PER BRINCH HANSEN

Information Science
California Institute of Technology

July 1975

CONCURRENT PASCAL INTRODUCTION

CONCURRENT PASCAL INTRODUCTION

Per Brinch Hansen

Information Science
California Institute of Technology
Pasadena, California 91125

July 1975

This an informal introduction to Concurrent Pascal, a programming language for structured programming of computer operating systems. It contains three papers entitled

1. The programming language Concurrent Pascal
2. Job control in Concurrent Pascal
3. Universal types in Concurrent Pascal

The development of Concurrent Pascal has been supported by the National Science Foundation under grant number DCR74-17331.

THE PROGRAMMING LANGUAGE CONCURRENT PASCAL

Per Brinch Hansen

Information Science
California Institute of Technology

February 1975

Key Words and Phrases: structured multiprogramming, concurrent programming languages, hierarchical operating systems, concurrent processes, monitors, classes, abstract data types, access rights, scheduling.

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. Part 1 of the paper explains these concepts informally by means of pictures illustrating a hierarchical design of a simple spooling system. Part 2 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.

1. THE PURPOSE OF CONCURRENT PASCAL

1.1. 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 [1, 2, 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.

1.2. 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 ?

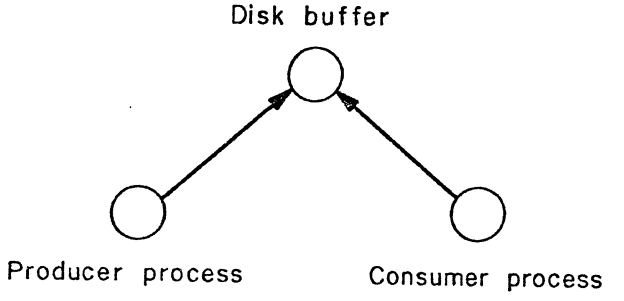

Fig. 1. Process communication

Figure 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.

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

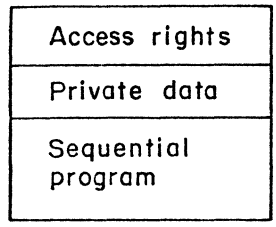

Fig. 2. A process

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.

1.3. 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 case, 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. Figure 3 shows a monitor in detail.

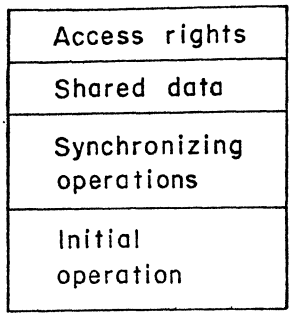

Fig. 3. A monitor

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.

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 procedures defined 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).

1.4. 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).

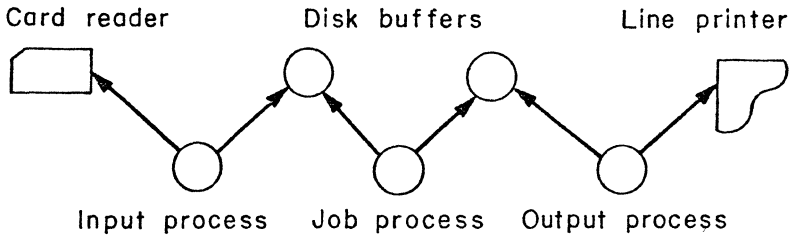

Fig. 4. A spooling system

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.

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.

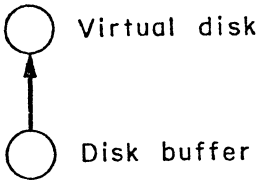

Fig. 5. Buffer refinement

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.

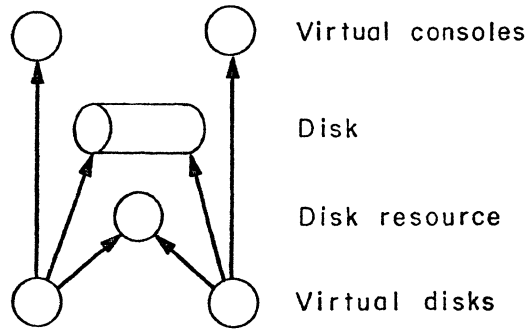

Fig. 6. Decomposition of virtual disks

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 to control disk scheduling within the monitor illusory. To give the programmer complete control of disk scheduling, processes should be able to enter the 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. Figure 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.

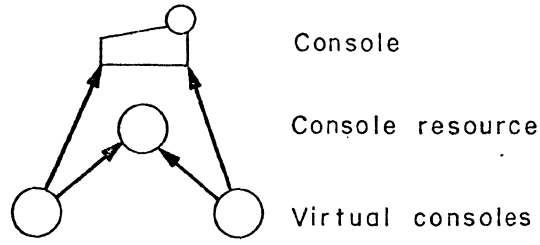

Fig. 7. Decomposition of virtual consoles

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.

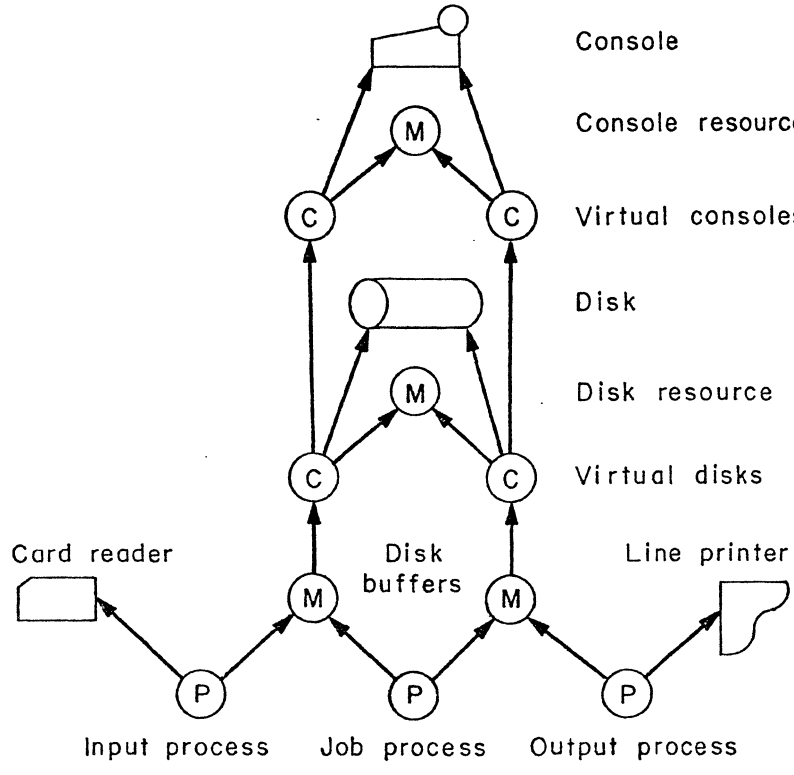

Fig. 8. Hierarchical system structure

1.5. 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") [2, 5, 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.

Figure 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 [8].

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) [2].

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) [8].

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].

1.6. 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 [2] and illustrated by examples in [3]. The queue variables needed by monitors for process scheduling were suggested in [5] 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.

2. THE USE OF CONCURRENT PASCAL

2.1. INTRODUCTION

In Part 1 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.

2.2. 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 buffer1, 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, buffer1 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(buffer1)
```

The init statement allocates space for the private variables of the reader process and starts its execution as a sequential

process with access to buffer1.

The access rights of a process to other system components, such as buffer1, 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(buffer1), master(buffer1, buffer2), writer(buffer2)
```

starts concurrent execution of the reader process (with access to buffer1), 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.

2.3. 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 latter 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. After 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 must be declared as permanent variables of other components.

2.4. 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. After 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 [3, 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.

2.5. CLASSES

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

```
type virtualdisk =
class(consoleaccess, diskaccess: resource);
var terminal: virtualconsole; peripheral: disk;

procedure entry read(pageno: integer; var block: page);
var 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 2.3 shows an example of how a virtual disk is declared and initialized (within a disk buffer).

A class can only be initialized once. After 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.

2.6. 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);
var terminal: console;
```

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

```
procedure entry read(var 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.

2.7. 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 [3]. 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
```

2.8. 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;
  buffer1, buffer2: diskbuffer;
  reader: inputprocess;
  master: jobprocess;
  writer: outputprocess;
begin
  init consoleaccess, diskaccess,
    buffer1(consoleaccess, diskaccess, base1, limit1),
    buffer2(consoleaccess, diskaccess, base2, limit2),
    reader(buffer1),
    master(buffer1, 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.

Acknowledgements

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

The project is now supported by the National Science Foundation under grant number DCR74-17331.

References

1. Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.

2. Brinch Hansen, P. Operating system principles. Prentice-Hall, July 1973.

3. Hoare, C. A. R. Monitors: an operating system structuring concept. Comm. ACM 17, 10 (Oct. 1974), 549-57.

4. Brinch Hansen, P. The nucleus of a multiprogramming system. Comm. ACM 13, 4 (Apr. 1970), 238-50.

5. Brinch Hansen, P. Structured multiprogramming. Comm. ACM 15, 7 (July 1972), 574-78.

6. Brinch Hansen, P. Concurrent programming concepts. ACM Computing Reviews 5, 4 (Dec. 1974), 223-45.

7. Brinch Hansen, P. A programming methodology for operating system design. Proc. IFIP 74 Congress. (Aug. 1974), 394-97.

8. Dijkstra, E. W. Hierarchical ordering of sequential processes. Acta Informatica 1, 2 (1971), 115-38.

9. Simon, H. A. The architecture of complexity. Proc. American Philosophical Society 106, 6 (1962), 468-82.

10. Dahl, O.-J., and Hoare, C. A. R. Hierarchical program structures. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Structured programming. Academic Press, 1972.

JOB CONTROL IN CONCURRENT PASCAL

Per Brinch Hansen

Information Science
California Institute of Technology

March 1975

Abstract

This is the second of two papers that describe 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 processes, monitors, and classes. The first paper explains the language informally. It is helpful (but not essential) to read that paper before this one. The present paper describes how an operating system written in Concurrent Pascal can start and preempt user programs written in sequential Pascal. It also explains how sequential Pascal can be used as a job control language and how sequential programs interact with an operating system.

Key Words and Phrases: Concurrent Pascal, operating systems, job control language, program execution, system interface, program preemption.

CR Categories: 4.2, 4.3

1. INTRODUCTION

This is the second of two papers that describe a new programming language for structured programming of computer operating systems. This language is called Concurrent Pascal. It extends the sequential programming language Pascal [1] with concurrent processes, monitors, and classes. 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.

Concurrent Pascal is explained informally by means of pictures and examples in [2]. The present paper describes how an operating system written in Concurrent Pascal can start and preempt user programs written in sequential Pascal. It also explains how these programs can call procedures defined within the operating system.

The discussion is informal and sketchy. It is helpful (but not essential) to read the previous paper on Concurrent Pascal [2]. A precise definition of the language and a complete description of one or more operating systems written in it will be given in future papers.

2. PROGRAM EXECUTION

An operating system written in Concurrent Pascal will consist of a fixed number of processes, monitors, and classes. It is assumed that both operating systems and user programs are written in high-level programming languages.

I see no difficulty in building a system that can compile and execute user programs written in a variety of different programming languages (by making certain common assumptions about the code generated by the compilers). However, since our project is a research effort, all user programs will be written in a single language (sequential Pascal).

We will assume that an operating system has access to a library of compiled user programs and discuss how these programs can be loaded in the internal computer store and executed.

It is tempting to try to make program loading an elementary operation in the system design language. This would make it possible for a compiler and its run-time environment to check that an executable program is loaded (and not an undefined collection of bits). To do this, the operating system code generated by a compiler must, of course, make many assumptions about the details of disk access, program files, and directories. But if the language makes rigid assumptions about one of the most central operating system components - a file system - it will obviously not be a very useful tool for the design of a variety of different operating systems.

So the language facilities for program loading and execution must be very simple and flexible. This will make them potentially dangerous to use, since the compiler has no way of knowing whether an operating system loads an executable program. It is wise therefore to hide the details of program loading within a single system component and make it look like a well-defined operation to the rest of the operating system.

In Concurrent Pascal, a process can load a compiled program into a data structure and call it as if it were a procedure. The loading is done by means of input operations as defined in [2]. When a compiled program terminates its execution it returns to the point inside the operating system where it was called by a process. (For the moment, we are ignoring the problem of how a program can be stopped if it causes a run-time error or exceeds a time limit.)

Figure 1 shows the simple idea of a user program being a procedure that is fetched and called by a system process. A process that controls the execution of a program will be called a job process.

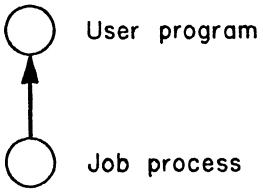

Fig 1. A job process that calls a user program

A compiled program must be stored in a variable declared within the job process:

```
var code: codestore
```

The data type codestore can, for example, be declared as an array of disk pages:

```
type codestore = array (...) of page
```

The job process must also include a declaration of the user program as a pseudo-procedure:

```
program job(v: codestore)
```

After loading, a program can be called as a procedure using its code variable as a parameter:

```
job(code)
```

If a user program completes its execution or causes a run-time error (say, an arithmetic overflow), it returns to the point where it was called by the job process. A standard function enables the job process to determine where and why the program

terminated. If the program failed, the job process can add a line number and an error message to its output.

3. SYSTEM INTERFACE

An operating system should be in complete control of resource allocation and input/output. But a user program must be able to call the operating system and ask it to perform these functions. Figure 2 shows how this is done.

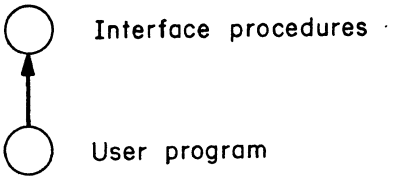

Fig. 2. Interface procedures

A user program is conceptually a procedure called by a job process. The program cannot access data inside the operating system (and vice versa). But the job process defines a set of procedures that can be called by a user program. They are called interface procedures.

As an example, consider a simple user program that only needs to input and output characters. To make this possible, two interface procedures, read and write, must be defined inside its job process:

```
procedure entry read(var c: char);
begin ... end;

procedure entry write(c: char);
begin ... end;
```

The details of these interface procedures are not relevant here. Interface procedures are marked with the word entry to distinguish them from local procedures used within the job process. A job process cannot call its own procedure entries (nor can any other system component).

The program declaration in the job process must be extended with a list of the interface procedures that are accessible to the user program:

```
program job(v: codestore);
entry read, write;
```

One can introduce several program declarations with different system interfaces inside a single job process. This makes it possible to give different access rights to different programs called by the same job process.

A sequential Pascal program must be prefixed by declarations of the interface procedures it can call and their parameter types. The following shows the job prefix of a user program that can call the read and write procedures inside its job process:

```
procedure read(var c: char);
procedure write(c: char);
program main;
... < user program > ...
```

The prefix must list the interface procedures in the order in which they appear in the program declaration inside the operating system.

A user could, of course, crash the system if he were able to write his own prefix. This can be avoided by having the operating system or the compiler automatically insert a standard prefix in front of all user programs. The compiler will then refuse to accept further interface definitions after the keyword program.

To a user program, its job process looks like a class that can be accessed by procedure calls only. The access rights of a user program to these procedures can be checked at compile time - a point that was emphasized in [2].

So far we have assumed that a program only has a single implicit parameter (representing its code). A job process can, however, pass explicit parameters to a program when it is called, and the program can return values when it terminates. The explicit program parameters and their types must be defined in the prefix as illustrated by the following example:

```
type T = ... ;
...
program main(param: T);
...
```

The Concurrent Pascal compiler assumes that the last parameter specified in a program declaration denotes the code to be executed. This parameter is not accessible within the sequential program. So a user program called with a single, explicit parameter would be declared as follows within a job process:

```
program job(param: T; code: codestore)
```

Only simple data types, arrays, records, and sets can be passed as explicit parameters to user programs. Procedures can only be passed as implicit parameters by means of the interface mechanism explained earlier. Processes, monitors, classes, and queues cannot be passed as parameters to user programs [2] (because user programs might misuse them and crash the operating system).

4. JOB CONTROL PROGRAMS

Sequential Pascal can also be used as a job control language to specify the execution of a sequence of programs. Figure 3 shows an example of this.

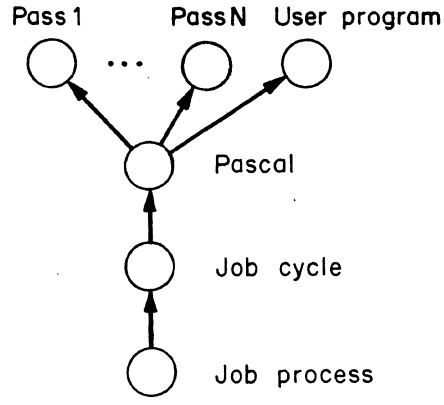

Fig. 3. Job control and user programs

Here a job process starts by executing a sequential program called the job cycle. The job cycle is a cyclical program that communicates with a user through a terminal. In this example, the user has asked the job cycle to call another program named pascal. This program in turn calls a sequence of other programs, pass 1 to pass N, that compiles a user program. If the compilation is successful, the user program will be executed. Afterwards control returns to the job cycle that will ask the user what to do next.

The job cycle and pascal programs are called job control programs because they control the execution of other programs. What a job control program needs is simply the ability to call other programs and pass parameters to them. Then the need for a separate job control language will vanish.

Nested program calls can be handled by an interface procedure run that takes a program identifier as an argument. The job process will look up the identifier in a directory and verify that it refers to an executable program. If so it will load the program and call it. After termination of a program, the job

process must reload the code of the previous program and return to it. To do this the process needs access to a stack of program identifiers - a trivial thing to implement by means of a class [2].

This leads to the following outline of the procedure run:

```
procedure entry run(id: identifier);
var oldid: identifier;
begin
  if executable(id) then
  begin
    load(id, code);
    push(id);
    job(code);
    pop(oldid);
    load(oldid, code);
  end;
end
```

The procedure can, of course, also include arguments to be passed to or from a called program (subject to the restrictions mentioned earlier).

This scheme essentially implements demand fetching of program code before and after each program call. The system only keeps the code of the current program in the internal store.

The variables of nested programs are, however, all kept in the stack of the job process until the programs terminate. This is not a serious problem since job control programs usually have few variables.

So although Concurrent Pascal uses a very primitive form of program loading, an operating system can make a program library look as a set of nested or recursive procedures. The integrity of a file system can be guaranteed by a suitable design of interface procedures that give user programs controlled access to programs and data.

5. PROGRAM PREEMPTION

How can an operating system force a sequential program to terminate if it goes into an endless loop or exceeds some time limit? Figure 4 shows the method used to solve this problem. The virtual machine associates a stop signal with every process. This signal is turned off by the operating system when the execution of a program begins and turned on when it must terminate.

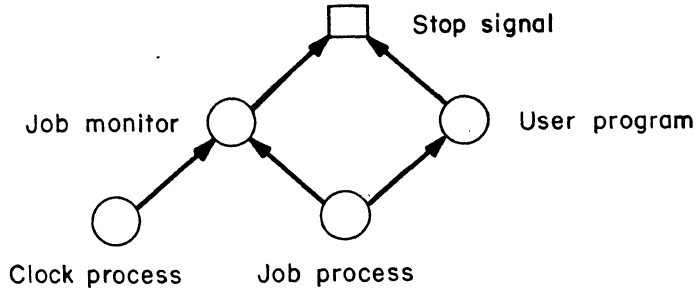

Fig. 4. Preemption of a user program

Before a job process starts a program it calls a job monitor and defines a time limit for the program. During program execution, the job monitor is called every second or so by a clock process. If the program has exceeded its time limit, the job monitor will call a standard procedure that turns the stop signal of the job process on.

The compiled user code examines this signal every time it repeats a loop. If the signal is on, the program terminates with the result time limit exceeded. (This mechanism can also be used to preempt a program at any time on request.)

One can argue about the details of this scheme, but it does have one major advantage: it makes preemption look like normal program termination to a job process. This simple solution only works because user code is generated by a reliable compiler

(and not by an assembly language programmer).

It may be too early to eliminate all use of assembly language programming in computer applications. But system programmers should realize that they are complicating their design problems tremendously if they assume that all programs potentially can be random collections of bits. A compiler can solve many of the reliability problems of programming systems in the simplest and most efficient manner (provided we are able to make a reliable compiler).

We must, of course, make sure that preemption does not take place while an interface procedure is being executed on behalf of a user program. Otherwise the data structures of an operating system might be left in an inconsistent state that could cause the system to crash. So it is only the code generated for sequential programs that examines stop signals; they are ignored by the code generated for concurrent programs.

Acknowledgements

This project is supported by the National Science Foundation under grant number DCR74-17331.

References

1. Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.

2. Brinch Hansen, P. The programming language Concurrent Pascal. Invited paper. ACM/IEEE International Conference on Reliable Software, Los Angeles, California, 21-23 April 1975. IEEE Transactions on Software Engineering 1, 2 (June 1975).

UNIVERSAL TYPES IN CONCURRENT PASCAL

Per Brinch Hansen

Information Science
California Institute of Technology

November 1974

Abstract

This note describes the use of universal types as a means of relaxing the checking of operand types where this is needed in system programming. The universal types are part of Concurrent Pascal, a new programming language for structured programming of computer operating systems.

Key Words and Phrases: Concurrent Pascal, programming languages, type checking, universal types.
CR Categories: 4.2, 4.3

This note describes a form of universal data types that seems to be convenient (if not necessary) in an abstract programming language for system programming. It is part of Concurrent Pascal, a new programming language for structured programming of computer operating systems [1, 2].

In most abstract programming languages operands and operators must be of compatible types. The type rules will allow you to add two integers, but not two booleans. The checking of such rules during compilation is vital for a programmer since it makes it possible for him to ignore the representation of data inside a computer store and think of them in terms of their abstract properties.

Occasionally, an operating system designer must, however, be able to relax the rules of type checking somewhat. The following describes how this can be done without going to the other extreme of introducing variables that are treated as typeless bit patterns throughout a program. (The latter extreme is, of course, the rule in assembly language and in some implementation languages.)

Consider an operating system procedure that writes a page of data on a disk:

```
procedure write(pageno: integer; page: charpage);
begin ... end
```

The details of this procedure do not matter here. We will assume that a charpage is defined elsewhere as an array of characters:

```
type charpage = array (...) of char
```

This procedure can now be used by an operating system to output a variable x as page number i on the disk:

```
var i: integer; x: charpage;
    ... write(i, x) ...
```

If we insist that the arguments of a procedure call must be of the same types as the parameters defined within the procedure then we cannot use the same procedure to output a page of another type, say an array of integers:

```
var j: integer; y: integerpage;
   ... write(j, y) ...
```

where

```
type integerpage = array (...) of integer
```

To make the write procedure more general we will use the key word univ to indicate that it can be called with any argument that occupies the same number of store locations as a charpage:

```
procedure write(pageno: integer; page: univ charpage);
begin ... end
```

The procedure can now be called with the argument y.

Before and after the call, the variable y is regarded as being strictly an integerpage. And within the procedure, the parameter page is considered to be strictly a charpage. The type checking is only relaxed at the point where the procedure is called.

There is also an occasional need for universal types in sequential system programs (usually handled by means of standard procedures). One example is the use of ordinal values of characters to convert constants from text form to numeric form. The following function defines the ordinal value of a character:

```
function ord(c: univ integer): integer;
begin ord:= c end
```

It can be called as follows:

```
var i: integer; c: char;
   ... i:= ord(c) ...
```

Another example would be a multipass compiler in which the passes communicate with one another by means of a sequence of integer values stored on disk. Here the first pass must be able to split a real constant into a sequence of integers ("machine words") and transmit them to the second pass. This can also be done by means of universal types.

A universal parameter type

```
univ T
```

represents the union of all arguments represented by the same number of store locations as the data type T. It seems reasonable to require that universal types not be used to perform undefined manipulation of data structures that contain pointers or are shared by concurrent processes.

Acknowledgements

The development of Concurrent Pascal is supported by the National Science Foundation under grant number DCR74-17331.

1. Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1, 2 (June 1975).

2. Brinch Hansen, P. Job control in Concurrent Pascal. Information Science, California Institute of Technology, March 1975.

BIBLIOGRAPHIC DATA SHEET	1. Report No. NSF-OCA-DCR74-17331-CP2	2.	3. Recipient's Accession No.
4. Title and Subtitle Concurrent Pascal Introduction			5. Report Date July 1975 6.
7. Author(s) Per Brinch Hansen			8. Performing Organization Rept. No. CIT-IS-TR 18
9. Performing Organization Name and Address Information Science 286-80 California Institute of Technology Pasadena, California 91125			10. Project/Task/Work Unit No. 11. Contract/Grant No. NSF DCR74-17331
12. Sponsoring Organization Name and Address National Science Foundation Office of Computing Activities Washington, D.C. 20550			13. Type of Report & Period Covered 1. Edition 14.

15. Supplementary Notes

16. Abstracts

This is an informal introduction to Concurrent Pascal, a programming language for structured programming of computer operating systems. It contains three papers entitled

1. The programming language Concurrent Pascal
2. Job control in Concurrent Pascal
3. Universal types in Concurrent Pascal

17. Key Words and Document Analysis. 17a. Descriptors

Structured multiprogramming, concurrent programming languages, hierarchical operating systems, concurrent processes, monitors, classes, abstract data types, access rights, scheduling, job control language, system interface, type checking, universal types

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 43
	20. Security Class (This Page) UNCLASSIFIED	22. Price