feasible and valuable way to isolate machine dependent components from other modules in a system. We have also introduced the <u>scope</u> phrase, <u>capabilities</u>, and <u>pointers</u> to provide flexible access control.

Elsewhere we analyze the relation of our language features to the other three goals [9]. We briefly summarize our results here. Throughout this paper we have respeatedly referred to the problem of data integrity. The semantic rules governing scope of access and the use of capabilities and pointers are specifically concerned with insuring that only one process at a time can ever access any variable. As long as programs do not escape from our language and violate its semantics, we have shown that data integrity is insured.

As we have described the problem, security has two aspects: data safety and absence of deadlock. The safety problem [4] is concerned with deciding what blocks can access each program variable. An exact solution requires knowledge of the execution flow of a program and is therefore unattainable in general. In [9], however, we show how to determine the potential access of each block at compile time. By potential access we mean those variables a program would access if it took every possible execution path.

As with the safety problem, an exact determination of whether or not a program is free of mutual blocking is in general undecidable. It is possible, however, to model any program using our language features, monitor its execution, and detect a deadlock if one occurs.

No language can both make it impossible to deadlock programs and allow programs to control scheduling (via waits and signals in resources). The best that can be hoped is that deadlock can be detected and guidelines for its avoidance can be enumerated.

The final goal, program verification, is the one we are furthest from achieving. The parallel phrase adds complexity to any proof because it adds parallel activity. Much work remains to be done here. Our consolation is that the same is true for all non-trivial languages.

No language proposal can or will be accepted until it has stood the test of extensive use. A necessary prelude is its implementation. This is one of our anticipated future tasks.

8.0 BIBLIOGRAPHY

1. Brinch Hansen,P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering Vol. SE-1, No. 2, June 1975, 199-207.

2. Brinch Hansen,P. The Solo operating system: a Concurrent Pascal program. Software Practices and Experience 6 (1976), 141-149.

3. Courtois,P.J.,Heymans,F., and Parnas,D.L. Concurrent control with readers and writers. Comm. ACM 14,10 (Oct. 1971), 667-668.

4. Harrison,M.A.,Ruzzo,W.L., and Ullman,J.D. Protection in operating systems. Comm. ACM 19,8 (August 1976),461-471.

5. Hoare,C.A.R. Monitors: an operating system structuring concept. Comm. ACM 17,10 (October 1974), 549-557.

6. Holt,R.C. Some deadlock properties of computer systems. Computing Surveys Vol. 4, No. 3, (September 1972), 179-196.

7. Howard, J.H. Proving monitors. Comm. ACM 19,5 (May 1976), 273-285.

8. Lampson,B.W., and Sturgis,H.E. Reflections on operating systems design. Comm. ACM 19,5 (May 1976), 273-279.

9. McGraw,J.R. Representing process interaction in a programming language. Ph.D. Thesis, Department of Computer Science, Cornell University (in preparation).

10. Shaw, A.C. et. al. A multiprogramming nucleus with dynamic resource facilities. Software Practices and Experience Vol. 5, 1975, 245-267.

11. Silberschatz,A.,Kieburtz,R.B., and Bernstein,A. Extending Concurrent Pascal to allow dynamic resource management. Technical Report #53, Department of Computer Science, State University of New York at Stony Brook, June 1976.

12. Wirth,N. The programming language Pascal. Acta Informatica, Vol. 1, 1971, 35-63.

13. Wirth,N. Modula: A language for modular multiprogramming. Technical Report #18, Institut für Informatik, ETH, March 1976.

14. Wirth,N. The use of Modula and Design and implementation of Modula. Technical Report #19, Institut für Informatik, ETH, June 1976.

15. Wulf, W.A. el. al. Hydra: the kernal of a multiprocessor operating system. Comm. ACM 17,6 (June 1974), 337-345.