Structure of an Operating System.

C.A.R. Hoare*, R.M. McKeag**

Draft. May 1977.

Second Draft. October 1977.

Summary: This paper suggests that the structure of an operating system can be clearly expressed as a hierarchy of communicating sequential processes. The suggestion is illustrated by the development of an absurdly simple multiprogrammed batch processing system. It is hoped that the structuring methods and notations may be more widely useful.

Key words and phrases: programming languages, operating systems, program structure, communicating sequential processes.

C.R. Categories. 4.22, 4.32, 4.35.

*Programming Research Group, University Computing Laboratory, Oxford OX2 6PE, England.

**Department of Computer Science, The Queen's University, Belfast BT7 1NN, N. Ireland.

1. Introduction.

Some of the reasons for planning and maintaining a clear structure of a large computer program are

(1) To enable the design to proceed in an orderly and intellectually manageable fashion.

(2) To enable different parts of the design to be implemented reliably by different programmers at different times.

(3) To enable the program to be tested systematically in a way that contributes to confidence in its overall correctness.

(4) To enable the program to be readily modified in its general configuration or in the detail of its parts, without risk of unexpected interactions.

(5) To enable the programming conventions which guarantee soundness of the whole structure to be enforced as far as possible by "compile time" checks.

Edsger W. Dijkstra [2] has suggested that an operating system should be structured as a series of levels, each of which uses the lower levels to implement a more desirable virtual machine for the benefit of the higher levels. The lowest level is the bare hardware, and the highest level consists of the virtual machines in which the user programs run. This paper suggests that the concept of the communicating sequential process is a suitable one for expressing such a structure, and illustrates the suggestion by the stepwise development of an absurdly simple batch processing system.

Of course, when an operating system is expressed in a higher level language, the lowest level implemented by software (or microcode, or even hardware) will be the necessary "run-time support" for that language, and cannot reasonably be implemented as part of a program expressed in that language. In the case of a language incorporating communicating sequential processes, the run-time support must include the allocation of local storage and processor(s) to the processes, and administration of the communication between them, as well

as any simulation required to make peripheral devices with interrupts look like communicating sequential processes. This will usually require several hundred machine code or microcode instructions, depending on the vagaries of hardware interface designs.

The language of communicating sequential processes has been described in a previous paper [4]. However, for the elucidation of the structure of an operating system, several extensions are desirable. Section 2 describes a method for dynamic establishment and disestablishment of communication channels between processes. Section 3 describes a scope rule which assists in the multilevel structuring of programs. Section 4 introduces the parallel repetitive command in which the number of activations of a process is not bounded a priori, but is determined by the needs of the rest of the program. Finally, for convenience and symmetry, we have used output commands as guards, in the same way as input commands.

These extensions are not so well suited to implementation on arrays of processors with disjoint main storage; and even when implemented on a single processor (or multiprocessor with shared main store), considerable optimisation of processor allocation, storage allocation, and message passing may be required to achieve reasonable efficiency. It is left as an open question how far such optimisations can be accomplished automatically by a translator, and how far the programmer can guide the optimisation.

The general structuring methods described in this paper reproduce some of the facility of the class and inner concepts of SIMULA 67; indeed, an operating system structure based on these SIMULA concepts was presented in [5]. The major advance of the present paper is the introduction of forms of parallelism, input, and output which seem to bring a great conceptual simplification and unification, though possibly at the expense of postponing problems of efficient implementation. If these problems can be solved, it is hoped that the usefulness of these concepts will extend more widely than operating systems.

2. Declarations of communication channels.

We allow a process name to feature in place of a type in the declaration of a variable, e.g.

```
&p:lineprinter                    x: X
```

If the declaration of "&p" (on the left) occurs in a process named "X", and the declaration of "x" (on the right) occurs in a process named "lineprinter", then these two declarations are executed simultaneously, and their effect is to set up a new communication channel between "lineprinter" and "X". The channel is broken down again as soon as either process exits from the scope of the declaration.

Within the scope of "&p", there may occur output (or input) instructions, e.g.

```
&p!heading; &p!concatenate ("COST IS", decimal (cost));
```

which communicate through the dynamically established channel to a corresponding input (or output) command within the scope of "x", e.g.

```
&:line;  x?&;
```

An attempt to communicate using a channel which has been broken down (by exit from scope) will fail, in the same way as communication with a terminated process; and this failure can cause termination of a repetitive command in which the input command appears as a guard.

The purpose of this extension is to permit processes internal to one process to set up communication channels with another process or even the latter's internal subprocesses. Since names of internal subprocesses are local, it is not possible for another process to use these names directly. That is why it is necessary for each subprocess to declare a new local name (e.g. "&p","x") by which it refers to its communicant. The declaration of a communication channel can appear as a guard, and will fail if the named process has terminated.

Example 1. Allocation of a single resource.

Problem: A single lineprinter is to be shared among the processes local to a process "X". The lineprinter is to be acquired by declaration (as described above); and the user process may then repeatedly output to it values of type "line". On exit from the scope of the declaration, the lineprinter is released, and is able to respond to further declarations from the same or another process in "X". But only one process at a time should use the lineprinter.

Use English instead of machine code for instructions to the hardware of the lineprinter.

Solution:

lineprinter:: *[x: X → *[ℓ: line; x?ℓ → ... print ℓ ...]]

Notes:

(1) The "lineprinter" process consists of a repetitive command, which terminates if and when the process "X" is terminated; this can happen only when all internal processes of "X" have terminated.

(2) Each repetition first "acquires" a client process from "X", and gives it a local name "x".

(3) It then embarks on an inner repetitive command, each repetition of which prints one line sent by "x".

(4) Each output line is single-buffered in "ℓ", so that "x" can proceed while "lineprinter" is waiting for the hardware to accept the line.

(5) The inner repetitive command terminates when the client process "x" leaves the block in which the given activation of a "lineprinter" was declared.

(6) The outer repetition is then ready again to respond to another output declaration of a "lineprinter" from within "X", either from the same or from a different process.

(7) If "X" contains a nested declaration of a "lineprinter", the "lineprinter" process can never respond to it, and the two processes will be deadlocked.

(8) However, normal scope rules provide a compile-time check against a process in "X" using the actual lineprinter before "acquiring" it, or after "releasing" it; and it is impossible to "forget" to release it (provided, of course, that "X" contains no machine code).

Often there will be not just one resource of a particular type but several resources; and it does not matter which of them is used on a particular occasion. This case is readily treated by a parallel command, with one process per resource; e.g.:

```
[resource 1 || resource 2 || resource 3]
```

If the code for the three resources is quite similar, it is more convenient to use the notation of the parallel array, e.g.

```
[Y(i: 1 .. 3) :: resource].
```

Here "resource" stands for the code which represents the resource. It may contain (but not assign to) the bound variable "i". The parallel array is equivalent to writing out the code for the resource three times, each time with a different value for "i", ranging between one and three:

$$[Y(1) :: \text{resource}_1 \;||\; Y(2) :: \text{resource}_2 \;||\; Y(3) :: \text{resource}_3]$$

Example 2.

Problem: Same as example 1, but with two lineprinters instead of one.

Solution:

```
lineprinter :: [Y(i: 1 .. 2) :: one lineprinter]
```

where "one lineprinter" stands for

```
*[x: X → *[ℓ: line; x?ℓ → ... print ℓ on printer i ...]]
```

Notes:

(1) The "lineprinter" process now contains an array of two internal subprocesses, each of which deals with one lineprinter, in the same way as in the solution to the previous problem.

(2) From within "X", or from within a subprocess of "X", a lineprinter may be acquired and used in exactly the same way as before: "ℓp: lineprinter". One of the two processes of the "lineprinter" array will respond to this declaration when it is ready to execute its input declaration. There is no way in which "x" may find out which lineprinter it is using.

(3) If "X" contains doubly nested lineprinter declarations, deadlock will result; or if each of two concurrent processes contains a singly nested declaration, there is a risk of deadly embrace [3].

(4) The name "Y" of the local array is never used, since the lineprinters do not need to communicate with each other.

Example 3. A simple multiprogramming system.

A simple multiprogramming system consists of a fixed number of processes (say three), each of which executes a batch of jobs submitted on one of (say) two cardreaders, and prints their output on one of (say) two lineprinters. Each user program is executed in one of three virtual machines. Thus the overall structure of the system is:

```
[X :: [Y(i: 1 .. 3) :: batch processor]
 || lineprinter :: ... see example 2 ...
 || cardreader :: ... left as an exercise ...
 || virtualmachine :: ... explained below ...
]
```

Each of the three batch processors consists of a repetitive command, which terminates when a switch is off; i.e. "batch processor" stands for:

```
*[switch? on ( ) → execute one job]
```

In order to execute a job, it is necessary to acquire a card reader, a lineprinter, and a "virtual machine", which provides the main storage within

which the user's job will be executed. We specify that this store is initialised to contain a standard user program, say a load-and-go compiler, or a control language interpreter. This program is triggered and proceeds in parallel with the batch processor. But since the virtual machine has no input or output devices, it must communicate with the operating system to perform all required reading and printing. It also informs the operating system on completion of each timeslice of (say) ten thousand instructions. This enables the operating system to maintain an account of the cost of each job, and print it out afterwards.

(In practice, the concept of a virtual machine will be implemented by setting base and limit registers, and using supervisor entry and exit instructions; the details are not relevant to the conceptual structure of the operating system. In fact, the relationship between a virtual machine and a batch processor is not necessary for an understanding of the remainder of this paper).

The following program assumes that the user's job always terminates after a reasonable time, and always reads the right number of cards. "execute one job" stands for:

```
cost: integer; cost := 0;
cr: cardreader; ℓ: line;
ℓp: lineprinter;
job: virtualmachine;
*[job? timeslice ( ) → cost := cost+timecharge
 []job?ℓ → ℓp!ℓ; cost := cost+linecharge
 []job? input ( ) → card: line; cr? card; job!card;
                    cost := cost+cardcharge
 ]; comment terminates when job is terminated;
 ℓp! concatenate ("COST IS", decimal (cost))
```

Notes:

(1) The operating system is not subject to deadly embrace, because the resources are always acquired in the same order by each batch processor, and they are all released before any of them is acquired again.

(2) We have assumed that each job behaves correctly, in that it reads exactly the right cards from the batch, and terminates after a reasonable time.

The operating system described above is very simple, but that is its only merit - it would be dreadful to use! Among other defects:

(a) It contains no provision for breakdown of hardware components.

(b) The assumption of note (2) is wholly unrealistic.

(c) In practice, shortage of peripheral equipment limits the actual degree of multiprogramming to two, because one of the batch processors will always be waiting for peripherals.

Defects (b) and (c) will be mitigated in the development of later examples.

3. Hole in scope.

In ALGOL 60, when the name of a procedure occurs inside the body of that same procedure, it refers to a recursive activation of that procedure. This is obviously a useful feature of the language, but it creates a slight difficulty in the multilevel structuring of a program. Suppose, for example, that a high level program uses the function "cos", without caring how it is implemented. At a lower level, it is decided that the standard function "cos" is not suitable, and it should be replaced by a programmed procedure. The scope rules of ALGOL 60 provide an excellent method of doing this, without changing the text of the high level program; simply replace the "high level program" by:

```
begin real procedure cos(x); ... new cos procedure body ...;
      high level program
end
```

All occurrences of the identifier "cos" in the high level program are "captured" by the local procedure declaration, and do not reach the more global standard function. But the difficulty occurs when the new cos procedure body needs to call the standard function "cos"; since in ALGOL 60 a use of this identifier would make a recursive call on the new "cos", which is certainly not wanted!

For this reason we adopt a different scope rule for names of communicating sequential processes. We permit the body of a process to mention its own name (or that of a textually enclosing process) in an input or output command, but specify that this denotes a non enclosing process with that same name. Thus the scope of a process name extends over all other processes in the same parallel command, but it does not include the body of the named process.* In all other respects, the normal ALGOL scope rules still apply - a name denotes the process to which that name is prefixed in the smallest enclosing parallel command. (In fact this is the only reasonable interpretation of a process which mentions its own name; since an attempt to communicate with itself (or an enclosing process) would be always unsuccessful and an attempt at recursive communication would seem meaningless. But further discussion of recursion is beyond the scope of this paper).

Less formally, if we regard a process as the "ancestor" of all its internal subprocesses, then a process name occurring in an input or output command always refers to its brother, or its uncle, or its great uncle, or its great great uncle, etc; and it always refers to the closest possible member of this series. It never refers to a direct ancestor.

---

* This rule conflicts with the use of process array names in [4].

Example 4.

Problem: The multiprogramming system of example 3 assumes that each job will read all the cards relevant to that job, and no more. This is an unrealistic assumption. We shall therefore stipulate that the cards of each job are followed by a special separator card, and we need to rewrite the system to ensure that if any job attempts to read beyond the separator its input requests will be met by the simple trick of replicating the separator card; and if the job terminates before the separator card is read, the remaining cards of the job (including the separator) are read and ignored, so that the next job will start properly at the beginning of its card deck.

An additional advantage of the separator card is that the cardreader can be deallocated as soon as the separator is read. Thus, if the jobs tend to finish their input early, it will often be possible for more than two jobs to run concurrently.

Solution: replace "execute one job" in example 3 by

```
[X :: execute one job || cardreader :: separate input]
```

where "separate input" is

```
x:X;
[cr: cardreader;
 c:line; cr?c; comment read one card ahead;
*[c≠separator; x!c → cr?c];
comment either c = separator or x has terminated;
*[c≠separator → cr?c]; comment skip unread cards (if any);
]; comment the real cardreader is released here;
*[x!separator → skip]
```

Notes:

(1) The declaration "x:X" responds to the declaration "cr: cardreader" in "execute one job", which is now the closest process with name "X".

(2) The declaration "cr: cardreader" acquires a real cardreader from the more global process with the name "cardreader", even though it occurs within a process which itself is named "cardreader".

(3) The last repetitive command sends separator cards to satisfy any additional input commands from "x".

Example 5.

Problem: The output produced by a batch also requires separator lines, so that material output by each job can be conveniently detached and returned to its owner. Adapt the system of example 4 to ensure that output from each job is followed by a separator; and that any attempt to output further lines after a separator is ignored.

It is advantageous also to delay acquisition of the "real" lineprinter until the first line has to be output. Thus if the jobs tend to engage in significant computation before their first output, it will often be possible for more than two jobs to run concurrently.

Solution: replace "execute one job" by

```
[X :: execute one job || lineprinter :: separate output]
```

where "separate output" is

```
x:X; once: Boolean; once := true; ℓ:line;
*[once; x?ℓ → ℓp: lineprinter; ℓp!ℓ;
    *[ℓ ≠ separator; x?ℓ → ℓp!ℓ];
     [ℓ = separator → skip [] ℓ ≠ separator → ℓp! separator];
    once := false
]; comment real lineprinter released here;
*[x?ℓ → skip]; comment ignore lines after separator;
```

Notes:

(1) The first loop is iterated at most once (and not at all, if a job has no output).

(2) In practice it would be a good idea to ensure that all separator lines are printed on double-page boundaries, to facilitate bursting by an operator.

(3) Removing the unnecessary nesting, the overall structure of the operating system is now:

```
[X :: [Y(i: 1 .. 3) :: *[switch?on ( ) →
                                    [X :: execute one job
                                    || cardreader :: separate input
                                    || lineprinter :: separate output
                                    ]
                        ]
      ]
|| cardreader: ... left as an exercise ...
|| lineprinter: ... see example 2 ...
|| virtualmachine: ... explained above ...
]
```

4. The parallel repetitive command.

A parallel repetitive command is like the normal sequential repetitive command, in that it involves a dynamically determined number of activations of its body; it differs only in that each activation proceeds in parallel with all those that started earlier. The notation for a parallel repetitive command will be the same as that of the sequential repetition, except that a double star ** will be used in place of the single star *.

To ensure disjointness, we must stipulate that the body of a parallel repetitive command must not update any global variables at all. Consequently, the normal method of termination of repetitive commands (when their Boolean guards become false) is not applicable; so the guards on a parallel repetitive

command must be input or output guards or declarations, which cause termination when all their sources and destinations have terminated.

Example 6.

Problem: The efficiency of a multiprogramming system can be greatly increased by the technique of spooled (pseudoofflined) output. The lines, when output by a job, are not transmitted to a real lineprinter; instead, they are copied to a file on backing store, and actual printing is started only when the job is complete. Adapt the system of example 5 by including spooled output; assume an implementation of the concept of a file, to which lines may be output, and which must be rewound before they can be input again.

Solution: The only change required is to replace the code of example 2 by

```
lineprinter :: [X :: output spooling || ... example 2 ...]
```

where "output spooling" stands for

```
**[x:X → f:file; ℓ: line;
  *[[redacted] x?ℓ → f!ℓ]; comment x has terminated;
  f!rewind ( );
  ℓp: lineprinter; comment only now, acquire a real lineprinter;
  *[[redacted] f?ℓ → ℓp!ℓ]; comment the file has terminated;
 ]
```

Notes:

(1) There is no a priori limit to the number of activations of this parallel repetitive command that may be in concurrent execution.

(2) Each activation is initiated by a declaration "ℓp: lineprinter" from within the more global "X", i.e., the batch processors.

(3) When "X" terminates, no further activations of the parallel repetitive command are initiated. The command then terminates after all outstanding activations have terminated.

(4) Of course, since there are only two real lineprinters, only two activations of the command can be executing their second loop simultaneously; and since there are only three jobs in concurrent execution, at most three of them can be executing their first loop. All the rest of them will be rewinding their files or waiting for a lineprinter or a file.

Example 7.

Problem: Efficiency of operation may also be increased by spooling of input.

Solution: replace the original "cardreader" process of example 3 (exercise) by cardreader :: [X :: input spooling || cardreader :: ...]
where "input spooling" stands for

```
**[x:X → f: file; c: line;
   [cr: cardreader; comment real card reader;
    cr?c;
    *[c≠separator → f!c; cr?c]; comment c = separator;
   ]; comment real cardreader released here;
   f!separator; f!rewind ( );
   comment now the user x can begin input;
   *[f?c → x!c];
   comment we have already ensured that the job will read
   all cards up to the separator, and no more;
  ]
```

The maximum number of concurrent instances of the input spooling process is equal to the number of concurrent batch processors; it would therefore be advantageous to increase the number of batch processors to (say) ten. Of course, it is likely that most of them will spend most of their time waiting for a virtual machine in which to run a job; but this is good, because it ensures that there will usually be a load of work waiting for the central processor(s).

5. Summary.

After all the developments of the previous sections, it is helpful to display the overall structure of the complete spooled multiprogrammed system (see figure 1).

```
[X :: [Y(i: 1 .. 10) :: *[switch?on ( ) →
                              X :: execute one job
                              || cardreader :: separate input
                              || lineprinter :: separate output
      ]                  ]
|| cardreader :: [X :: input spooling || cardreader :: ... exercise ... ]
|| lineprinter :: [X :: output spooling || lineprinter :: ... example 2 ... ]
|| virtual machine :: ... explained above ...
]
```

Figure 1.

The persistent (even perverse) reuse of the same identifiers at every branch and every level of the structure is the result of the way we have chosen to present its development by stepwise enrichment. The structure can also be displayed pictorially without redundant names. Figure 2 omits the virtual machines and the filing system; it indicates physical containment of processes and subprocesses by solid lines, and communication channels by dotted lines. A process with more than one instance is indicated by a double box.

For practical use, this multiprogramming system still suffers from many defects, including

(1) There should be a cost limit imposed on each job.

(2) Cards unread by a job should be printed out, to assist in diagnosis.

(3) Better methods are required for job identification and accounting.

(4) No provision is made for rerunning jobs which have already been input when the hardware breaks down.

(5) It is not possible to ensure that more urgent jobs overtake less urgent ones.

(6) There is no way of avoiding the spooling of exceptionally long files.

(7) No job can use more than one input and one output file.

(8) No job can use files as temporary working storage, ...

(9) ... or for long term storage of information to be processed by succeeding jobs.

A remedy of these defects requires introduction of a "job description card" and a major revision of "execute one job"; and the overall system would be much more complicated than that described in this paper. It is to be hoped that the same structuring methods may be helpful in controlling the additional complexity by assisting in the achievement of some of the objectives of program structure listed at the beginning of this paper.

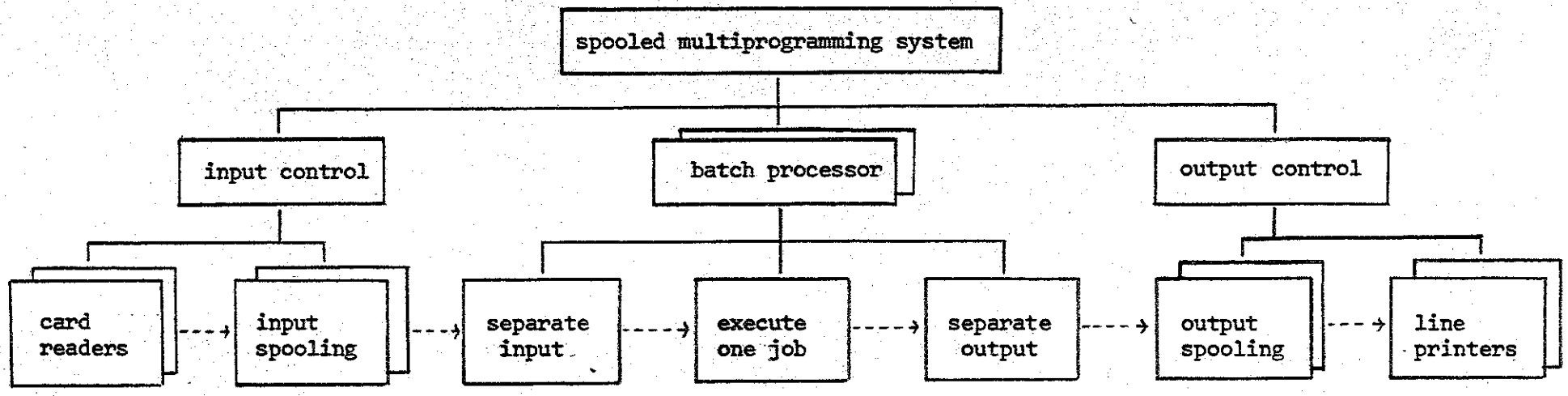
spooled multiprogramming system
input control
batch processor
output control
card readers
input spooling
separate input
execute one job
separate output
output spooling
line printers

Figure 2.

Acknowledgements

The work of the first author was supported by a Senior Fellowship of the Science Research Council of Great Britain.

The ideas reported arose from a research project into "A Model Operating System", also supported by the SRC; and have benefitted from an implementation of a similar approach by D.W. Bustard and S.A.J. Clarke.

They have also been improved by the useful advice of M.V. Wilkes, E.W. Dijkstra, C. Hewitt, M.K. Harper and D.W. Bustard.

References

1. Birtwhistle, G.M. Simula Begin. Auerbach, London, 1973.

2. Dijkstra, E.W. The structure of the T.H.E. multiprogramming system. C.A.C.M. 11, 5 (May 1968), 341-346.

3. Dijkstra, E.W. Co-operating sequential processes. In Programming Languages (F. Genuys, ed.), 43-112. Academic Press, London, 1968.

4. Hoare, C.A.R. Communicating sequential processes. Accepted for publication in Commun. A.C.M.

5. Hoare, C.A.R. The Structure of an Operating System. Draft May 1975 in "Language Hierarchies and Interfaces" Lecture Notes in Computer Science No.46 Springer 1976.