Open access • Journal Article • DOI:10.1109/TSE.1977.229907

Experience with Modular Concurrent Programming — Source link

Per Brinch Hansen

Institutions: University of Southern California

Published on: 01 Mar 1977 - IEEE Transactions on Software Engineering (IEEE)

Topics: Concurrent Pascal, Compiler, Concurrent computing, Data structure and Modular design

Experience with Modular Concurrent Programming*

(1977)

This paper summarizes the initial experience with the programming language Concurrent Pascal in the design of three model operating systems. A Concurrent Pascal program consists of modules called processes, monitors, and classes. The compiler checks that the data structures of each module are accessed only by the operations defined in the module. The author emphasizes that the creative aspect of program construction is the initial selection of modules and the connection of them into hierarchical structures. By comparison the detailed implementation of each module is straightforward. The most important result is that it is possible to build concurrent programs of one thousand lines out of one-page modules that can be comprehended at a glance.

1 Introduction

This paper summarizes the initial experience with the abstract programming language *Concurrent Pascal* in the design of three model operating systems. A Concurrent Pascal program consists of *modules* called processes, monitors, and classes (Brinch Hansen 1975a). The compiler checks that the data structures of each module are accessed only by the operations defined in the module. *The most important result so far is that it is possible to build a concurrent program of one thousand lines of text out of one-page modules that can be comprehended at a glance* (Brinch Hansen 1976b).

When this research project was started four years ago we had four main goals:

*P. Brinch Hansen, Experience with modular concurrent programming, *IEEE Transactions on Software Engineering 3*, 2 (March 1977), 156–159.

1. To develop an effective method for constructing large, reliable concurrent programs from trivial modules that can be defined, programmed, tested, and described one at a time.

2. To design an abstract programming language that supports a precise form of modularity and hides irrelevant machine detail.

3. To make a compiler that checks whether program modules use one another properly (in a restricted sense).

4. To build useful minicomputer operating systems exclusively by means of this abstract programming language.

The first attempt by Hoare and myself to invent a structured language for concurrent programs led to a notation for shared variables, critical regions, and scheduling queues (Hoare 1972a; Brinch Hansen 1972). In 1972 we combined these concepts into a single program module called a *shared class* (Brinch Hansen 1973) or a *monitor* (Hoare 1974). This was inspired by Dahl's *class* module (Dahl 1972; Hoare 1972b) and Dijkstra's *secretaries* (1971).

To experiment with these ideas on modularity I designed the programming language Concurrent Pascal (Brinch Hansen 1975b). It extends the sequential language Pascal (Wirth 1971) with processes, monitors, and classes. A Concurrent Pascal compiler for the PDP 11/45 computer (written in Pascal) was completed in January 1975 (Hartmann 1975).

A Concurrent Pascal program consists of a fixed number of processes executed simultaneously. Each process performs a sequence of operations on a data structure that is inaccessible to other processes. Processes can only communicate by means of monitors. A monitor is a module that defines all the possible operations on a shared data structure. It can, for example, define the *send* and *receive* operations on a message buffer. Finally, a class is a module that defines all the possible operations on a data structure used by a single process only. It can, for example, define the *open*, *close*, *read*, and *write* operations on a disk file.

The compiler checks that the data structures of each process, monitor, and class are accessed only by the operations defined within that module. The controlled access to data structures tends to confine programming errors within single modules and prevent them from causing obscure effects in other modules. This makes systematic testing of modules fast and effective.

Concurrent Pascal has been used to write three model operating systems: a single-user operating system (Solo), a job stream system for small jobs, and a real-time scheduler for process control (Brinch Hansen 1975d, 1976a, 1976b). They have been running successfully on a PDP 11/45 computer for more than a year.

The following describes the modular structure of these operating systems. I hope to show that *the creative aspect of concurrent programming is the initial selection of modules and the connection of them into hierarchical structures. The detailed implementation of each module is quite straightforward.* These details are described in the three papers mentioned above.

2 Program Modules

I will begin with an example from the *job stream system* which compiles and executes a stream of Pascal jobs. Input, execution, and output take place simultaneously using large buffers stored on disk.

Figure 1 shows two processes in the job stream system connected by a disk buffer. The circles are program modules (processes, monitors, and classes); the arrows show how they call one another.

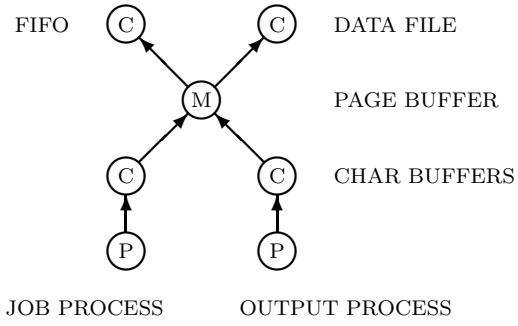

Figure 1 Job stream system.

A *job process* sends one character at a time to a *character buffer* module which assembles them into disk pages. When a page is full, the character buffer sends it through a *page buffer* module which in turn calls a *data file* module to store the page in a disk file of fixed length. The disk file is used as a cyclical buffer. The page buffer uses a *fifo* (first-in, first-out) module to

keep track of the order in which pages are transferred to and from the disk file.

An *output process* receives one character at a time from another character buffer which calls the page buffer when it needs another page from the disk.

Each of the modules consists of a data structure and the possible operations on it. Take for example the *page buffer*,

```
type pagebuffer =
monitor
var ...

procedure send(block: page)
begin ... end

procedure receive(var block: page)
begin ... end

begin initialize buffer end
```

It is defined as a *data type* that can be used to transmit pages from one process to another by means of *send* and *receive* operations.

A page buffer is represented by a *data file* and a *fifo* sequence. It also uses two *queues* to delay the sending and receiving processes when the buffer is full or empty:

```
var file: datafile; next: fifo;
  sender, receiver: queue
```

These variables are declared within the module and are not accessible outside it. They can only be used by the routines of the modules, for example

```
procedure receive(var block: page)
begin
  if next.empty then delay(receiver);
  file.read(next.departure, block);
  continue(sender)
end
```

This page buffer routine in turn calls other routines

next.empty next.departure file.read

defined within the fifo and data file modules, *next* and *file*.

A particular page buffer can be declared and used as follows by two processes:

var buffer: pagebuffer; text1, text2: page;

buffer.send(text1) buffer.receive(text2)

3 Hierarchical Structures

The job stream module that implements data files on disk was borrowed from the *Solo* operating system. Figure 2 shows the hierarchical structure of the Solo filing system (simplified a bit).

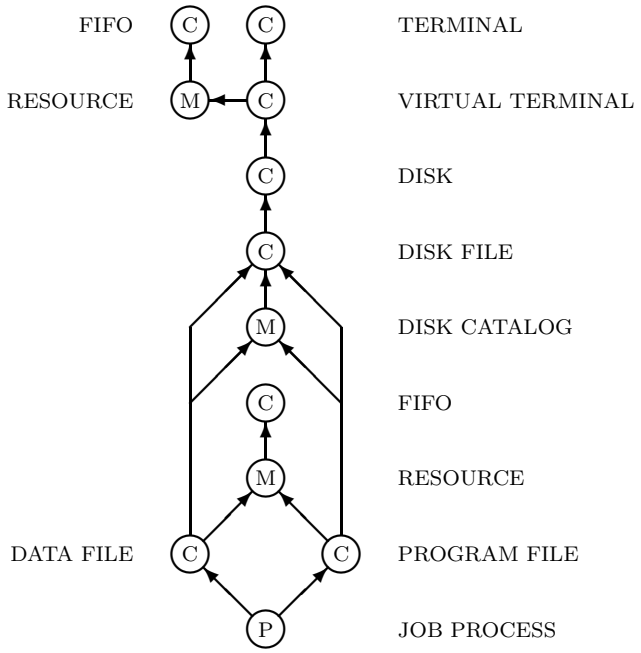

Figure 2 Solo system.

The heart of Solo is a *job process* that compiles and executes programs. It can access disk files through *data* and *program file* modules. Since there are other processes in the system, the file modules must use a *resource* module

to get exclusive access to the disk during page transfers. *Disk catalog* and *disk file* modules are used to locate a named file and its pages on the disk. A *disk* module handles the details of transferring a single page to or from the device. If the disk fails this is reported to the operator through a *virtual terminal*. Each process has its own virtual terminal. They all use the same resource module to get exclusive access to a single real *terminal*. A resource module uses a *fifo* module to implement first-come, first-served scheduling.

The Solo system was written eight months before the job stream system. It was a pleasant surprise to discover that 14 modules from Solo could be used unchanged in the job stream system. This may be the first example of different kinds of operating systems using the same modules.

Figure 3 shows the structure of another concurrent program. This is the *real-time scheduler* which executes a fixed number of task processes regularly with frequencies chosen by an operator. It is based on an existing process control system (Brinch Hansen 1967).

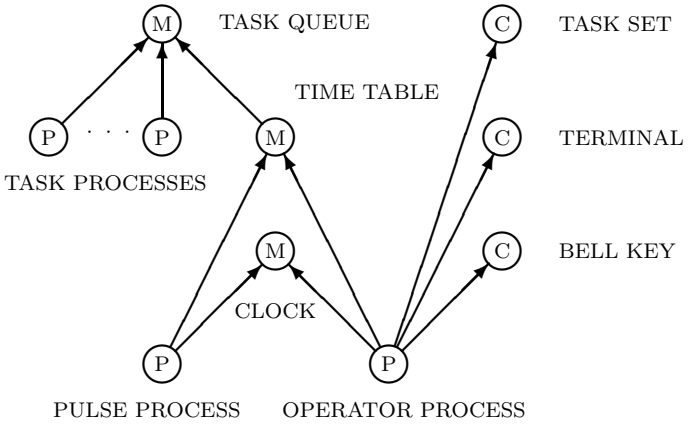

Figure 3 Real-time scheduler.

Each *task* is a cyclical process that waits inside a *task queue* module until a timing signal from a *time table* module wakes it up. After performing its task the process waits for the next signal. The time table defines the start time and frequency of each task. A *pulse process* updates a *clock* module every second and calls the time table which then starts all tasks that are due.

The operator contacts the scheduler by pushing the *bell key* on a terminal. This wakes up an *operator process* which then accepts a command from the operator through a *terminal* module. The command may cause the operator process to either set the clock or change the time table. The scheduler uses a *task set* to keep track of task names.

4 Program and Module Size

The model operating systems illustrate how one can build a concurrent program as a hierarchy of modules. Each module implements a data type and its operations. Other modules can use these operations, but cannot access the components of the data type directly.

So modules "know" very little about each other. This makes it possible to program them one at a time. The programming experience with Concurrent Pascal illustrates the success of this approach.

	Solo	Job Stream	Real Time
Lines	1300	1400	600
Modules	23	24	13
Lines/Module	57	58	46
Routines/Module	5	4	4
Lines/Routine	11	15	12

The table shows that each of the model operating systems is a Concurrent Pascal program of about 1000 lines of text divided into 15–25 modules. A module is roughly one page of text (50–60 lines) with about 5 routines of 10–15 lines each.

These three examples consistently show that it is possible to compose nontrivial concurrent programs of very simple modules that can be studied one page at a time as one reads a book. This must surely be the main goal of structured programming on a larger scale.

Each of the model operating systems corresponds to an assembly language program of about 4000 machine instructions. But fortunately they are written in an abstract programming language that hides machine detail, such as registers, addresses, bit patterns, interrupts, store allocation, and processor multiplexing. As a result it was possible for me to design, program, test, and describe each of these programs in a matter of weeks. Compared to assembly language, Concurrent Pascal has reduced my design effort for concurrent programs by an order of magnitude and has made them

so simple that a journal could publish the complete text of a 1300 line program (Brinch Hansen 1976b).

5 Reliability and Efficiency

The integrity of modules enforced during compilation tends to make programs practically correct before they are even tested. The modules of a concurrent program are tested one at a time starting with those that do not depend on other modules. Each module is tested by means of a short test process that calls the module and makes it execute all its statements. A detailed example of how this is done is described in Brinch Hansen (1975d).

When a module works, another one is tested on top of it. The compiler now makes sure that the new (untested) module only uses the routines of the old (tested) module. Since these routines already work, the new module cannot make the old one fail. This makes it quite easy to locate errors during testing.

Although systematic testing theoretically does not guarantee correctness it is very successful in practice. The Solo system was tested in 27 test runs during April 1975. It has since been used daily without software failure. The job stream system and the real-time scheduler were tested in 10 and 21 test runs. So the initial experience has been that *a concurrent program of one thousand lines requires a couple of compilations followed by one test run per component. And then it works.*

The checking of access rights to data structures is almost exclusively done during compilation. It is not supported by hardware protection mechanisms during execution. The elimination of consistency checks at run time makes routine calls between modules about as fast as routine calls within modules:

	μs
simple routine call	60
class routine call	80
monitor routine call	200

The static allocation of store among a fixed number of processes also contributes to efficiency (Brinch Hansen 1975c).

The Solo and job stream systems compile programs at the speed of the line printer (10 lines/s) and are not limited by the speed of the computer.

6 Final Remarks

As the first abstract language for modular concurrent programming Concurrent Pascal will no doubt turn out to have deficiences in detail, but the overall modular approach to program design by means of processes, monitors, and classes seems to be a fertile direction for further research.

Since a concurrent program can be composed of semi-independent modules of one page each, there is reason to believe that verification techniques for small, sequential programs can be extended to concurrent programs as well. Some of this work has already been started by Hoare (1972a, 1972b, 1974), Howard (1976), and Owicki and Gries (1976). It would be a worthy achievement to verify parts of a working operating system, such as Solo.

The greatest value of a formal approach to correctness is probably the extreme rigor and structure that it must impose on the design process from the beginning to be succesful. This cannot fail to improve our informal understanding of programs as well.

References

Brinch Hansen, P. 1967. The RC 4000 real-time control system at Pulawy. *BIT 7*, 4, 279–288. *Article 1.*

Brinch Hansen, P. 1972. Structured multiprogramming. *Communications of the ACM 15*, 7 (July), 574–578. *Article 4.*

Brinch Hansen, P. 1973. *Operating System Principles.* Prentice Hall, Englewood Cliffs, NJ, (July).

Brinch Hansen, P. 1975a. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering 1*, 2 (June), 199–207. *Article 7.*

Brinch Hansen, P. 1975b. Concurrent Pascal report. Information Science, California Institute of Technology, Pasadena, CA, (June).

Brinch Hansen, P. 1975c. Concurrent Pascal machine. Information Science, California Institute of Technology, Pasadena, CA, (October).

Brinch Hansen, P. 1975d. A real-time scheduler. Information Science, California Institute of Technology, Pasadena, CA, (November).

Brinch Hansen, P. 1976a. The job stream system. Information Science, California Institute of Technology, Pasadena, CA, (January).

Brinch Hansen, P. 1976b. The Solo operating system. *Software—Practice and Experience 6*, 2 (April–June), 141–205. *Articles 8–9.*

Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. 1972. *Structured Programming.* Academic Press, New York.

Dijkstra, E.W. 1971. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 115–138.

Hartmann, A.C. 1975. A Concurrent Pascal compiler for minicomputers. Information Science, California Institute of Technology, Pasadena, CA, (September).

Hoare, C.A.R. 1972a. Towards a theory of parallel programming. In *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrott, Eds., Academic Press, New York.

Hoare, C.A.R. 1972b. Proof of correctness of data representations. *Acta Informatica 1*, 271–281.

Hoare, C.A.R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM 17*, 10 (October), 549–557.

Howard, J.H. 1976. Proving monitors. *Communications of the ACM 19*, 5 (May), 273–279.

Owicki, S., and Gries, D. 1976. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM 19*, 5 (May), 279–285

Wirth, N. 1971. The programming language Pascal. *Acta Informatica 1*, 35–63.