Multiprogramming Coordination*

LEON PRESSER

Department of Electrical Engineering and Computer Science, University of California, Santa Barbara, California 93106

This is a tutorial paper on the coordination of parallel activities It commences with an overview of multiprogrammed operating systems that uncovers an architecture consisting of cooperating, but competing processes working in parallel This is followed by a formal treatment of processes, and an analysis of the fundamental coordination needs of concomitant processes. The analysis leads to a set of two coordination primitives originally defined by Dijkstra. In the rest of the paper, an evolutionary series of examples of increasing coordination complexity is formulated and solved. As the various examples are discussed, cumulative extensions to the original set of coordination primitives are justified and formally defined.

Keywords and Phrases: multiprogramming, operating systems, multiprocessing, coordination, synchronization, parallelism, processes, architecture.

CR Categories: 4.32, 4.35, 6.20

. . . Suffering, as I am, from the sequential nature human of communication.

—*E. W. Dijkstra* [7]

INTRODUCTION

The trend in the design of modern computer (operating) systems is to effect a high degree of parallelism, that is, to maintain as many system components as possible working concurrently. Consequently, an important design criterion is to produce an architecture that facilitates parallel operation. This objective requires an integrated (hardware and software) system design approach. In particular, it requires the identification of a set of operations (i.e., primitives) that are to serve as the fundamental building blocks for the implementation of a specific architecture.

*This work was supported in part by the National Science Foundation Grant GJ-31949.

In this tutorial paper we examine such fundamental operations in an evolutionary form that leads us to a relatively high-level set of primitives for the coordination of parallel activities.

We commence with an overview of modern operating systems that uncovers an architecture consisting of cooperating, but competing activities working in parallel. This is followed by a formal treatment of the basic concept of processes in order to provide a framework for the discussion of concomitant activities in the rest of the paper. In particular, multiprogramming systems are placed in perspective in this section. Next, an analysis of the fundamental coordination needs of parallel processes is presented, and

CONTENTS

a well-known set of coordination primitives is defined. In the next section three classical coordination problems that illustrate the use of the primitives are postulated and solved. In the remaining sections of the paper an evolutionary series of problems of increasing coordination complexity is formulated and solved. As the various problems are discussed, cumulative extensions to the original set of coordination primitives are justified and formally defined. Finally, the overall paper is summarized and pertinent references are listed.

1. OPERATING SYSTEM OVERVIEW

In order to create a framework for the treatment of the coordination needs of modern operating systems we display a functional overview of an operating system in Figure 1. In that diagram a cloud represents a functional component, while a rectangle represents a data structure. Let us discuss the figure in some detail.

A computer system receives directives for the execution of jobs through *job control language* statements, possibly from multiple consoles. A typical job directive would be *compile*, *load* and *execute* (i.e., *go*). This directive includes an aggregate of information which contains the name of the compiler needed as well as the location of the source inputs and the desired repository for the outputs. The incoming sets of independent job directives are placed in a System Job List as indicated on the left of Figure 1. A job directive set is transformed, through the Process Creator, into a collection of processes, capable of parallel operations, whose completion would correspond to the completion of the job from which they originated. These processes, which represent refined units of work, are listed in the System Ready List. For example, the compile-load-execute directive may be transformed into the following group of processes: load compiler, load source program, execute compiler, load data, execute object program. Each one of these processes may, during its execution, give rise to other processes. For instance, load compiler would lead to the creation of a process to find main memory

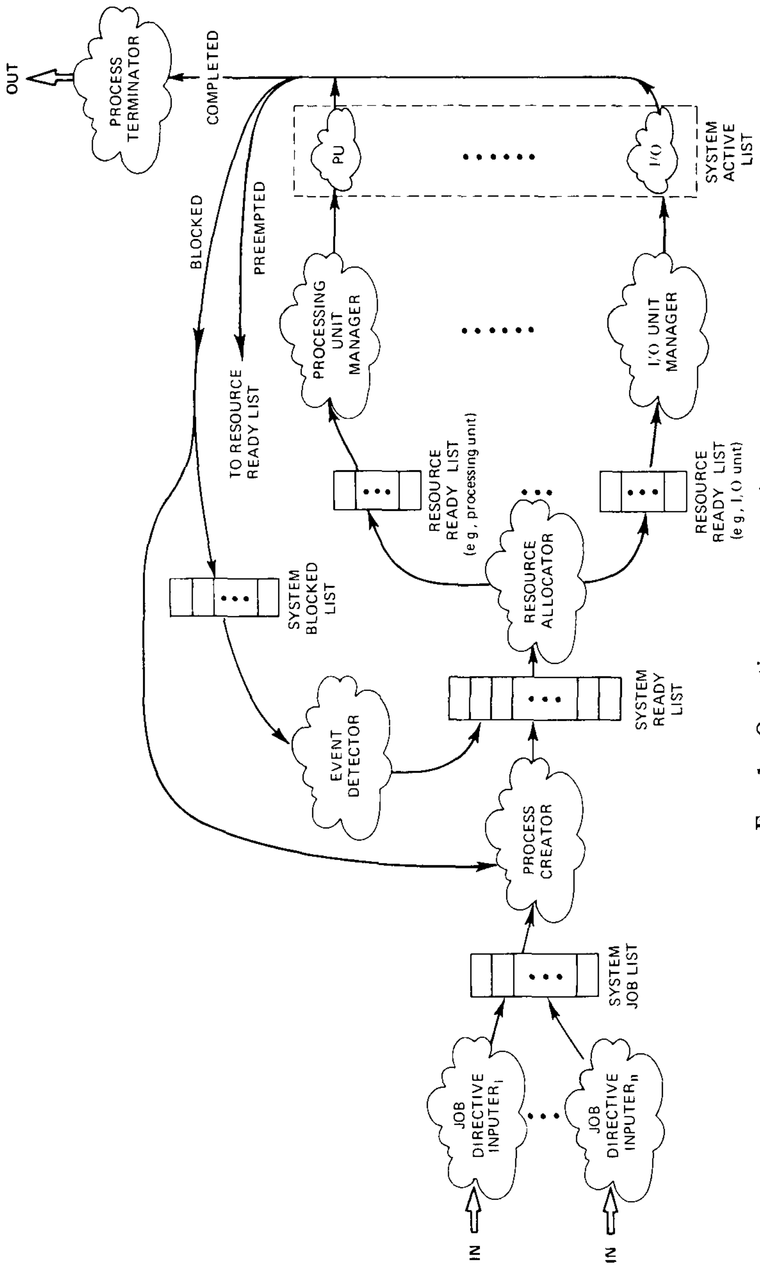

FIG 1. Operating system overview.

space for the incoming compiler. Furthermore, each one of these processes needs to have appropriate resources allocated to it before it can execute.

It is the responsibility of the Resource Allocator to manage and implement a (global) resource allocation policy. In order to do so it needs to call on an inventory of processes such as a deadlock prevention process. The Resource Allocator, which may also be viewed as a Process Allocator, selects processes from the System Ready List and places them on specific Resource Ready Lists. For each resource there is a Unit Manager process whose responsibility is to manage the resource. These managers select processes from their associated Resource Ready List for execution on the resource. It is convenient, as will be seen later, to view all the actively executing processes in the system as part of a System Active List. Processes executing on certain resources such as processing units may be preempted by the Unit Manager, in which case they are returned to the appropriate Resource Ready List. Often a process in execution cannot proceed any further until some other process completes its work. For example, a process executing on a processing unit and reading its input data from a buffer must wait when the buffer is empty for some input process to place new data in the buffer. In such a situation it is not efficient to let the processing unit idle while the input operation is taking place. Rather, the Unit Manager moves the blocked process to the System Blocked List, and gives the processing unit to another process that can use it. If the input process has not been previously created it is necessary not only to make an entry in the System Blocked List, but also to ask the Process Creator to generate the needed input process. The operating system contains a systemwide Event Detector process whose responsibility it is to detect the occurrence of any event being waited for by processes in the System Blocked List. When such events occur the Event Detector moves the appropriate processes from the System Blocked List to the System Ready List. Eventually all processes should complete and release their resources through the Process Terminator, as indicated on the right of Figure 1. Note that the Process Terminator may also be viewed as a Resource Releaser.

In addition to these functional components, modern operating systems include processes for purposes of providing: protection, clock services, performance monitoring, accounting, recovery procedures, and utilitarian (e.g., listing the contents of a file) services.

Processes will be formally defined in the next section. It must be clear, however, that processes may be executed in parallel. Referring to Figure 1, and our previous discussion, we see that processes (i.e., clouds in the figure) display two basic types of *natural* coordination needs:

Producer/consumer: A process produces information that is consumed by another process. For instance, the Process Creator produces information for consumption by the Resource Allocator, which in turn produces for the Unit Managers.

Mutual exclusion: When two or more processes share a resource only one process at a time may modify the shared resource. For instance, if both the Process Creator and the Event Detector simultaneously attempt to insert a new entry into the System Ready List, one of the new entries may be lost, since both processes may place their entry in the same next available slot. Thus, mutually exclusive access to the resource must be inforced.

The two types of coordination operations described are of a relatively high architectural level—that is, high-level with respect to a basic low-level signaling mechanism that, when appropriately programmed, would suffice to implement these operations. This point will be treated in more detail in a later section. Our overview of the functional components of a modern operating system makes it quite clear that the above types of coordination needs surface in a rather natural and ubiquitous manner. Consequently, in later sections we will devote a fair amount of discussion to their implementation.

Before proceeding any further, we must formalize our treatment of processes.

2. PROCESSES

We may view a modern computer system as a collection of simpler pseudo-computers that could operate, to a large extent, in parallel. Examples of such pseudo-computers, shown in Figure 2, are a (central) processing unit together with a section of main memory space; and an I/O channel together with a section of main memory space, a device controler, and an I/O device. Assuming the availability of resources, each such pseudo-computer is logically capable of independent operation after appropriate initialization.

When a computer system is presented with jobs (i.e., programs and data) it is the responsibility of its operating system Process Creator to subdivide the work into subunits (i.e., subprograms) that can be assigned to the various pseudo-computers. If this is done properly the jobs will be completed rapidly, since parallel operation would be effected, and resources would be well utilized. We shall refer to a (sub)program that is executing on a pseudo-computer as a *process* [14]. Somewhat more formally we define a *process* as the occurrence of a sequence of asynchronous events. That is, the progress of a process is independent of the time elapsed between the occurrence of its successive events. In essence, the term "process" denotes a dynamic entity, while the term "program" denotes a static entity. For further discussion, as well as for several alternative definitions of the process concept, the reader is referred to the papers by Horning and Randell [11] and by Denning [6].

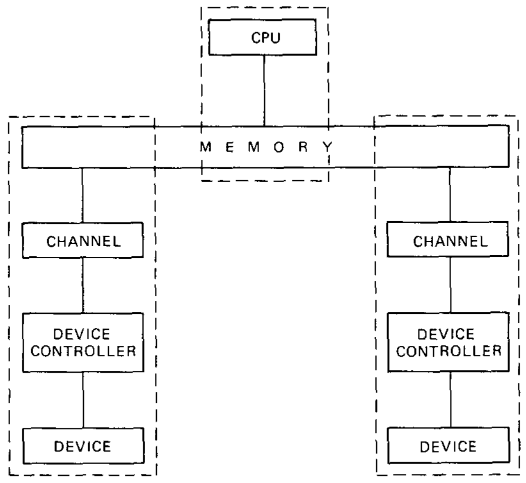

FIG. 2 Computer architecture viewed as a collection of pseudo-computers.

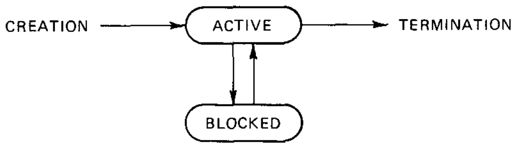

FIG. 3. Process states—unlimited resources.

We say that a process is *created* when a (sub)program is assigned to a pseudo-computer. Similarly, we say that a process is *terminated* when the assignment is invalidated. When unlimited resources exist a created process may be viewed as a logical entity that is in one of two execution states, as depicted in Figure 3. It is in the *active* state when it is executing pseudo-computer instructions, and it is in the *blocked* state when it is waiting for some event in some other process to occur before it can proceed to the next instruction. In practice, however, we have a limited number of resources. Therefore, we need to define a new state, called the *ready* state, to accommodate those processes that, after their creation, must wait for the associated pseudo-computer to become available. For example, such a situation occurs if there is not enough unused main memory space available to satisfy the needs of a created process. The new state diagram is shown in Figure 4.

Note that a process is in the ready state

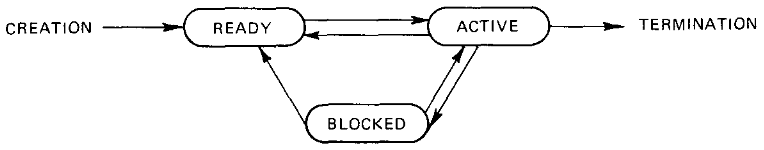

FIG. 4. Process states—limited resources.

if it is waiting for physical resources, and it is in the blocked state if it is waiting for a signal from some other process. Referring back to Figure 1 we see that all processes in a system are essentially in one of three lists, depending on whether their status is ready, active, or blocked.

When a process moves from the active into the blocked state it is possible in the case of certain pseudo-computers to record the current status of the process in a compact form, and subsequently to release the associated pseudo-computer. The process would be reactivated at a later time from the compacted information, which is referred to as the *state vector* of the process. In the case of a processing unit the state vector typically consists of the appropriate register contents, pointers to the program and data areas (which are saved), and the program counter value. At this point the reader should refer to Figure 4 and observe that a compacted blocked process must pass through the ready state to obtain active status anew. This is so because blocked processes that have been compacted have had their resources released. The architectural support provided for the implementation of process state transitions is a major consideration in the design of modern computer systems.

As a result of the discussion thus far we can view a computer system as a collection of cooperating, but competing sequential processes working in parallel. This implies the concurrent execution of a number of (sub)programs. Hence, we will employ the term *multiprogramming* to refer to such an environment. Popularly, the term is applied when the various (sub)programs originate from different jobs.* In the next section we examine the basic coordination needs of a multiprogrammed system.

3. COORDINATION FUNDAMENTALS

Given two parallel processes, A and B, composed of ordered events $a_1, a_2, \cdots, a_n$ and $b_1, b_2, \cdots, b_m$ respectively, we can *reason* that the following type of control statement is powerful enough to enforce all possible coordination requirements between the two processes:

> *a_i can only occur after b_j has occurred*
> where $1 \leq i \leq n$ and $1 \leq j \leq m$

Note that as a result of our definition of a process, a_{i+1} can only occur after a_i has occurred.

Let us now direct our attention to the implementation of the above control statement. We identify three structures of interest. First, we must place a logical *gating mechanism* just before the first line of code of a_i. Second, we must associate an *event variable* with the event b_j; its value depends on whether or not event b_j has occurred. Third, we associate a *waiting cell* with the event b_j. This cell is to contain the address of any event a_i waiting for the occurrence of b_j. If no event a_i is waiting for b_j, this cell should indicate that fact. We consider the gating mechanism, the event variable, and the waiting cell as an intrinsic trio for purposes of coordination. The structures discussed are depicted in Figure 5.

We want this system to behave as follows. When process A is in the active state and its locus of control (i.e., program counter value) reaches the gating mechanism located just before a_i, process A is to commence event a_i only if the event variable associated with b_j indicates that b_j has occurred. If b_j has not taken place, process A is to store the address of the gating mechanism, which precedes the first line of code of a_i, in the waiting cell associated with event b_j, and process A is to enter the blocked state. If process A is compacted its resources are released and the Resource Allocator is activated. When event b_j occurs, the corresponding event variable is to change state in order to indicate this fact, and if the associated waiting cell contains an address, process A is to attempt to resume progress at the indicated address.

Our discussion so far has been limited to two concomitant processes. In the general case we allow for a number of processes to work in parallel. We can *reason* that the coordination control statement now takes the following forms:

AND: *A certain event can only occur if all*

*The term *multiprocessor* is conventionally used to refer to a computer system that includes two or more (central) processing units.

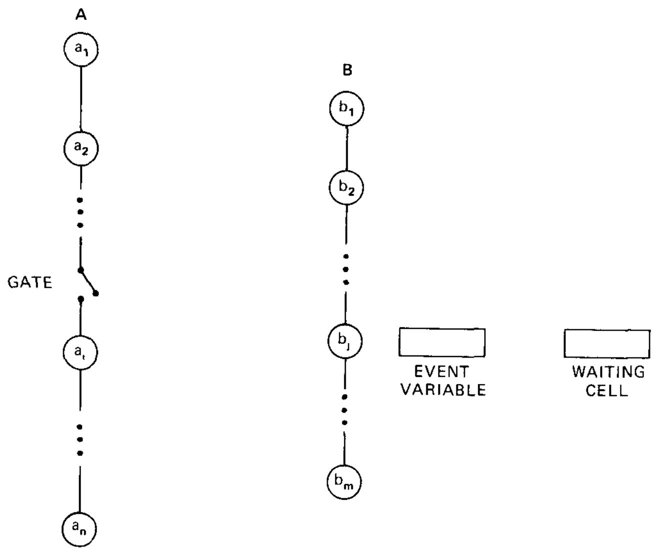

FIG. 5. Two cooperating sequential processes working in parallel.

the members of a specified set of events has occurred.

OR: *A certain event can only occur if any of the members of a specified set of events has occurred.*

Note that, as a consequence of previous definitions, each one of the members of a specified set of events belongs to a different process.

The original control statement has been divided into two versions. The AND case is to function so that, when the locus of control of a process reaches a coordination gate, the process in question is only allowed to proceed if all of the required events have already occurred. The events specified in the AND gate are scanned, and if any of these events has not occurred the process places its name and the relative address of the coordination gate in the *waiting list* of the first event that is found not to have taken place. Subsequently, the process that issued the AND control operation enters the blocked state, releasing its resources and activating the Resource Allocator if compacted. When the event being waited for occurs, and the address of this gate is selected from the associated waiting list, a new attempt is made to pass through the gate. That is, all of the events specified for that gate are tested anew. Further on in this paper we will define in a precise manner the behavior that has been described. (Please note the introduction of a waiting list to replace the waiting cell, since now there may be more than one process waiting for the occurrence of an event.)

We could detail the workings of the OR type of coordination control as we have done for the AND control. (Indeed, the reader may find it an instructive exercise to do so.) However, we shall not do it here. The reason is that in the sequel we will define coordination primitives that implement the AND type of control in a precise form. These primitives will be defined in such a manner that OR type of control is also obtained. This point will be clarified through an example (Section 5, Example 3) later in the paper. In the next section we present formal definitions of the coordination operations discussed in this section.

4. THE P/S COORDINATION PRIMITIVES

E. Dijkstra and his associates [7] and [8] defined and implemented two classical indivisible coordination primitives that mechanize the kind of operations described in the preceding section. Indivisibility implies that, once started, an operation runs to completion without interruption; therefore, while in progress, it must be guaranteed exclusive access to any resources it requires. Dijkstra named his primitives *P* and *V*. *P* and *V* originate from the Dutch words *Passeren* and *Vrijgeven*, which mean *to pass* (*by*) and *to give free* respectively. We shall employ the terms *Pass and Signal*, or simply *P and S*, to refer to these primitives. These indivisible primitives operate only on special purpose integer variables which we shall call *event variables* to emphasize their relation with the events of interest. (Dijkstra employed the term *semaphores* to refer to these variables.) There is a waiting list associated with each event variable, and event variables are always initialized. We let *E* represent an event variable and $|L_E|$ be the number of entries in its waiting list. For convenience we define the two primitives in a form somewhat different from the original, as follows:

```
P(E):
  IF $E \geq 1$
  THEN $E \leftarrow E - 1$
      and the process issuing the P continues
      its progress.
  ELSE an entry of the form (name of process issuing P, address of P) is placed in
      the waiting list associated with E, and
      the process issuing the P enters the
      blocked state.

S(E):
    $E \leftarrow E + 1$
    IF $|L_E| \geq 1$
    THEN remove one of the entries from
        the waiting list and change the status
        of the corresponding process to the
        ready state. The process issuing the
        S is placed in the ready state.
    ELSE the process issuing the S continues its progress.
```

Several important observations can be made with respect to the above primitives:

- P and S are indivisible operations.
- Event variables must always be initialized.
- An event variable can only obtain a negative value by initialization.
- No specific queueing discipline is implied for a waiting list. In a given implementation, however, the global scheduling strategy must be designed such that the possibility of a process waiting indefinitely for the resources it requires is taken into consideration. (See Example 5 in Section 6 for further discussion of this issue.)
- It is a totally competitive environment. This is so because when a process issues a signal operation and causes another process to move to the ready state, the process issuing the signal also moves to the ready state, thus, the two processes will be competing for resources.
- The P and S operations can only address one event variable at a time.
- The P operation employs a fixed test value of 1.
- The P operation employs a fixed decrement value of 1, which is identical to the fixed test value. Similarly, the S operation employs a fixed increment value of 1.

Later in the paper we shall address most of these observations in detail. At this point we proceed to illustrate the application of the P/S operations.

5. COORDINATION EXAMPLES

The relative simplicity and power of the P/S primitives is well-known. In order to insure that this is clearly comprehended, and to place the primitives in perspective, we present solutions to the two basic types of coordination needs identified in our overview of multiprogrammed operating systems.

Example 1: Producer/consumer

A typical instance of this problem is an input process (i.e., a producer) placing records of information onto a buffer concomitantly with a computation process (i.e., a consumer) taking records from the buffer. We must coordinate the situation so that the consumer does not run ahead of the producer, and insure that the producer does not write over records before the consumer has had a chance to read them. Assume a buffer *N* records in length and define two event variables, *RECORDS* and *SPACE: RECORDS* indicates whether or not there are one

```
                    {PRODUCER, CONSUMER} PROCESSES
                    BUFFER IS N RECORDS LONG
                    INITIAL (RECORDS) = 0
                    INITIAL (SPACE) = N

      PRODUCER                                  CONSUMER

LOOP                                     LOOP
  P (SPACE)                                P (RECORDS)
  PLACE A RECORD IN BUFFER                 OBTAIN A RECORD FROM BUFFER
  S (RECORDS)                              S (SPACE)
END LOOP                                 END LOOP
```

FIG. 6. Producer/consumer.

or more records in the buffer for the consumer to take; *SPACE* indicates if a space is available in the buffer for the producer to place a new record without over-writing records that have not been read by the consumer. *RECORDS* and *SPACE* are initialized to 0 and *N* respectively. A solution is presented in Figure 6.

We emphasize that the producer and consumer processes are working in parallel, and that they are continuously looping through their code. Before the consumer process takes a record from the buffer it executes the P (*RECORDS*) code which places it in the blocked state if and only if the producer has fallen behind. If the producer falls behind the consumer it will eventually catch up and remove the barrier on the consumer's progress by executing S (*RECORDS*). The producer continues to place records onto the buffer until the buffer is full. Each time a record is placed in the buffer the contents of the event variable *SPACE* is decremented by 1. When *SPACE* equals 0 the buffer is full, and a barrier is placed on the progress of the producer process. When the consumer takes a record from the buffer it executes S (*SPACE*) which increments the event variable *SPACE*, and removes the barrier on the producer's progress if it was in effect. Note that the solution presented is rather simple and elegant.

Example 2: Mutual exclusion between equal priority processes competing for a single resource

The problem, as previously discussed, is to coordinate a set of events in different processes so that no two events occur simultaneously. For example, the event that represents the modification of a given resource (e.g., modifying the System Ready List) cannot overlap (in time) in two or more processes. A simple solution to the problem is obtained if we associate an event variable *with the resource* in question to indicate whether or not the resource is free. To illustrate this in more detail, let the event variable employed for this purpose be called *MX* and be initialized to 1; then, if k looping processes are competing for access to the single resource, each competing process will include code of the form:

```
LOOP
      :
  P(MX)
  access to resource
  S(MX)
      :
END LOOP
```

Again, the solution presented is rather simple and elegant. It is formalized in Figure 7 where the code for the *i*th process is shown. Note that the P(*MX*) and S(*MX*) code surrounds the event that represents the utilization of the shared resource. The reader should convince himself that mutually exclusive access to the resource is indeed being enforced. The power of primitives like P and S can only be fully appreciated after attempting to solve a problem like mutual exclusion with simpler primitives. The reader is advised to attempt such a solution with the use of other primitives. (E.g., Dijkstra [7] discusses its solution with conventional *load* and *store* instructions; an attempt at a

```
        {E_1, , E_i, , E_k}   EVENTS IN k DIFFERENT PROCESSES
        INITIAL (MX) = 1

LOOP
  P (MX)
  EVENT E_i                                          1 ≤ i ≤ k
  S (MX)
END LOOP
```

FIG. 7. Mutual exclusion between k equal priority processes competing for a single resource—event variable associated with resource.

solution based on IBM's *wait* and *post* primitives would also be instructive.) Baer [1] and Shaw [15] examine various primitives in some detail. It is important to point out that for purposes of clearer and more reliable software design (higher-level) syntactic constructs based on P and S have been proposed [2] and [10]. In particular, Brinch-Hansen [2] motivates such constructs in the context of the mutual exclusion problem.

Earlier, we noted that we would define primitives that implement AND type of coordination control in such a manner that OR type of control was also obtained. We illustrate this point through an example.

Example 3: OR coordination

Suppose we have a process A that wishes to enforce OR type of coordination at some point along its progress. At that point it desires to proceed if either event b in process B, or event c in process C has occurred. We may depict the situation as follows:

```
      Process A            Process B    Process C
          ⋮                    ⋮            ⋮
proceed only if b or c      event b      event c
          ⋮                    ⋮            ⋮
```

A solution is easily obtained if we define an event variable, let us call it *B-OR-C*, which is initialized to 0, and code as follows:

```
 Process A     Process B     Process C
     ⋮             ⋮             ⋮
P(B-OR-C)     S(B-OR-C)     S(B-OR-C)
     ⋮             ⋮             ⋮
```

In the next section we identify some shortcomings and inconveniences of the P/S operations and define extensions that overcome the problems identified.

6. EXTENSIONS TO THE P/S PRIMITIVES

As we have shown in the preceding sections, the P/S primitives are relatively simple and powerful, as well as *natural* instruments for the solution of the basic producer/consumer and mutual exclusion coordination problems. However, as previously defined these primitives do not provide us with a general form of AND coordination control. In order to illustrate the problem we consider the following example.

Example 4: Mutual exclusion between equal priority processes competing for a set of resources

This version of the mutual exclusion problem, which we shall also call the multiple resource problem, differs from that of Example 2 (Figure 7) in that each process needs to obtain mutually exclusive access to a set of resources rather than to a single resource. Let us consider a specific case. Suppose we have two looping processes A and B that operate in parallel. Both processes require a card reader and a printer in order to carry out their work. We assume that when a process acquires a resource it keeps it until the process is terminated. Process A will request the card reader first and the printer second, while process B will request these resources in the opposite order.

Solution based on current P/S definitions

An attempt at a solution would lead us to the definition of two event variables, designated *READER* and *PRINTER*, respectively, to enforce mutually exclusive access to the associated resources, and to the code detailed in Figure 8.

```
                  {A, B} PROCESSES
                  INITIAL (READER) = 1
                  INITIAL (PRINTER) = 1

        A                                        B

LOOP                                    LOOP
  P (READER)                              P (PRINTER)
  USE OF READER                           USE OF PRINTER
  P (PRINTER)                             P (READER)
  USE OF PRINTER                          USE OF READER
  S (READER)                              S (PRINTER)
  S (PRINTER)                             S (READER)
END LOOP                                END LOOP
```

FIG. 8. Mutual exclusion between two equal priority processes competing for a set of resources—attempted solution.

The problem with this proposed solution is that if process A obtains control of the reader and process B control of the printer, a *deadlock* situation arises since no resource is released until process termination. Deadlock means that both processes enter the blocked state and have no way to exit from it. Hence, the code listed in Figure 8 is not a solution to the problem posed. A deadlock-free solution could be obtained if we have a process request, in a single indivisible operation, all the resources it needs. We exploit this idea next.

Solution based on multiple event variables extension

An extension to the original version of the P/S primitives due to Patil [13] can be employed to obtain a correct solution to the problem under consideration. It consists of allowing the specification of multiple event variables within a single P or S operation. Formally, the extended primitives are (where $1 \leq i \leq k$):

```
P(E_1, ···, E_i, ···, E_k):
  IF for all i we have $E_i \geq 1$
  THEN for all i we do $E_i \leftarrow E_i - 1$
    and the process issuing the P continues its progress.
  ELSE an entry of the form (name of process issuing P, address of P) is
    placed in the waiting list of the first $E_i$ found such that $E_i < 1$, and the
    process issuing the P enters the blocked state.
```

```
S(E_1, ···, E_i, ···, E_k):
  For all i we do $E_i \leftarrow E_i + 1$
  IF there exists a value of i such that $|L_{E_i}| \geq 1$
  THEN for each i such that $|L_{E_i}| \geq 1$
    remove all the entries from the associated waiting list and change the
    status of the corresponding processes to the ready state. The process issuing
    the S is placed in the ready state.
  ELSE the process issuing the S continues its progress.
```

The P primitive now represents a general AND coordination facility; hence, when a line of P code is attempted anew as a result of an S operation, all of the specified event variables are tested again. A subtle point is the need to remove all the process entries in a waiting list when an S operation is performed. This is necessary to insure that any process that is logically capable of progress would indeed do so. For example, suppose that a process includes the line of code $P(E_1, E_2)$ while another process contains the code $P(E_1, E_3)$. Further suppose that $E_1 = 0$ and both processes are waiting on E_1. When an $S(E_1)$ occurs, if we were to select only one of the processes in E_1's waiting list, say the one containing $P(E_1, E_2)$, and if $E_2 = 0$, then this process would be placed on E_2's waiting list. At that point we would like the other process to attempt to continue its progress, which it could do if $E_3 > 0$; however, this would only be achieved if we change all the

```
                    {A, B} PROCESSES
                    INITIAL (READER) = 1
                    INITIAL (PRINTER) = 1

        A                                          B

LOOP                                       LOOP
  P (READER PRINTER)                         P (READER PRINTER)
  USE OF RESOURCES                           USE OF RESOURCES
  S (READER PRINTER)                         S (READER, PRINTER)
END LOOP                                   END LOOP
```

FIG. 9. Mutual exclusion between two equal priority processes competing for a set of resources.

processes waiting on an event variable from the blocked to the ready state when a signal on that event variable is performed.

Returning to the problem under discussion we can now generate an elegant deadlock-free solution, since a process can request, in a single indivisible operation, all the resources it needs. Such a solution is presented in Figure 9.

Solution based on arrays of event variables extension

An alternative type of extension to the original version of the P/S primitives was suggested by Dijkstra [7] and in Parnas [12] to solve this type of problem. It requires that we allow the declaration of arrays of event variables and their manipulation through the P/S operations. To illustrate the approach we present a solution to the problem under discussion.

Let X [1:2] be an array of two event variables, and let j be an integer used to index the array. We also define an event variable MX which will be utilized to enforce mutual exclusion. The index j as well as all elements of X are initialized to 0, while MX is initialized to 1. In addition to X, j, and MX we introduce two processes, let us call them READER and PRINTER, and two other event variables designated *READER* and *PRINTER*, which are initialized to 1 to indicate that the associated resources are available. We are now able to present, in Figure 10, the code for the four concurrent processes.

The basic idea of the strategy employed in Figure 10 is that processes READER and PRINTER which are associated with the resources in question will set X[1] to a positive value whenever one of the resources becomes available. X[2] will be set to 1 only if both resources are available. The first line of coordination code in processes A and B tests X[2], and it is passed successfully only if both resources are free. In other words, the array X (with the help of index j) serves as a "bulletin board" that processes A and B can access to determine if the two resources are simultaneously available. X[2] is to be set only when both resources are available. It is the responsibility of processes READER and PRINTER to set the appropriate bulletin board element when a resource becomes available. Eventually, either process A or B will reset X[2] when the two available resources are allocated. It should be noted that the relative order of the two S operations in processes A and B is not important. Also, if we wished to be concerned with possible overflow of X[1] we could add the following additional process to the concurrent mix:

```
LOOP
  P(X[1])
END LOOP
```

Solution based on parametrized test value extension

The solution based on the array of event variables employed an integer counter j which was set to the value 2 only when both resources were free. If the test value in a P primitive were not fixed to 1 it would have been possible to define j as an event variable which is incremented through S operations; then, we could have the first line of coordi-

```
{READER, PRINTER, A, B} PROCESSES
INITIAL (READER) = 1
INITIAL (PRINTER) = 1
INITIAL (MX) = 1
INITIAL (x[1]) = INITIAL (x[2]) = 0
j ← 0

        READER                                   PRINTER
LOOP                                     LOOP
  P (READER)                               P (PRINTER)
  P (MX)                                   P (MX)
    j ← j + 1                                j ← j + 1
    S (x[j])                                 S (x[j])
  S (MX)                                   S (MX)
END LOOP                                 END LOOP

          A                                        B
LOOP                                     LOOP
  P (x[2])                                 P (x[2])
  j ← 0                                    j ← 0
  USE RESOURCES                            USE RESOURCES
  S (READER)                               S (PRINTER)
  S (PRINTER)                              S (READER)
END LOOP                                 END LOOP
```

FIG. 10. Mutual exclusion between two equal priority processes competing for a set of resources.

nation code in processes A and B test if j = 2, and continue execution of the processes only if this is the case. That is, if the original version of the P operation were extended to allow the parametrization of the test value, another alternative solution to the multiple resource problem could be obtained. We would continue to leave the test value identical to the decrement value. Next, we extend [3, 16] the original definition of P and S so that the test value is a parameter; then, we present an alternative solution to the problem under discussion. Let m be an integer such that $m \geq 0$. Then:

```
P(E, m):
    IF $E \geq m$
    THEN $E \leftarrow E - m$
        and the process issuing the P continues its progress.
    ELSE an entry of the form (name of process issuing P, address of P) is
        placed in the waiting list associated with E, and the process issuing the P
        enters the blocked state.
```

```
S(E, m):
    $E \leftarrow E + m$
    IF $|L_E| \geq 1$
    THEN remove one of the entries from the waiting list and change the status
        of the corresponding process to the ready state. The process issuing the S
        is placed in the ready state.
    ELSE the process issuing the S continues its progress.
```

We can now write another possible solution. READER and PRINTER are two processes as before, and *READER* and *PRINTER* are two event variables which are initialized to 1. Let *J* be an event variable that is initialized to 0. The desired code is shown in Figure 11.

This last solution to the multiple resource problem demonstrates that the variable test value extension also provides additional power to the original P/S primitives, since the problem could not be solved correctly with the original set, as was shown in Figure 8. However, the variable test value exten-

```
{READER, PRINTER, A, B} PROCESSES
INITIAL (READER) = 1
INITIAL (PRINTER) = 1
INITIAL (J) = 0

        READER                              PRINTER
LOOP                                 LOOP
  P (READER 1)                         P (PRINTER, 1)
  S (J, 1)                             S (J, 1)
END LOOP                             END LOOP

        A                                   B
LOOP                                 LOOP
  P (J, 2)                             P (J, 2)
  USE RESOURCES                        USE RESOURCES
  S (READER, 1)                        S (PRINTER, 1)
  S (PRINTER, 1)                       S (READER, 1)
END LOOP                             END LOOP
```

FIG. 11. Mutual exclusion between two equal priority processes competing for a set of resources

```
{A, B, C} PROCESSES
INITIAL (READER) = INITIAL (PRINTER) = INITIAL (TAPE) = 1

        A                        B                        C
LOOP                       LOOP                       LOOP
  P (READER, PRINTER)        P (READER, TAPE)           P (PRINTER, TAPE)
  USE OF RESOURCES           USE OF RESOURCES           USE OF RESOURCES
  S (READER, PRINTER)        S (READER, TAPE)           S (PRINTER, TAPE)
END LOOP                   END LOOP                   END LOOP
```

FIG. 12. Mutual ex clusion between three equal priority processes competing for a set of resources.

sion is less powerful than the other two extensions discussed so far. To illustrate the point we consider a different version of the multiple resource problem.

Example 4: revisited

Suppose we have three looping processes A, B and C that operate concurrently, and compete for three resources: a reader, a printer and a tape device. Process A needs the reader and the printer to carry out its work. Process B requires the reader and the tape, while process C must have the printer and the tape in order to operate properly. As in the previous example, we now attempt to generate three different solutions to this problem, which are based on the three extensions presented so far for the P/S primitives. Let us define three event variables called *READER*, *PRINTER* and *TAPE* these variables are initialized to 1.

Solution based on multiple event variables extension

As before, the problem is easily solved, as shown in Figure 12.

Solution based on arrays of event variables extension

We employ the same strategy as before. Let $X[1:6]$ be an array of event variables, and let j be an integer used to index the array. The index j as well as all elements of X are initialized to 0. We also define an event variable MX that is initialized to 1. We introduce three processes designated READER, PRINTER and TAPE, as well as three other event variables called

```
{READER, PRINTER, TAPE, A, B, C} PROCESSES
INITIAL (x[i]) = 0     1 ≤ i ≤ 6
INITIAL (MX) = INITIAL (READER) = INITIAL (PRINTER) = INITIAL (TAPE) = 1
j ← 0

READER                    PRINTER                   TAPE
LOOP                      LOOP                      LOOP
  P (READER)                P (PRINTER)               P (TAPE)
  P (MX)                    P (MX)                    P (MX)
    j ← j + 1                 j ← j + 2                 j ← j + 4
    S (x[j])                  S (x[j])                  S (x[j])
  S (MX)                    S (MX)                    S (MX)
END LOOP                  END LOOP                  END LOOP

A                         B                         C
LOOP                      LOOP                      LOOP
  P (x[3])                  P (x[5])                  P (x[6])
  j ← 0                     j ← 0                     j ← 0
  USE RESOURCES             USE RESOURCES             USE RESOURCES
  S (READER)                S (READER)                S (PRINTER)
  S (PRINTER)               S (TAPE)                  S (TAPE)
END LOOP                  END LOOP                  END LOOP
```

FIG. 13 Mutual exclusion between three equal priority processes competing for a set of resources.

READER, *PRINTER* and *TAPE*, which are initialized to 1. The code for the six processes is detailed in Figure 13. It should be clear that $X[3]$ is set to 1 only if the reader and printer are available; $X[5]$ is set to 1 only if the reader and the tape are available; and $X[6]$ is set to 1 only if the printer and tape are free.

Comparing the multiple event variables to the arrays of event variables extension for the realization of deadlock-free solutions, this author believes that the former is the better choice. The main reasons are that: the bulk of the work is placed on the system; the work of the programmer is made simpler; the resulting code offers a clearer documentation; and a smaller number of processes is required to implement a solution. These points become stronger as the complexity of the problems we are trying to solve increases. Further, current hardware technology is such that, if implemented in hardware, the relative differences in cost and speed incurred by extending the P and S primitives may not be of much importance. The reader should be aware, however, that other workers in the field seem to disagree with the above reasoning; see Parnas [12] and Dijkstra [9, pp. 132–134].

Solution based on parametrized test value extension

We now attempt to solve the problem under consideration employing event variables and initial values defined as in the preceding solution. Instead of the array of event variables X we define the event variable J. This attempt is shown in Figure 14. This code does not work correctly. The reason is that we have lost the identity of the resources. For example, if the tape is available, J is incremented by 4 units, which allows process A to start even though the reader and the printer may not be available. This solution strategy succeeded before because at that time we were only interested in detecting the case when all the resources were available; knowledge of their identity was not necessary.

We now pause and consolidate the P/S extensions that have been discussed.

```
{READER, PRINTER, TAPE, A, B, C} PROCESSES
INITIAL (MX) = INITIAL (READER) = INITIAL (PRINTER) = INITIAL (TAPE) = 1
INITIAL (J) = 0

      READER                    PRINTER                   TAPE

LOOP                      LOOP                      LOOP
  P (READER, 1)             P (PRINTER, 1)            P (TAPE, 1)
  S (J, 1)                  S (J, 2)                  S (J, 4)
END LOOP                  END LOOP                  END LOOP

        A                         B                         C

LOOP                      LOOP                      LOOP
  P (J, 3)                  P (J, 5)                  P (J, 6)
  USE RESOURCES             USE RESOURCES             USE RESOURCES
  S (READER, 1)             S (READER, 1)             S (PRINTER, 1)
  S (PRINTER, 1)            S (TAPE, 1)               S (TAPE, 1)
END LOOP                  END LOOP                  END LOOP
```

FIG. 14. Mutual exclusion between three equal priority processes competing for a set of resources—attempted solution.

Combined extensions to the P/S coordination primitives

Before proceeding to consider more complex coordination examples, we generate a new set of definitions for the P and S primitives that combine the various extensions discussed up to this point. Let $1 \leq i \leq k$, and $m_i \geq 0$, then:

```
P(E_1, m_1; ···; E_i, m_i; ···; E_k, m_k)
  IF for all i we have $E_i \geq m_i$
  THEN for all i we do $E_i \leftarrow E_i - m_i$
    and the process issuing the P continues its progress.
  ELSE an entry of the form (name of
    process issuing P, address of P) is
    placed in the waiting list of the first
    $E_i$ found such that $E_i < m_i$, and the
    process issuing the P enters the
    blocked state.

S(E_1, m_1; ···; E_i, m_i; ···; E_k, m_k):
  For all i we do $E_i \leftarrow E_i + m_i$
  IF there exists a value of i such that
    $|L_{E_i}| \geq 1$
  THEN for each i such that $|L_{E_i}| \geq 1$
    remove all the entries from the associated waiting list and change the
    status of the corresponding processes
    to the ready state. The process issuing
    the S is placed in the ready state.
  ELSE the process issuing the S continues its progress.
```

Let us now tackle a challenging coordination problem.

Example 5: Mutual exclusion between processes prioritized by levels and competing for a single resource

In order to study a specific case, suppose we have three concurrent processes A, B, and C which compete for access to a resource. We wish not only to insure mutually exclusive access to the resource, but also to enforce access to the resource according to preassigned process priorities. Let us say that A has the highest, B the next highest, and C the lowest priority. No preemption is permitted. That is, when a process obtains the resource it keeps the resource until it is finished using that resource. We assume that a process holds the resource for a finite period of time.

Solution based on current P/S definitions

Let us attempt a solution of this problem of mutual exclusion with priorities. Let MX be an event variable that is initialized to 1, and let L_a, L_b, L_c be event variables that are initialized to 0, 1, and 2 respectively. The

```
                         {A, B, C} PROCESSES
                         INITIAL (MX) = 1
                         INITIAL (La) = 0
                         INITIAL (Lb) = 1
                         INITIAL (Lc) = 2

LINE NO          A                        B                        C

         LOOP                     LOOP                     LOOP
  1        P (Lb, 1, Lc, 1)         P (Lc, 1)                ——
  2        P (MX, 1, La, 0)         P (MX, 1, Lb, 1)         P (MX, 1, Lc, 2)
  3        S (La, 0)                S (Lb, 1)                S (Lc, 2)
  4        USE OF RESOURCE          USE OF RESOURCE          USE OF RESOURCE
  5        S (MX, 1, Lb, 1, Lc, 1)  S (MX, 1, Lc, 1)         S (MX, 1)
         END LOOP                 END LOOP                 END LOOP
```

FIG. 15. Mutual exclusion between three prioritized processes competing for a single resource—attempted solution.

code for each of the three concurrent processes is detailed in Figure 15.

The reader should pause at this point and make sure that he understands the logic of the code in Figure 15. The best way to accomplish this is to exercise the code in various combinations and timing sequences. In essence, *MX* enforces mutual exclusion while the latches L_a, L_b, and L_c are employed to implement the desired priority scheme. Note that L_c is initialized to the value 2 in order to allow for two higher priority interrupts; L_b is initialized to 1 to allow for one higher priority interrupt; and L_a is initialized to 0 since it is associated with the highest priority process.

Most readers would be convinced that the code in Figure 15 works correctly; however, this is not the case. The reasons are subtle, and typical of the type of timing problems that can occur in multiprogrammed environments. Let us explain in detail.

One problem we encounter is that if we indeed assume that each process is continuously looping through its code, as shown in Figure 15, then processes A and B will monopolize the resource. That is, process C will never obtain access to the resource. This is so because A and B have a higher priority of access than C. To make the situation more explicit we expand the code presented in Figure 15. In actuality the code for a process, for example process A in Figure 15, would have the following format:

```
          A
LOOP
  SOME CODE
      :
  P (Lb, 1; Lc, 1)
  P (MX, 1; La, 0)
  P (La, 0)
  USE OF RESOURCE
  S (MX, 1; Lb, 1; Lc, 1)
      :
  SOME CODE
END LOOP
```

The implication of the above format is that each process spends a finite amount of time within its code, and that this time is process dependent. Therefore, if we wanted to make absolutely certain that processes A and B would not be able to lock out process C indefinitely, either by coincidence or by conspiracy, we would have to incorporate specific coordination code into these processes. For example, if G is an event variable initialized to n(n $\geq$ 0) and if we insert in Figure 15 G, 1 into the first line of code in the loop of processes A and B, and into the last line of code in the loop of process C, then, at any point in time process C is guaranteed access to the resource within n + x + 1 accesses, where a, b, c are the number of previous accesses by processes A, B, C respectively, and x = c − (a + b). It must be realized, however, that the definition of the problem (i.e., Example 5) under consideration indeed states that the higher priority process be given preference without concern for the waiting time of any lower priority process. Nevertheless, we have brought to the surface an important issue that needs further elucidation. In particular, the reader should be aware of the fact that the possibility of a process waiting indefinitely has been with us since the beginnings of this paper; it is present in the code of Figure 7. Therefore, let us address the basic issue.

In section 4 we observed that our definition of the P/S primitives assumes no queueing (i.e., scheduling) discipline for the waiting lists. Such generality carries with it the danger of indefinite waits, as discussed above. An alternative strategy [10] would be to design the primitives so that a signal operation always reactivates the longest waiting process, thus, indefinite waits are precluded. However, this is a restrictive approach of limited applicability. Consequently, we opted, for purposes of this paper, to make no assumption of queueing discipline in our definition of the P/S primitives. The relationship between waiting list queueing disciplines, other than first-in-first-out, and the problem of indefinite waits is presently not well understood.

The other problem we have with the code in Figure 15 is best explained as follows. First of all let us recall that we make no assumptions about timing relationships between independent P and S operations. Now, assume that process C becomes active, and consider the point in time when its locus of control (i.e., instruction counter) is after its second line of code but before its third line of code, that is, L_c contains the value 0 and it has not yet been set anew to the value 2. At this point in time processes A and B become active. Both A and B will attempt to execute their first line of code and, consequently, will be placed on the waiting list associated with L_c. After process C completes execution of its third line of code (with processes A and B waiting on L_c), and goes on to eventually release MX, we have no way to insure that process A will perform its first line of code before process B executes its second line of code. Hence, priority of A over B is not guaranteed. Note that appropriate priorities are maintained if process A has executed its first line of code by the time MX is released by process C; then, process B could not proceed past its second line of code, since it would fail the test on L_b. The crux of the problem is that lines 2 and 3 in process C represent a *test and set* operation on L_c which is carried out in two distinct parts rather than as an indivisible unit. Thus, we are led to our last extension of the P and S operations, that is, to allow the specification of a test and set operation within an indivisible primitive. This facility is obtained if we differentiate the test value from the decrement value in a P operation. We incorporate this extension into our last set of P/S *definitions*, and arrive at our most sophisticated version of the primitives.

Further extension to the P/S coordination primitives

```
Let $1 \leq i \leq k$, $t_i \geq 0$, and $\delta_i \geq 0$,
    then:
P($E_1, t_1, \delta_1; \cdots; E_i, t_i, \delta_i; \cdots; E_k, t_k, \delta_k$):
    IF for all $i$ we have $E_i \geq t_i$
    THEN for all $i$ we do $E_i \leftarrow E_i - \delta_i$
        and the process issuing the P continues its progress.
    ELSE an entry of the form (name of process issuing P, address of P) is
        placed in the waiting list of the first $E_i$ found such that $E_i < t_i$, and the
        process issuing the P enters the blocked state.
S ($E_1, \delta_1; \cdots; E_i, \delta_i; \cdots; E_k, \delta_k$):
    For all $i$ we do $E_i \leftarrow E_i + \delta_i$
    IF there exists a value of $i$ such that
        $|L_{E_i}| \geq 1$
    THEN for each $i$ such that $|L_{E_i}| \geq 1$
        remove all the entries from the associated waiting list and change the
        status of the corresponding processes to the ready state. The process issuing
        the S is placed in the ready state.
    ELSE the process issuing the S continues its progress.
```

Solution based on parametrized test and decrement/increment values extension

We are now able to present a correct solution to the problem of mutual exclusion with

```
                         {A, B, C} PROCESSES
                         INITIAL (MX) = 1
                         INITIAL (La) = 0
                         INITIAL (Lb) = 1
                         INITIAL (Lc) = 2

           A                          B                          C

    LOOP                      LOOP                      LOOP
      P (Lb, 1, 1, Lc, 1, 1)    P (Lc, 1, 1)              ----
      P (MX, 1, 1, La, 0, 0)    P (MX, 1, 1, Lb, 1, 0)    P (MX, 1, 1, Lc, 2, 0)
      USE OF RESOURCE           USE OF RESOURCE           USE OF RESOURCE
      S (MX, 1, Lb, 1, Lc, 1)   S (MX, 1, Lc, 1)          S (MX, 1)
    END LOOP                  END LOOP                  END LOOP
```

FIG. 16 Mutual exclusion between three prioritized processes competing for a single resource.

```
HIGHEST PRIORITY ---+              +-------LOWEST PRIORITY
                    v              v
                 {E1,   , Ei,   , Ek} EVENTS IN k PROCESSES
                 INITIAL (MX) = 1
                 INITIAL (Li) = i - 1      1 ≤ i ≤ k
                 k + 1 EVENT VARIABLES (i e , MX PLUS k LATCHES)

          LOOP
            P (Li+1, 1, 1,    , Lk, 1, 1)
            P (MX, 1, 1, Li, i - 1, 0)
            EVENT Ei
            S (MX, 1, Li+1, 1,    , Lk, 1)
          END LOOP
```

FIG 17. Mutual exclusion between k processes prioritized by levels and competing for a single resource.

priorities. We employ the same event variables and initial values defined in the solution attempted earlier. The code is detailed in Figure 16.

The only difference between the codes in Figures 15 and 16 is that in the latter the test and set operations previously discussed are carried out within an indivisible P operation; thus, priority information is not lost, as in the case of Figure 15. We have, however, increased the complexity of the P/S primitives and, consequently, we have extended the length of our smallest units of uninterruptable operation. The reader should study the solution presented in Figure 16 and convince himself that it is, indeed, correct, that is, that it enforces the desired behavior in all possiblecases.

General solution—processes prioritized by levels and competing for a set of resources

In Figure 16 we presented a solution to the specific problem of three prioritized processes competing for mutually exclusive access to a single resource. It is not difficult to extend this solution to the general case of k prioritized processes competing for a single resource; the corresponding code for the *i*th level is shown in Figure 17.

Figure 18 details the general case of k prioritized processes competing for mutually exclusive access to a set of r resources.

Next, we treat a rather general priority problem.

Example 6: Mutual exclusion between processes prioritized by classes and competing for a single resource

In Example 5 we dealt with a number of processes, each of which was associated with a given priority level. Here we go one step further and allow each priority level to become a priority class. Each class contains a group of equal priority processes. All processes in a given class may be active simul-

```
HIGHEST PRIORITY ──┐                      ┌──── LOWEST PRIORITY
                  $\{E_1, \quad , E_i, \quad , E_k\}$ EVENTS IN k PROCESSES
                  INITIAL $(L_i) = i - 1$        $1 \le i \le k$
                  INITIAL $(RESOURCE_j) = 1$     $1 \le j \le r$
                  k + r EVENT VARIABLES

LOOP
   P $(L_{i+1}, 1, 1, \quad , L_K, 1, 1)$
   P $(RESOURCE_1, 1, 1, \quad , RESOURCE_r, 1, 1, L_i, i - 1, 0)$
   EVENT $E_i$
   S $(RESOURCE_1, 1 \quad , RESOURCE_r, 1, L_{i+1}, 1, \quad , L_k, 1)$
END LOOP
```

Fig. 18. Mutual exclusion between k processes prioritized by levels and competing for a set of r resources.

```
HIGHEST PRIORITY ──────┐                ┌──── LOWEST PRIORITY
                 $\{E_1, \quad , E_i, \quad , E_k\}$ EVENT (PROCESS) CLASSES
          $c_i$ MEMBERS IN $E_i$ CLASS           $1 \le i \le k$

          INITIAL $(MX_i) = c_i$        INITIAL $(L_i) = c_{t_i} = \begin{cases} \sum_{1}^{i-1} c_i \text{ for } 1 < i \le k \\ 0 \text{ for } i = 1 \end{cases}$

            2·k EVENT VARIABLES (i e , k MX's PLUS k L's)

LINE NO   LOOP
   1        P $(L_{i+1}, 1, 1, \quad , L_K, 1, 1)$
   2        P $(MX_i, 1, 1, MX_{i+1}, c_{i+1}, 0, \quad , MX_k, c_k, 0, L_i, c_{t_i}, 0)$
   3        EVENT IN $E_i$ CLASS
   4        S $(MX_i, 1, L_{i+1}, 1, \quad L_K \ 1)$
          END LOOP
```

Fig. 19 Mutual exclusion between k processes prioritized by classes and competing for a single resource

taneously. When a process in a higher priority class is waiting, no further initiations of processes in lower priority classes is permitted. All active processes (or more precisely, specific events in active processes) are always allowed to run to completion, that is, there is no preemption. Figure 19 displays a general solution that enforces the desired coordination for the case of k priority classes competing for mutually exclusive access to a single resource; the code for a process in the *i*th class is shown in the figure.

The easiest way to understand the code is to view it as an extension of that presented in Figure 17. In Figure 19 we have a different *MX* variable for each class, and we allow each MX_i to be initialized to the value c_i which represents the total membership of the *i*th class; this permits c_i processes to have simultaneous (and exclusive) access to the resource being competed for. Note that each L_i is initialized to a value equal to the total number of higher priority class members, c_{t_i}, that could issue an interrupt to the *i*th class. Also, the second line of code of Figure 19 includes a test not present in Figure 17. This test insures that this line of code is not passed successfully until all processes in all lower priority classes are inactive. Next, we extend the problem under discussion one further step.

General solution—processes prioritized by classes and competing for a set of resources

,As a final complexity in our evolutionary study of coordination problems, we extend the solution presented in Figure 19 so that we have k priority classes competing for

```
HIGHEST PRIORITY ↓                 ↓ LOWEST PRIORITY
                 $\{E_1, \ldots, E_i, \ldots, E_k\}$ EVENT (PROCESS) CLASSES
                 $c_i$ MEMBERS IN $E_i$ CLASS        $1 \le i \le k$
                 r RESOURCES                         $1 \le j \le r$
                 INITIAL $(MX_i) = c_i$              INITIAL $(L_i) = c_{t_i} = \begin{cases} \sum_{1}^{i-1} c_i \text{ for } 1 < i \le k \\ 0 \text{ for } i = 1 \end{cases}$
                 INITIAL $(RESOURCE_j)$ = DEPENDS ON RESOURCE     $2 \cdot k + r$ EVENT VARIABLES

LOOP
    P $(L_{i+1}, 1, 1, \ldots, L_k, 1, 1)$
    P $(RESOURCE_1, 1, 1, \ldots, RESOURCE_r, 1, 1, MX_i, 1, 1, MX_{i+1}, c_{i+1}, 0, \ldots, MX_k, c_k, 0, L_i, c_{t_i}, 0)$
    EVENT IN $E_i$ CLASS
    S $(RESOURCE_1, 1, \ldots, RESOURCE_r, 1, MX_i, 1, L_{i+1}, 1, \ldots, L_k, 1)$
END LOOP
```

FIG 20. Mutual exclusion between k processes prioritized by classes and competing for a set of r resources.

mutually exclusive class access to a set of r resources. Figure 20 shows the desired code for a process in the *i*th class.

It may be viewed as an extension of the solution presented in Figure 18. Note that in Figure 20 we allow the event variable associated with a resource to be initialized to the number of processes that may access the resource in parallel. For example, for a printer this value would be 1, while for a read-only file it would be some positive integer greater than 1.

At this point we discuss a specific type of priority coordination problem that has received much attention [4, 5].

Example 7: Readers and writers

A classical example of mutual exclusion by priority class without preemption is the coordination of a group of readers and writers that access a common data base. Let us consider several variations of this problem.

Strong reader preference solution

This version of the readers and writers coordination problem may be summarized as follows:

- There are R readers and W writers sharing a data base
- Any number of readers may have simultaneous access to the data base
- Only one writer at a time may access the data base
- Readers and writers may not have simultaneous access to the data base
- A writer may initiate only when all readers are idle
- There is no preemption
- No writer is allowed to initiate if a reader is waiting (i.e., when an active writer completes, waiting readers *are* to have preference over waiting writers.)

This problem is precisely of the form discussed as Example 6; its solution was presented in Figure 19. In this (strong reader preference) case we have two priority classes, readers and writers, with the former having a higher priority. Let us emphasize that we have a situation here where 1 + W mutually exclusive classes are competing for access to a data base. The solution formula of Figure 19 may be applied directly to solve this problem. However, a simpler solution may be obtained if we view the problem solution in two stages. In the first stage we enforce mutual exclusion between a group of W writers; in the second stage we enforce mutual exclusion between a single writer and R

```
                  {R + W} PROCESSES
                  INITIAL (L) = R = # OF READERS
                  INITIAL (MX) = 1
                  INITIAL (MX_WRITERS) = 1

       READER                                  WRITER

LOOP                                 LOOP
  P (L, 1, 1)                          P (MX_WRITERS, 1, 1)
  P (MX, 1, 0)                         ----
  READER ACCESS                          P (MX, 1, 1, L, R, 0)
  S (L, 1)                               WRITER ACCESS
END LOOP                                 S (MX, 1)
                                       S (MX_WRITERS, 1)
                                     END LOOP
```

FIG 21. Strong reader preference.

```
                  {R + W} PROCESSES
                  INITIAL (L) = R = # OF READERS

       READER                                  WRITER

LOOP                                 LOOP
  P (L, 1, 1)                          P (L, R, R)
  READER ACCESS                        WRITER ACCESS
  S (L, 1)                             S (L, R)
END LOOP                             END LOOP
```

FIG. 22. Weak reader preference.

readers. Such a solution is presented in Figure 21.

It should be clear that the first phase of the solution is implemented by the P and S operations on the event variable *MX—WRITERS*, while the second phase follows directly from Figure 19 when only two priority levels are considered.

Weak reader preference solution

This version of the readers and writers problem may be summarized just like the previous one, except that the last statement of the problem is changed as follows:

- When an active writer completes, waiting readers *are not* guaranteed preference over waiting writers

A simple solution [3] to this problem is shown in Figure 22.

Observe that readers will initiate as long as $L > 0$ but a writer will not initiate unless $L = R$, that is, until all readers are idle. When a writer is active, no reader or writer can initiate, since $L = 0$. Waiting readers are not guaranteed preferential treatment.

Writer preference solution

This variation of the readers and writers coordination problem may also be summarized as before, except that the last statement of the problem is modified as follows:

- No reader is allowed to initiate if a writer is waiting

A solution is presented in Figure 23. It is similar to that in Figure 21 except that now the writers have the highest priority.

SUMMARY

In this paper we commenced with an overview of modern operating systems that uncovered an architecture consisting of cooperating, but competing processes working in parallel. Subsequently, we presented a formal treatment of processes, and analyzed the fundamental coordination requirements of concomitant processes. The analysis led to

```
                    {R + W} PROCESSES
                    INITIAL (L) = 1
                    INITIAL (MX) = R
                    INITIAL (MX_WRITERS) = 1

        WRITER                                    READER

LOOP                                      LOOP
  P (MX_WRITERS, 1, 1)                      ——
    P (L, 1, 1)                             P (MX, 1, 1, L, 1, 0)
    P (MX, R, 0)                            READER ACCESS
    WRITER ACCESS                           S (MX, 1)
    S (L, 1)                              END LOOP
  S (MX_WRITERS, 1)
END LOOP
```

FIG. 23. Writer preference.

the definition of a set of two well-known coordination primitives. In the remainder of the paper an evolutionary series of problems of increasing coordination complexity was formulated and solved. As the various problems were discussed, cumulative extensions to the original set of coordination primitives were justified and formally introduced.

At this point it behooves us to review the various extensions in detail. The parametrized test value extension [3, 16] supplied an increment of power to the original primitives. The multiple event variables [13] or the arrays of event variables [7] provided an additional increment of power. Finally, the differentiation of the test and decrement (increment) values extension (for which the author is pleased to acknowledge valuable comments from his students) added a further capability to the P/S primitives. These extensions are not mutually exclusive; in fact, when combined they provide a powerful, and relatively easy to use, set of primitive operations. (Such a set has been implemented as part of the JOSSLE system [17]). We wish to emphasize that it was not the intent of this paper to establish a hierarchical classification of the additional power provided by the various extensions to the basic P/S set. Indeed, we offered no formal proofs for purposes of such classification. Rather, by means of examples we illustrated apparent limitations and/or inconveniences that justified the extensions presented.

Whether one implements the extended set of operations as hardwired primitives, or whether one hardwires the original and simpler set and then builds upon these to implement the more complex ones in software is not an issue at this level of discussion. The important point is that we have identified a well-defined set of fundamental operations. The degree of sophistication needed in these operations depends on the intended applications. Many operating system needs can be satisfied with the basic set introduced by Dijkstra. However, the concise design of a priority class interrupt system, for example, appears to necessitate all the power of the most sophisticated version of the primitives defined in this paper.

ACKNOWLEDGMENT

I am indebted to my students Charles Dana, Mark Scott Johnson, Fred Martinez, Curtis Mosso, and John Pickens for their helpful comments. A similar note of gratitude is due to Professors Rick Bunt and Jean-Loup Baer, as well as to the referees of COMPUTING SURVEYS.

REFERENCES

[1] BAER, J L. "A survey of some theoretical aspects of multiprocessing." *Computing Surveys* 5, 1 (March 1973), 31-80.

[2] BRINCH-HANSEN, P. "Concurrent programming concepts." *Computing Surveys* 5, 4 (Dec. 1973), 223-245.

[3] CERF, V. G. "Multiprocessors, semaphores, and a graph model of computation." UCLA

Computer Science Dept. Report ENG-7223, Univ. of California, Los Angeles, Calif., April 1972.

[4] COURTOIS, P. J.; HEYMANS, R.; AND PARNAS, D. L. "Concurrent control with readers and writers." *Comm. ACM* 14, 10 (Oct. 1971), 667-668.

[5] COURTOIS, P. J.; HEYMANS, R.; AND PARNAS, D. L. "Comments on a comparison of two synchronizing concepts by P. Brinch-Hansen." *Acta Informatica* 1 (1972), 375-376.

[6 DENNING, P. J. "Third generation computer systems." *Computing Surveys* 3, 4 (Dec. 1971), 175-216.

[7] DIJKSTRA, E. W. "Cooperating sequential processes." In *Programming languages*, F. Genuys (Ed.), Academic Press, New York, 1968, 43-112.

[8] DIJKSTRA, E W. "The structure of THE-multiprogramming system." *Comm. ACM* 11, 5 (May 1968), 341-346.

[9] DIJKSTRA, E. W. "Hierarchical ordering of sequential processes." *Acta Informatica* 1, 2 (1971), 115-138.

[10] HOARE, C. A. R. "Monitors: an operating system structuring concept." *Comm. ACM* 17, 10 (Oct. 1974), 549-557.

[11] HORNING, J. J.; AND RANDELL, B. "Process structuring." *Computing Surveys* 5, 1 (March 1973), 5-30.

[12] PARNAS, D. L. "On a solution to the cigarette smokers' problem." Carnegie-Mellon Computer Science Dept. Report, Carnegie-Mellon University, Pittsburgh, Pa., July 1972.

[13] PATIL, S. S "Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes." MIT Project MAC Computation Structures Group Memo 57, MIT, Cambridge, Mass., Feb. 1971.

[14] SALTZER, J. H. "Traffic control in a multiplexed computer system." MIT Project MAC Report MAC-TR-30, MIT, Cambridge, Mass. July 1966.

[15] SHAW, A. C. *The logical design of operating systems*. Prentice-Hall, Englewood Cliffs, N. J., 1974.

[16] VANTILBORGH H.; AND VAN LAMSWEERDE A. "On an extension of Dijkstra's semaphore primitives." *Information Processing Letters* 1 (1972), 181-186.

[17] WHITE, J R.; JOHNSON, M. C.; AND PRESSER, L. "A basic guide to Jossle." Dept. of Electrical Engineering and Computer Science, Univ. of California, Santa Barbara, Calif., Dec. 1973.