# The Programming Language Concurrent Pascal

PER BRINCH HANSEN

*Abstract-The* paper describes a new programming language for structured programming of computer operating systems. It extends the sequential programming language Pascal with concurrent programming tools called processes and monitors. Section I explains these concepts informally by means of pictures illustrating a hierarchical design of a simple spooling system. Section II uses the same example to introduce the language notation. The main contribution of Concurrent Pascal is to extend the monitor concept with an explicit hierarchy Of access' rights to shared data structures that can be stated in the program text and checked by a compiler.

*Index* Terms-Abstract data types, access rights, classes, concurrent processes, concurrent programming languages, hierarchical operating systems, monitors, scheduling, structured multiprogramming.

## 1. THE PURPOSE OF CONCURRENT PASCAL

### A. Background

SINCE 1972 I have been working on a new programming language for structured programming of computer operating systems. This language is called Concurrent Pascal. It extends the sequential programming language Pascal with concurrent programming tools called processes and monitors [1J-[3]'

This is an informal description of Concurrent Pascal. It uses examples, pictures, and words.to bring out the creative aspects of new programming concepts without getting into their finer details. I plan to define these concepts precisely and introduce a notation for them in later papers. This form of presentation may be imprecise from a formal point of view, but is perhaps more effective from a human point of view.

### B. Processes

We will study concurrent processes inside an operating system and look at one small problem only: how can large amounts of data be transmitted from one process to another by means of a buffer stored on a disk?

Fig. 1 shows this little system and its three components: a process that produces data, a process that consumes data, and a disk buffer that connects them.

The circles are *system components* and the arrows are the *access rights* of these components. They show that both processes can use the buffer (but they do not show that data flows from the producer to the consumer). This kind of picture is an *access graph.*

Manuscript received February 1, 1975. This project is supported by the National Science Foundation under Grant DCR74-17331.

The author is with the Department of InformatIOn SCIence, California Institute of Technology, Pasadena, Calif. 91125.

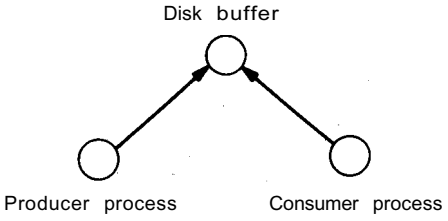


Fig. 1. Process communication.

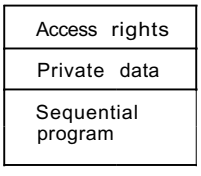


Fig. 2. Process.

The next picture shows a process component in more detail (Fig. 2).

A *process* consists of a *private data* structure and a *sequential program* that can operate on the data. One process cannot operate on the private data of another process. But concurrent processes can share certain data structures (such as a disk buffer). The *access rights* of a process mention the shared data it can operate on.

### C. Monitors

A disk buffer is a data structure shared by two concurrent processes. The details of how such a buffer is constructed are irrelevant to its users. All the processes need to know is that they can *send* and *receive* data through it. If they try to operate on the buffer in any other way it is probably either a programming mistake or an example of tricky programming. In both cases, one would like a compiler to detect such misuse of a shared data structure.

To make this possible, we must introduce a language construct that will enable a programmer to tell a compiler how'a shared data structure can be used by processes. This kind of system component is called a monitor. A monitor can synchronize concurrent processes and transmit data between them. It can also control the order in which competing processes use shared, physical resources. Fig. 3 shows a monitor in detail.

A *monitor* defines a *shared data* structure and all the operations processes can perform on it. These synchronizing operations are called *monitor procedures.* A monitor also defines an *initial operation* that will be executed when its data structure is created.

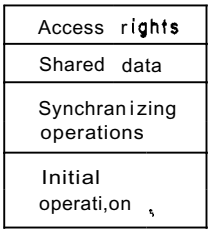


Fig. 3. Monitor.

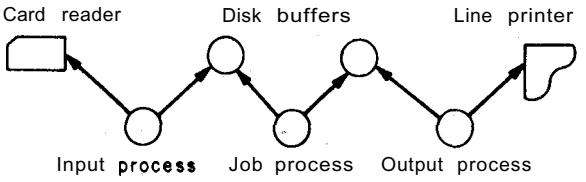


Fig. 4. Spooling system.

We can define a *disk buffer* as a monitor. Within this monitor there will be shared variables that define the location and length of the buffer on the disk. There will also be two monitor procedures, *send* and *receive.* The initial operation will make sure that the buffer starts as an empty one.

Processes cannot operate directly on shared data. They can only call monitor procedures that have access to shared data. A monitor procedure is executed as part of a calling process (just like any other procedure).

If concurrent processes simultaneously call monitor procedures that operate on the same shared data these procedures must be executed strictly one at a time. Otherwise, the results of monitor calls will be unpredictable. This means that the machine must be able to delay processes for short periods of time until it is their turn to execute monitor procedures. We will not be concerned about how this is done, but will just notice that a monitor procedure has *exclusive access* to shared data while it is being executed.

So the (virtual) machine on which concurrent programs run will handle *short-term scheduling* of simultaneous monitor calls. But the programmer must also be able to delay processes for longer periods of time if their requests for data and other resources cannot be satisfied immediately. If, for example, a process tries to receive data from an empty disk buffer it must be delayed until another process sends more data.

Concurrent Pascal includes a simple data type, called a *queue,* that can be used by monitor procedures to control *medium-term scheduling* of processes. A monitor can either *delay* a calling process in a queue or *continue* another process that is waiting in a queue. It is not important here to understand how these queues work except for the following essential rule: a process only has exclusive access to shared data as long as it continues to execute statements within a monitor procedure. As soon as a process is delayed in a queue it loses its exclusive access until another process calls the same monitor and wakes it up again. (Without this rule, it would be impossible for other processes to enter a monitor and let waiting processes continue their execution.)

Although the disk buffer example does not show this yet, monitor procedures should also be able to call proceduresdefined within other monitors. Otherwise, the language will not be very useful for hierarchical design. In the case of a disk buffer, one of these other monitors could perhaps define simple input/output operations on the disk. So a monitor can also have *access rights* to other system components (see Fig. 3).

### D. System Design

A process executes a sequential program-it is an active component. A monitor is just a collection of procedures that do nothing until they are called by processes-it is a passive component. But there are strong similarities between a process and a monitor: both define a data structure (private or shared) and the meaningful operations on it. The main difference between processes and monitors is the way they are scheduled for execution.

It seems natural therefore to regard processes and monitors as *abstract data types* defined in terms of the operations one can perform on them. If a compiler can check that these operations are the only ones carried out on the data structures, then we may be able to build very reliable, concurrent programs in which *controlled access* to data and physical resources is guaranteed before these programs are put into operation. We have then to some extent solved the *resource protection* problem in the cheapest possible manner (without hardware mechanisms and run time overhead).

So we will define processes and monitors as data types and make it possible to use several instances of the same component type in a system. We can, for example, use two disk buffers to build a *spooling system* with an input process, a job process, and an output process (Fig. 4). I will distinguish between definitions and instances of components by calling them *system types* and *system components.* Access graphs (such as Fig. 4) will always show system components (not system types).

*Peripheral devices* are considered to be monitors implemented in hardware. They can only be accessed by a single procedure *io* that delays the calling process until an input/output operation is completed. Interrupts are handled by the virtual machine on which processes run.

To make the programming language useful for stepwise system design it should permit the division of a system type, such as a disk buffer, into smaller system types. One of these other system types should give a disk buffer access to the disk. We will call this system type a *virtual disk.* It gives a disk buffer the illusion that it has its own private disk. A virtual disk hides the details of disk input/output from the rest of the system and makes the disk look like a data structure (an array of disk pages). The only operations on this data structure are *read* and *write* a page.

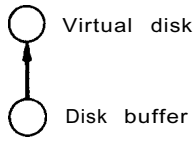


Fig. 5. Buffer refinement.

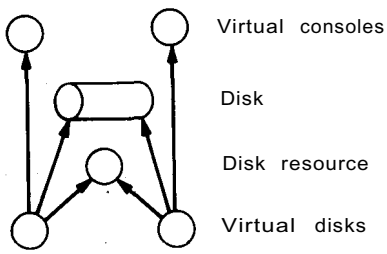


Fig. 6. Decomposition of virtual disks.

Each virtual disk is only used by a single disk buffer (Fig. 5). A system component that cannot be called simultaneously by several other components will be called a *class.* A class defines a data structure and the possible operations on it (just like a monitor). The exclusive access of class procedures to class variables can be guaranteed completely at compile time. The virtual machine does not have to schedule simultaneous calls of class procedures at run time, because such calls cannot occur. This makes class calls considerably faster than monitor calls.

The spooling system includes two virtual disks but only one real disk. So we need a single *disk resource* monitor to control the order in which competing processes use the disk (Fig. 6). This monitor defines two procedures, *request* and *release* access, to be called by a virtual disk before and after each disk transfer.

It would seem simpler to replace the virtual disks and the disk resource by a single monitor that has exclusive access to the disk and does the input/output. This would certainly guarantee that processes use the disk one at a time. But this would be done according to the built-in short-term scheduling policy of monitor calls.

Now to make a virtual machine efficient, one must use a very simple short-term scheduling rule (such as first come, first served) [2]. If the disk has a moving access head this is about the worst possible algorithm one can use for disk transfers. It is vital that the language make it possible for the programmer to write a medium-term scheduling algorithm that will minimize disk head movements [3]. The data type *queue* mentioned earlier makes it possible to implement arbitrary scheduling rules within a monitor.

The difficulty is that while a monitor is performing an input/output operation it is impossible for other processes to enter the same monitor and join the disk queue. They will automatically be delayed by the short-term scheduler and only allowed to enter the monitor one at a time after each disk transfer. This will, of course, make the attempt

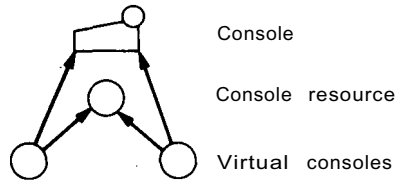


Fig. 7. Decomposition of virtual consoles.

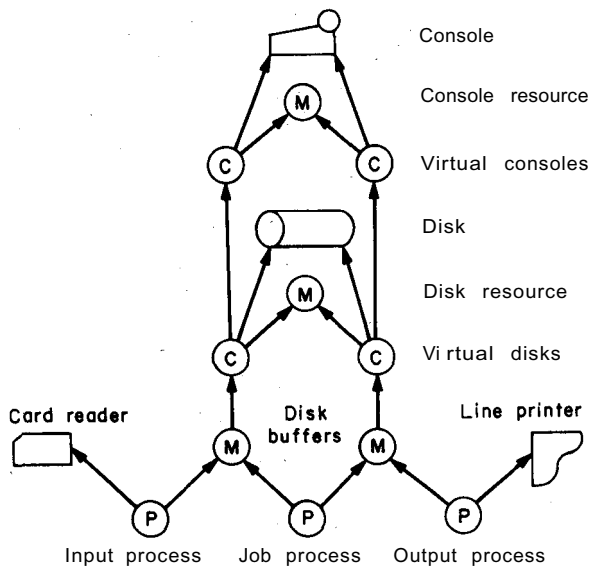


Fig. 8. Hierarchical system structure.

to control disk scheduling within the monitor illusory. To give the programmer complete control of disk scheduling, processes should be able to enter th'e disk queue during disk transfers. Since *arrival* and *service* in the disk queueing system potentially are simultaneous operations they must be handled by different system components, as shown in Fig. 6.

If the disk fails persistently during input/output, this should be reported on an operator's console. Fig. 6 shows two instances of a class type, called a *virtual console.* They give the virtual disks the illusion that they have their own private consoles.

The virtual consoles get exclusive access to a single, real console by calling a *console resource* monitor (Fig. 7). Notice that we now have a standard technique for dealing with virtual devices.

If we put all these system components together, we get a complete picture of a simple spooling system (Fig. 8). Classes, monitors, and processes are marked *C, M,* and *P.*

### *E. Scope Rules*

Some years ago I was part of a team that built a multiprogramming system in which processes can appear and disappear dynamically [4]. In practice, this system was used mostly to set up a fixed configuration of processes, Dynamic process deletion will certainly complicate the semantics and implementation of a programming language considerably. And since it appears to be unnecessary for

a large class of real-time applications, it seems wise to exclude it altogether. So an operating system written in Concurrent Pascal will consist of a fixed number of processes, monitors, and classes. These components and their data' structures will exist forever after system initialization. An operating system can, however, be extended by recompilation. It remains to be seen whether this restriction will simplify or complicate operating system design. But the poor quality of most existing operating systems clearly demonstrates an urgent need for simpler approaches.

In existing programming languages the data structures of processes, monitors, and classes would be called "global data." This term would be misleading in Concurrent Pascal where each data structure can be accessed by a single component only. It seems more appropriate to call them *permanent data structures.*

I have argued elsewhere that the most dangerous aspect of concurrent programming is the possibility of *time-dependent programming errors* that are impossible to locate by program testing ("lurking bugs") [2J, [5J, [6]. If we are going to depend on real-time programming systems in our daily lives, we must be able to find such obscure errors before the systems are put into operation.

Fortunately, a compiler can detect many of these errors if processes and monitors are represented by a structured notation in a high-level programming language. In addition, we must exclude low-level machine features (registers, addresses, and interrupts) from the language and let a virtual machine control them. If we want real-time systems to be highly reliable, we must stop programming them in assembly language. (The use of hardware protection mechanisms is merely an expensive, inadequate way of making arbitrary machine language programs behave almost as predictably as compiled programs.)

A Concurrent Pascal compiler will check that the private data of a process only are accessed by that process. It will also check that the data structure of a class or monitor only is accessed by its procedures.

Fig. 8 shows that *access rights* within an operating system normally are not tree structured. Instead they form a directed graph. This partly explains why the traditional scope rules of block-structured languages are inconvenient for concurrent programming (and for sequential programming as well). In Concurrent Pascal one can state the access rights of components in the program text and have them checked by a compiler.

Since the execution of a monitor procedure will delay the execution of further calls of the same monitor, we must prevent a monitor from calling itself recursively. Otherwise, processes can become *deadlocked.* So the compiler will check that the access rights of system components are hierarchically ordered (or, if you like, that there are no cycles in the access graph).

The *hierarchical ordering* of system components has vital consequences for system design and testing [7].

A hierarchical operating system will be tested component by component, bottom up (but could, of course, be conceived top down or by iteration). When an incomplete operating system has been shown to work correctly (by proof or testing), a compiler can ensure that this part of the system will continue to work correctly when new untested program components are added on top of it. Programming errors within new components cannot cause old components to fail because old components do not call new components, and new components only call old components through well-defined procedures that have already been tested.

(Strictly speaking, a compiler can only check that single monitor calls are made correctly; it cannot check sequences of monitor calls, for example whether a resource is always reserved before it is released. So one can only hope for compile time assurance of *partial correctness.)*

Several other reasons besides program correctness make a hierarchical structure attractive:

1) a hierarchical operating system can be studied in a stepwise manner as a sequence of *abstract machines* simulated by programs [8J;

2) a partial ordering of process interactions permits one to use *mathematical induction* to prove certain overall properties of the system (such as the absence of deadlocks) [2J;

*3) efficient resource utilization* can be achieved by ordering the program components according to the speed of the physical resources they control (with the fastest resources being controlled at the bottom of the system) [8J;

4) a hierarchical system designed according to the previous criteria is often *nearly decomposable* from an analytical point of view. This means that one can develop stochastic models of its dynamic behavior in a stepwise manner [9].

### F. Final Remarks

It seems most natural to represent a hierarchical system structure, such as Fig. 8, by a two-dimensional picture. But when we write a concurrent program we must somehow represent these access rules by linear text. This limitation of written language tends to obscure the simplicity of the original structure. That is why I have tried to explain the purpose of Concurrent Pascal by means of pictures instead of language notation.

The class concept is a restricted form of the class concept of Simula 67 [10]. Dijkstra suggested the idea of monitors [8]. The first structured language notation for monitors was proposed in [2J, and illustrated by examples in [3]. The queue variables needed by monitors for process scheduling were suggested in [5J and modified in [3].

The main contribution of Concurrent Pascal is to extend monitors with explicit access rights that can be checked at compile time. Concurrent Pascal has been implemented at Caltech for the PDP 11/45 computer. Our system uses sequential Pascal as a job control and user programming language.

## II. THE USE OF CONCURRENT PASCAL

### A. Introduction

In Section I the concepts of Concurrent Pascal were explained informally by means of pictures of a hierarchical spooling system. I will now use the same example to introduce the language notation of Concurrent Pascal. The presentation is still informal. I am neither trying to define the language precisely nor to develop a working system. This will be done in other papers. I am just trying to show the flavor of the language.

### B. Processes

We will now program the system components in Fig. 8 one at a time from top to bottom (but we could just as well do it bottom up).

Although we only need one *input process,* we may as well define it as a general system type of which several copies may exist:

```
type inputprocess =
process(buffer: diskbuffer) ;
var block: page;
cycle
  readcards(block);
  buffer.send(block) ;
end
```

An input process has access to a *buffer* of type diskbuffer (to be defined later). The process has a private variable *block* of type page. The data type page is declared elsewhere as an array of characters:

```
type page = array (.1..512.) of char
```

A process type defines a *sequential program-in* this case, an endless cycle that inputs a block from a card reader and sends it through the buffer to another process. We will ignore the details of card reader input.

The *send* operation on the buffer is called as follows (using the block as a parameter) :

```
buffer.send(block)
```

The next component type we will define is a *job process:*

```
type jobprocess =
process(input, output: diskbuffer);
var block: page;
cycle
  input.receive(block) ;
  update(block) ;
  output.send(block) ;
end
```

A job process has access to two disk buffers called *input* and *output.* It receives blocks from one buffer, updates them, and sends them through the other buffer. The details of updating can be ignored here.

Finally, we need an *output process* that can receive data from a disk buffer and output them on a line printer:

```
type outputprocess =
process(buffer: diskbuffer) ;
var block: page;
cycle
  buffer.receive(block) ;
  printlines(block) ;
end
```

The following shows a declaration of the main system components:

```
var bufferl, buffer2: diskbuffer;
  reader: inputprocess;
  master: jobprocess;
  writer: outputprocess;
```

There is an input process, called the *reader,* a job process, called the *master,* and an output process,called the *writer.* Then there are two disk buffers, *buffed* and *buffer2,* that connect them.

Later I will explain how a disk buffer is defined and initialized. If we assume that the disk buffers already have been initialized, we can initialize the input process as follows:

```
init reader(bufferl)
```

The *init* statement allocates space for the *private variables* of the reader process and starts its execution as a sequential process with access to bufferl.

The *access rights* of a process to other system components, such as bufferl, are also called its *parameters.* A process can only be initialized once. After initialization, the parameters and private variables of a process exist forever. They are called *permanent variables.*

The init statement .can be used to start concurrent execution of several processes and define their access rights. As an example, the statement

```
init reader(bufferl), master(buffer1, buffer2),
  writer(buffer2)
```

starts concurrent execution of the reader process (with access to bufferl), the master process (with access to both buffers), and the writer process (with access to buffer2).

A process can only access its own parameters and private variables. The latter are not accessible to other system components. Compare this with the more liberal scope rules of block-structured languages in which a program block can access not only its own parameters and local variables, but also those declared in outer blocks. In Concurrent Pascal, all variables accessible to a system component are declared within its type definition. This access rule and the init statement make it possible for a programmer to state access rights explicitly and have them checked by a compiler. They also make it possible to study a system type as a self-contained program unit.

Although the programming examples do not show this, one can also define constants, data types, and procedures within a process. These objects can only be used within the process type.

### C. Monitors

The *disk buffer* is a monitor type:

```
type diskbuffer =
monitor(consoleaccess, diskaccess: resource;
  base, limit: integer);

var disk: virtualdisk; sender, receiver: queue;
  head, tail, length: integer;

procedure entry send(block: page);
begin
  if length = limit then delay(sender);
  disk.write(base + tail, block);
  tail:= (tail + 1) mod limit;
  length:= length.+ 1;
  continue(receiver);
end;

procedure entry receive(var block: page);
begin
  if length = 0 then delay(receiver);
  disk.read(base + head, block);
  head:= (head + 1) mod limit;
  length:= length - 1;
  continue(sender);
end;

begin "initial statement"
  init disk(consoleaccess, diskaccess);
  head:= 0; tail:= 0; length:= 0;
end
```

A disk buffer has access to two other components, *consoleaccess* and *diskaccess,* of type resource (to be defined later). It also has access to. two integer constants defining the *base* address and *limit* of the buffer on the disk.

The monitor declares a set of *shared variables:* the *disk* is declared as a variable of type virtualdisk. Two variables of type queue are used to delay the *sender* and *receiver* processes until the buffer becomes nonfull and nonempty. Three integers define the relative addresses of the *head* and *tail* elements of the buffer and its current *length.*

The monitor defines two *monitor procedures,* send and receive. They are marked with the word *entry* to distinguish them from local procedures used within the monitor (there are none of these in this example).

*Receive* returns a page to the calling process. If the buffer is empty, the calling process is *delayed* in the receiver queue until another process sends a page through the buffer. The receive procedure will then read and remove a page from the head of the disk buffer by calling a *read* operation defined within the virtual disk type:

```
disk.read(base + head, block)
```

Finally, the receive procedure will *continue* the execution of a sending process (if the la.tter is waiting in the sender queue).

*Send* is similar to receive.

The queuing mechanism will be explained in detail in the next section.

The *initial statement* of a disk buffer initializes its virtual disk with access to the console and disk resources. It also sets the buffer length to zero. (Notice, that a disk buffer does not use its access rights to the console and disk, but only passes them on to a virtual disk declared within it.)

The following shows a declaration of two system components of type resource and two integers defining the base and limit of a disk buffer:

```
var consoleaccess, diskaccess: resource;
  base, limit: integer;
  buffer: diskbuffer;
```

If we assume that these variables already have been initialized, we can initialize a disk buffer as follows:

```
init buffer(consoleaccess, diskaccess, base, limit)
```

The *init* statement allocates storage for the parameters and shared variables of the disk buffer and executes its initial statement.

A monitor can only be initialized once. *Mter* initialization, the parameters and shared variables of a monitor exist forever. They are called *permanent variables.* The parameters and local variables of a monitor procedure, however, exist only while it is being executed. They are called *temporary variables.*

A monitor procedure can only access its own temporary and permanent variables. These variables are not accessible to other system components. Other components can, however, call procedure entries within a monitor. While a monitor procedure is being executed, it has *exclusive access* to the permanent variables of the monitor. If concurrent processes try to call procedures within the same monitor simultaneously, these procedures will be executed strictly one at a time.

Only monitors and constants can be permanent parameters of processes and monitors. This rule ensures that processes only communicate by means of monitors.

It is possible to define constants, data types, and local procedures within monitors (and processes). The local procedures of a system type can only be called within the system type. To prevent *deadlock* of monitor calls and ensure that access rights are hierarchical the following rules are enforced: a procedure must be declared before it can be called; procedure definitions cannot be nested and cannot call themselves; a system type cannot call its own procedure entries.

The absence of recursion makes it possible for a compiler to determine the store requirements of all system components. This and the use of permanent components make it possible to use *fixed store allocation* on a computer that does not support paging.

Since system components are permanent they inust be declared as permanent variables of other components.

### D. Queues

A monitor procedure can delay a calling process for 'any length of time by executing a *delay* operation on a queue variable. Only one process at a time can wait in a queue. When a calling process is delayed by a monitor procedure' it loses its exclusive access to the monitor variables until another process calls the same monitor and executes a continue operation on the queue in which the process is waiting.

The *continue* operation makes the calling process return from its monitor call. If any process is waiting in the selected queue, it will immediately resume the execution of the monitor procedure that delayed it. Mter being resumed, the process, again has exclusive access to the permanent variables of the monitor.

Other variants of process queues (called "events" and "conditions") are proposed in [3J, [5]. They are multi-process queues that use different (but fixed) scheduling rules. We do not yet know from experience which kind of queue will be the most convenient one for operating system design. A single-process queue. is the simplest tool that gives the programmer complete control of the scheduling of individual processes. Later, I will show how multi-process queues can be built from single-process queues.

A queue must be declared as a permanent variable within a monitor type.

### E. Classes

Every disk buffer has its own virtual disk. A virtual disk is defined as a class type:

```
type virtualdisk =
class (consoleaccess, diskaccess: resource) ;

val' terminal: virtualconsole; peripheral: disk;

procedure entry read (pageno: integer; val' block: page) ;
val' error: boolean;
begin
  repeat
    diskaccess.request;
    peripheral.read (pageno, block, error) ;
    diskaccess.release;
    if error then terminal.write ('disk failure') ;
  until not error;
end;

procedure entry write (pageno: integer; block: page) ;
begin "similar to read" end;

begin "initial statement"
  init terminal (consoleaccess), peripheral;
end
```

A virtual disk has access to a console resource and a disk resource. Its permanent variables define a virtual console and a disk. A process can access its virtual disk by means of *read* and *write* procedures. These procedure entries *request* and *release* exclUSIve access to the real disk before and after each block transfer. If the real disk fails, the virtual disk calls its virtual console to report the error.

The *initial statement* of a virtual disk initializes its virtual console and the real disk.

Section H.C shows an example of how a virtual disk is declared and initialized (within a disk buffer).

A class can only be initialized once. Mter initialization, its parameters and private variables exist forever. A class procedure can only access its own temporary and permanent variables. These cannot be accessed by other components.

A class is a system component that cannot be called simUltaneously by several other components. This is guaranteed by the following rule: a class must be declared as a permanent variable within a system type; a class can be passed as a permanent parameter to another class (but not to a process or monitor). So a chain of nested class calls can only be started by a single process or monitor. Consequently, it is not necessary to schedule simultaneous class calls at run time-they cannot occur.

### F. Input/Output

The real *disk* is controlled by a class

```
type disk = class
```

with two procedure entries

```
read (pageno, block, error)
write (pageno, block, error)
```

The class uses a standard procedure

```
io (block, param, device)
```

to transfer a block to or from the disk device. The io parameter is a record

```
var param: record
             operation: iooperation;
             result: ioresult;
             pageno: integer
           end
```

that defines an input/output operation, its result, and a page number on the disk. The calling process is delayed until an io operation has been completed.

A *virtual console* is also defined as a class

```
type virtualconsole =
class (access: resource) ;
val' terminal: console;
```

It can be accessed by read and write operations that are similar to each other:

```
procedure entry read (val' text: line) ;
begin
  access.request;
  terminal.read (text) ;
  access.release;
end
```

The real *console* is controlled by a class that is similar to the disk class.

### G. Multiprocess Scheduling

Access to the console and disk is controlled by two monitors of type *resource.* To simplify the presentation, I will assume that competing processes are served in first-come, first-served order. (A much better disk scheduling algorithm is defined in [3J. It can be programmed in Concurrent Pascal as well, but involves more details than the present one.)

We will define a multiprocess queue as an array of single-process queues

```
type multiqueue = array (.0..qlength-1.) of queue
```

where qlength is an upper bound on the number of concurrent processes in the system.

A first-come, first-served scheduler is now straightforward to program:

```
type resource =
monitor

var free: Boolean; q: multiqueue;
  head, tail, length: integer;

procedure entry request;
var arrival: integer;
begin
  if free then free:= false else
  begin
    arrival:= tail;
    tail:= (tail + 1) mod qlength;
    length:= length + 1;
    delay(q(.arrival.));
  end;
end;

procedure entry release;
var departure: integer;
begin
  if length = 0 then free:= true else
  begin
    departure:= head;
    head:= (head + 1) mod qlength;
    length:= length - 1;
    continue(q(.departure.));
  end;
end;

begin "initial statement"
  free:= true; length:= 0;
  head:= 0; tail:= 0;
end
```

### H. Initial Process

Finally, we will put all these components together into a concurrent program. A Concurrent Pascal program consists of nested definitions of system types. The outermost system type is an anonymous process, called the initial process. An instance of this process is created during system loading. It initializes the other system components.

The initial process defines system types and instances of them. It executes statements that initialize these system components. In our example, the initial process can be sketched as follows (ignoring the problem of how base addresses and limits of disk buffers are defined) :

```
type
  resource = monitor---end;
  console = class---end;
  virtualconsole =
    class(access: resource);---end;
  disk = class---end;
  virtualdisk =
    class(consoleaccess, diskaccess: resource);---end;
  diskbuffer =
    monitor(consoleaccess, diskaccess: resource;
      base, limit: integer);---end;
  inputprocess =
    process(buffer: diskbuffer);---end;
  jobprocess =
    process(input, output: diskbuffer);---end;
  outputprocess =
    process(buffer: diskbuffer);---end;
var
  consoleaccess, diskaccess: resource;
  bufferl, buffer2: diskbuffer;
  reader: inputprocess;
  master: jobprocess;
  writer: outputprocess;
begin
  init consoleaccess, diskaccess,
    buffer1(consoleaccess, diskaccess, basel, limitl),
    buffer2(consoleaccess, diskaccess, base2, limit2),
    reader(bufferl),
    master(bufferl, buffer2),
    writer(buffer2);
end.
```

When the execution of a process (such as the initial process) terminates, its private variables continue to exist. This is necessary because these variables may have been passed as permanent parameters to other system components.

## ACKNOWLEDGMENT

It is a pleasure to acknowledge the immense value of a continuous exchange of ideas with C. A. R. Hoare on structured multiprogramming. I also thank my students L. Medina and R. Varela for their helpful comments on this paper.

## REFERENCES

[1] N. Wirth, "The programming language Pascal," *Acta Informatica,* vol. 1, no. 1, pp. 35-63, 1971.
[2] P. Brinch Hansen, *Operation System Principles.* Englewood Clift's, N. J.: Prentice-Hall, July 1973.
[3] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.,* vol. 17, pp. 549-557, Oct. 1974.
[4] P. Brinch Hansen, "The nucleus of a multiprogramming

system," *Commun. Ass. Comput. Mach.,* vol. 13, pp. 238-250, Apr. 1970.
[5] ——, "Structured multiprogramming," *Commun. Ass. Comput. Mach.,* vol. 15, pp. 574-578, July 1972.
[6] ——, "Concurrent programming concepts," Ass. *Comput. Mach. Comput. Rev.,* vol. 5, pp. 223-245, Dec. 1974.
[7] ——, "A programming methodology for operating ,system design," in *1974 Proc. IFIP Congr.* Stockholm, Sweden: North-Holland, Aug. 1974, pp. 394-397.
[8] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica,* vol. 1, no. 2, pp. 115-138, 1971.
[9] H. A. Simon, "The architecture,of complexity," in *Proc. Amer. Philosophical Society,* vol. 106, no. 6, 1962, pp. 468-482.
[10] O.-J. Dahl and C. A. R. Hoare, "Hierarchical_program structures," in *Structured Programming,* O.-J. Dahl, E. W. Dijkstra; and C. A. R. Hoare. New York: Academic, 1972.

Per Brinch Hansen was born in Copenhagen, Denmark, oil November 13, 1938. He received the M.S. degree in electronic engineering from the Technical University of Denmark, Copenhagen, in 1963.

Mterwards he joined the Danish computer manufacturer, Regnecentralen, as a systems programmer and designer. In 1967 he became head of the department at Regnecentralen which developed the architecture of the RC 4000 computer and its multiprogramming system. From 1970 to 1972 he visited Carnegie-Mellon University, Pittsburgh, Pa., where he wrote the book *Operating System Principles* (Englewood Cliffs, N. J., Prentice-Hall, July 1973). This book contains the first proposal of the *monitor concept* on which the programming language Concurrent Pascal is based. In 1972 he became Associate Professor of Computer Science at the California Institute of Technology, Pasadena. He has been a consultant to Burroughs Corporation; Control Data Corporation, Jet Propulsion Laboratory, Philips, and Varian Data Machines. His main research interests are computer architecture and programming methodology.

Dr. Brinch Hansen is a member of the Working Group 2.3 on Programming Methodology sponsored by the International Federation for Information Processing.

# On the Problem of Uniform References to Data Structures

CHARLES M. GESCHKE AND JAMES G. MITCHELL

*Abstract-The* cost of a change to a large software system is often primarily a function of the size of the system rather than the complexity of the change. One reason for this is that programs which access some given data structure must operate on it using notations which are determined by its exact representation. Thus, changing how it is implemented may necessitate changes to the programs which access it. This paper develops a programming language notation and semantic interpretations which allow a program to operate on a data object in a manner which is dependent only on its logical or abstract properties and independent of its underlying concrete representation.

*Index* Terms-Abstract data types, compilation, data description, extensible languages, generic functions, Simula 67, sparse arrays, structured values, systems programming language, uniform references.

## INTRODUCTION

IF A programmer needs to represent a table which associates a sum of money with a name for all the checking accounts in some bank branch office, he must first decide what operations are germane to an account table. Adding an amount to someone's account (e.g., increase John Smith's account by $5.03), subtracting an amount from an account (possibly with a check against overdrawing), closing an account (deleting the entry in the table for John Smith), and opening a new account (e.g., making a new entry in the table for Jane Doe with an initial balance of $0.00) are one possible set. Once such a set of *logical* operations on the account table are known, the programmer can make decisions on how to represent one in the computer.

Any specific way of implementing an account table will have to provide storage and storage management for the table itself, as well as *concrete* operations corresponding to the logical operations on an account table. To distingUish between the notion of an account table and the specific way in which it is implemented in the computer, we will refer to the former as an *abstract data structure* (or simply an *abstraction)* and to the latter as a *representation* of it. For instance, representing an account table as two arrays, one containing strings (the names associated with the accounts), and the other containing numbers (the funds in the accounts), would mean that the

Manuscript received February 1, 1975.
The authors are with the Palo Alto Research Center, Xerox Corporation, Palo Alto, Calif. 94304.