



Operating Systems

B. RANDELL, Editor

On Multiprogramming, Machine Coding, and Computer Organization

NIKLAUS WIRTH

Eidgenössische Technische Hochschule
Zurich, Switzerland*

The author feels that the interrupt feature which is available in most modern computers is a potent source of programming pitfalls and errors, and that it therefore may heavily contribute to the unreliability of programs making use of it. A programming scheme is presented which avoids the concept of the interrupt and permits the specification of concurrent (or pseudoconcurrent) activities in a supposedly more perspicuous manner. It is intended to serve as a basis for the construction of operating systems, which are prime examples of programs with concurrent activities. The scheme includes a set of basic instructions for the generation, termination, and synchronization of parallel processes. A set of routines representing these instructions and thereby simulating a hypothetical machine organization has been implemented and tested on the IBM System/360. Two programs using these instructions, written in PL360, are presented.

KEY WORDS AND PHRASES: multiprogramming, parallelism, interrupt, input-output, computer organization, file handling, PL360

CR CATEGORIES: 4.21, 4.22, 4.32, 4.41, 6.21

1. Introduction and Motivation

Most modern data processing systems consist of several machines which can operate concurrently, and most programs make use of many of these concurrently operating units. This is particularly the case for input and output (I/O) activities. It is therefore surprising that most programming languages do not reflect this basic fact and instead still convey the impression that programs consist of a single sequence of instructions. This situation can be explained as follows:

1. Concurrent execution is usually confined to input/output operations. The "programs" of I/O equipment are standard and need therefore not be specified in detail by every problem programmer. They can be supplied as system subroutines.

* Fachgruppe Computer-Wissenschaften

2. The difficulties arising with multiprogramming compared with uniprogramming (programming a single sequential algorithm) are such that every effort is made to save the programmer from having to specify concurrent operations.

On the machine code level, however, these difficulties cannot be circumvented: the activities of every unit must be specified in detail. Unfortunately this level offers the least safety against programming mistakes, because there is no redundancy as provided by higher level languages, which can be used by compilers to check against errors. It is therefore particularly desirable that at the machine code level the programmer be able to work on the basis of clear and systematic concepts of concurrent activities—their initiation, termination, and synchronization. Present computers hardly satisfy even low standards in this respect. In this paper a proposal is given for a machine organization (a set of computer instructions) conceived with these problems in mind.

The prospects for reliability of programs—and of large programs in particular—are considerably enhanced if the programmer restricts himself to tools which he is convinced he can master, if he avoids pitfalls, and if he disciplines himself to a clean style. It has become well known that, for the sake of reliability and clarity of programs, use of instructions which modify other instructions should be avoided. This recognition has even influenced the design of several computers (such as the B5500 and the IBM 360—subroutine return address deposited in registers instead of within the code, and "execute" instruction permitting the length of a variable length instruction to be computed and used through a register instead of by insertion in the actual instruction). An even more dangerous source of pitfalls is the concept of the *interrupt*. Its danger stems from the same cause as that of self-modifying programs: *The program which is actually being executed is not directly visible on the coding sheet.* The high degree of difficulty with interrupts is because an interrupt effectively consists of the insertion of one or several instructions at points of the program which can be neither foreseen nor reconstructed.

It is this impossibility to reconstruct the program being executed which makes it highly desirable to devise multiprogramming systems on the basis of thoroughly clear concepts (if not even on a mathematical theory admitting proofs) instead of using the common techniques of trial and error. A multiprogram—a program consisting of several concurrent activities—cannot be exhaustively "debugged" by the usual heuristic methods; it must be

designed according to principles which allow the deduction of a set of test cases whose successful application guarantees the program's correct behavior under all admissible circumstances. Furthermore it seems that any notation amenable to a technique of program verification must be based on the premise that the program executed is precisely that which is written on the paper (cf. also [9]).

The key of the proposal is thus the *elimination of the concept of interrupt*. Obviously, this concept will continue to exist in some form in the actual hardware, e.g. as communication pulses between the various cooperating processing units. The essential point is that it will not manifest itself to the programmer in this dangerous form. An analogy can be drawn to the elimination of the necessity to perform instruction (address) modifications through the introduction of the index registers: address modifications still occur in modern computers; the need for program self-modification has been eliminated by the introduction of instructions with a slightly higher complexity. A similar trade-off will be the basis of this proposal.

This paper is organized as follows. In Section 2, a set of machine instructions is specified which embody the new organization. These instructions are formulated as generally as possible, i.e. without making reference to any particular existing computer. A simple programming example is then given to demonstrate the use of these instructions. How this organization presents itself as incorporated in a machine code is demonstrated in Section 3. The scheme has been implemented on the IBM System/360 computer, and the *machine language* used in this presentation is PL360 [6]. The sample problem given in Section 1 is programmed in detail in PL360. It should be noted that at this level no reference to the particular I/O organization of the 360 computer is made, nor that knowledge about it is necessary. This knowledge is essential only in Section 4, where the detailed implementation of the subroutines which represent the new instructions is discussed. These subroutines will of course differ from one computer to another. But it should be kept in mind that the proposal suggests that these subroutines be incorporated in the hardware of future machines or, as suggested in Section 6, possibly be implemented in the form of microprograms. Finally, in Section 5 a more sophisticated application example of this multiprocess facility is presented, namely the programming of a sequential file system. This example is again written in PL360 and was tested on the 360 computer. The program is organized as a collection of subroutines which is conceived as being part of the run-time package belonging to a higher level language implementation.

2. Proposed Instruction Set

The proposed set is intended to replace the I/O instructions of the conventional type and to augment them with instructions which initiate, terminate, and synchronize parallel processes. It is necessary to define some terminology about parallel processes.

1. A distinction is made between *processors* (workers) and *processes* (jobs). Many processors can operate on a single process during consecutive periods of time, and vice versa—a single processor can operate on many processes, switching its attention from one to another at certain moments. A processor is in reality a unit of a computer system, e.g. a central processing unit, a card reader, or a tape drive; a process is a logical piece of a program, which at any instance contains a program point denoting the instruction currently being executed.

2. We never insist that processes *must* be executed by individual processors in real concurrency. We imply that the m parallel processes $P_1, \dots, P_m$, where each P_i consists of a sequence of instructions $P_i = P_{i1} P_{i2} \dots P_{in_i}$ may be executed in any sequence by any number of processors (even only one!), as long as P_{ij} precedes P_{ik} for all i and all $j < k$.

The proposed instructions are START, HALT, P, V and the generic DOIO. Any process under execution may *initiate* another process by the instruction

START(X) (1)

where the parameter X denotes the instruction sequence to be activated. The sequence ends with the instruction

HALT (2)

The instructions START and HALT effectively correspond to the instructions *fork* and *quit* as proposed by Conway [1] and Dennis and Van Horn [2].

Synchronization among concurrent processes is performed by means of variables common to those processes. *Dijkstra* has thoroughly investigated the problems arising from access of common variables (and the reader is encouraged to study his treatise at this point [3, 7]). He postulates two basic operations which constitute an interlocking mechanism, named P and V. These operations act on variables which assume integral values (0, 1, 2, ...). *Dijkstra* appropriately calls these variables *semaphores*, and it is subsequently postulated that P and V be the *only operations* applicable to variables designated as semaphores. (The observation of this postulate will be crucial for the implementation discussed below.) The operation

V(S) (3)

increments the value of the semaphore S by 1. The operation

P(S) (4)

can only be performed when the value of S is positive, which is decremented by 1. From this it follows that P may effectuate a delay of program execution until another process performs a V operation on the semaphore S. Thereby a synchronization of the two processes is obtained.

If semaphores are restricted to assume only the values 0 and 1, then the operators P and V correspond respec-

tively to the operators *lock* and *unlock* described in [2] and the TSL instruction of [8].

Input/output operations are all of the form:

DOIO(U,L) (5)

where the parameter U designates the I/O unit requested to perform, and L is a list of parameters specifying the operand(s), such as the storage address of the data to be transmitted, the size of the data block to be transmitted, and the location where status information is to be returned. The notable point is that such an instruction does not only initiate the data transfer, but performs it to completion strictly according to the principle of sequential program execution.

As a result, the programmer is encouraged and enabled to decompose his program into concurrent parts natural to its own inner logic instead of according to a system configuration typical for a particular installation.

A simple example, taken from [3], indicates how the given facilities can be effectively used. The sample program consists of two concurrent cyclic processes, named *Producer* and *Consumer*. The Producer generates a portion of data during each cycle and then puts this portion into a buffer (with a capacity of N portions). The Consumer in each cycle gets a portion from the buffer and then "consumes" it. (See Figure 1.)

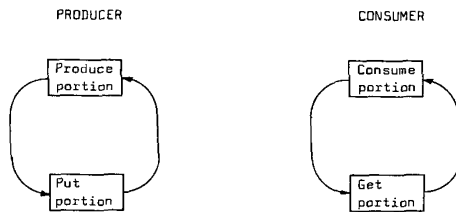


FIG. 1

It is obvious that these two (nearly independent) processes require some synchronization; namely the Producer has to wait whenever the buffer is full, and the Consumer has to wait whenever the buffer is empty. This waiting is effectuated by means of P operations on semaphores *f* and *e*, whose values denote the number of full and empty buffer portions respectively. (See Figure 2.)

The characteristic of the discussed scheme is that the form of the program does not depend on the details of the sections named "produce" and "consume." They may involve I/O operations or not; but if they do, the computer system can take advantage of possible concurrency in operating I/O units.

3. An Example Written in PL360

The outlined scheme has been implemented and tested on the IBM System/360 computer. PL360 was chosen as machine language, because it offers the possibility of exhibiting processes as program blocks (similar to ALGOL) due to its structural properties, and thus of *exhibiting the*

functional structure of a program by its textual structure. By convention, a process shall be written in the form of a procedure declaration as follows:

```

procedure (process identifier) (R1);
begin OPEN; (sequence of PL360 statements);
          CLOSE;
end

```

OPEN and CLOSE are calls of two-system subroutines whose use shall by convention be confined to occurrences in exactly this context. The postulated START instruction is then represented by a procedure statement which effectively activates the identified process. CLOSE corresponds to the postulated HALT instruction which had been renamed to exhibit the bracketing nature of the OPEN/CLOSE pair. The generic instruction DOIO is represented by an entire set of I/O instructions for operating card readers and punches, line printers, tapes, and the operator console. The individual instructions are listed below, accompanied by the specifications about the particular subroutine parameters to be transmitted in general purpose registers.

OPEN Register R1 contains the address of the instruction which called the process (i.e. which corresponds to a START instruction). R2 contains the address of a storage area of 8 double words. This area will be of no further concern to the programmer but serves the subroutines to store information about the status of the process; it is called the *process record*.

CLOSE terminates a process and has no parameters.

P R1 contains the address of the semaphore to be decremented.

V R1 contains the address of the semaphore to be incremented.

The following instructions are instances of DOIO. R0 contains the address of the block of data to be transmitted, R1 the number of bytes to be transmitted (except for READ, PUNCH, and WRITE, where the block-size is standard), and R2 contains the logical device

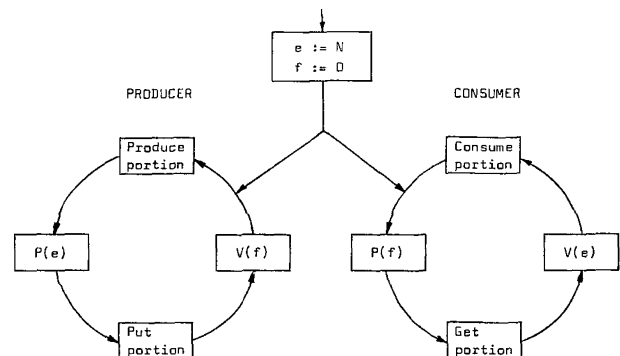


FIG. 2

number in the case of READTAPE and WRITETAPE.

READ read a card
 WRITE print a line
 PUNCH punch a card
 READTAPE read a tape record
 WRITETAPE write a tape record
 READTYPE read a message from the typewriter
 WRITETYPE write a message on the typewriter

Program 1 represents the Producer-Consumer example of the previous section. The Producer constitutes the only explicitly declared process; its "produce" section consists

```
BEGIN COMMENT PROGRAM 1. PRODUCER - CONSUMER ;
FUNCTION OPEN (0, #0A12), CLOSE (0, #0A13);
FUNCTION P (0, #0A10), V (0, #0A11);
ARRAY 8 LONG REAL PROCESSRECORD;
SHORT INTEGER E = 5;
SHORT INTEGER F = 0;
SHORT INTEGER LOCK = 0;
ARRAY 132 BYTE CONSUMERBUFFER = 132(" ");
ARRAY 80 BYTE PRODUCERBUFFER; COMMENT WORK AREA OF PRODUCER;
ARRAY 400 BYTE BUFFER; COMMENT 5 PORTIONS OF 80 BYTES;

PROCEDURE PRODUCER (R1);
BEGIN COMMENT THIS IS A PROCESS. R2 IS BUFFER POINTER;
OPEN; R2 := R4; R0 := @PRODUCERBUFFER;
L: READ;
  R1 := @E; P;
  IF R2 = R4 THEN R2 := @BUFFER(0) ELSE R2 := R2 + 80;
  MVC(79, MEM(R2), PRODUCERBUFFER);
  R1 := @F; V;
  CL1("%", PRODUCERBUFFER); IF "=" THEN GOTO L;
  R1 := @LOCK; V;
CLOSE;
END ;

R4 := @BUFFER(400); COMMENT POINTER TO THE LAST PORTION;
R2 := @PROCESSRECORD; PRODUCER; COMMENT ACTIVATE PRODUCER;
CONSUMER:
  R2 := R4; R0 := @CONSUMERBUFFER;
  R1 := @F; P;
  IF R2 = R4 THEN R2 := @BUFFER(0) ELSE R2 := R2 + 80;
  MVC(79, CONSUMERBUFFER, MEM(R2));
  R1 := @E; V;
  CL1("%", CONSUMERBUFFER); IF "=" THEN
  BEGIN COMMENT CONSUME DATA IN CONSUMERBUFFER;
  WRITE; -GOTO L;
  END ;
  R1 := @LOCK; P;
END .
```

PROGRAM 1

of a READ instruction referring to the card reader. The Consumer is represented by the main program (which itself can be regarded as the one permanent process); its "consume" portion consists of a WRITE instruction referring to the line printer. The processes terminate when an "end-of-file" is read, represented by a record with "%" in the first position.

4. Implementation on the 360 Computer

The proposed instructions have been implemented as a closed subroutine package on the 360 computer. Each subroutine is called by a supervisor-call instruction, thus permitting a program status change to the effect that all interrupts are masked during the execution of any such "instruction."

Every logical process is associated with a so-called process record, a storage area containing the relevant status information of the process. It can be considered as an extension of the 360's program status word (PSW).

The process record format is illustrated by Figure 3. The field named LINK serves to connect all (active) processes in a circular chain. A pointer called CP (current process) designates the process record belonging to the process currently being processed by the central processor.

This processor must *switch* to another process whenever (1) the current process must synchronize with another process, i.e. it executes a P operation on a nonpositive semaphore, or (2) it is incapable of further serving the current process and must delegate it to another processor. This is the case for I/O transfers.

Process switching includes the following operations:

1. The PSW is saved in the PSW-field of the current process record;
2. Registers R0-R3 and R15 are saved in the REG-field of the process record (R4-R14 are considered as "global variables");

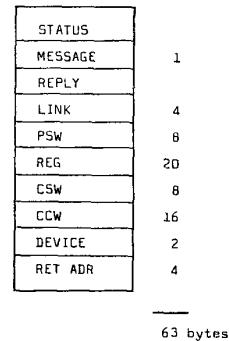


Fig. 3

3. Passing through the process record chain, a search is made for any process ready for resumption; if none is found, the processor enters the wait-state;
4. CP is set to point to the new process record;
5. Registers R0-R3 and R15 are loaded from the new process record;
6. The processor's PSW is loaded from the record's PSW field.

The search for a process ready for resumption is made by inspection of a process record field named STATUS with the following possible values:

- 0: the process is available for resumption (it is "free"),
- 1: the process is waiting on a semaphore,
- 2: the process is being serviced by an I/O processor (channel).

It must be noted that the organization of the active processes in one circular chain does not necessarily represent an optimal switching strategy. It might be more advantageous to arrange the processes in several queues according to their status, particularly in systems where the number of processes is assumed to be large. It is not the subject of this article to investigate various process-switching strategies, but it is to be pointed out that any strategy could be incorporated in the discussed scheme, and that the circular method has been chosen merely for its obvious simplicity. (Moreover, it proved to be the most efficient one when the number of processes is relatively small.)

4.1. INSTRUCTIONS OPEN AND CLOSE. The OPEN instruction inserts the storage area designated by the

pointer in R2 into the chain of active process records. Then the status of the current process (which activated the new process) is saved and CP is set to the new process, whose execution is thereby initiated. The CLOSE instruction merely deletes the current process from the chain. (Note that no storage release is involved.)

4.2. INSTRUCTIONS P AND V. It is absolutely crucial that a scheme is used which does not necessitate a repeated polling of semaphores on which processes are waiting. Such a scheme is realizable through the restriction that P and V be the only operations applicable to semaphores, because then only a V operation can cause the resumption (waking-up) of a process waiting on a semaphore. In this implementation, the semaphore values are allowed to become negative; a negative value $-n$ then signifies that n processes are executing a P operation and are consequently waiting for the "reopening" of that same semaphore. The chosen implementation of P and V is illustrated by Figure 4. The symmetry of the two operations should be noticed.

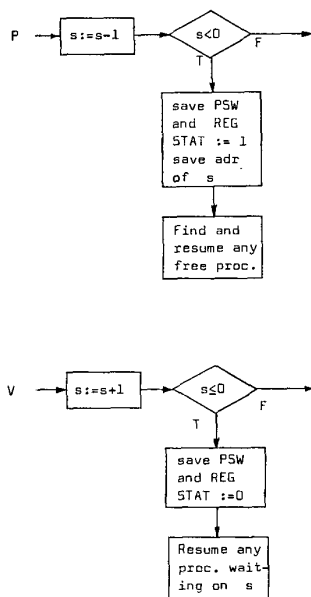


FIG. 4

4.3. GENERIC INSTRUCTION DOIO. While the previous paragraphs made only scant reference to a particular machine, the implementation of DOIO strongly depends on the object computer.

The 360's I/O scheme is based on the premise that data transfers are executed by channels which execute their own programs (CCW chains). This execution is initiated by a single central processor instruction (SIO). Termination of the transfer causes an interrupt of the central processor. This situation requires that the DOIO instruction first construct an appropriate program for the

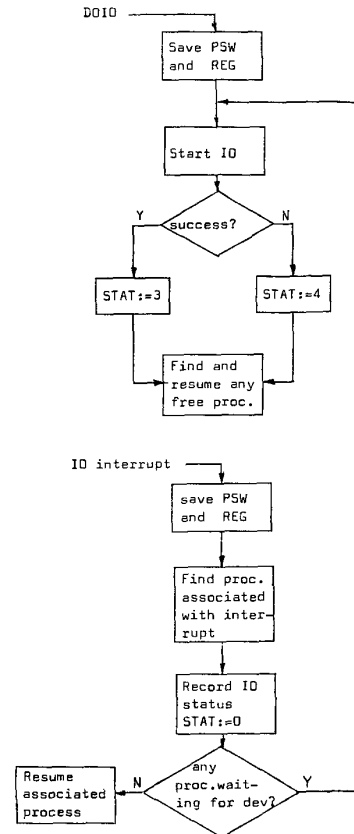


FIG. 5

designated channel according to its parameters, whereupon that channel is activated. The central processor is then switched to any other free process. If the channel is found to be already engaged, then this is marked in the process record by a STATUS-field value of 3, implying that the data transfer has to be reattempted whenever the designated channel signals termination of its operation. After such a termination, the process which had specified the I/O operation is resumed by the central processor. This scheme is illustrated in Figure 5.

An extension of this simple algorithm is necessitated by three circumstances:

1. A single I/O operation can give rise to more than one interrupt. This is the case e.g. when the data channel dissociates itself from the I/O operation before the device has terminated its execution. Examples are the operation of hardware buffered card readers or line printers, or the rewinding of magnetic tapes. In the latter case, three interrupts may be generated, the first being associated with the channel, the second with the tape controller, the third with the tape drive.
2. An SIO instruction may, under certain circumstances, clear a pending interrupt. This case is identified by specific channel status information accompanied with an indication that the new operation has not been started.
3. I/O operations may fail to be initiated because a

While the first two cases may be considered as being due to the specific characteristics of the System/360, the third one is an inherent phenomenon with which every system has to cope. After closer inspection, however, it turns out that case 1 is also of a general nature, since often several parts of the electronic equipment participate in the execution of an I/O action, and may dissociate themselves at different moments. Nevertheless, the solution adopted to cope with points 1 and 2 shall be treated separately from point 3. The enlarged DOIO routine is outlined by Figure 6. It is essential to know that the identification of the nature of the interrupt (channel, control unit, or device end) is provided by a so-called channel status word (CSW), which in System/360 is stored at a fixed location of core whenever an I/O interrupt occurs.

both cases the (human) operator has to be notified. In the first case the message could most conveniently assume the form of an acoustical or optical signal originating at the selected device; the operator replies by making the device ready. In the second case, either the operation has to be automatically retried or a message is sent to the operator and the process is delayed until he provides a reply. Communication with the operator is usually performed by a console typewriter (or display). The situation becomes complicated because several processes may wish to communicate with the console simultaneously. Thus the routine for DOIO has to be enlarged into a small program which decodes the status of I/O devices and accordingly performs another DOIO on the console. It seemed most desirable that this recursivity be avoided. The solution was found in establishing a *permanet process* in charge of communicating with the human operator. This process, called *Operator*, has access to the status information of all processes.

4.4. OPERATOR PROCESS. Synchronization between the processes having to communicate with the operator and the Operator is established by a semaphore called OS. As far as the processes are concerned, it assumes the values 1 (Operator ready) and 0 (Operator busy). Actually, a negative value of $-n$ denotes the number of processes waiting to be serviced by the Operator. When active, the Operator searches through the chain of process records to find any one requiring attention, which is indicated by a status value of 4. The Operator then inspects the I/O status field of the relevant process and communicates with the operator console accordingly, either requesting a reply or not. In the case of no reply (device not ready), the concerned process is reactivated when an interrupt signal originates from the relevant device due to the operator's pressing the READY button. In the case of a malfunction, the operator's reply determines whether the process is being reactivated by the Operator process (by changing its status), or whether it is being closed. Figure 7 illustrates the entire DOIO routine, and Figure 8 is the diagram of the cyclic process Operator.

The subroutine package was written and tested using OS/360 Assembly code, and the approximate storage requirements are:

PROGRAMS

PL360 I/O routines	300
P and V operations	100
DOIO and interrupt handling	400
OPERATOR	200
OPEN and CLOSE	200
	1200 bytes

DATA

two process records	128
variables and constants	<u>200</u>
	328 bytes

It is worth noting two details:

1. The routine which finds free processes must check

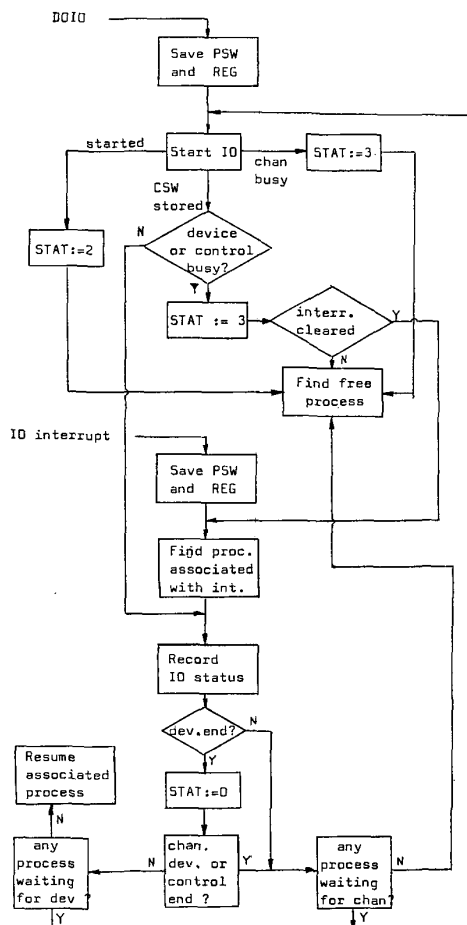


Fig. 6

whether an operator's reply has been attached to the process about to be resumed, and it must interpret such a reply. Two replies are possible in the present implementation: (1) cancel the process, and (2) retry the attempted I/O operation. The execution of a retry may vary in details depending on the nature of the I/O operation.

2. The operator's console may be used as an I/O device by several processes. It is therefore essential that the reply be uninterruptably connected with the message. To this end, a primitive I/O operation called *Typeoutin* is provided additionally (it results in the chaining of a write-CCW and a read-CCW). The operator process therefore waits for a reply as soon as a message is typed out, and no other message can be sent until the reply has been received. This seems to be a sensible philosophy since the human operator constitutes a single sequential process, too, and is better served if, at any time, at most one action is requested from him. If this philosophy is rejected, then it is necessary to alter the structure of the

Operator process, and the human operator is forced to prefix every reply with an identification of the logical process to which the reply is directed. This prefix must then be interpreted by a message-switching routine which is part of DOIO.

5. An Application: Sequential Tape Files

5.1. DECLARATIONS AND OPERATIONS ON FILES. Sequential tape files with constant record length, subsequently simply to be called "files", occur in almost every commercial data processing application, and computers have, on purpose, been geared to allow tapes to be operated in parallel with the central processor to handle this class of applications efficiently. In this section a proposal is made on how sequential tape files could be incorporated as a type of data structure in a high level programming language, and how an implementation can be organized on a computer with the proposed parallel processing facilities. The purpose is to demonstrate the effects of this organization on the conceptual clarity and consequent trustworthiness of programs, regardless of whether they are "user programs," or "system programs" such as the file-handling package to be described below.

The language in which the file-handling facilities are to be embedded is described in [4]. It contains a record class declaration by which the structure of a record (contiguous block of storage) can be defined. The record class declaration provides a proper basis for describing files which can be considered as a finite sequence of records of a specified class [5]. A file declaration defined by the following syntax

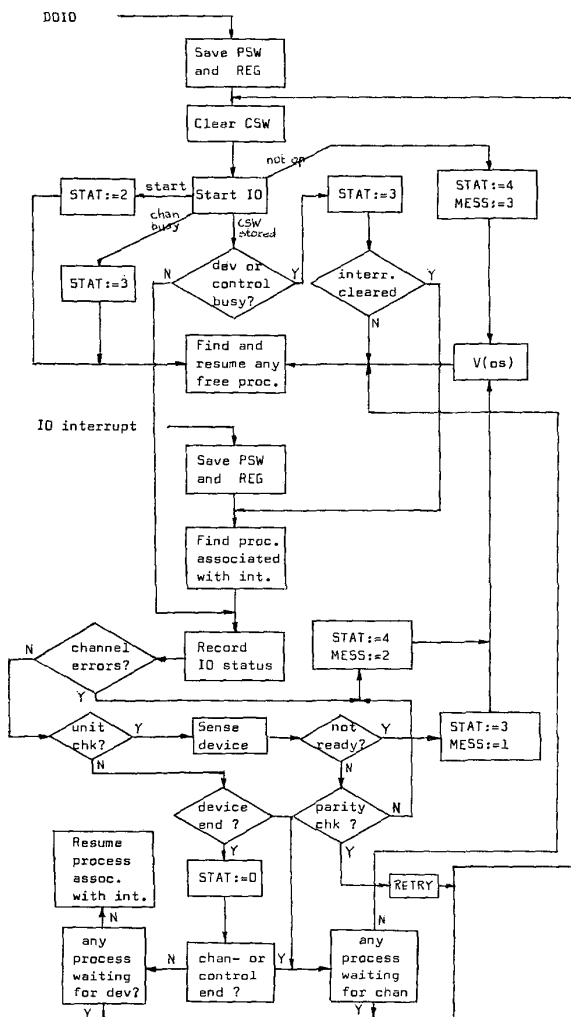


Fig. 7

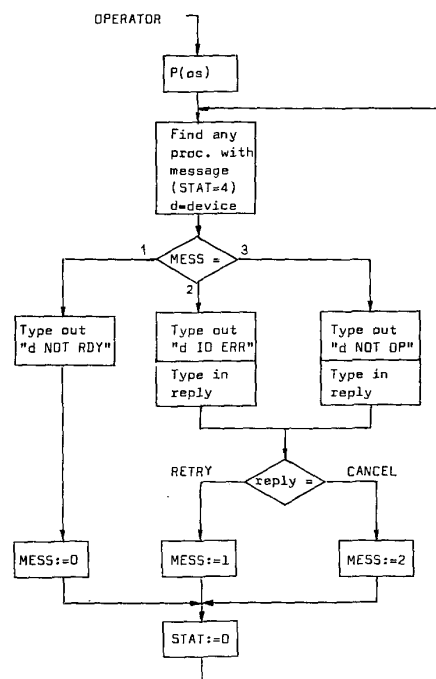


Fig. 8

augments the syntax of [4]:

```

<declaration> ::= <file declaration>
<file declaration> ::= <record class identifier> file <identifier>
    <<file parameters>>
<file parameters> ::= <file label>, <file mode>, <block factor>,
    <buffer factor>, <disposition>
<file label> ::= <string>
<file mode> ::= input | output
<block factor> ::= <unsigned integer number>
<buffer factor> ::= <unsigned integer number>
<disposition> ::= release | keep | save

```

Examples of record class declarations are:

```

record Person (string (20) name; integer age; Boolean male)
record Town (integer population; real area; string (30) name)

```

Examples of file declarations are

```

Person file Johnson ('XY25Z', input, 8, 4, release)
Town file Calaveras ('Calif. 98765', output, 10, 1, save)

```

Opening of a file is performed on entry to the block containing the file declaration; closing upon exit from that block.

It is characteristic of a sequential file that at any instance one and only one record is readily accessible. It is therefore possible to use the name of the file itself to denote that specific record, i.e. to identify by the file name an implicit pointer to the currently accessible record. This is achieved by using the file identifier as a reference in field designators (cf. [4, Sec. 6.7]):

```

<simple reference expression> ::= <file identifier>

```

where a file identifier is understood to be an identifier which was declared to denote a file.

Examples of field designators referring to a file are:

```

age (Johnson)
name (Johnson)
population (Calaveras)

```

In order to gain access to records sequentially, a primitive operation *next* is introduced, which causes the file pointer to be moved to the next record in sequence. Operations on the file pointer are represented by a simple statement and may also include actions such as setting the pointer to the beginning of the file (*rewind*).

```

<simple statement> ::= <file operator> <file identifier>
<file operator> ::= next | rewind

```

5.2. IMPLEMENTATION OF FILES. It is essential that the implementation should ensure utmost efficiency in handling magnetic tapes. It is therefore mandatory that

(1) records, once transferred from magnetic tape into core storage (or vice versa) are not again copied from one location to another (which is common practice in current operating systems providing buffered files),

(2) the length of blocks of information can be adjusted to obtain a reasonable block length for information transfers, so that the information density on tape is adequate,

(3) advantage can be taken of the parallel operation of I/O channels,

(4) storage allocation occur only once, namely when the block is entered where the file is declared.

The approach subsequently described satisfies these requirements. (It is assumed that the records contain no fields of type **reference**.) Upon block entry two areas of storage are allocated: one called *filedescriptor*, in which all information relevant to the file is contained; and the other called *buffer*, in which selected records of the file are stored. The size of the buffer is computed as

$$\text{recordsize} * \text{blockfactor} * \text{bufferfactor}$$

The size of a logical record is deduced by the compiler from the record class declaration. The file-handling routines are organized so that each record on tape consists of n logical records, where n is the specified blockfactor. A tape record is called a *block*. It should be noted that n is determined by the programmer through the file declaration. A large number of n assures high information density on tape due to a reduced number of interrecord gaps, but increases the necessary core buffer size. The file descriptor contains the information provided by the file declaration. In addition, several fields are provided for the variables on which the file routines operate. One field holds the process record of the process which performs the reading from or writing on the tape, depending on whether an input file or an output file is considered.

In Program 2 the details are specified. The main program illustrates a short compiled ALGOL program, and it constitutes the last two blocks. The code corresponding to the file opening upon block entry, the next-statements, and the file closings upon block exit, is represented by procedure calls to the appropriate system subroutines, supposedly belonging to an ALGOL run-time package. As subroutine parameter, register R3 contains the address of the relevant file descriptor.

The file-handling procedures are constructed so that they do not contain local quantities, except for those stored in the (automatically local) registers R0-R3. As a consequence, the same routines can be called by many processes independently; in particular the file-reader and file-writer programs can represent many concurrent processes which operate on individual files. (PL360 programs are code-invariant, and therefore reentrant.) We stress here the distinction between *programs*, which are pieces of text, and *processes*, which constitute computational activities.

Admittedly, the present "operating system" is primitive: Upon opening of a file, the (human) operator receives the message

```

MOUNT <file label>

```

and has to respond by typing the logical number of the unit on which he has mounted the tape; upon file closing he receives the message

```

RELEASE <file label>

```

or

```

KEEP <file label>

```

or

```

SAVE <file label>,

```

```
BEGIN COMMENT PROGRAM 2. FILE HANDLING ;
```

```
COMMENT LAYOUT OF FILE DESCRIPTOR. LENGTH = 128 BYTES;
INTEGER BUFORG SYN B3( 0); COMMENT POINTER TO BUFFER;
ARRAY 8 BYTE MESSAGE SYN B3( 4);
ARRAY 16 BYTE LABEL SYN B3(12);
SHORT INTEGER MODE SYN B3(28); COMMENT FILE LABEL;
RECLENG SYN B3(30); COMMENT NO. OF BYTES / RECORD;
BLKFAC SYN B3(32); COMMENT NO. OF RECORDS / BLOCK;
BUFFAC SYN B3(34); COMMENT NO. OF BLOCKS IN BUFFER;
DISP SYN B3(36); COMMENT 1=INPUT, 2=OUTPUT;
DEVICE SYN B3(38); COMMENT LOGICAL DEVICE NUMBER;
BLKLENG SYN B3(40); COMMENT NO. OF BYTES / BLOCK;
N1 SYN B3(42); COMMENT SEMAPHORE;
N2 SYN B3(44); COMMENT SEMAPHORE;
LOCK SYN B3(46); COMMENT LOCK USED FOR CLOSING;
INTEGER RECPNT SYN B3(48); COMMENT POINTER TO CURRENT REC;
NEXT1 SYN B3(52); COMMENT POINTER TO NEXT BLOCK;
NEXT2 SYN B3(56); COMMENT POINTER TO NEXT BLOCK;
BUFEND SYN B3(60); COMMENT POINTER TO BUFFER END;
ARRAY 64 BYTE PROCREC SYN B3(64);
```

```
PROCEDURE FILEREADER (R1);
COMMENT INPUT PROCESS FOR TAPE-FILE. R3 = @FILEDESCRIPTOR;
BEGIN R2 := @PROCREC; OPEN;
R2 := DEVICE; REWIND;
L: R1 := @N2; P;
R0 := NEXT2; R1 := BLKLENG; READTAPE;
IF THEN
BEGIN R0 := R0+R1; IF R0 = BUFEND THEN R0 := BUFORG;
NEXT2 := R0;
R1 := @N1; V; GOTO L;
END ;
REWIND;
R1 := @LOCK; V;
CLOSE;
END ;
```

```
PROCEDURE FILEWRITER (R1);
COMMENT OUTPUT PROCESS FOR TAPE-FILE. R3 = @FILEDESCRIPTOR;
BEGIN R2 := @PROCREC; OPEN;
R2 := DEVICE; REWIND;
L: R1 := @N2; P;
R1 := NEXT2; CLI(#E0,B1);
IF THEN
BEGIN R0 := R1; R1 := BLKLENG; WRITETAPE;
R0 := R0+R1; IF R0 = BUFEND THEN R0 := BUFORG;
NEXT2 := R0;
R1 := @N1; V; GOTO L;
END ;
MARKTAPE; REWIND;
R1 := @LOCK; V;
CLOSE;
END ;
```

```
PROCEDURE NEXT (R1);
COMMENT MOVE "RECPNT" TO THE NEXT RECORD IN THE FILE. R3 = @FILEDESCR;
BEGIN STM(R0,R1,MESSAGE);
R0 := RECPNT + RECLENG;
IF R0 = NEXT1 THEN
BEGIN COMMENT THE NEXT RECORD BELONGS TO THE NEXT BLOCK;
R1 := @N1; P;
IF R0 = BUFEND THEN R0 := BUFORG;
R1 := R0 + BLKLENG; NEXT1 := R1;
R1 := @N2; V;
END ;
RECPNT := R0;
LM(R0,R1,MESSAGE);
END ;
```

```
PROCEDURE OPENFILE (R9);
COMMENT R3 = ADDRESS OF DESCRIPTOR OF FILE TO BE OPENED.
THE FOLLOWING FIELDS MUST HAVE GIVEN VALUES:
BUFORG, LABEL, MODE, RECLENG, BLKFAC, BUFFAC, DISP ;
BEGIN COMMENT TYPE OUT FILE LABEL, THEN ASK DEVICE NUMBER;
FUNCTION NC(5,#0400);
MVC(7,MESSAGE,"MOUNT: ");
R0 := @MESSAGE; R1 := #00010018; R2 := @DEVICE(1); TYPEOUTIN;
NC(1,DEVICE,#000FS); COMMENT REQUEST DEVICE NUMBER;
R0 := 0; LOCK := R0;
R0 := RECLENG * BLKFAC; BLKLENG := R0;
R0 := BLKLENG * BUFFAC + BUFORG; BUFEND := R0;
R0 := BUFORG; RECPNT := R0; NEXT2 := R0;
R0 := R0 + BLKLENG; NEXT1 := R0;
```

```
R0 := 0; R1 := BUFFAC; R2 := MODE;
CASE R2 OF BEGIN
BEGIN N1 := R0; N2 := R1; FILEREADER;
END ;
BEGIN N2 := R0; N1 := R1; FILEWRITER;
END ;
END ;
R1 := @N1; P;
END ;
PROCEDURE CLOSEFILE (R9);
BEGIN
R2 := MODE;
CASE R2 OF BEGIN
NULL;
BEGIN COMMENT GENERATE EOF RECORD;
R1 := RECPNT; MVI(1,R1); NEXT;
R1 := @N2; V; R1 := @N1; P;
R1 := NEXT1; IF R1 = BUFEND THEN R1 := BUFORG; MVI(#E0,B1);
R1 := @N2; V;
END ;
END ;
R1 := DISP;
CASE R1 OF BEGIN
MVC(7,MESSAGE,"RELEASE ");
MVC(7,MESSAGE,"KEEP ");
MVC(7,MESSAGE,"SAVE ");
END ;
R0 := @MESSAGE; R1 := 24; WRITETYPE;
R1 := @LOCK; P;
END ;
```

```
BEGIN COMMENT MAIN PROGRAM BLOCK 1 ;
ARRAY 16 LONG REAL OUTFILEDES;
ARRAY 128 INTEGER OUTFILEBUF;
```

```
OUTFILEDECLARATION:
R3 := @OUTFILEDES;
R0 := @OUTFILEBUF; BUFORG := R0;
R0 := 2; MODE := R0;
R0 := 32; RECLENG := R0;
R0 := 8; BLKFAC := R0;
R0 := 2; BUFFAC := R0;
R0 := 1; DISP := R0;
MVC(15,LABEL,"ANY SCRATCH TAPE");
OPENFILE;
```

```
WRITEFILE:
FOR R5 := 1 STEP 1 UNTIL 729 DO
BEGIN R3 := @OUTFILEDES;
COMMENT ASSIGN VALUES TO THE RECORD DENOTED BY "RECPNT" ;
NEXT;
END ;
R3 := @OUTFILEDES; CLOSEFILE;
END ;
```

```
BEGIN COMMENT MAIN PROGRAM BLOCK 2 ;
ARRAY 16 LONG REAL INFILEDES;
ARRAY 256 INTEGER INFILEBUF;
ARRAY 132 BYTE LINE = 132(" ");
LONG REAL DEC;
```

```
INFILEDECLARATION:
R3 := @INFILEDES;
R0 := @INFILEBUF; BUFORG := R0;
R0 := 1; MODE := R0;
R0 := 32; RECLENG := R0;
R0 := 8; BLKFAC := R0;
R0 := 4; BUFFAC := R0;
R0 := 3; DISP := R0;
MVC(15,LABEL,"SAME TAPE-FILE ");
OPENFILE;
```

```
R5 := 0;
READFILE:
R3 := @INFILEDES;
COMMENT PROCESS RECORD DENOTED BY "RECPNT" ;
R4 := RECPNT; CLI(1,MEM(R4));
IF THEN BEGIN R5 := R5+1; NEXT; GOTO READFILE; END ;
CVD(R5,DEC); MVC(23,LINE,"---/--- RECORDS PROCESSED");
ED(5,LINE,DEC(5)); R0 := @LINE; WRITE;
R3 := @INFILEDES; CLOSEFILE;
END ;
END .
```

PROGRAM 2

according to the disposition of the file. It would be most desirable if the system would itself maintain knowledge about the whereabouts of kept tape files, particularly in case of scratch files and of free tape units. To this end, the file opening and closing routines must have access to a permanent (global) table with one entry per tape unit. Simultaneous use of these routines by several processes must as a consequence be prohibited. Their mutual ex-

clusion can be achieved by declaring a global semaphore

```
SHORT INTEGER MUTEX = 1
```

and incorporating the statements

```
R1 := @MUTEX; P
```

and

```
R1 := @MUTEX; V
```

at the beginning and end respectively of *openfile* and *close-*

file. This example displays the fundamental problem of providing security from mutual interference of processes which access common data. The semaphores represent a necessary and sufficient mechanism for the solution of this problem [3].

6. Concluding Remarks

1. The scheme of linking all records of active processes together in a single circular chain is appealing because of its simplicity, and sufficiently efficient as long as only relatively few processes are to be governed. If many processes exist in a system, a scheme which groups process records into separate chains would be desirable for reasons of efficiency in process switching. One circular chain would contain all free (ready) processes, while to each semaphore and each I/O device would be associated a (possibly empty) chain containing all processes waiting for that semaphore or device. It is evident, however, that the appending and disconnecting routines would be relatively complicated. On System/360, the situation is particularly tricky, because a process could at the same time be waiting for (availability or completion of) a channel, a control unit, and a device, and moreover because a control unit is not necessarily identifiable. It supplies a completion signal only if it dissociates itself from the I/O operation at a time different from that of either channel-termination or device-termination, i.e. only in rather unusual circumstances. The possibility of maintaining a process queue for control units in System/360 is therefore doubtful.

2. "Time-sharing" systems are examples of cases with a relatively large number of processes (namely at least one per console). A "time-sharing" system is a special case of a multiprocess system, where it is guaranteed that every process is serviced by a processor at least once within a given period of time. In order to be able to host a time-sharing system, the presented multiprocess system nucleus would have to be augmented by a real-time clock causing a process switch to occur at least once within a given time interval. It should be noted that the present system nucleus does not include any kind of storage management. A time-sharing system relying on dynamic allocation and/or swapping techniques would require that the present nucleus be enlarged by a set of storage management routines to be activated each time a process switching is initiated.

3. During the experimental implementation on System/360 (Model 40) it was realized that the organization of the actual hardware is entirely different from, and in some aspects more convenient than the one made visible to the programmer via the built-in microprogram. It seems that the proposed scheme is particularly attractive for microprogrammed computers, whose underlying hardware may still contain the conventional features of "start I/O instruction" and interrupt, or even more primitive I/O interaction facilities. On such machines, it is recommended that the central parts of the described scheme, namely the process switching, the DOIO, and the Operator routines be implemented completely by microprogram. In this case the following hardware instructions would facilitate

their implementation:

- $p(x)$ decrement the contents of x by 1; trap to a given, fixed location, if the result is negative.
- $v(x)$ increment the contents of x by 1; trap to a given fixed location, if the result is not positive.

As usual in the case of a trap, the present machine status would be stored in another fixed location. Obviously, these microinstructions could also be used conveniently in the program for the P and V macroinstructions.

4. The scheme can most easily be adapted to a true multi-CPU configuration, where all processors (with equal capabilities) have entirely equal status. Every processor owns a CP pointer, and the process-switching routine must be made a "critical section" [3] which can be executed by, at most, one processor at any time.

5. The presentation of parallel activities in the form of textually separate sequential programs is also natural for higher level languages. Questionable, however, is the appropriateness of the semaphores and their P and V operations as synchronizers. Experience indicates that they may be too primitive for convenient use. Synchronization of loosely coupled concurrent processes naturally consists of the interchange of messages (as in ordinary life). The synchronizing facility is then a message file (or queue), and the operations consist of adding messages by one process at the beginning of the queue and of subtracting messages by the other process at the end of the queue.

It is interesting that the semaphore can be regarded as a file containing null messages. The V operation adds a null message, and the P operation subtracts a null message (and implies delay if the file is already empty). The semaphore value is merely a count of the number of null messages in the "file." Viewed in this light, they again emerge as appropriately chosen synchronizing primitives for the low level of machine code programming, which lend themselves to a natural generalization in the realm of high level languages.

RECEIVED JUNE, 1968; REVISED FEBRUARY, 1969 AND MARCH, 1969

REFERENCES

1. CONWAY, M. E. A multiprocessor system design. Proc. AFIPS 1963 Fall Joint Comput. Conf., Vol. 24, Spartan Books, New York, pp. 140-146.
2. DENNIS, J. B., AND VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Comm. ACM* 9, 3 (Mar. 1966), 143-155.
3. DIJKSTRA, E. W. *Cooperating Sequential Processes*. Technical U. Eindhoven, Netherlands, 1966.
4. WIRTH, N., AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM* 9, 6 (June 1966), 413-432.
5. HOARE, C. A. R. File processing. *Algol Bull.* AB25.3.3, Mar. 1967.
6. WIRTH, N. "PL360—A programming language for the 360 computers. *J. ACM* 15, 1 (Jan. 1968), 37-74.
7. DIJKSTRA, E. W. The structure of the "THE"-Multiprogramming system. *Comm. ACM* 11, 5 (May 1968), 341-346.
8. LAMPSON, B. W. A scheduling philosophy for multiprocessor systems. *Comm. ACM* 11, 5 (May 1968), 347-360.
9. DIJKSTRA, E. W. A constructive approach to the problem of program correctness. *BIT* 38, (Dec. 1968), 174-186.