15

THE SOLO OPERATING SYSTEM: A CONCURRENT PASCAL PROGRAM*

PER BRINCH HANSEN

(1976)

This is a description of the single-user operating system Solo written in the programming language Concurrent Pascal. It supports the development of Sequential and Concurrent Pascal programs for the PDP 11/45 computer. Input/output are handled by concurrent processes. Pascal programs can call one another recursively and pass arbitrary parameters among themselves. This makes it possible to use Pascal as a job control language. Solo is the first major example of a hierarchical concurrent program implemented in terms of abstract data types (classes, monitors and processes) with compile-time control of most access rights. It is described here from the user's point of view as an introduction to another paper describing its internal structure.

INTRODUCTION

This is a description of the first operating system *Solo* written in the programming language Concurrent Pascal (Brinch Hansen 1975). It is a simple, but useful single-user operating system for the development and distribution of Pascal programs for the PDP 11/45 computer. It has been in use since May 1975.

From the user's point of view there is nothing unusual about the system. It supports editing, compilation and storage of Sequential and Concurrent Pascal programs. These programs can access either console, cards, printer,

*P. Brinch Hansen, The Solo operating system: a Concurrent Pascal program. *Software—Practice and Experience 6*, 2 (April–June 1976), 141–149.

tape or disk at several levels (character by character, page by page, file by file, or by direct device access). Input, processing, and output of files are handled by concurrent processes. Pascal programs can call one another recursively and pass arbitrary parameters among themselves. This makes it possible to use Pascal as a job control language (Brinch Hansen 1976a).

To the system programmer, however, Solo is quite different from many other operating systems:

1. Less than 4 per cent of it is written in machine language. The rest is written in Sequential and Concurrent Pascal.

2. In contrast to machine-oriented languages, Pascal does not contain low-level programming features, such as registers, addresses and interrupts. These are all handled by the virtual machine on which compiled programs run.

3. System protection is achieved largely by means of compile-time checking of access rights. Run-time checking is minimal and is not supported by hardware mechanisms.

4. Solo is the first major example of a hierarchical concurrent program implemented by means of abstract data types (classes, monitors, and processes).

5. The complete system consisting of more than 100,000 machine words of code (including two compilers) was developed by a student and myself in less than a year.

To appreciate the usefulness of Concurrent Pascal one needs a good understanding of at least one operating system written in the language. The purpose of this description is to look at the Solo system from a user's point of view before studying its internal structure (Brinch Hansen 1976b). It tells how the user operates the system, how data flow inside it, how programs call one another and communicate, how files are stored on disk, and how well the system performs in typical tasks.

JOB CONTROL

The user controls program execution from a display (or a teletype). He calls a program by writing its name and its parameters, for example:

```
move(5)
read(maketemp, seqcode, true)
```

The first command positions a magnetic tape at file number 5. The second one inputs the file to disk and stores it as sequential code named maketemp. The boolean true protects the file against accidental deletion in the future.

Programs try to be helpful to the user when he needs it. If the user forgets which programs are available, he may for example type:

```
help
```

(or anything else). The system responds by writing:

```
not executable, try
    list(catalog, seqcode, console)
```

The suggested command lists the names of all sequential programs on the console.

If the user knows that the disk contains a certain program, but is uncertain about its parameter conventions, he can simply call it as a program without parameters, for example:

```
read
```

The program then gives the necessary information:

```
try again
    read(file: identifier; kind: filekind; protect: boolean)
using
    filekind = (scratch, ascii, seqcode, concode)
```

Still more information can be gained about a program by reading its manual:

```
copy(readman, console)
```

A user session may begin with the input of a new Pascal program from cards to disk:

```
copy(cards, sorttext)
```

followed by a compilation:

```
pascal(sorttext, printer, sort)
```

If the compiler reports errors on the program listing:

```
pascal:
compilation errors
```

the next step is usually to edit the program text:

```
edit(sorttext)
...
```

and compile it again. After a successful compilation, the user program can now be called directly:

```
sort(...)
```

The system can also read job control commands from other media, for example:

```
do(tape)
```

A task is preempted by pushing the bell key on the console. This causes the system to reload and initialize itself. The command *start* can be used to replace the Solo system with any other concurrent program stored on disk.

DATA FLOW

Figure 1 shows the data flow inside the system when the user is processing a single text file sequentially by copying, editing, or compiling it.

The input, processing, and output of text take place simultaneously. Processing is done by a *job process* that starts input by sending an argument through a buffer to an input process. The argument is the name of the input device or disk file.

The *input process* sends the data through another buffer to the job process. At the end of the file the input process sends an argument through yet another buffer to the job process indicating whether transmission errors occurred during the input.

Output is handled similarly by means of an *output process* and another set of buffers.

In a single-user operating system it is desirable to be able to process a file continuously at the highest possible speed. So the data are buffered in core instead of on disk. The capacity of each buffer is 512 characters.

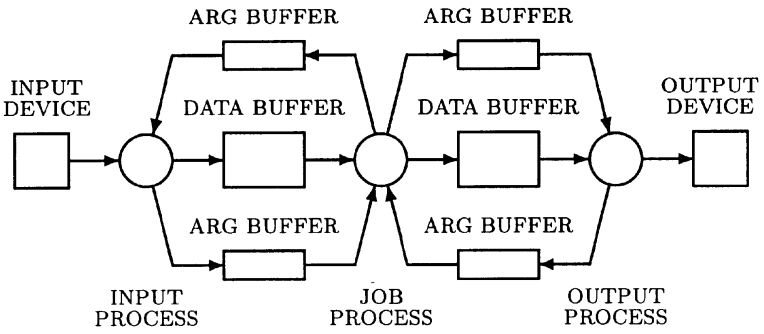

Figure 1 Processes and buffers.

CONTROL FLOW

Figure 2 shows what happens when the user types a command such as:

edit(cards, tape)

After system loading the machine executes a Concurrent Pascal program (Solo) consisting of three processes. Initially the input and output processes both load and call a sequential program *io* while the job process calls another sequential program *do*. The do program reads the user command from the console and calls the *edit* program with two parameters, *cards* and *tape*.

The editor starts its input by sending the first parameter to the io program executed by the input process. This causes the io program to call another program *cards* which then begins to read cards and send them to the job process.

The editor starts its output by sending the second parameter to the io program executed by the output process. The latter then calls a program *tape* which reads data from the job process and puts them on tape.

At the end of the file the cards and tape programs return to the io programs which then await further instructions from the job process. The editor returns to the do program which then reads and interprets the next command from the console.

It is worth observing that the operating system itself has no built-in drivers for input/output from various devices. Data are simply produced

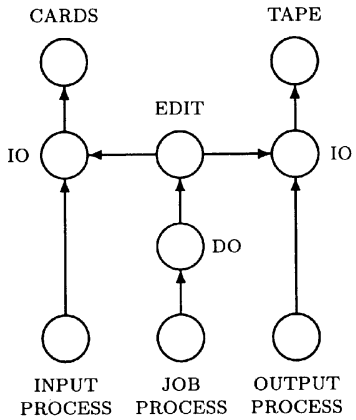

Figure 2 Concurrent processes and sequential programs.

and consumed by Sequential Pascal programs stored on disk. The operating system only contains the mechanism to call these. This gives the user complete freedom to supplement the system with new devices and simulate complicated input/output such as the merging, splitting and formatting of files without changing the job programs.

Most important is the ability of Sequential Pascal programs to call one another recursively with arbitrary parameters. In Fig. 2, for example, the do program calls the edit program with two identifiers as parameters. This removes the need for a separate (awkward) job control language. *The job control language is Pascal.*

This is illustrated more dramatically in Fig. 3 which shows how the command:

pascal(sorttext, printer, sort)

causes the do program to call the program *pascal*. The latter in turn calls seven compiler passes one at a time, and (if the compiled program is correct) *pascal* finally calls the filing system to store the generated code.

A program does not know whether it is being called by another program or directly from the console. In Fig. 3 the program *pascal* calls the filing system. The user, may, however, also call the file system directly, for example, to protect his program against accidental deletion:

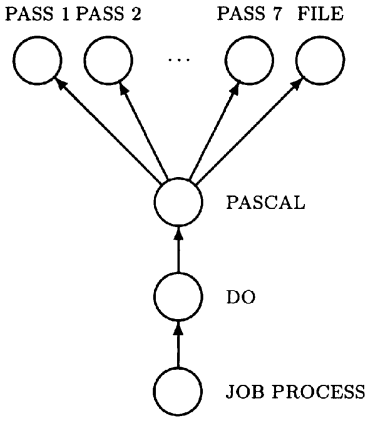

Figure 3 Compilation.

```
file(protect, sort, true)
```

The Pascal *pointer* and *heap* concepts give programs the ability to pass arbitrarily complicated data structures among each other, such as symbol tables during compilation (Jensen 1974). In most cases, however, it suffices to be able to use identifiers, integers, and booleans as program parameters.

STORE ALLOCATION

The run-time environment of Sequential and Concurrent Pascal is a kernel of 4 K words. This is the only program written in machine language. The user loads the kernel from disk into core by means of the operator's panel. The kernel then loads the Solo system and starts it. The Solo system consists of a fixed number of processes. They occupy fixed amounts of core store determined by the compiler.

All other programs are written in Sequential Pascal. Each process stores the code of the currently executed program in a fixed core segment. After termination of a program called by another, the process reloads the previous program from disk and returns to it. The data used by a process and the programs called by it are all stored in a core resident stack of fixed length.

FILE SYSTEM

The backing store is a slow *disk* with removable packs. Each user has his own disk pack containing the system and his private files. So there is no need for a hierarchical file system.

A disk pack contains a *catalog* of all files stored on it. The catalog describes itself as a file. A *file* is described by its name, type, protection and disk address. Files are looked up by hashing.

All system programs check the *types* of their input files before operating on them and associate types with their output files. The Sequential Pascal compiler, for example, will take input from an ascii file (but not from a scratch file), and will make its output a sequential code file. The possible file types are scratch, ascii, seqcode and concode.

Since each user has his own disk pack, files need only be *protected* against accidental overwriting or deletion. All files are initially unprotected. To protect one the user must call the file system from the console as described in Section 4.

To avoid compacting of files (lasting several minutes), file pages are scattered on disk and addressed indirectly through a *page map* (Fig. 4). A file is opened by looking it up in the catalog and bringing its page map into core.

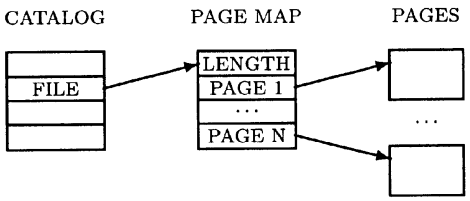

Figure 4 File system.

The resident part of the Solo system implements only the most frequently used file operations: lookup, open, close, get and put. A nonresident, sequential program, called *file*, handles the more complicated and less frequently used operations: create, replace, rename, protect, and delete file.

DISK ALLOCATION

The disk always contains a scratch file of 255 pages called *next*. A program creates a new file by outputting data to this file. It then calls the file system to associate the data with a new name, a type, and a length (≤ 255). Having done this the file system creates a new instance of *next*.

This scheme has two advantages:

1. All files are initialized with typed data.
2. A program creating a file need only call the nonresident file system once (after producing the file). Without the file *next* the file system would have to be called at least twice: before output to create the file, and after output to define its final length.

The disadvantages of having a single file *next* is that a program can only create one file at a time.

Unused disk pages are defined by a powerset of page indices stored on the disk.

On a slow disk special care must be taken to make *program loading* fast. If program pages were randomly scattered on the disk it would take 16 seconds to load the compiler and its input/output drivers. An algorithm described in Brinch Hansen (1976c) reduces this to 5 seconds. When the system creates the file *next* it tries to place it on consecutive pages within neighboring cylinders as far as possible (but will scatter the pages somewhat if it has to). It then rearranges the page indices within the page map to minimize the number of disk revolutions and cylinder movements needed to load the file. Since this is done before a program is compiled and stored on disk it is called *disk scheduling at compile time*.

The system uses a different allocation technique for the two temporary files used during compilation. Each pass of the compiler takes input from a file produced by its predecessor and delivers output to its successor on another file. A program *maketemp* creates these files and interleaves their page indices (making every second page belong to one file and every second one to the other). This makes the disk head sweep slowly across both files during a pass instead of moving wildly back and forth between them.

OPERATOR COMMUNICATION

The user communicates with the system through a console. Since a task (such as editing) usually involves several programs executed by concurrent

processes these programs must identify themselves to the user before asking for input or making output:

```
do:
edit(cards, tape)
edit:
...
do:
...
```

Program identity is only displayed every time the user starts talking to a different program. A program that communicates several times with the user without interruption (such as the editor) only identifies itself once.

Normally only one program at a time tries to talk to the user (the current program executed by the job process). But an input/output error may cause a message from another process:

```
tape:
inspect
```

Since processes rarely compete for the console, it is sufficient to give a process *exclusive access* to the user for input or output of a single line. A conversation of several lines will seldom be interrupted.

A Pascal program only calls the operating system once with its identification. The system will then automatically display it when necessary.

SIZE AND PERFORMANCE

The Solo system consists of an operating system written in Concurrent Pascal and a set of system programs written in Sequential Pascal:

Program	Pascal lines	Machine words
operating system	1,300	4 K
do, io	700	4 K
file system	900	5 K
concurrent compiler	8,300	42 K
sequential compiler	8,300	42 K
editor	400	2 K
input/output programs	600	3 K
others	1,300	8 K
	21,800	110 K

(The two Pascal compilers can be used under different operating systems written in Concurrent Pascal—not just Solo.)

The amount of code written in different programming languages is:

Language	%
machine language	4
Concurrent Pascal	4
Sequential Pascal	92

This clearly shows that a good sequential programming language is more important for operating system design than a concurrent language. But although a concurrent program may be small it still seems worthwhile to write it in a high-level language that enables a compiler to do thorough checking of data types and access rights. Otherwise, it is far too easy to make time-dependent programming errors that are extremely difficult to locate.

The kernel written in machine language implements the process and monitor concepts of Concurrent Pascal and responds to interrupts. It is independent of the particular operating system running on top of it.

The Solo system requires a core store of 39 K words for programs and data:

Programs	K words
kernel	4
operating system	11
input/output programs	6
job programs	18
core store	39

This amount of space allows the Pascal compiler to compile itself.

The speed of text processing using disk input and tape output is:

Program	char/sec
copy	11,600
edit	3,300–6,200
compile	240

All these tasks are 60–100 per cent disk limited. These figures do not distinguish between time spent waiting for peripherals and time spent executing operating system or user code since this distinction is irrelevant to the user. They illustrate an overall performance of a system written in a high-level language using straightforward code generation without any optimization.

FINAL REMARKS

The compilers for Sequential and Concurrent Pascal were designed and implemented by Al Hartmann and me in half a year. I wrote the operating system and its utility programs in 3 months. In machine language this would have required 20–30 man-years and nobody would have been able to understand the system fully. The use of an efficient, abstract programming language reduced the development cost to less than 2 man-years and produced a system that is completely understood by two programmers.

The low cost of programming makes it acceptable to throw away awkward programs and rewrite them. We did this several times: An early 6-pass compiler was never released (although it worked perfectly) because we found its structure too complicated. The first operating system written in Concurrent Pascal (called *Deamy*) was used only to evaluate the expressive power of the language and was never built (Brinch Hansen 1974). The second one (called *Pilot*) was used for several months but was too slow.

From a manufacturer's point of view it is now realistic and attractive to replace a huge ineffective "general-purpose" operating system with a range of small, efficient systems for special purposes.

The kernel, the operating system, and the compilers were tested very systematically initially and appear to be correct.

Acknowledgements

The work of Bob Deverill and Al Hartmann in implementing the kernel and compiler of Concurrent Pascal has been essential for this project. I am also grateful to Gilbert McCann for his encouragement and support.

Stoy and Strachey (1972) recommend that one should learn to build good operating systems for single-users before trying to satisfy many users simultaneously. I have found this to be very good advice. I have also tried to follow the advice of Lampson (1974) and make both high- and low-level abstractions available to the user programmer.

The Concurrent Pascal project is supported by the National Science Foundation under grant number DCR74–17331.

References

1. P. Brinch Hansen 1974. Deamy—A structured operating system. Information Science, California Institute of Technology, (May), (out of print).
2. P. Brinch Hansen 1975. The programming language Concurrent Pascal. *IEEE Trans. on Software Engineering*, **1**, 2 (June).

3. P. Brinch Hansen 1976a. The Solo operating system: job interface. *Software—Practice and Experience*, **6**, 2 (April–June).

4. P. Brinch Hansen 1976b. The Solo operating system: processes, monitors and classes. *Software—Practice and Experience*, **6**, 2 (April–June).

5. P. Brinch Hansen 1976c. Disk scheduling at compile-time. *Software—Practice and Experience 6*, 2 (April–June), 201–205.

6. K. Jensen and N. Wirth 1974. Pascal–User manual and report. *Lecture Notes in Computer Science*, **18**, Springer-Verlag, New York.

7. B. W. Lampson 1974. An open operating system for a single-user machine. In *Operating Systems, Lecture Notes in Computer Science*, **16**, Springer Verlag, 208–217.

8. J. E. Stoy and C. Strachey 1972. OS6—an experimental operating system for a small computer. *Comput. J.*, **15**, 2.