'75 Boston IFAC Real Time Workshop

ON THE REAL-TIME SUPERVISORY OPERATING SYSTEM OF THE FUTURE

E. H. Bristol
The Foxboro Company
Foxboro, Massachusetts 02035

The evolution to hierarchical and distributed systems based on microprocessors is becoming clearer. For large systems this hardware is likely to include distributed control processors under a supervisory computer system consisting of many microprocessors dividing their labor on traditional computer architechtural grounds. An operating system, distributed as one or more programs throughout the system, manages the system and its resources. Given the required flexibility, the independent character of the subordinate elements, and the independent evolution of the different control programs, there will be frequent inconsistencies between programs. The total system must continue to function in the face of these inconsistencies.

Traditional priority-based coordination and communication systems will be inadequate for this new environment. In place of priority, the system will have to make more refined decisions based on a better-detailed knowledge of the application behavior. Three nontraditional control concepts are proposed:

1. The operating system should be based on an operating system language which provides a mental image of the relationship between programs.

2. The concept of virtual devices should be extended as broadly as possible.

3. Major application programs should be supported with a "written contract" which defines, for the system, the expected usage of the program, and allows the definition of alternatives whenever required resources are not available.

Where the program characteristics are not simple enough, a "negotiator," that is, a program which can compute expected resource requirements on the basis of user-provided heuristics, may augment the contract. All of this depends on a more formal classification of program function. The paper will discuss these and other anthropomorphic strategies for coordinating multiprocessor real-time systems.

INTRODUCTION

The evolution to hierarchical and distributed systems based on microprocessors is becoming clearer. The elementary subsystems in such a hierarchical system will have to be both functionally powerful and independent of each other to an extent not presently appreciated. This will be necessary if these systems are to achieve the reliability, data rate reduction, and design manageability expected from them. The basic hardware and system software elements will be independent of each other. The application programs will evolve on-line, in parts which are in independent hardware elements. This will magnify the likelihood of design errors and inconsistencies.

The simplest kinds of inconsistencies will be those relating to use of resources: the deadly embrace and other wild beasts from the traditional preserve of operating systems. But the hierarchical systems will also complicate the issues of data ownership, parallel processing, consistency of independent data bases, and a whole host of other application-oriented issues. These systems will be expected to minimize the effects of these inconsistencies for the same reasons that we now minimize the effects of deadly embraces: to simplify independent design of independent application subsystems.

The operating system (taken in the broadest sense) in such systems will consist of a supervisory operating system in the supervisory computer system, and independent operating systems in each of the subsystem elements. Its job will be to manage the system as a whole, to allocate resources, and to order the communication between system elements. Many of these problems parallel problems in our normal business or social environment. The solutions to social problems provide rich analogies particularly useful as the individual subsystems acquire the human characteristics of independence and freedom of action.

The resources of the system can be hardware elements, individual programs, or sections of data base. These correspond to individuals of suborganizations in society. The coordination and communication of these systems exist on two major planes: system communication and application communication. The system communication plane includes all detailed communication and coordination strategy necessary to provide working tools to the application designer. The system communication strategies may be formal or informal for local kinds of communication

(accesses within a program, etc.). Much of this strategy is hidden from the application designer.

The application communication plane includes all system tools available to an application designer. This plane is analogous to the formal communication in a social system, the procedures designed into the society to deal with normal requirements of the society. This is the subject of the rest of the paper.

AN OPERATING SYSTEM BASED ON SOCIAL ANALOGIES

In order for the operating system to effectively manage the system, it must "understand" the function of the different programs and resources. In fact, all independent elements of the system must be sufficiently aware of each other so that they "understand" any requests made of them, not only in terms of what is being asked (commanded), but in terms of the more general consequences of the request: on other resource allocation, on data base consistency, on recognition of potential application errors. If the system is to allocate resources based on "understanding" of function, then there is a trade-off between the complexity of the allocation system and the detail of program characteristics included in the allocation analysis. Unless the allocation algorithms are going to take much more time than the programs analyzed, it will be necessary that they be given simplified descriptions (heuristics) of the processes that they manage. These simplified descriptions are analogous to the (hopefully) white lies many people tell their bosses to facilitate understanding and speed effective decision-making and resource allocation.

An operating system built to include this thinking might have the following features:

1. It would be language. Today's operating systems are packages of routines like the scientific packages written before FORTRAN. The user is denied the organized thinking possible with a well-designed language structure.

2. The language would control the usage of hardware devices, data files, and other programs by a uniform semantic structure which considers all these things as processes. Data files acquire the properties of a process by virtue of a well-designed set of access functions. Where other programming languages were included in the system, it would be possible to write programs (processes) in these languages and thereafter treat them like any system resource.

3. The most basic (primitive) resources and the application programs are opaque to a system; the system has no natural means of "understanding" them. The system must provide standard means for describing the characteristics of resources to whatever precision or imprecision desired or required. Where use of one resource restricts the use of other resources, the description of these relationships to the process should be straightforward.

4. The system would contain effective integral instrumentation for measuring the actual behavior of any and all processes during operating conditions.

5. The effective use of a resource depends on how well the designer has described that resource to the system. To the considerable extent that it is not possible to exactly and optimally schedule a resource, the performance depends on the approximations used in this description. Note that the cost of allocation of a resource is part of the cost of the resource, and this trade-off is also part of the designer's responsibility. To the greatest extent possible, the system would penalize only the users of a given resource for the inefficiency of the resource usage. (The consequences of this will be described later, but, in outline, the system would be based on statistical prediction of resource behavior which includes both user-supplied information and system-generated information. The resulting predictions will clearly be most refined and efficient when the user provides an accurate base model.) This places the burden of good resource definition on those people most affected and most able to recognize or deal with the problem. However, the penalties will be principally in terms of speed of execution rather than failure to execute.

6. The operating system would include integrated, generalized means for coordinating and communicating between processes with a standardized error handling strategy. Integral with the operating system language are the means for manually loading, initializing, and operating the system (in its digital computer as opposed to its application sense). These features should be more appealing and have more general purpose than existing JCL languages.

Key to all of this is the development of different descriptive images or standard models of a process. Here are some suggestions:

1. A standard diagrammatic language for representing sequential and parallel processes.

2. Program priority based on the real value of the program and its cost in time and resources.

3. Use of virtual device techniques.

4. The use of "contracts" as a process model or heuristic to define detailed mutual expectation between a process and the rest of the system.

5. Interprocess communication based on post office or telephone analogies.

These concepts are now defined more explicitly:

1. A standard language for representing the relative operation of sequential or parallel processes. The author has been developing a

language, illustrated in Figure 1. It is a variant of the Petri net, but is designed to be more clearly related to traditional flow chart and logical languages. The figure shows seven symbols, although a full language contains other structures, including subroutine or macro-structures. The language effectively combines the normal flow of time and operation of the flow chart (represented by the Time sequence line, the Operation box, and the Node circle in the figure), with the Boolean logic of a logic diagram (represented by the Dashed logic line and the Slash Not symbol). The Branch-out (Fork) allows representation of logical fan-out for dashed lines, and initiation of parallel operations for solid or flow chart lines. The Branch-in (Join) can be numbered, indicating how many paths in must be completed in a flow chart sense for the output path to be taken. A Branch-in of dashed lines is used to represent And or Or logic. Mixed (solid-dashed) Branch-in's allow the gating of a flow chart path and provide the control logic usually represented in flow charts by a diamond symbol.

Later publications will develop the language further. But, by way of example, Figure 1b can be interpreted as follows: on starting at A, execute operations B, C, and D in parallel (or in any convenient order). When all three operations are complete (the Branch-in with a 3), execute operation E and wait (the mixed branch) until all three conditions F, G, and Not H are True (as recognized by the dashed Branch-in). Then execute I and exit at J.

Such a point of view gives an intuitively useful, diagrammatic language framework; for representing parallel processing, the point of view is as powerful as any existing. In the operating system it would provide the vehicle for exact representation of time relationships where these are needed.

Priority based on value. Priority is the opposite of an exact specification of the relationship between processes. A priority is a "me against the world" comparison of the program against the set of all programs. Early users of computer programs found that basing execution priority of processes on the value of the process led to bad misallocations, in that fast operating programs such as Teletype handlers may be slowed down unreasonably. The simplest priority based on value ignores the fact that many repeated uses of a low value program may be collectively more valuable than the single execution of a slower, more valued program. The proposed system would express the value of a program as a function of time, and would include as part of the system's description of program some means for estimating program timing. The scheduling algorithm would then allocate resources to maximize expected computing value. Where an exact allocation is impractical to compute, many analogies from society may be useful: marketplace analogies which allow the value of service routines to be defined by the bidding of application programs. Objective standards of value would be required to avoid "inflation" among programs.

3. The "virtual anything" concept. For any limited resource such as a line printer or a given element of data base, the system should be able to develop virtual device files so that the request for a device or data base access can be divorced from the actual scheduling of the operation. In this manner the scheduling of many resources is simplified to the point where independent scheduling algorithms independently handle each resource. Where the nature of a resource precluded virtual device techniques, an integral use of monitors (3,4) or similar strategies would accomplish the same end.

4. "The use of contracts." This social analogy is useful to the control of deadly embrace and related problems. The contract is a model of the process and its requirements. It is written by the process designer as the main description of the process. For a program, for instance, the contract defines the expected value of the program and the expected usage of resources, using parameters like working set and run time. The contract allows for all degrees of certainty of resource usage, for definition of alternative resources in case of resource failure or unavailability. It defines circumstances where resources may be required in combination. Each contract will include "act of God" clauses which define how the program will be handled in case continued execution becomes impossible. The contract performance is evaluated by the system in terms of previous behavior of the program by an evaluation routine, which is able to simply model the error between the behavior predicted by the contract and actual program performance. The system uses the contract together with this model to predict process behavior. This in turn affects the effective scheduling and resource allocation in a way that rewards effective contract definition and penalizes faulty definition. For more complex processes a special program called a "negotiator" may take the place of the contract. Since the negotiator (or even the contract interpreter) is a program requiring resources, it may also be introduced to the system by a simple contract. Since all contract and negotiator calculations will in effect be charged to the program, the program designer must select a level of "litigation" consistent with the value of the program. Both contracts and negotiators provide all information used by the scheduling routines. This simplifies the scheduling routines because program look-ahead is avoided. Any prediction of behavior must be in the contract or negotiator.

One of the functions of a contract is to soften the effects of deadlock by providing for predefined alternative resource usage. Good alternatives radically diminish probability of deadlock. For the casual user, default or

standard ("boiler plate") contracts should be available. For certain kinds of resources, real-estate-based contract analogies may be helpful.

5. Formalized interprocess communication. Messages or requests for resources as well as transmissions of answers to requests should always include destination and source addresses, as well as any routing information and information relating to purpose. Useful analogies can be drawn from postal services (including dead letter handling strategies), telephone routing strategies, and the normal order slips within a business. Such systems are designed to control the transmission of data under uncertainty of actual resource location or other contingencies. Several advantages come from inclusion of general usage data with the messages. When the sender or receiver is dynamically located, a more interpretive address scheme minimizes the effort involved in keeping messages on course. Where the solution of a problem involves the independent handling of several messages, their coordinated processing by intervening elements of the system ignorant of their purpose is better handled this way. The problem is like that faced by a factory in which many parts are shipped around for assembly of different jobs; each part has an order slip to ensure its final proper receipt. The coordination of messages would include and simplify the handling of programs waiting for multiple resources. It would also facilitate the reorder of lost information.

CONCLUSIONS

The interplay in future real-time systems will be more involved as the application requirements get greater. It will be further complicated by the distribution of hardware. In such an environment the allocation and communication between resources are more complex. The operating system must know more about the operation of all resources and programs if its decision is to be sensible. The paper suggests a scheduling algorithm which works with simplified models of all processes and some view of program value to allocate resources to get the best expected usage of the system.

The paper proposes a number of analogies from social allocation strategy. The ideas are highly related to thoughts already circulating in the computer community, particularly for heuristic search and artificial intelligence. The use of heuristics in the form of contracts is most appropriate for two reasons: the allocation of resources need not be optimum, but merely workable and forgiving of error; and the heuristics for each resource are written by the designer of the resource, and turnable by him, to whatever degree of precision he can justify or accomplish.

REFERENCES

1. Petri, C. A., "Kommunikation mit Automaten," Schriften des Reinisch - West Falischen Inst. Instrumentelle Math., und der Universitat Bonn, Nr. 2, Bonn, 1962.

2. Holt, A. W., Shapero, R. M., Saint, H., and Warshall, S., "Final Report for the Information System Theory Project," Applied Data Research Inc., February 1968, ADR Ref. No. 6608.

3. Hoare, C. A., 1974 Monitors, "An Operating System Concept," Comm. ACM 17, 10, 1974, pp. 549-557.

4. Brinch Hansen, P., "Operating System Principles," Prentice-Hall, New Jersey, 1973.

5. Nilsson, N. J., "Problem-Solving Methods in Artificial Intelligence," McGraw-Hill Book Co., New York, 1971.

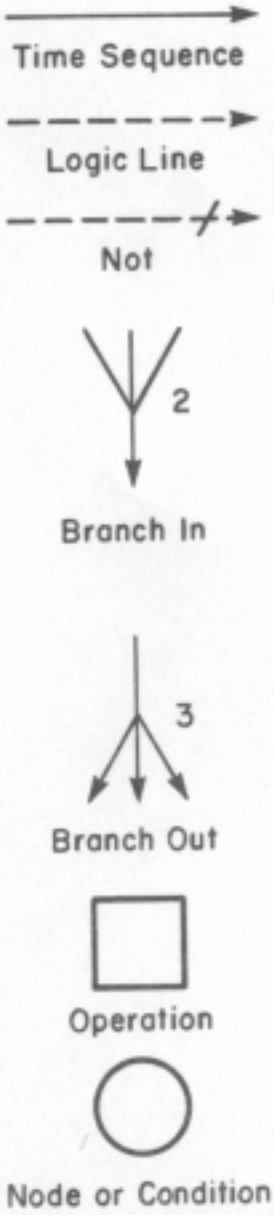

Fig. Ia

Symbols for Operation System Language

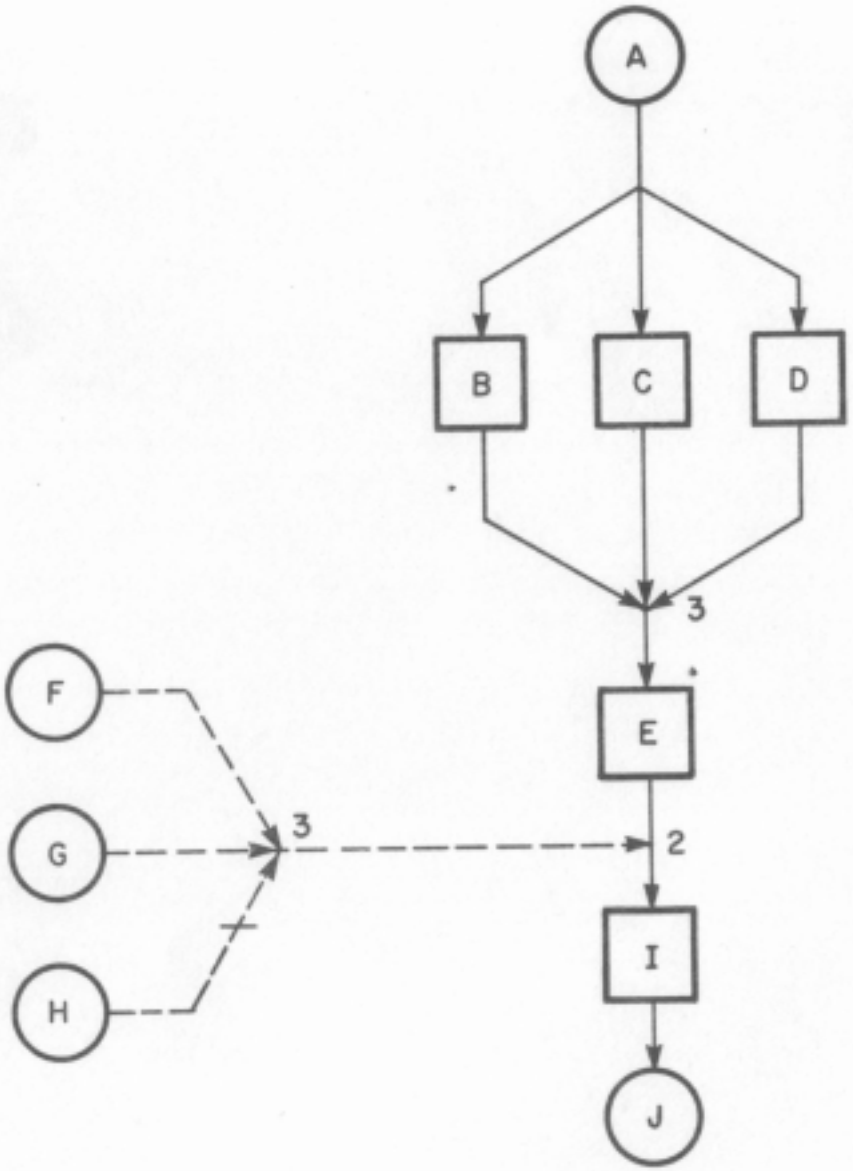

Fig. Ib

Operation System Diagram