# SCHEDULING PARALLEL PROCESSES WITHOUT A COMMON SCHEDULER*

Extended Abstract

George Holober and Lawrence Snyder

Technical Report # 154, July 1979

*This work was funded in part by the Office of Naval Research grant N00014-75-C-0752.

# SCHEDULING PARALLEL PROCESSES WITHOUT A COMMON SCHEDULER†

EXTENDED ABSTRACT

George Holober and Lawrence Snyder
Department of Computer Science
Yale University
New Haven, Connecticut 06520

Abstract: An algorithm which solves the critical section problem for distributed processes is presented. We extend the solution of Lamport [LL76] by continuing to allow processes to access their respective critical sections in any arbitrary user-specified order, but with greatly reduced storage requirements for each process. In addition, we supply a facility for testing the presence of deadlock among processes waiting to enter their critical code. We show our scheme to be tolerant of several malfunctioning processors, and derive an equation relating the probability of total system failure to the probability of many individual failures occurring simultaneously among the processors.

## INTRODUCTION

The "critical section" problem, which involves developing a synchronization scheme for a set of processes that enforces solo occupancy of common code, is further complicated when we generalize the circumstances under which the scheme will work or restrict the allowable solutions in some manner. For example, we will assume that the processes execute asynchronously (i.e. nothing is known about one process' rate of execution relative to that of another process nor to the same process' rate of execution at a different time) and that each process must have the same solution as every other process. Another reasonable objective is to avoid possible deadlock resulting from two or more processes waiting for each other.

A number of solutions to the critical section problem have been developed and studied since Dijkstra's initial paper [EWD65]. The results reported in that paper, along with the subsequent refinements outlined by Knuth [DEK], deBruijn [deB], and Eisenberg and McGuire [EM], assumed that concurrent processes would be implemented on multiprogrammed systems. These systems allow different processes to read from or write into any memory location.

Only recently have researchers begun to look at multiprocessor or distributed systems. In such a system, a process may read or write in its local memory and may read from another processor's memory, but may not write into another processor's address space. This restriction prevents the use of global variables, but does yield one important advantage over multiprogrammed computers: if one process fails, the entire systems does not *necessarily* crash, though system performance will likely be degraded.

One of the first examinations of distributed systems was done by Dijkstra [EWD74]. This paper studied the possibility of processors independently recognizing that they had failed and correcting themselves to some prescribed state. At about the same time Lamport [LL74] presented a solution to Dijkstra's original problem with critical sections that obeyed the constraints of distributed computers. Rivest and Pratt [RP] improved upon this scheme by bounding the values of the variables necessary for inter-process coordination and by preventing a process that continually fails and restarts from deadlocking the system. Further improvements (in terms of smaller ranges of values for variables, greater fairness when sequencing processes for entry into their critical regions, and reduced waiting times for processes before entering their respective regions) were developed by Peterson and Fischer [PM]. Finally, Katseff [HPK] incorporated the best aspects of each of these solutions, including the servicing of processes in the order in which they arrive (FIFO), into one algorithm.

Taking a somewhat different approach, Lamport [LL76] recognized the fact that it is not always desirable to allow processes to enter their critical regions in the same order in which they attempted to access these regions. It is frequently the case that a process may not

---

†This work was funded in part by Office of Naval Research Grant N00014-75-C-0752.

conflict with another process in the sense that they may enter critical regions simultaneously, though both these processes may conflict with a third process. Furthermore, given a set of processes that are currently prevented from entering their critical regions, we may wish to impose some priority on these processes so that when conflicting processes eventually do leave their critical regions, the process having the highest priority, rather than the process that has been waiting the longest, will be the first to access its own region.

In this paper we present a modification of Lamport's system that corrects some drawbacks of both his and Katseff's solutions. In particular:
1) We maintain the basic capabilities of Lamport's design but add a facility to detect the formation of anamalous situations in which a set of processes will deadlock because each process believes another process has priority over it.
2) One variable that is used in Lamport's solution may grow unboundedly large (though in practice this <u>may</u> have little effect). We show how to limit to a finite range the possible values of all variables used for synchronization purposes.
3) Lamport's and Katseff's code requires that each process contain an array, the length of which is equal to the total number of processes. With the recent advances in computer-on-a-chip hardware designs, it is quite likely that future machine architectures will involve huge numbers of communicating processors (capable of running a proportionately large number of processes), each processor possessing a fairly limited amount of memory. Such a hardware scheme is clearly incompatible with Lamport's and Katseff's routines. In our program, each process will need to keep track of only a constant number of other processes.

## <u>SYSTEM</u> <u>OVERVIEW</u>

The architecture of the system we will use for our studies is conceptually simple: we have a set of processors, each processor capable of executing at most one process from a set of N processes, and each processor communicating with a subset of the other processors. By "communicate" we mean that one processor may read from another's memory or possibly transmit an interrupt signal (this latter condition is not essential); however, conforming to the definition of a truly distributed system, it may not store into any memory but its own.

We further assume that a processor may fail, though it does so in a somewhat orderly fashion. A read request issued to a process immediately after this process has malfunctioned may return arbitrary values. Eventually only some default value will be returned by read requests to a failing processor, hence it is impossible to accurately examine the memory contents of such a processor. Each processor has the ability to detect its own deviation from normal operating protocol and shut itself down without transmitting spurious interrupts and without writing incorrect information on a disk to which it is linked. The process that had been running on a processor until that processor malfunctioned may be restarted at some predefined point.

As noted in the previous section, the early solutions to the critical section problem require disjoint processes to store into common memory locations. Many of the synchronization schemes that have been proposed to date, such as PV [EWD68], monitors [CARH74], and path expressions when implemented in terms of semaphores [CH, ANH], seem to rely upon a dedicated scheduling routine. Unfortunately, such schemes are incompatible with the desired autonomy of processors. For if the processor in which global data is stored or a dedicated scheduler should fail, the entire system fails. Lamport has explored many aspects of a synchronization scheme that avoids this drawback, though he only touches briefly upon the issue of scheduling. We examine this last issue in greater detail.

The synchronization primitive used by Lamport is an extension of the conditional critical region first proposed by Hoare [CARH71] and later described by Brinch-Hansen [PBH72, PBH73a, PBH73b]. This new primitive takes the form

<u>region</u> <mode> <u>when</u> <condition>
<u>do</u> <critical-section> <u>od</u>

The metavariable <mode> is an expression (typically a constant or a single variable) which evaluates to an element of some arbitrary finite set M (subject to Restriction #1 below); <condition> is a Boolean expression; <critical-section> is an arbitrary length of code (subject to Restriction #3) which comprises the critical region.

It may not be the case that all critical regions will conflict with all other critical regions in the sense that we may desire two processes to be executing their critical regions simultaneously, though either or both of these processes may in turn prevent a third process from entering its region. To formalize this notion, we define a symmetric, time-independent function <u>conflict</u>: M x M —> {true, false}. We then say that two processes <u>conflict</u> if and only if they are both attempting to execute <u>region</u> statements with respective <mode> values of <u>mode1</u> and <u>mode2</u>, and <u>conflict</u> (<u>mode1</u>, <u>mode2</u>) = true.

The semantics of the <u>region</u> statement can be stated quite simply: the code in the <critical-section> may not begin execution if a conflicting process has already entered the <critical-section> of a <u>region</u> statement or if <condition> evaluates to false. To prevent certain anomalous situations from arising, we must enforce the following restrictions on our synchronization primitive:

Restriction #1: The value of <mode> must remain constant during the entire execution of the

region statement to which it is associated.

Restriction #2: To prevent races between instructions which alter and examine a when <condition>, arguments of the <condition> of one process' region statement which are stored in the memory of another process may only be modified by this second process within a region statement which conflicts with the first region statement.

Note that if the <condition> of a region statement does not depend upon the contents of another process' address space, then this <condition> must always evaluate to true, for if this were not so, then the process would enter the region statement, halt execution until the <condition> became true, thereby preventing assignments to the very variables that can satisfy the <condition> and causing the process to deadlock with itself.

Restriction #3: A region statement may not be one of the instructions in the <critical-section> of another region statement.

One problem frequently associated with conditional critical regions is the difficulty they pose in expressing some synchronization problems. These problems usually have a "scheduling" flavor to them: given a set of conflicting processes that are all competing to enter their respective critical regions, which will take precedence? To remedy this flaw, we define a new function must precede: {1, 2, ..., N} x {1, 2, ..., N} --> {true, false} which may depend upon any information that is available to the system. Therefore given a particular i and j in the set {1, ..., N}, must precede (i, j) need not remain constant over a period of time. (Lamport actually calls this function "should precede"; we will save this term to denote a different function.)

This very general definition of must precede is actually too permissive. The following argument illustrates this point. Suppose that in addition to i and j, the names of the two processes, the function must precede depends on K other sources of information, e.g. which processes are in their critical regions, which processes are awaiting permission to enter their regions and how long they've been in this state, which processes have failed, the values stored in the memories of various processes, etc. It is very unlikely that a process can examine all K+2 arguments and instantly determine the value of must precede (i, j). Rather, the process would probably scan one or two arguments at a time and combine this information with previously computed results to obtain a partial answer. This procedure would repeat this until all arguments have been examined and must precede (i, j) has been fully determined. Consider the case in which a process is scanning the xth argument of must precede (x is in the interval [2, ..., K+2]) when another process alters the value of the yth argument (y is in the interval [1, ..., x-1]). The first process will never rescan the yth argument, so the value it finally obtains for must precede (i, j) will be incorrect. To overcome this difficulty, Lamport assumes that must precede is strongly constant, meaning that its value will not change when we are in the midst of computing it. This convention simplifies matters greatly (and in fact probably does not pose a severe restriction), so we will adopt it as well.

The interpretation of the must precede function is self-evident, but it is important to point out that it has meaning only on those processes that are simultaneously waiting to enter their critical regions and that conflict with one another. Putting together the mechanisms we have described so far, it becomes clear that a process i can enter the <critical-section> of a region statement only if the following three conditions are satisfied:

Condition #1: All processes that conflict with process i are executing code outside of their critical regions.

Condition #2: The when <condition> evaluates to true.

Condition #3: For all processes j that are presently executing region statements but have not yet entered their <critical-section>'s, and that conflict with process i

$$\text{must precede}\ (i, j) = \begin{cases} \text{true} & \text{if } j \text{ has been waiting longer than } i \\ \text{false} & \text{if } i \text{ has been waiting longer than } j \end{cases}$$

In other words, of all the processes that do not conflict with another process that is in a <critical-section> (#1), that have true when <condition>'s (#2), and that have no predecessors (in the sense that there is no conflicting process j for which must precede (j, i) holds true), time of arrival is the final arbiter (#3). We impose one more condition on our system that guarantees that no process can be locked out of a <critical-section> once it has begun executing a region statement:

Condition #4: Assuming no further processes encounter region statements, a process satisfying Conditions 1 - 3 will enter its <critical-section> after a finite delay.

This condition will follow if we assume that all processes make progress executing their instructions (though our previous assumption of asynchronous operation may make this progress very slow) and if a permanent deadlock situation does not exist among the processes that are waiting to enter their critical regions.

## THE ALGORITHM

In the last section we briefly mentioned the possibility of two or more processes causing a deadlock while waiting to enter critical regions. To see how this might happen, consider the most trivial case for the moment. Suppose that process i has just encountered the statement

region model when true do <anything> od

where conflict (model, model) = true and must precede (i, i) = true. Using our rules for selecting processes to enter their <critical-section>'s, process i must wait for itself to leave its <critical-section> before it can enter it, a clear impossibility. A deadlock is present, and Condition #4 is violated (unless must precede (i, i) changes to false at some future point). Although this may seem like a contrived example, and therefore not a very convincing justification for our attempts to determine the existence of deadlocks, these deadlocks can arise in far more subtle ways. The following theorem characterizes the situations in which a deadlock will be present.

Cycle Theorem: A deadlock will exist among the processes that are awaiting entrance to their critical regions if and only if there exists a subset {P(0), P(1), ..., P(L)} of these processes which form a "cycle" in the sense that for all i in the set {0, 1, ..., L}

(1) P(i) is in a region statement with <mode> value M(i), and
(2) the functions must precede (P(i), P(i+1 mod L)) and conflict (M(i), M(i+1 mod L)) evaluate to true.

Proof: The "if" part follows immediately from our definitions. The "only if" part stems from the following fact: if we trace backwards over the must precede and conflict relations on a finite set of processes, we must eventually either return to a process which has already been visited (thereby showing the presence of a cycle), or else we will arrive at a process i for which there are no processes j such that must precede (j, i) = true and processes i and j are in conficting region statements. In this latter case there is no cycle, but process i can enter its <critical-section> and there is no deadlock.

We must establish several ground rules for manipulating faulty processes so that we will have a common convention with which to work. In addition to assuming that a failing process does not behave "maliciously," e.g. it sends off spurious interrupts to the remaining operational processes, we further assume that we have some reliable mechanism for determining whether a particular process has failed. A process can be thought of as emitting a "carrier signal"; when the signal dies, the process has failed.

Processes which fail while on the queue remain there until some external device repairs them so that they can eventually enter their critical sections. We adopt this convention on the basis of its being the most general scheme for dealing with the failure of enqueued processes. "Most general," in the sense used here, means the ability of this scheme to simulate any other scheme. This generality arises from the flexibility of scheduling provided by the must precede. For example, we could easily alter the value of must precede to effectively ignore the presence of a failed process on the queue. Of course, we are assuming that in such a situation the values of the arguments to must precede can be determined despite the loss of accessibility to data that has been stored by malfunctioning processes.

Processes which fail while executing their critical sections can block many other processes with which they conflict, thereby causing serious degradation in system performance. We will assume such processes are to be removed from their critical sections by the external mechanism before being repaired and returned to normal operation. Note that once in its critical section, a process is beyond the effects of the must precede function. Thus we do not have the run-time flexibility we had when dealing with the failure of enqueued processes, and we appear to be quite rigidly bound by whatever scheme we choose for servicing processes which fail in the midst of their critical code.

It would be unreasonable to assume that a process can be made to stop, perform some desired operation, and resume unless it is under our control. Thus we cannot expect the cooperation of processes which are executing their critical sections or non-critical sections. The only times a process does come under our control so that it can be made to perform synchronization tasks is when it is waiting on the queue and leaving its critical section.

Because concurrent computations are inherently difficult to understand (and rigorous mathematical proofs of their correctness are even more difficult to comprehend), we will break down the development of the algorithm into three steps. In the first version, we deal with a sequential program that will temporarily serve as our scheduler and that is easy to comprehend. In the next version, we transform the sequential program into a parallel program. At this point we are halfway to our target program: control of instruction sequencing has been removed from the central scheduler and is now managed by the individual processes, but shared memory is still utilized. In the final version, we convert this parallel program into fully distributed code by passing out the common storage locations among the component processes. (For notational convenience, we say i ==> j if conflict(i,j) = true and must precede(i,j) = true.)

There are several advantages to treating the development of a distributed program as code synthesis beginning with a simple statement of the solution rather than as a programming task followed by a verification phase. Not only are

proofs of programs (especially parallel programs) difficult to devise, they are almost as difficult to understand due to their ad hoc nature. Even if the rules of verification could be formalized, mechanical verifiers invariably suffer from extremely poor efficiency, as the task they are meant to perform is almost surely intractible. Synthesizing code by means of simple transformations need not require a major effort, just as the compilation of high level sequential languages into machine level code can be accomplished efficiently and in a straightfoward manner (presumably because this is a well understood task). Furthermore, programming techniques demanding verification suffer because it is difficult to build each new program upon old programs. Instead, many papers dealing with parallel processes seem to begin afresh, defining low level features, expanding upon them, and finally verifying what has been developed. On the other hand, synthesis begins with a small collection of requisite parameters, and modifies these to mesh with the low level features of the system in a top-down fashion.

Version 1

In this initial version, we are dealing with a very simple sequential program. The scheduler exists as a separate routine (which we will presently assume is immune to failure), and governs the operation of all other processes. A macroscopic view of the operation of the scheduler is given by the flowchart in Figure 1.

There is one very important issue that we have avoided so far: how do we deal with two or more processes that simultaneously begin execution of region statements? Or in terms of our system, how do we treat processes that signal their intention to interact with the scheduler when the scheduler is already busy servicing some other process? Before proceeding with our description of the algorithm, we must put this issue to rest by establishing a method for determining the relative ordering of such processes.

Optimally, we would like the scheduling routine to service processes in the same chronological order these processes signal the scheduler. One solution to this problem, performed at the implementation level of the system, would be to let each process dispatch an interrupt when it wants the attention of the scheduler. The scheduler, in turn, serves as our interrupt handler, and it disables all other interrupts until the process requesting attention is fully serviced. In this solution, we have pushed the problem back onto the hardware mechanism.

Another possible solution might be to let each process maintain a timer while it is awaiting the attention of the scheduler. The timer could be a mechanical clock, or we could let the program idle in a loop. On each iteration of the loop, a variable TIMER would be incremented by one. When the scheduler becomes available, it picks the process whose timer indicates the longest wait. This solution suffers several drawbacks. Depending upon the response time of the scheduler, the value stored in the timer could grow unboundedly large. Even worse, we are dealing with an asynchronous system, so the timer may not reflect a true measure of the waiting time (though if we assume a finite bound on the speed of one process relative to another, we are guaranteed that all processes will eventually command the attention of the scheduler).

In both of these solutions, we have relied upon an external agent to assume the burden of the problem. Is it possible to avoid the use of an external device entirely? We maintain the answer is no. In any realistic system, there will be a lower bound on the length of time that can be measured. If two events occur within this time span, we are faced with the problem of taking these seemingly simultaneous events and determining which of them actually came first. What choice do we have, but to rely upon an external arbiter to resolve this dilemma? Hopefully, such an arbiter would either be capable of measuring time on a more refined basis, or would have some other information, unknown to us, for ordering events.

In our system, the lower bound for measuring time is the maximum response time of the scheduler. What we have done, in effect, is to treat time as a resource, and to insist that mutual exclusion be maintained on this resource at those points in time when a process is interacting with the scheduler. We note in passing that many systems that have been described in the literature finesse the issue of simultaneity by assuming the availability of indivisible or atomic operations.

Version 2

Continuing with the synthesis of our final program, we now "snip" the control mechanism to eliminate the explicit scheduler. The scheduler, which is still failure-free, can instead be thought of as existing only in an conceptual form, transmitting instructions to the individual processes. By this we mean that the scheduler issues an instruction which all the processes compete for and execute. The execution of such an instuction is finished when all the processes have completed their portions of the code, or have failed. The result is a parallel program which utilizes shared memory.

In reality, each process will have a copy of the scheduler. These individual copies will operate in asynchronous parallel manner by using a "mutual handshake" concept. When one component of the scheduler finishes some instruction, it polls the other components to determine if they have finished their respective instructions, and waits until they have done so before proceeding with the next instruction. Setting a flag at the beginning and end of each instuction would be a simple mechanism for determining whether or not each process had finished its scheduling instuction. Figure 2 illustrates a sample instruction for enqueuing a process that begins

executing a region statement.

We also begin to decompose the queue at this point. Instead of having one process, the common scheduler, store the configuration of the enqueued processes, we now let each component process remember its location within the queue. The processes on the queue will be strung together in a linear sequence by a set of multiple pointers. Each process contains s-element arrays BEFORE and AFTER. The value of BEFORE[i] is the identifier of the process which arrived on the queue i arrivals before the process in which this array is stored. AFTER has the complementary meaning. We will sometimes subscript a variable with index i to emphasize that this variable is local to process i. Figure 3 provides a global view of the structure of these arrays.

The purpose of the multiple links between processes is twofold. First, should a process fail, we can still determine which processes follow it or precede it on the queue simply by following an alternative link around the malfunctioning process. And second, the redundancy of these pointers can be useful for detecting the failure of processes. Many previous solutions to the critical section problem assume that when a process fails, it turns on some sort of signal that beacons its failure to the remaining functional processes, so that the operation of these processes will not be affected. Clearly this is not an entirely realistic assumption. We note that if $BEFORE_i[j] = k$ and $AFTER_k[j]$ does not equal i, then it is quite likely that either or both of processes i and k have failed. Further tests involving comparisons with links from other processes could aid in pinpointing the exact identity of the malfunctioning process.

Version 3

In the third and final version of our routines, we are ready to eliminate the scheduler completely and to distribute both the memory and control mechanism to the individual processes. Each process has a copy of the scheduler and can be thought of as issuing instructions to itself. The processes then operate in conjuction to determine which instructions should be executed and when.

Possibly the first feature of version 2 that strikes the reader is that memory management has been almost entirely divided among the constituent processes. This division of memory management has been one of our prime objectives from the beginning, for in order to conform to the definition of a distributed system and reap the fault-tolerant capabilities such systems have to offer, we must insure that individual processes perform write operations only on their own local memories. An examination of the instructions of version 2 reveals that all of the instuctions cause process i to alter only the contents of its own memory.

We are not quite finished, however, due to the memory requirements that would result from a naive implementation of the instructions. A restriction we have placed on our system, along with the need for a distributed control mechanism, is that each process use a limited amount of memory. In other words, each process should have an address space whose size is independent of n, the number of processes. Nearly all of the instructions obey this property, the sole exception being the deadlock-test operation.

The Cycle Theorem tells us that testing for deadlock is equivalent to testing for the presence of cycles in the ==> relation. Phrasing this another way, a deadlock will exist if and only if some process p obeys the relationship $p \overset{*}{\Longrightarrow} p$, where $\overset{*}{\Longrightarrow}$ is the (non-reflexive) transitive closure of ==>. A deterministic algorithm for computing the transitive closure on n objects will undoubtedly proceed by following the ==> relation from one object to the next and backtracking where necessary. To prevent some sequence a ==> b ==> c ==> ... ==> z of processes from being examined repeatedly, it appears necessary to keep a record of the processes along such chains that have already been scanned and need not be re-examined. The number of markers needed to maintain this record yields O(n) space complexity in the worst case. Linear space complexity is unfortunate from our point of view, for even though an amount of memory proportional to n will be needed to test for deadlock, no single process can directly utilize that much space. Thus each of the n processes must devote a constant amount of memory toward executing the deadlock-test instruction.

To see if process i has caused a deadlock, process i turns a flag CYCLE to ON. Each process k other than i checks to see if there is a process j such that $CYCLE_j$ = ON and j ==> k. If so, process k sets $CYCLE_k$ to ON, establishing one more link in the potential deadlock cycle. Eventually, either no more processes can set their values of CYCLE to ON (in which case there can be no deadlock) and the test ends, or some process k such that k ==> i sets $CYCLE_k$ to ON, and process i notes the completed cycle and announces the presence of a deadlock. This deadlock check algorithm is outlined by the flowchart in Figure 4.

An analysis of the requirements for the deadlock-test instruction shows that it can fail in either of two circumstances: more than the designated number of consecutive processes fail simultaneously (in which case the remaining operational processes will not be able to assume responsibility for all of their malfunctioning counterparts), or all the processes on the queue fail simultaneously. However, neither of these conditions is too important. We have ruled out the first case (or at least know the probability of its happening). In the second case, there are no operational processes on the queue, so that none could possibly enter their critical sections,

and the existence of a deadlock is therefore inconsequential.

Note that we have been very liberal in allowing the user to risk potential deadlock situations. As a result, our deadlock detection routine incurs a great deal of run-time expense in the form of process cross-talk. One possible alternative to the scheme presented here is somewhat more conservative in nature. Instead of permitting the possibility of deadlock at compile-time and checking for its presence at run-time, we disallow definitions of must precede that would allow a deadlock to develop when certain combinations of processes are enqueued. This compile-time check is simple: we assume all processes are on the queue, and use our deadlock tester to see if a cycle is present. If no cycle exists under these circumstances, no cycle can ever exist, and the system is guaranteed to be deadlock-free. Otherwise, the user is informed that deadlock may develop in the future. Thus we need to test for deadlock only when must precede changes, and not whenever a new process enters the queue.

## FAILURE ANALYSIS

One drawback of our system is that under extreme circumstances the entire system may fail. Such a situation would arise if groups of operational enqueued processes were separated by so many failed processes that the former could not use the information contained in BEFORE and AFTER to derive the relative ordering of the groups. If each of these arrays has s elements, at least 2s consective processes on the queue must be down at the same time for the system to collapse. The probability of such a failure occurring is given by the formula

$$\sum_{i=0}^{n-s-1}\left(1+\left[\sum_{j=1}^{\lfloor i/s+1\rfloor}\binom{i-js}{j}\,[(p-1)p^{s}]^{j}\right]\right)\cdot(1-p)\cdot p^{s}\cdot(1-p^{n-i-s-1})$$

where p is the probability that an individual process will be nonoperational at any particular moment. By making s as large as we desire, this probability becomes arbitrarily small.

## CONCLUSIONS

We have demonstrated a solution to the critical section problem for distributed systems that satisfies the stated design requirements:
1) It permits arbitrary processes to conflict/not confict depending upon the particular critical regions they are attempting to enter.
2) It allows granting access to critical regions based upon an arbitrary scheduling function. The order of requests for entering the critical regions is maintained and can be used for scheduling purposes.
3) All variables involved in the synchronization process assume values from a finite range.
4) All processes need to store only a small amount of data to maintain the synchronization scheme. By "small" we mean an amount that is independent of of the number of processors in the system.
5) The failure and subsequent restart of any individual process or even a reasonably small subset of processes will not cause a widespread system malfunction.
6) The creation of a cycle of must-precede-related processes and the resulting deadlock can be detected, though we do not specify what course of action should be taken from that point on.

Most importantly, we have demonstrated a technique for transforming an easy-to-understand sequential program into a distributed program. Each step of the transformation is reasonably straightforward. We have attempted to find natural lines along which to decompose our program. With a greater effort, we might hope to formalize the transformation process, possibly to the point where it could be mechanized.

Our results point to several other areas that should be examined. For example, we have described one notion of deadlock, when in fact there exists another rather obvious form of deadlock with which we have not dealt. If a process is waiting on the queue for its when condition to turn true, but no other conflicting process has yet arrived which can alter this when condition, then this process, along with all enqueued conflicting processes which it must-precede, will sit idle. Determining whether a process will alter any variables and thereby change when conditions is recursively undecidable, so it may not be feasible to build a mechanism to accurately detect or correct this type of deadlock. Is this an important consideration among real parallel routines? If so, will heuristic deadlock testers suffice to make this a negligible problem?

Furthermore, we have been able to develop a reasonably simple algorithm by passing the details of scheduling, in the form of conflict and must-precede relations, to the user. While this gives the user a great deal of flexibility, this flexibility must be accompanied by a certain measure of responsibility. Is all this flexibility necessary? Or must the user pay for it in terms of the extreme care taken to program scheduling relations? And are there techniques he might employ developing these relations that would allow the synchronization protocols to execute with greater efficiency?

Another area for further study centers around the implementation of the queue. Our multiply-linked list is a "stretched-out" data structure, in the sense that it does not require a large set of malfunctioning processes to form a

cut set and thereby cause the system to fail. Are there alternative data structures which require a larger cut set to separate and therefore present a lower probability of system failure? And exactly what would be the tradeoff between the improved reliability of these structures and the increased complexity and reduced efficiency of the code for the critical section problem?

Acknowledgements: Prof. Mark R. Brown provided some useful suggestions related to the failure analysis. Prof. Alan J. Perlis made several very insightful comments upon the nature and limitations of parallel systems.

## REFERENCES


[PBH72] Per Brinch-Hansen, "A Comparison of Two Synchronizing Concepts," Acta Informatica, Vol. 3, Fasc. 1, 1972, pp. 190 - 199.

[PBH73a] Per Brinch-Hansen, "Concurrent Programming Concepts," ACM Computing Surveys, Vol. 6, No. 4, Dec. 1973, pp. 223 - 245.

[PBH73b] Per Brinch-Hansen, Operating System Principles, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[deB] N.G. deBruijn, "Additional Comments on a Problem in Concurrent Programming Control," Comm. ACM, Vol. 10, No. 3, March 1967, pp. 137 - 138.

[CH] R.H. Campbell and A.N. Habermann, "The Specification of Process Synchronization by Path Expressions," Operating Systems, Lecture Notes in Computer Science, Vol. 16, Springer-Verlag, 1974, pp. 89 - 102.

[EWD65] E.W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," Comm. ACM, Vol. 8, No. 9, Sept. 1965, p. 569.

[EWD74] E.W. Dijkstra, "Self-stabilizing Systems in Spite of Distributed Control," Comm. ACM, Vol. 17, No. 11, Nov. 1974, pp. 643 - 644.

[EM] M.A. Eisenberg and M.R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Problem," Comm. ACM, Vol. 15, No. 11, Nov. 1972, p. 999.

[ANH] A.N. Habermann, "Path Expressions," Technical Report, Dept. of Computer Science, Carnegie-Mellon University, June 1975.

[CARH71] C.A.R. Hoare, "Towards a Theory of Parallel Programming," International Seminar on Operating System Techniques, Belfast, Northern Ireland, Aug. - Sept. 1971. Also in Operating Systems Techniques, Ed. by C.A.R. Hoare and R.H. Perrott, Academic Press, 1972, pp. 61 - 71.

[CARH74] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," Comm. ACM, Vol. 17, No. 10, Oct. 1974, pp. 549 - 557.

[HPK] H.P. Katseff, "A Solution to the Critical Section Problem with a Totally Wait-free FIFO Doorway," Internal Memorandum, Computer Science Division, University of California, Berkley. Extended abstract in "A New Solution to the Critical Section Problem," Proc. of the Tenth Annual ACM Symp. on Theory of Computing, May 1978, pp. 86 - 88.

[DEK] D.E. Knuth, "Additional Comments on a Problem in Concurrent Programming Control," Comm. ACM, Vol. 9, No. 5, May 1966, pp. 321 - 322.

[LL74] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," Comm. ACM, Vol. 17, No. 8, August 1974, pp. 453 - 455.

[LL76] L. Lamport, "The Synchronization of Independent Processes," Acta Informatica, Vol. 7, Fasc. 1, 1976, pp. 15 - 34.

[PF] G.L. Peterson and M.J. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System," Proc. of the Ninth Annual ACM Symp. on Theory of Computing, May 1977, pp. 91 - 97.

[RP] R.L. Rivest and V.R. Pratt, "The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report," Proc. of the 17th Annual Symp. on Foundations of Computer Science, Oct. 1976, pp. 1 - 8.

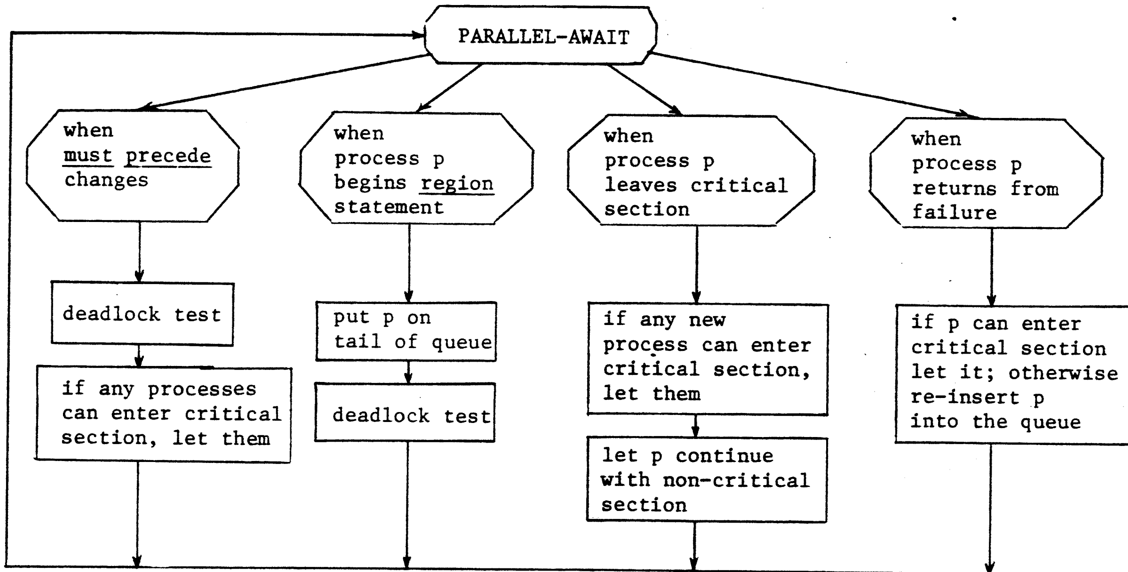


Figure 1: Version 1 Common Scheduler

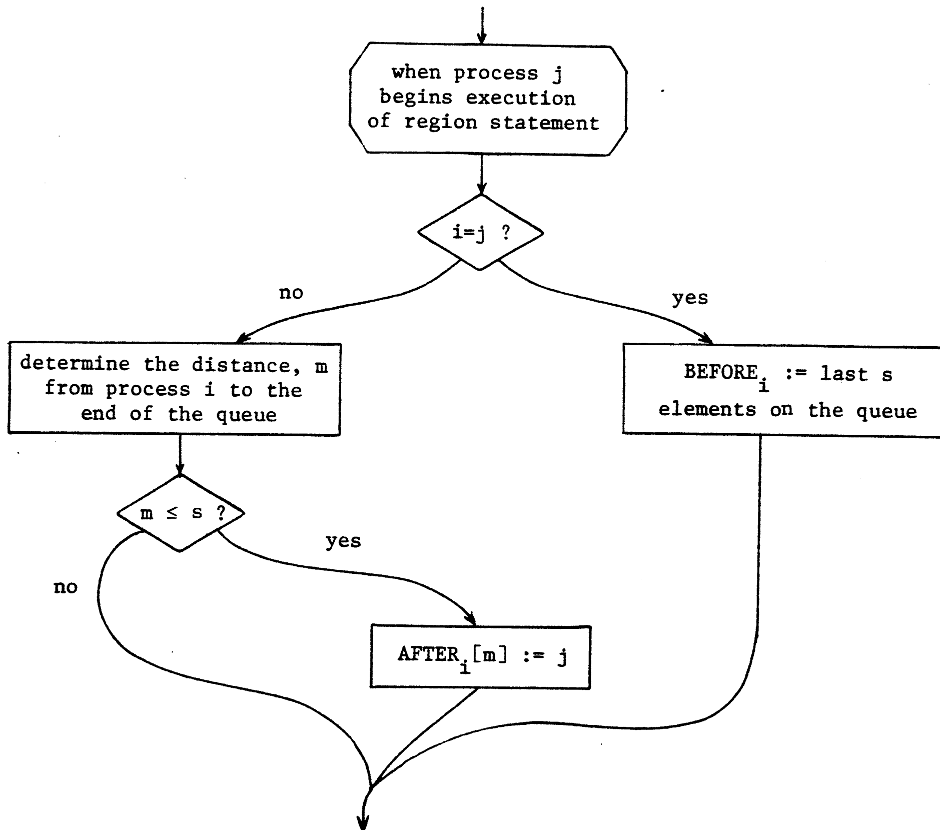


Figure 2: Version 2 Instruction for Process i : Enqueue Process j.

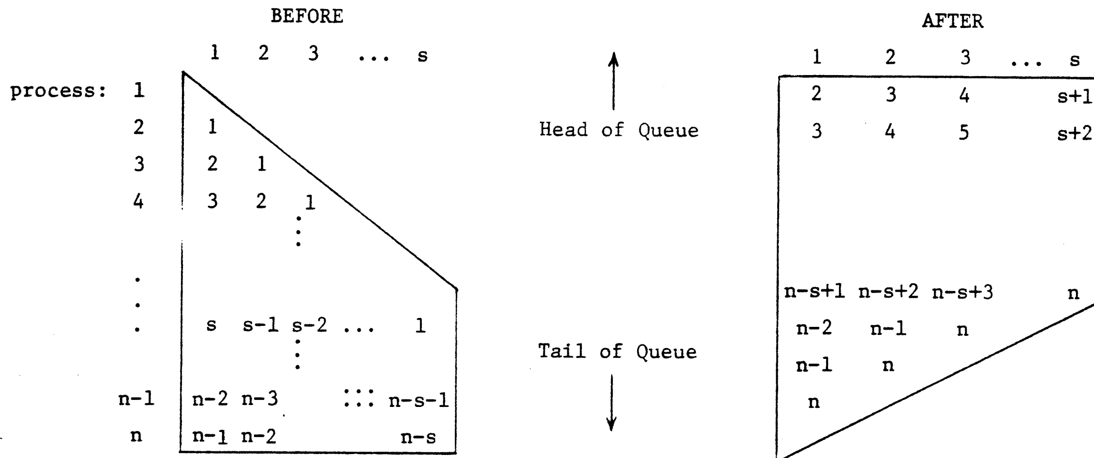


Figure 3: Structure of the BEFORE and AFTER arrays.

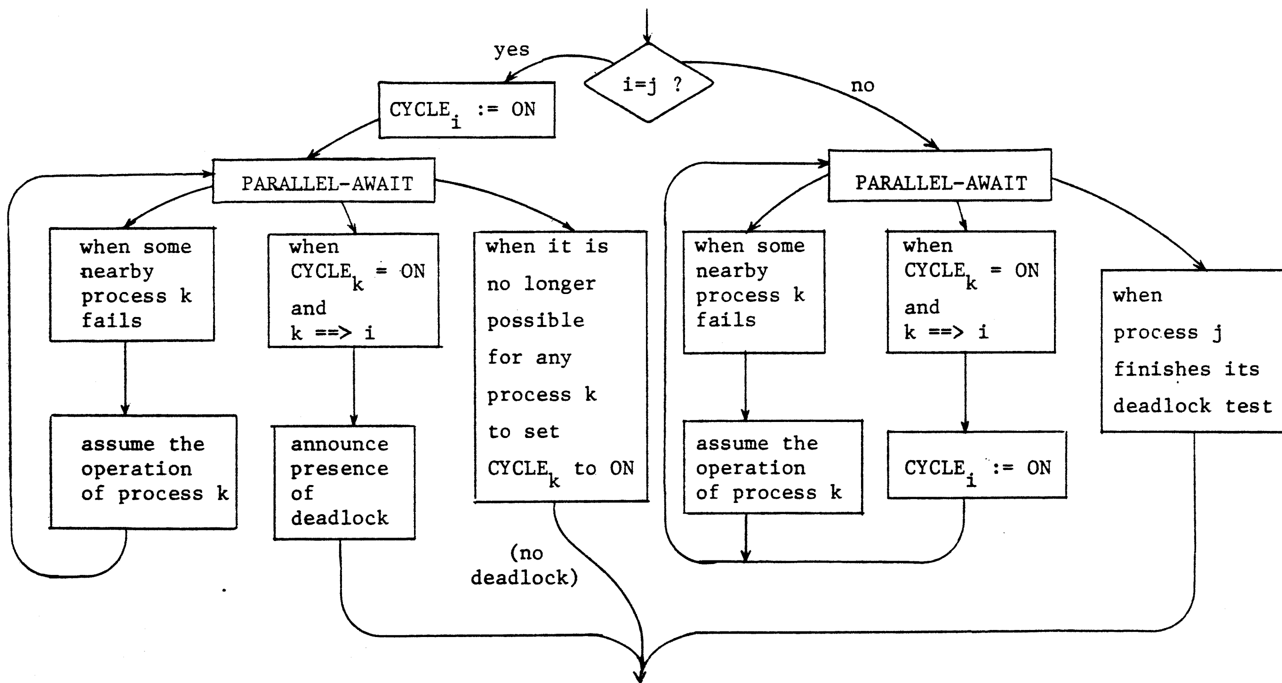


Figure 4: Version 3 instruction for process i to determine if process j is caught in a deadlock.