A SURVEY OF SYNCHRONIZATION PROBLEMS*

George Holober

Technical Report #181, February 1980

*This research was supported in part by Office of Naval Research Grant N00014-75-C-0752. A large portion of this work was done while the author was a summer visitor with the Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

ABSTRACT

Decreased hardware costs will facilitate the use of parallel computation in the near future. Synchronization primitives will be needed to implement concurrent algorithms. Many such primitives have been proposed to date. Unfortunately, their power can only be accurately measured in terms of their ability to solve a particular set of synchronization problems. A correspondingly large number of such problems have been proposed along with the primitives. We present some of these synchronization problems and outline their parameters, variations and histories.

INTRODUCTION

The past fifteen years have witnessed a tremendous shift in the factors that contribute to the cost of computation. Whereas hardware was at one time the dominant factor, it is software development and maintenance that now overwhelmingly influence the price of computing. The decline in the cost of hardware, both in relative and absolute terms, has made feasible the application of multiple processing units to a single program as a technique for speeding up the execution of that program. The trend toward parallel computation is likely to increase as VLSI circuit technology lowers the cost of computing systems even further, and as the physical limits on the speed of a single processor are approached.

Concurrent computation is not without its difficulties. People seem best suited to solving problems in a sequential fashion, and the use of von Neumann-style machines since the 1940's has inculcated us with the idea that programs must also run sequentially. One of the primary problems with implementing parallel algorithms on an asynchronous computer is the need for <u>synchronization</u>. Processes must

cooperate if they are to maintain the integrity of globally accessible data. In addition, cooperation implies the need for interprocess communication, which in turn necessitates a synchronizing mechanism that enables messages to be handled properly. One notion of why synchronization may be difficult to achieve is described by Parnas and Siewiorek [PS75].

Literally dozens of synchronization primitives have been studied since the difficulties inherent to parallel computation were first discovered. (A very nice overview of the history of problems and mechanisms related to concurrency has been presented by Brinch Hansen [BH79].) The "power" of a such primitives (by whatever metric synchronization performance can be measured) can only be determined relative to their ability to solve a particular class of problems. One plausible notion of the power of a synchronization primitive is due to Lipton [Lip73], [Lip74]: primitive A is more powerful than primitive B with respect to some problem if A can solve the problem while B cannot. Further research along these lines has been pursued by Lipton, Snyder and Zalcstein [LSZ74], [LSZ75], Dolev [Dol79], and Henderson and Zalcstein [HZ80].

When introducing a new primitive, a researcher will invariably describe how it can applied to some select set of problems. In a display of modesty, he may even explain why the solution to some problems lies beyond the scope of his synchronization mechanism, or why it suffers from some other undesirable characteristic (such as inefficiency stemming from busy-waiting). Not surprisingly, the invention of many synchronization primitives has led to the development

of correspondingly large set of synchronization problems. It is important for computer scientists to appreciate and understand some of these problems if they are to have a basis for judging the capabilities of the new synchronization schemes that are sure to be proposed over the coming years.

EXAMPLES OF SYNCHRONIZATION PROBLEMS

In this section we examine numerous synchronization problems and their variations. These problems have several sources of origin. As mentioned above, some were developed to prove or refute the efficacy and/or efficiency of a particular synchronization mechanism. Some are problems that might be encountered when designing an actual multiprocessing system. And between these two extremes of "toy" vs. "real-world" problems, we find some problems that bridge the gap: they have been distilled from actual problems through the removal of low-level details so that the result will more readily succumb to rigorous analysis.

The difficulties associated with developing precise specifications for these problems has been studied by Czaja [Cza79], who has also examined some critical implementation issues [Cza78]. In addition to those listed below, a number of other problems arising from memory management in operating systems can be found in [Hoa73].

Mutual Exclusion

1. No Restrictions. The most elementary version of this problem is simply stated. Every process has a portion of code known as the critical region or critical section. The processes execute in parallel, though a process is forbidden from entering its critical section if any other process is in the midst of executing its respective critical section. Mutual exclusion is generally needed to prevent more than one process from simultaneously accessing some resource if the characteristics of the resource deem such multiple accesses meaningless. For example, if two or more processes were to attempt to store different values into the same memory location or to utilize the same printer at the same time, the result would most likely be gibberish. Dijkstra [Dij65], [Dij68a] was the first to examine mutual exclusion as well as to develop instruction protocols and primitive mechanisms with which to implement it. (As an aside, we note that Dijkstra's PV semaphore operations are not merely of theoretical interest; they have in fact found application in the construction of the Venus [Lis72] and Boss 2 [Lau75] operating systems.)

2. Bounded Number Exclusion. This version of the problem is exactly the same as the previous version, but we relax our requirements slightly to allow as many as k processes to execute their critical sections simultaneously. The value of k is finite and fixed.

3. Deadlock-free Exclusion. A solution to any of the previous problems would be vacuously correct if the introduction of synchronization

protocols caused all the processes to cease doing productive work. At the very least, we would want any solution to a synchronization problem to be deadlock-free, i.e. there is always one process which continues to make progress through its critical region. Though all mutual exclusion schemes in the literature implicitly satisfy this requirement, it was not stated explicitly until Burns et al. [BFJLP78] determined the limits of the size of inter-process messages used by exclusion mechanisms satisfying, among other properties, deadlock-freeness.

4. Lockout-free Exclusion. Dijkstra's [Dij65] original algorithm contained a subtle but important flaw. Though it would perform as specified, it permitted some processes to compete unfairly for the use of its critical section. In particular, if a set of processes were in contention for exclusive access to some resource, it would be possible for some of these processes to enter, leave and reenter their respective sections before the remaining processes could ever begin executing their own critical sections. Continuing in this mannner, some of the processes would never be granted access to the resource. These processes are said to suffer from lockout, starvation, or livelock. Freedom from lockout implies freedom from deadlock.

Knuth [Knu66] was the first to recognize the difficulties caused by lockout and to offer a solution that avoids it altogether. To prevent lockout, we must guarantee that each process contending for access to the critical section will eventually be granted that right. The delay time between requesting and gaining entry to the critical section is therefore finite, though it need not be bounded. Two other schemes that provide freedom from lockout have been developed by Kessels and Martin

[KM79] and by Morris [Mor79]. A weak primitive that allows fair exclusion (or can alternatively be used to allow an unbounded number of processes to leave or enter the system) has been proposed by Goodman [Goo76].

5. Bounded Waiting. The time a process waits between requesting access to its critical section and receiving that right is bounded by some fixed constant k. It would be difficult, if not impossible, to make k some absolute measure, such as the number of seconds passing on a clock. The reason for this is simple; since we are dealing with totally asynchronous systems, we cannot know (by definition of such systems) precisely how long it will take any process to execute any instruction at any moment, other than it will be a finite amount of time. As a result, we cannot know how long a process will occupy its critical section, other than this too must be a finite time span. If k were bounded even though the occupancy time of a critical section were not (as is the case here), then a situation could arise wherein process A has waited the full k seconds to enter its critical section while process B is still in its critical section and is not yet ready to yield its exclusive occupancy of that region. If the bound k is to be respected, then A must also begin executing instructions in its critical section at the same time as B, a clear violation of mutual exclusion.

More commonly, k will be a relative measure of passing events. A frequent definition used for k is the maximum number of times each of the processes (other than the one waiting to execute its critical section) may enter and leave its own section before the process in question is finally granted the access it desires. Whatever metric is

chosen for k, it must be such that no two processes could reach and exceed it at the same time. For if this were to happen, both processes would simultaneously be required to enter their critical sections, and the most basic constraint of mutual exclusion would be violated. Knuth's lockout-free scheme did in fact provide for bounded waiting, and the bound established in his paper was subsequently lowered by DeBruijn [DeB67] and by Eisenberg and McGuire [EM72]. A more recent proposal by Elgot and Miller [ElM79] achieves a very tight bound by having the right to enter the critical section passed among the processes in a round-robin fashion.

6. Generalized Fair Exclusion. We have a set $P = \{P_0,P_1,\ldots,P_m\}$ of processes. Each process in this set spends a possibly infinite amount of time thinking followed by a finite period of eating, and this protocol is repeated an indeterminate number of times. We also have a set $S = \{S_0,S_1,\ldots,S_n\}$, where each S_i, $0 \leq i \leq n$, is a subset of P. These subsets define the exclusion constraints. If $\{P_b,P_c,\ldots,P_z\}$ is the set of processes that are currently eating (initially this set if empty), then another process P_a can stop thinking and begin eating only if $\{P_a,P_b,\ldots,P_z\}$ is a subset of some S_i, $0 \leq i \leq n$. To prevent trivial cases of lockout, we assume that every process P_j is an element of at least one subset S_i.

Lockout is nevertheless possible under such a scheme. Consider the problem described by the set of five processes $P = \{P_0,P_1,P_2,P_3,P_4\}$ and the set $S = \{\{P_0,P_2\}, \{P_0,P_3\}, \{P_1,P_3\}, \{P_1,P_4\}, \{P_2,P_4\}\}$. These parameters define the five dining philosophers problem. In a later section, we show that this problem may subject processes to starvation.

Dijkstra [Dij72a] was instrumental in developing this problem and in establishing criteria needed to prevent starvation. A process that wishes to eat is said to be "hungry," and a hungry process that cannot be serviced immediately will sleep until the time comes when it can eat. Two properties are then necessary and sufficient to guarantee no starvation:

1) If a process is sleeping then some other process must be eating or is about to resume thinking.
2) If process P_i is hungry, then permission to eat cannot be granted to other processes more than N_i times before P_i itself is allowed to eat, where N_i is a predetermined constant that serves as an upper bound for P_i.

These two properties clearly induce a modified form of bounded waiting on the system. Devillers and Lauer [DL76] have since developed a general mechanism for solving this problem.

7. First-In-First-Out (FIFO) Scheduling. Processes enter their critical sections in the same order in which they requested access to it, thereby making this an even stronger condition than bounded waiting. FIFO scheduling is, in one sense, the "fairest" of all scheduling policies, for the worst case waiting time of any individual process is minimized. Katseff [Kat78] was the first to develop protocols for implementing this type of mutual exclusion.

8. Last-In-First-Out (LIFO) Scheduling. Processes enter their critical sections in the reverse order from the way in which they requested access to it. Such a scheduling policy leaves open the possibility of

lockout in the following circumstance: process A is in its critical section while processes B and C are awaiting entry to their critical sections. B will be granted priority over C, since C began waiting before B. Process A leaves its critical section, and B enters its own critical section. Before B releases exclusive possession of its critical section, A makes a request to reenter its own section. Since this request certainly comes after the request of C, A will have priority over C. Now the situation is exactly as it was in the beginning, with only the roles of A and B reversed. As long as these two processes continue to swap control of the critical section in this fashion, C will never gain the right to enter its critical section, and would therefore suffer from lockout.

9. Arbitrary Exclusion and Scheduling. Lamport [Lam76] developed this problem to illustrate the use of one plausible synchronization mechanism that might be used in a distributed, fault-tolerant system. (Lamport first examined mutual exclusion in a distributed environment several years earlier [Lam74].) When process A signals its intent to enter a critical section, it chooses and is assigned a mode value M_A that cannot be altered until the critical section has been successfully executed. There is a symmetric binary predicate conflict on the set of mode values that may not change over time. There is also a predicate should-precede which is very flexible; given two processes and an arbitrary assortment of other (possibly time-dependent) data, it determines which of these two processes should have precedence over the other when both are waiting to access their respective critical sections.

One restriction is placed on the nature of should-precede. The arguments to the function may not change value while the function is being evaluated. This constraint makes impossible the existence of a function that cannot return an answer before the values of its operands become obsolete. A strongly constant function (as Lamport terms a function which obeys this restriction) will not cause race or other ill-defined situations.

A process A that has mode value M_A and that is waiting to enter its critical section will be allowed to proceed providing three conditions are satisfied:

1) There is no process B presently executing its critical section such that B has mode value M_B and conflict(M_A,M_B) = true;
2) There is no process B that is also waiting to access its critical section and that has mode value M_B, where conflict(M_A,M_B) = true and should-precede(B,A) = true; and
3) Of all the processes awaiting access to their critical sections that satisfy the first two conditions, process A has been waiting the longest.

The tremendous power provided by the conflict and should-precede functions enable us to simulate any of previous eight versions of the mutual exclusion problem. In fact, virtually every imaginable variation of mutual exclusion could be described in terms of this problem. Of course, with all this flexibility, we also run the risk of underspecifying the functions. If this were to happen, deadlock, lockout or almost any other unwanted system behavior could develop. Ford [For78] has shown that by sacrificing a small measure of this

flexibility, efficient solutions to mutual exclusion and related readers/writers problems can be obtained. Similar, though more constrained, results can be achieved using PVchunk [VvL72] or pe/ve [Age77] instructions.

The concept of mutual exclusion appears to be so essential to any form of synchronization that it is referred to, either implicitly or explicitly, in virtually every paper dealing with this subject. While the need for critical regions can be directly attributed to Dijkstra, the idea of using special instructions to regulate parallelism can be traced at least back to the fork/join [Con63, And65] and do-together/hold [Opl65] operations. Other early devices to control simultaneous resource access include lock/unlock [DVH66] and count matrices [VH66]. Some of the theoretical aspects that govern solutions to mutual exclusion problems are discussed by Gilbert and Chandler [GC72], along with a technique for synthesizing solutions to problems based on the states of individual processes. In addition, a large class of high level language constructs, such as enclosures [DD76], have been proposed to aid the programmer's task of specifying the exclusion constraints.

Readers/Writers

The readers/writers problem and two of its variants (reader priority, writer priority) were developed by Courtois, Heymans and Parnas [CHP71] to illustrate the flexibility of Dijkstra's semaphore primitives for solving various exclusion and scheduling problems. Inadvertently, they may also have shown just how impractical P and V can

be when dealing with non-trivial problems. Their code is complex and difficult to understand; the thought of deriving solutions to similar or more involved problems is mind-boggling.

Some resource is accessed by a set of processes. These processes are categorized in one of two ways: either as a reader or as a writer. Readers examine the state of the resource but make no attempt to alter it. Since a reader preserves the integrity of this state, any number of readers can access the resource concurrently and each will interpret the state of the resource correctly. Writers, on the other hand, do modify the resource in some manner. If a reader were to examine the resource at the same time a writer were changing it, the reader might see part of the old state and part of the new state (plus any intermediate states). Since the outcome of such a read would be incorrect (or even meaningless), the operation of a reader and a writer must be mutually exclusive events. If two writers attempt to modify the resource simultaneously, the result is not likely to be the target state of one writer or the other writer, but a combination of the two (i.e. gibberish). Consequently, writers must exclude other writers when altering the state of the resource.

1. Reader Priority. Readers have absolute priority to use the resource. If a read request is made and a writer is not currently active, the reader is serviced immediately. If a writer is active, then all pending readers are granted access to the resource as soon as the writer has finished. A writer can become active only if there are no readers either pending or active and there are no other writers currently active. Clearly, the rapid and continuous arrival of read

requests could cause writers to be locked out indefinitely. In fact, since no ordering is placed upon the pending writers, a policy of servicing writers after a bounded interval of time will not prevent individual writers from suffering lockout.

2. Writer Priority. The above-mentioned priority is reversed. A writer may access the resource providing no other process (reader or writer) is in the midst of doing the same. Only when no writers are either pending or being serviced will all the readers be granted the opportunity to perform their function.

3. First-Come-First-Served (FCFS) Scheduling. Reader priority (respectively, writer priority) scheduling will permit individual processes to be locked out, since the rapid arrival of readers (respectively, writers) will prevent long waiting writers (respectively, readers) from gaining access to the resource. To avoid lockout, we could simply service processes in the same order in which they request access to the resource and without regard to the type of process. A writer gains access to the resource after all the processes which have been waiting for a longer period of time are through examining or modifying the resource. A reader may be serviced if no other process has been waiting longer than it has and a writer is not currently in control of the resource (though other readers may be). Lockout is avoided using this scheme, though throughput suffers a decrease since readers will back up behind writers that have been waiting longer, even if other readers are presently accessing the resource.

4. Fair Readers Scheduling. Hoare [Hoa74] described this scheduling technique, which prevents lockout and gives readers some measure of priority over writers. If a read request is made while no write requests are either pending or active, then the read is serviced immediately. If, on the other hand, some writers are pending or active when the read is requested, then one of these writers must be fully serviced before the reader can access the resource. When a writer finishes, all pending readers are granted access to the resource. Among only the writers, requests are serviced on the basis of first-come-first-served.

5. Structured and Dynamic Resources. The readers/writers problem takes on an entirely new dimension if the resource is "structured," i.e. it has many interrelated components, each of which may or may not be accessed concurrently. The problem is further complicated if this structure can be altered over a period of time. Synchronization schemes for tree-shaped resources are described in [BS77], [BN78], [MS78], [Com79], [Ell79] and [KuL79]. Owicki [Owi77] has developed techniques for verifying parallel access to complex resources, such as buffers and queues. Even more elaborate structures can be manipulated by the methods of Kung and Robinson [KuR79]. Generalizing this problem completely, the resource changes in nature from data stucture to data base. Ullman [Ull80, chapter 10] outlines the difficulties associated with multiple processes that are simultaneously attempting to access a dynamically changing data base.

Readers/writers has been one of the most intensely studied synchronization problems. The literature that makes reference to it is vast. Brinch Hansen [BH72a] contrasts solutions based on semaphores and conditional critical regions, and his conclusions are debated back and forth in [CHP72] and [BH73a]. The application of various P/S semaphore primitives to a variety of readers/writers problems has been surveyed by Presser [Pre75]. Additional solutions have been developed over the years, utilizing such varied synchronization schemes as PVchunk [VvL72], PVgeneral [Cer72], up/down [Wod72], conditional critical regions and await statements [BH72b], conditional critical regions with priority [Pet75], sets of interacting processes [New75], pe/ve [Age77], a conditional wait variation of monitors [Kes77a], P* [Con77] (which is corrected and modified in [SJ78]), capability managers [KS78], distributed processes [BH78], guarded regions [BHS78], flow expressions [Sha78], sentinels [Kel78], predicate path expressions [An79] and serializers [HA79].

A host of issues related to readers/writers have been detailed in the literature. Included among these issues are:

1) deadlock detection and recovery [BN76];

2) efficiency [Sch76];

3) simultaneous reading and writing [Lam77b];

4) allowing writers to read as well as write, as in the "secure readers/writers problem" described and solved (using eventcounts and sequencers) by Reed and Kanodia [RK79];

5) formal semantics [Gre75] and specification [Gre77];

6) non-procedural solutions [Kes77b];

7) hardware design (e.g. a method for implementing lock/unlock

instructions using a read/interlock protocol is presented in [Hil73]); and

8) verification techniques designed to utilize the properties of Petri nets [Kel76], PV operations [Lev72] and programs employing no special synchronizing device other than exclusive memory access [Bab79].

Producer/Consumer

When two processes A and B execute in parallel with some joint goal, they will undoubtedly have to share information at some point. A situation may arise wherein A is ready to receive information from B before B has produced it. Dijkstra [Dij68b] was among the first to describe and implement a solution to this problem in the "THE" multiprogramming operating system; other aspects of the problem are discussed by Habermann [Hab72a] and by Brinch Hansen [BH73b]. The development of synchronization mechanisms which allow communicating processes to overcome the difficulties inherent to this situation is known as the producer/consumer problem. Process B, which will eventually create the message, is the producer, and process A, which will receive the message, is the consumer.

One technique that is frequently used to implement interprocess communication is the establishment of a buffer in a global portion of memory. The buffer has a role analogous to that of a mailbox: the producer leaves messages in the buffer, where they remain until the consumer is ready to examine them. Since the size of a buffer is likely to be specified at system generation time, we will probably not have the option of dynamically increasing this size as more messages arrive.

(Nor would we necessarily want to expand the buffer. Presumably, after the consumer has processed awaiting messages, the content of these messages becomes obsolete, and the space they occupy can be made available to store heretofore undelivered messages.) Fixed buffer size adds a new constraint to the problem. Not only must the consumer wait if messages are not available for it, but now the producer must wait before storing new messages in the mailbox if the box is already full of information that has not yet been examined by a consumer. Difficulties of this nature are likely to arise when message passing protocols are used as the basis of programming languages, such as PLITS [Fel79], or as the basis of operating systems, such as those employed by the Regnecentralen 4000 [BH70] and CDC 6600 [Gai72] computers. Several variations of this problem have been studied in the literature, and they are outlined below.

1. Single Slot Mailbox. The mailbox has room for a single message. Producers and consumers must therefore alternate accessing the buffer. Since we would not want a consumer to examine a message before it has been completely written by a producer, and since we would not want a new message to be written before the old message has been thoroughly examined by the consumer, a process (be it producer or consumer) must have exclusive control of the resource. Clearly, the producer/consumer problem will assume many of the characteristics of the readers/writers problem. This highly restrictive version of the problem was first mentioned by Habermann [Hab72a] and was later used by Campbell and Habermann [CH74] to demonstrate one application of path expressions.

2. Bounded Buffer (Multiple Slot Mailbox). This problem is identical to the one described above, though the size of the buffer is increased to allow more than one message to reside in it at any time. Typically, a circular buffer would serve as the mailbox, with pointers indicating the bounds of the non-obsolete messages. Hoare described this problem along with a solution based upon the monitor concept [Hoa74] and again with a solution based upon the concept of communicating sequential processes [Hoa78]. Methods of synthesizing solutions to the bounded buffer problem have developed by Lavenberg [Lav78] and van Lamsweerde and Sintzoff [vLS79], the latter of which places heavy emphasis on ideas for avoiding deadlock and starvation.

3. Parallel Bounded Buffer. When moving from the single to multiple slot mailbox versions of the producer/consumer problem, we preserve the constraint that prevents two or more processes from accessing the resource concurrently. In the second version of the problem, this constraint becomes unnecessarily strong, since we only wish to avoid the possibility of multiple processes accessing the same portion of the buffer at the same time. In our new variation of the problem, processes may deposit or remove messages in parallel, providing each process is treating a unique message. Habermann [Hab72a], Owicki and Gries [OG76a], Lamport [Lam77a] and Yonezawa [Yon77] use this problem to illustrate their ideas on program verification.

4. Information Streams. Dijkstra [Dij72b] invented and solved this modification of the bounded buffer problem. Additional solutions were developed by Cooprider et al. [CHCP74] and by Kessels [Kes77a]. There

is a collection of pairs of producers and consumers. Each $pair_i$ is coupled via an information stream of n_i messages. All messages are the same length, and all the messages must be stored in a buffer having size tot. As in the other versions of this problem, we wish to synchronize the producers and consumers so that at any time $0 \leq n_i$ for all i and $\Sigma_i n_i \leq tot$.

Further constraints are placed on the problem. If a consumer is unable to continue its function for lack of waiting messages, it must be allowed to sleep until new messages become available. If the sleeping consumers are then reawakened in some even-handed fashion (such as FIFO ordering), a slow consumer has the opportunity to retard all the other consumers to its speed. Still worse is the consumer that has stopped. Since its information stream may occupy the entire buffer, all other streams are blocked. The additional constraints on the problem would prevent such inefficiencies from occurring.

The producer/consumer problem nearly rivals the readers/writers problems in terms of the amount of attention it has received in the literature. Early references to producer/consumer-like problems included solutions utilizing ports [Wal72] and block/wakeup [Sal66]. (This latter paper was extended in [Lam68] to allow priority scheduling.) Solutions based on different semaphore operations are examined by Presser [Pre75]. Much more recently, a system called "COSY" has been devised to solve a wide variety of producer/consumer problems [LTS79]. Other mechanisms include conditional critical regions with await statements [BH72b], regular expressions [Sc76], Modula signals [Wir77a, Wir77b], distributed processes [BH78], communicating sequential

processes [Hoa78], eventcounts and sequencers [RK79], flow expressions [Sha78], sentinels [Kel78] and synchronizing resources [And79].

Other topics related to this problem have been studied by researchers. A few of these topics are:

1) operation in a real-time environment [Wir77c];
2) permitting a process to produce or consume a variable number of messages at one time [VvL72]; and
3) verification methodologies, such as the scheme developed by Howard [How76b] for use with monitors.

Joint Checking Account

The joint checking account problem was developed by Howard [How76a] to demonstrate the relative capabilities of various signaling protocols for monitors. A bank account is shared among a number of people (in his examples, Howard uses just two people: a husband and wife). Any person may access the account by using the operations deposit(k) and withdraw(k). The first of these operations causes the balance in the account, B, to be increased by k dollars. The second operation has precisely the opposite effect: it decrements B by k.

There are two further restrictions on the sum we may maintain in the bank at any time. It may never fall below zero, and it may never rise above some predetermined upper bound M (presumably the FDIC will not insure amounts greater than M). To insure these conditions are never violated, we might only permit a withdraw(k) operation to be serviced if $k \leq B$. Going in the opposite direction, a request to

<u>deposit</u>(k) might be serviced only if $k \leq M-B$. Any operation which cannot be serviced will be left pending until the condition upon which it is predicated becomes true.

In fact, we will extend the conditions of servicability to permit single transactions of greater than M dollars. Assume operations <u>withdraw</u>(w_1), <u>withdraw</u>(w_2), ..., <u>withdraw</u>(w_a) and <u>deposit</u>(d_1), <u>deposit</u>(d_2), ..., <u>deposit</u>(d_b) are pending. They all become eligible for servicing if $0 \leq (d_1+d_2+...+d_b)-(w_1+w_2+...+w_a) \leq M-B$. Of course, the mechanism which implements this procedure would need to break down and splice back together the individual operations in such a fashion so as to prevent B from dropping below 0 or from rising above M.

To see that lockout is possible under this arrangement, consider the following example. The account is initially empty, so B=0. A request to withdraw 10 dollars is made, but must remain pending until $B \geq 10$. Next, a <u>deposit</u>(5) operation is initiated and serviced, so B rises to 5. Following that, a <u>withdraw</u>(5) operation is initiated, and it too is serviced to completion, reducing B to 0. We are then back at the initial state, and the <u>withdraw</u>(10) operation still must remain pending. If small deposits and withdrawals continue to alternate in this fashion, the larger withdrawal can never be serviced, even though it has been waiting longer than any other request.

Without too much difficulty, we could dream up a scheduling policy that implements the following idea. The longer a withdraw (respectively, deposit) request cannot be serviced, the more deposit (respectively, withdraw) requests will be left pending, until this withdrawal (respectively, deposit) and the deposits (respectively,

withdrawals) all become serviceable (even if some of the deposits (respectively, withdrawals) can be serviced at an earlier time). Assuming a fairly regular mix of operations (imagine what would happen if, contrary to this assumption, deposits greatly outnumbered withdrawals, or vice versa), no one operation will be locked out forever. The throughput of the system will suffer as a result of detaining operations which are eligible for servicing.

Five Dining Philosphers

This problem was developed by Dijkstra [Dij71]. It was used soon thereafter in a discussion of synchronization problems written by Hoare [Hoa72], who used it again later to illustrate one application of communicating sequential processes [Hoa78]. Five philosophers {$Philosopher_0$, $Philospher_1$, ..., $Philosopher_4$} sit around a table. In front of each of them lies a plate full of spaghetti, and to the right of each plate lies a fork (i.e. for each i, $0 \leq i \leq 4$, $Fork_i$ is by the right arm of $Philospher_i$). This arrangement is shown in Figure 1.

Each philospher divides his time between thinking and eating. Unfortunately, because the spaghetti is very tangled, a philospher will need the use of the two closest forks in order to eat (i.e. $Philosopher_i$ must have $Fork_i$ and $Fork_{(i-1) \bmod 5}$ before he can begin eating). Only one philospher can be using a particular fork at any time. When a philospher finishes eating and resumes thinking, he replaces his two forks back on the table so that they will be available in the future when he or others may need them.

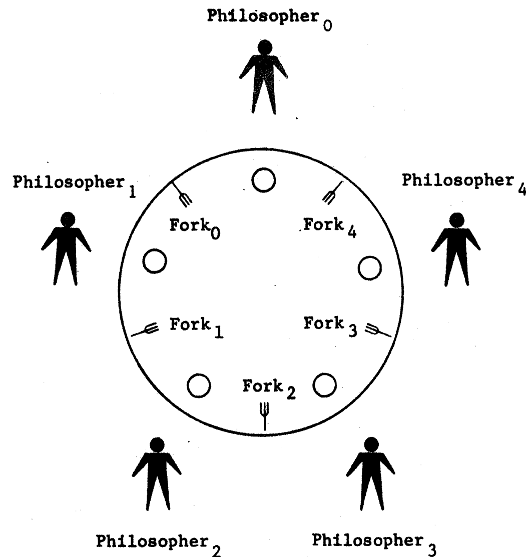

Figure 1: Five Dining Philosophers

The crux of this problem is making sure that Philospher_i never grabs Fork_i unless he is also sure that he can grab $\text{Fork}_{(i-1) \bmod 5}$. Assume Philospher_i were to grab Fork_i, and before he could also pick up $\text{Fork}_{(i-1) \bmod 5}$, $\text{Philospher}_{(i-1) \bmod 5}$ were to grab it. Philosopher_i would then be left holding Fork_i but would be unable to eat with it. Assume too that $\text{Fork}_{(i+1) \bmod 5}$ were lying on the table and that $\text{Philospher}_{(i+1) \bmod 5}$ wanted to eat. He would be prevented from doing so because Fork_i is unavailable, even though Philosopher_i is not using it productively. Thus $\text{Philospher}_{(i+1) \bmod 5}$ would be unable to proceed, and quite unnecessarily; the overall result is a decrease is system throughput.

Even worse situations than the one described above are possible. Assume that each Philosopher_i has grabbed Fork_i. Every philosopher gained control of exactly one fork before any philospher could gain

control of two. Now every philosopher has a fork that another philospher needs, and since no one of the men can proceed, the forks needed by others are never released. The system is therefore deadlocked, and productivity drops to zero.

Of course, even if these problems are remedied, there remains a possibility of lockout. Suppose that $\text{Philosopher}_{(i-1) \bmod 5}$ is eating, and before he finishes, $\text{Philosopher}_{(i+1) \bmod 5}$ also begins to eat. In additon, suppose that before the latter philosopher finishes eating, the former resumes attending to his plate of spaghetti. If these two philosphers continually alternate eating in this fashion, one or both of $\text{Fork}_{(i-1) \bmod 5}$ and Fork_i will be in use at all times. Since the two forks that Philosopher_i needs will never be available simultaneously, he is forever prevented from eating, and he will suffer, quite literally, from starvation. One strategy for preventing this occurrence is developed by Courtois and Georges [CG77]. A paradigm for proving the correctness of solutions has been outlined by Owicki and Gries [OG76b].

Cigarette Smokers

The cigarette smokers problem centers around the distribution of three resources: tobacco (abbreviated by "T"), paper ("P") and matches ("M"). There are three agents, each of whom possesses two of the resources, and no two agents possess the same two resources. The agent possessing resources X and Y will be labeled Agent_{XY}. There are also three smokers. Smoker_T already has tobacco but needs paper and matches in order to smoke, Smoker_P has paper but needs tobacco and matches, and Smoker_M similarly has matches but needs tobacco and paper.

Between the agents and the smokers lies a table. The table is such that at most one agent can access it at any time, though a smoker may access the table even if another smoker or agent presently has access to it. An agent (say $Agent_{TP}$) will gain control of the table, and will not relinquish that control until he has placed his two resources upon the table, one after another. The agents then vie again for control of the table (each agent has an infinite supply of its resources, so the choice of agent to go next is made at random). Before any agent can go again, however, the smoker needing the two resources that have just been put down by the agent (in this case, $Smoker_M$) must access the table and clear it off. This methodology is illustrated in Figure 2.

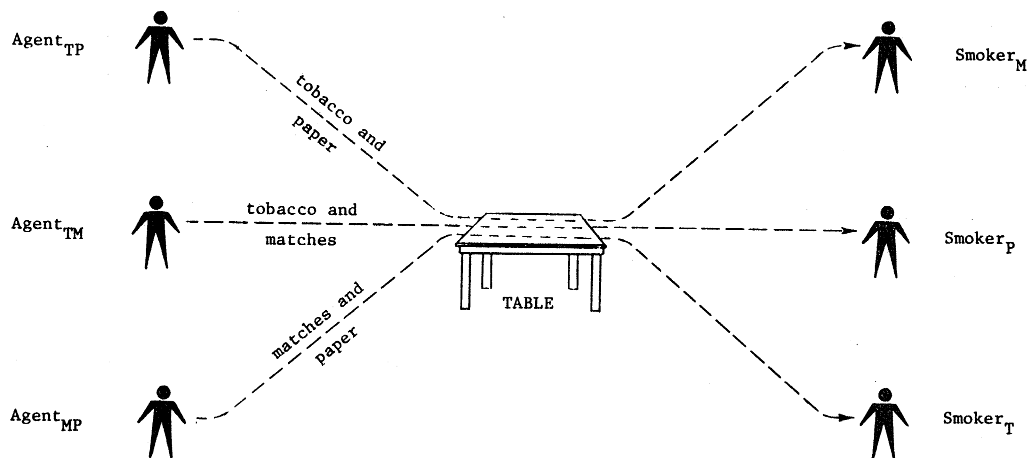

Figure 2: Cigarette Smokers

The difficulty with this problem results from one further restriction, albeit an implicit one. We could arrive at a simple solution if smokers could determine which agent most recently had

control of the table. For example, Smoker_M could wait for Agent_{TP} to finish laying its resources on the table. Instead, the smokers will be aware only of the resources that are presently displayed on the table, and not the identity of the agent that placed them there.

Complications arise in the following situation. Assume Agent_{TP} has control of the table, he has placed the tobacco upon it, but has yet to do the same with the paper. Smoker_M and Smoker_P spot the tobacco, but are unable to determine whether it was deposited by Agent_{TP} or Agent_{TM}. Both of these smokers need the tobacco, and both lay claim to it. Finally Agent_{TP} sets the paper on the table. Now the identity of Agent_{TP} can be inferred by the smokers, but it may be too late: Smoker_M completes his takeover of the resources by grabbing the paper, while Smoker_P, realizing his initial mistake in claiming the tobacco, must somehow relinquish that claim. Whatever synchronization mechanism is used must be sufficiently flexible to allow the forfeiture of claims, or must allow claims to be delayed until the transaction can be completed with certainty.

Patil [Pat71] was the first to describe the cigarette smokers problem. Using an elaborate proof based upon Petri Nets, Patil claimed this problem was not solvable using Dijkstra's P and V operations on semaphores [Dij68a]. In place of P and V, Patil suggested a new, more powerful operation known as PVmultiple. Parnas [Par75] replied to this claim by showing that Patil's assumptions concerning the use of P and V were too strong, and that in fact this synchronization problem is solvable using just these operations. A variety of solutions to the cigarette smokers problem and some generalizations have since been

produced by Habermann [Hab72b] (whose results were later extended by Devillers and Louchard [DL73]), by Lauer and Campbell [LC75] and by Reddi [Red77]. An additional solution using B/F operations has been proposed by Ramsperger [Ram77].

Disk Scheduling

Disk packs are among the most common mass storage devices in use today. They consist of a collection of flat disks which are joined through their centers by a spindle which enables them to rotate at high speed. Each disk has one or two surfaces that are magnetized in order to record data. Every surface is logically decomposed into a set of concentric circles known as tracks, and the tracks are further decomposed into records or sectors. The choice of surface, track and sector collectively determine the address of a particular unit of information. If the tracks on the surfaces are consecutively labeled 1 through n, then cylinder i consists of all tracks labeled i. Above each surface is a read/write head which can store or retrieve data from its respective surface. The heads are joined by an arm, the purpose of which is to position the heads. Figure 3 outlines the structure and major components of a disk pack. Given this configuration for disks, there are still a great number of variables. The answers to synchronization problems will be very sensitive to the way in which these variable factors are fixed for a particular system. The following questions illustrate some of the pertinent concerns.

1) Is the arm stationary, with one head per track, as in a fixed head disk? Or can the arm be positioned so that one set of heads can visit

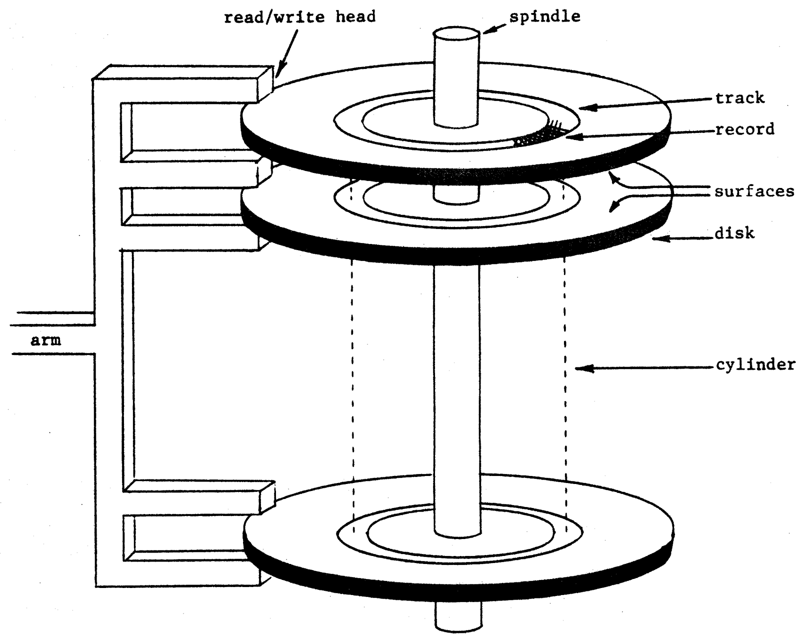

Figure 3: Disk Pack

all the cylinders, as in a moving head disk?

2) Is there just one arm, or could we envision devices with multiple arms, each of them moving independently from the others? The use of a single arm physically constrains us to access distinct cylinders in a mutually exclusive manner.

3) If the arm is positioned at a particular cylinder, can some heads be reading on one set of tracks while other heads are writing on a different set of tracks?

4) At what point is the arm committed to accessing a particular cylinder or record? For example, suppose a head is currently examining record A, and record B is the next one due to be accessed. While seeking record B, a request to access record C is received. If record C is encountered before record B, should the heads temporarily preempt their search for B

in order to examine C, or should they ignore C for the moment and continue on their way to B?

Disks are mechanical devices, and as such are relatively slow. Because of this, their use can be very expensive in terms of the time required for data access and transfer. A well thought-out strategy for utilizing disks can therefore greatly reduce overall processing times. Five disk scheduling policies were compared and analyzed by Teorey and Pinkerton [TP72]. These five alternatives are listed below in order of increasing complexity and presumably better performance.

1. First-Come-First-Served (FCFS) Scheduling. This policy is very similar in nature to the FIFO scheme described under Mutual Exclusion. Requests to access the disk are serviced in the same order in which they are received. Lockout is not possible, though FCFS scheduling "...does not take advantage of positional relationships between I/O requests in the current queue or of the current cylinder position of the read/write heads." These heads, which are movable, will travel in a random pattern seeking the appropriate track. The potential for inefficient use being made of the disk is therefore increased.

2. Shortest-Seek-Time-First (SSTF) Scheduling. The request requiring the shortest seek time is allowed access to the disk first. Seek time refers to the delay incurred when the read/write heads move from their present position to the proper cylinder. SSTF scheduling, while more efficient than FCFS in terms of number of requests serviced during a sufficiently long interval, can lead to discrimination against individual requests. For example, if a stream of requests to nearby

cylinders is continuously received by the disk scheduler, these requests will be serviced quickly. However, if requests to distant cylinders are also enqueued, there may be a long delay before they are serviced. Carried to an extreme, these latter requests can be locked out.

3. SCAN Scheduling. This policy was developed by Denning [Den67]. The heads sweep back and forth across the disk surface, changing direction only at the extreme inside and outside cylinders. The motion of the heads is halted at any track containing a record to which an access request has been made, and the motion resumes in the same direction only after the information transfer has been completed. Among its other virtues (e.g. ease of implementation, good performance), SCAN does not permit the type of discrimination that can occur with SSTF.

Unfortunately, lockout is still a possibility, as the following scenario illustrates. Assume that the scheduler is servicing some request to a record on a particular cylinder. If a new request is made to a record on this same cylinder before the original request is satisfied, the heads will not move and the new request will be the next one serviced. If new requests continually arrive in this manner, the heads will never move, and all pending requests to records on cylinders other than the one currently being scanned will never be granted the attention of the scheduler. Hoare [Hoa74] has overcome this drawback through a scheme that minimizes the frequency at which the heads change direction.

4. N-Step SCAN. Once a set of disk requests have been completed, a new set of up to N pending requests (where N might be variable) is selected

for servicing. Once the members of this set are chosen, the selections cannot be modified. These requests are then fully serviced before another such set of inputs is located. Within each set, an optimal algorithm or near-optimal heuristic is used to determine the ordering of requests that will yield the fastest completion time. The origin of this strategy can be traced to the NSCAN scheduling method described by Frank [Fra69]. It is advantageous in that it preserves the rate of system throughput while lowering the variance of wait times for individual disk access requests.

5. Eschenbach. This scheduling policy was developed by Weingarten [Wei66] for use in message switching systems that are often subject to heavy loads. Assume there are M sectors on each track and that on each revolution of the disk, the heads can access some fraction M/E of these sectors. The heads will begin at the outermost cylinder, where they will remain for E revolutions so that all sectors on these tracks may be accessed. The heads then proceed to the next cylinder for E revolutions, and this process continues until the innermost cylinder is reached. After servicing this last cylinder, the heads return immediately to the outermost cylinder and the entire scan of the disk's surface repeats. This "order E scheme" tends to improve both rotation and seek times.

Of course, these five scheduling algorithms do not exhaust all the possibilities. Many other alternatives have been studied, such as the CSCAN [TP72], FSCAN [CKR72] and MTPT [Ful74] disciplines. The first of these methods is identical to SCAN, except requests are serviced only

while the heads are moving from the outside cylinder inward. In the second method, requests that are made while the heads are sweeping across the face of the disk are serviced on the following sweep. The name "MTPT" is suggestive of its function: it minimizes the total processing time associated with the disk requests.

Alarm Clock

The alarm clock problem was described by Hoare [Hoa74] as a simple example of a scheduled wait. Underlying this problem is a resource which serves as the system clock. There are two operations which act upon this clock. The first operation is tick, whose execution changes the state of the clock to reflect the passage of one unit of time. The other operation is wakeme(k). When a process encounters this instruction, its execution is suspended until k tick operations have subsequently been issued by other (presumably non-sleeping) processes. In addition to the monitor-based solution to this problem presented by Hoare, a solution that uses distributed processes has been developed by Brinch Hansen [BH78].

Deadlock Detection

Habermann [Hab69] was the first to study the problems associated with deadlock. Intuitively, deadlock arises when process 1 cannot proceed until it obtains some resource currently held by process 2, process 2 cannot proceed until it obtains some resource currently held by process 3, and so on, with process n unable to proceed until it

obtains some resource currently held by process 1. The complexity of deadlock avoidance, detection and recovery can be very great, making these potentially costly resource allocation tasks. Questions relating to deadlock come in so many variations, and such a large number of papers have been written on the subject that we could not possibly survey the entire area here. However, Coffman et al. [CES71] and Holt [Hol72] present accurate descriptions of the fundamental characteristics of deadlock.

Garbage Collection

The problem of garbage collection goes back nearly to the beginning of automatic computing. Simply stated, garbage collection involves the location of resources that have been used but are no longer needed by the system, and the return of these resources to a state in which they may be reutilized. Only recently have the added difficulties of garbage collection in a parallel system been analyzed. See [Ste75], [Gri77], [KuS77] and [DLMSS78], along with their references, for a description of some of these problems.

Firing Squad Synchronization

The firing squad synchronization problem is unique among all the problems we have examined thus far. Whereas the other problems are motivated by asynchronous systems, this problem originates from the theory of self-reproducing automata and synchronous logical devices. In addition, it predates the other problems by several years. Moore

[Moo64] gives the first written description of the firing squad problem, and he attributes its origin to some 1957 work of John Myhill. Since then, the problem has been modified, generalized, studied and solved in numerous contexts. Relevant papers include [Wak66], [Bal67], [ML68], [Var69], [VMP70], [Her72], [RFH72], [HLRW74], [Shi74], [Gra75], [Rom76], [LMS77], [Vol77], [Kob77], [Kob78] and [Gol78].

The problem centers around an abstract model of a firing squad. Each soldier is represented by a finite state machine, and the state-to-state transitions of each machine is governed by the state of that machine along with the states of some subset of the other machines. The states of the machines are designated either "quiescent" or "firing." Initially all of the machines are in a quiescent state, and one machine is designated the "general." The general will give the command to fire (he does so by changing states) and thereby initiates a sequence of state transformations among all the machines. The object is for all the machines to reach a firing state for the first time within some small time interval. To make the problem non-trivial, this time interval must be less than the maximum time it takes for any two machines to communicate with each other. The trick is to design the state set and transition rules of the individual machines in such a way that the firing squad correctly performs its function no matter how many machines it is comprised of; i.e. the size of the state sets must be independent of the number of soldiers.

Typically, the machines are arranged in a linear array with the general stationed at one end of the array. The transition rules of a soldier depend upon the state of that soldier and the states of the two

adjacent soldiers (or the single adjacent soldier in the case of the general and the soldier at the opposite extreme of the line). The time interval in which all the soldier must fire has zero length. In other words, the machines fire simultaneously. This scheme is illustrated in figure 4.

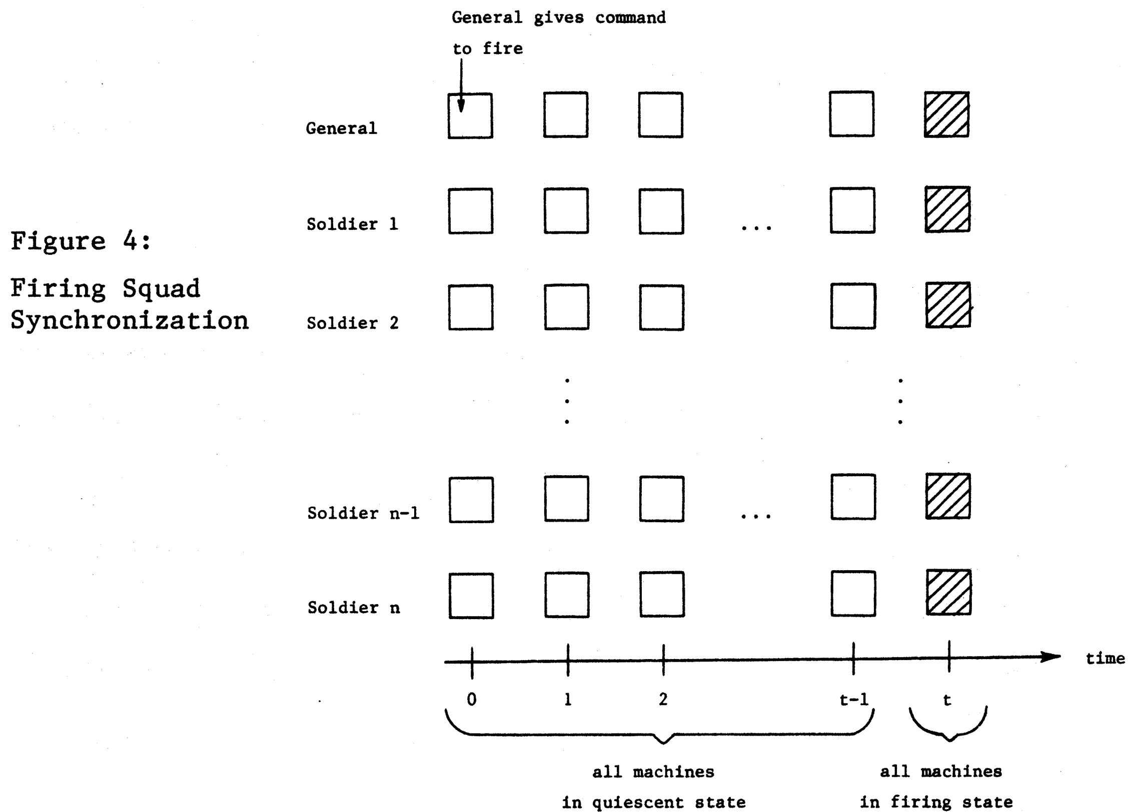

Figure 4:
Firing Squad Synchronization

CONCLUSIONS

We have examined many of the synchronization problems that have been proposed by researchers to date. Parallel computers today are still relatively rare. The concurrent architectures that do exist today tend to be primarily of the SIMD (Single Instruction Stream - Multiple Data Stream), or vector variety, which do not suffer the pitfalls (nor offer the benefits) of asynchronism and synchronization. Machines that

do support truly concurrent processes tend to display this concurrency largely among specialized devices, such as the communication protocol that serves as an interface between central processing units and data channels. Thus it is somewhat difficult to guage whether or not the problems presented in this paper accurately reflect some of the difficulties one in likely to encounter when implementing an actual system. In addition to questions of synchronizing concurrent processes, there also remain very important issues dealing with language features and computational efficiency that remain largely untouched.

As parallel computers become more of a reality, so too will synchronization problems become more realistic. For example, Fischer et al. [FLBB79] and Rivest and Pratt [RP76] have already made preliminary investigations of some variants of the mutual exclusion problem when the processes are subject to failure. Similarly, Russell and Brendt [RB75] and Wurges [Wur77] have examined techniques for coping with errors in producer/consumer problems. Our understanding of the underlying nature of concurrency will hopefully grow, and we will be able to expand our set of synchronization problems that can be rigorously analyzed and solved to include those exhibiting greater complexity.

ACKNOWLEDGEMENT

I would like to thank Dr. Calvin C. Elgot for inviting me to spend a summer with IBM and for suggesting this survey.

REFERENCES

[Age77] Tilak Agerwala
"Some Extended Semaphore Primitives"
Acta Informatica, vol. 8, 1977, pp. 201 - 220.

[And65] James A. Anderson
"Program Structures for Parallel Processing"
CACM, vol. 8, no. 12, Dec. 1965, pp. 786 - 788.

[An79] Sten Andler
"Predicate Path Expressions: A High-Level Synchronization Mechanism"
Ph.D. Dissertation. Carnegie-Mellon University, Computer Science Dept., Report CMU-CS-79-134, Aug. 1979. Also in Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, Jan. 1979, pp. 226 - 236.

[And79] Gregory R. Andrews
"Synchronizing Resources"
Cornell University, Computer Science Dept., Technical Report, Feb. 1979.

[Bab79] Alan F. Bablich
"Proving Total Correctness of Parallel Programs"
IEEE Transactions on Software Engineering, vol. SE-5, no. 6, Nov. 1979, pp. 558 - 574.

[Bal67] R. M. Balzer
"An 8-State Minimal Time Solution to the Firing Squad Synchronization Problem"
Information and Control, vol. 10, no. 1, 1967, pp. 22 - 42.

[BS77] R. Bayer and M. Schkolnick
"Concurrency of Operations on B-Trees"
Acta Informatica, vol. 9, 1977, pp. 1 - 21.

[BN76] M. Boari and A. Natali
"Some Properties of Deadlock Detection and Recovery in Readers and Writers Problems"
Information Processing Letters, vol. 5, no. 4, Oct. 1976, pp. 118 - 123.

[BN78] M. Boari and A. Natali
"Multiple Access to a Tree in the Context of Readers and Writers Problem"
Information Processing Letters, vol. 7, no. 2, Feb. 1978, pp. 112 - 121.

[BH70] Per Brinch Hansen
"The Nucleus of a Multiprogramming System"
CACM, vol. 13, no. 4, April 1970, pp. 238 - 241, 250.

[BH72a] Per Brinch Hansen
"A Comparison of Two Synchronizing Concepts"
Acta Informatica, vol. 1, 1972, pp. 190 - 199.

[BH72b] Per Brinch Hansen
"Structured Multiprogramming"
CACM, vol. 15, no. 7, July 1972, pp. 574 - 578.

[BH73a] Per Brinch Hansen
"A Reply to Comments on `A Comparison of Two Synchronizing Concepts'"
Acta Informatica, vol. 2, 1973, pp. 189 - 190.

[BH73b] Per Brinch Hansen
"Concurrent Programming Concepts"
Computing Surveys, vol. 5, no. 4, Dec. 1973, pp. 223 - 245.

[BH78] Per Brinch Hansen
"Distributed Processes: A Concurrent Programming Concept"
CACM, vol. 21, no. 11, Nov. 1978, pp. 934 - 941.

[BH79] Per Brinch Hansen
"A Keynote Addre in Concurrent Programming"
Computer, vol. 12, no. 5, May 1979, pp. 50 - 56.

[BHS78] Per Brinch Hansen and Jorgen Staunstrup
"Specification and Implementation of Mutual Exclusion"
IEEE Transactions on Software Engineering, vol. SE-4, no. 5, Sept. 1978, pp. 365 - 370.

[BFJLP78] James E. Burns, Michael J. Fischer, Paul Jackson, Nancy A. Lynch, and Gary L. Peterson
"Shared Data Requirements for Implementation of Mutual Exclusion Using a Test-and-Set Primitive"
Proceedings of the International Conference on Parallel Processing, August 1978, pp. 79 - 87.

[CH74] R. H. Campbell and A. Nico Habermann
"The Specification of Process Synchronization by Path Expressions"
Lecture Notes in Computer Science, vol. 16, Springer-Verlag, 1974, pp. 89 - 102.

[Cer72] Vinton G. Cerf
"Multiprocessors, Semaphores and a Graph Model of Computation"
Ph.D. Dissertation. UCLA, Computer Science Dept., ENG-7226, April 1972.

[CKR72] E. G. Coffman, L. A. Klimko, and Barbara Ryan
"Analysis of Scanning Policies for Reducing Disk Seek Times"
SIAM J. Comput., vol. 1, no. 3, Sept. 1972, pp. 269 - 279.

[CES71] E. G. Coffman, M. J. Elphick, and A. Shoshani
"System Deadlocks"
Computing Surveys, vol. 3, no. 2, June 1971, pp. 67 - 78.

[Com79] Douglas Comer
"The Ubiquitous B-Tree"
Computing Surveys, vol. 11, no. 2, June 1979, pp. 121 - 137.

[Con77] Reidar Conradi
"Some Comments on 'Concurrent Readers and Writers'"
Acta Informatica, vol. 8, 1977, pp. 335 - 340.

[Con63] Melvin E. Conway
"A Multiprocessor System Design"
AFIPS Conference Proceedings, vol. 24, Fall Joint Computer Conference, Spartan Books, Inc.: Baltimore, Md., 1963, pp. 139 - 146.

[CHCP74] Lee W. Cooprider, F. Heymans, P. J. Courtois, and David L. Parnas
"Information Streams Sharing a Finite Buffer: Other Solutions"
Information Processing Letters, vol. 3, no. 1, July 1974, pp. 16 - 21.

[CG77] P. J. Courtois and J. Georges
"On Starvation Prevention"
RAIRO Informatique/Computer Science, vol. 11, no. 2, 1977, pp. 127 - 141.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas
"Concurrent Control with 'Readers' and 'Writers'"
CACM, vol. 14, no. 10, Oct. 1971, pp. 667 - 668.

[CHP72] P. J. Courtois, F. Heymans, and D. L. Parnas
"Comments on 'A Comparison of Two Synchronizing Concepts'"
Acta Informatica, vol. 1, 1972, pp. 375 - 376.

[Cza78] Ludwig Czaja
"Implementation Approach to Parallel Systems"
Information Processing Letters, vol. 7, no. 5, April 1978, pp. 244 - 249.

[Cza79] Ludwig Czaja
"A Specification of Parallel Programs"
Information Processing Letters, vol. 8, no. 4, April 1979, pp. 162 - 167.

[DeB67] N. G. DeBruijn
"Additional Comments on a Problem in Concurrent Programming Control"
CACM, vol. 10, no. 3, March 1967, pp. 137 - 138.

[DD76] K. Delcour and A. J. W. Duijvestein
"Enclosures: An Access Control Mechanism with Applications in Parallel Programs and Other Areas of Systems Programming"
Information Processing Letters, vol. 5, no. 5, Nov. 1976, pp. 125 - 135.

[Den67] Peter J. Denning
"Effects of Scheduling on File Memory Operations"
AFIPS Conference Proceedings, vol. 30, Spring Joint Computer Conference, AFIPS Press, Montvale, N.J., 1967, pp. 9 - 21.

[DVH66] Jack B. Dennis and Earl C. Van Horn
"Programming Semantics for Multiprogrammed Computations"
CACM, vol. 9, no. 3, March 1966, pp. 143 - 155.

[DL73] R. Devillers and G. Louchard
"Realization of Petri Nets without Conditional Statements"
Information Processing Letters, vol. 2, no. 4, Oct. 1973, pp. 105 - 107.

[DL76] R. Devillers and G. Louchard
"A General Mechanism for Avoiding Starvation with Distributed Control"
Information Processing Letters, vol. 7, no. 3, April 1976, pp. 156 - 158.

[Dij65] Edsger W. Dijkstra
"Solution of a Problem in Concurrent Programming Control"
CACM, vol. 8, no. 9, Sept. 1965, p. 569.

[Dij68a] Edsger W. Dijkstra
"Cooperating Sequential Processes"
In Programming Languages, ed. F. Genuys, Academic Press, New York, 1968.

[Dij68b] Edsger W. Dijkstra
"The Structure of the 'THE'-Multiprogramming System"
CACM, vol. 11, no. 5, May 1968, pp. 341 - 346.

[Dij71] Edsger W. Dijkstra
"Hierarchical Ordering of Sequential Processes"
Acta Informatica, vol. 1, 1971, pp. 115 - 138. Also in Operating Systems Techniques, ed. C.A.R. Hoare and R.H. Perrott, Academic Press, New York, 1972, pp. 72 - 93.

[Dij72a] Edsger W. Dijkstra
"A Class of Allocation Strategies Inducing Bounded Delays Only"
AFIPS Conference Proceedings, vol. 40, Spring Joint Computer Conference, AFIPS Press, Montvale, N.J., 1972, pp. 933 - 936.

[Dij72b] Edsger W. Dijkstra
"Information Streams Sharing a Finite Buffer"
Information Processing Letters, vol. 1, no. 5, Oct. 1972, pp. 179 - 180.

[DLMSS78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens
"On-the-fly Garbage Collection: An Exercise in Cooperation"
CACM, vol. 21, no. 11, Nov. 1978, pp. 966 - 975.

[Dol79] Danny Dolev
"Commutation Properties and Generating Sets Characterize Slices of Various Synchronization Primitives"
Theoretical Computer Science, vol. 8, no. 3, June 1979, pp. 379 - 391.

[EM72] M. A. Eisenberg and M. R. McGuire
"Further Comments on Dijkstra's Concurrent Programming Control Problem"
CACM, vol. 15, no. 11, Nov. 1972, p. 999.

[ElM79] Calvin C. Elgot and Raymond E. Miller
"On Coordinated Sequential Processes"
IBM Research Report RC 7778, July 1979.

[Ell79] Carla J. Ellis
"The Design and Evaluation of Algorithms for Parallel Processing"
Ph.D. Dissertation. Technical Report 79-07-01, Dept. of Computer Science, University of Washington, July 1979.

[Fel79] Jerome A. Feldman
"High Level Programming for Distributed Computing"
CACM, vol. 22, no. 6, June 1979, pp. 353 - 368.

[FLBB79] Michael J. Fischer, Nancy A. Lynch, James E. Burns and Allan Borodin
"Resource Allocation with Immunity to Limited Process Failure: Preliminary Report"
Proceedings of the Twentieth Annual Symposium on Foundations of Computer Science, 1979, pp. 234 - 254.

[For78] Warwick S. Ford
"Implementation of a Generalized Critical Region Construct"
IEEE Transactions on Software Engineering, vol. SE-4, no. 6, Nov. 1978, pp. 449 - 455.

[Fra69] H. Frank
"Analysis and Optimization of Disk Storage Devices for Time-Sharing Systems"
JACM, vol. 16, no. 4, Oct. 1969, pp. 602 - 620.

[Ful74] Samuel H. Fuller
"Minimal-Total-Processing-Time Drum and Disk Scheduling Disciplines"
CACM, vol. 17, no. 7, July 1974, pp. 376 - 381.

[Gai72] R. Stockton Gaines
"An Operating System Based on the Concept of a Supervisory Computer"
CACM, vol. 15, no. 3, March 1972, pp. 150 - 156.

[GC72] Philip Gilbert and W. J. Chandler
"Interference Between Communicating Parallel Processes"
CACM, vol. 15, no. 6, June 1972, pp. 427 - 437.

[Gol78] Ulrich Golze
"(A-)Synchronous (Non-)Deterministic Cell Spaces Simulating Each Other"
Journal of Computer and System Sciences, vol. 17, no. 2, Oct. 1978, pp. 176 - 193.

[Goo76] Nathan Goodman
"Coordination of Parallel Processes in the Actor Model of Computation"
MIT/LCS/TR-173, Dec. 1976.

[Gra75] Antonio Grasselli
"Synchronization in Cellular Arrays: The Firing Squad Problem in Two Dimensions"
Information and Control, vol. 28, no. 2, June 1975, pp. 113 - 124.

[Gre75] Irene Greif
"Semantics of Communicating Parallel Processes"
Ph.D. Dissertation. MIT Project MAC TR-154, Sept. 1975.

[Gre77] Irene Greif
"A Language for Formal Problem Specification"
CACM, vol. 20, no. 12, Dec. 1977, pp. 931 - 935.

[Gri77] David Gries
"An Exercise in Proving Parallel Programs Correct"
CACM, vol. 20, no. 12, Dec. 1977, pp. 921 - 930.

[Hab69] A. Nico Habermann
"Prevention of System Deadlock"
CACM, vol. 12, no. 7, July 1969, pp. 373 - 377, 385.

[Hab72a] A. Nico Habermann
"Synchronization of Communicating Processes"
CACM, vol. 15, no. 3, March 1972, pp. 171 - 176.

[Hab72b] A. Nico Habermann
"On a Solution and a Generalization of the Cigarette Smoker's Problem"
Technical Report, Carnegie-Mellon University, Department of Computer Science, August 1972.

[HZ80] Peter B. Henderson and Yechezkel Zalcstein
"Synchronization Problems Solvable by Generalized PV Systems"
JACM, vol. 27, no. 1, Jan. 1980, pp. 60 - 71.

[Her72] Gabor T. Herman
"Models for Cellular Interactions in Development without Polarity of Individual Cells, Part II: Problems of Synchronization and Regulation"
International Journal of System Science, vol. 3, 1972, pp. 149 - 175.

[HLRW74] Gabor T. Herman, Wu-Hung Liu, Stuart Rowland, and Adrian Walker
"Synchronization in Growing Cellular Arrays"
Information and Control, vol. 25, no. 2, June 1974, pp. 103 - 122.

[HA79] Carl E. Hewitt and Russell R. Atkinson
"Specification and Proof Techniques for Serializers"
IEEE Transactions on Software Engineering, vol. SE-5, no. 1, Jan. 1979, pp. 10 - 23.

[Hil73] J. Carver Hill
"Synchronizing Processors with Memory-Content-Generated Interrupts"
CACM, vol. 16, no. 6, June 1973, pp. 350 - 351.

[Hoa72] C. A. R. Hoare
"Towards a Theory of Parallel Programming"
In Operating Systems Techniques, ed. C.A.R. Hoare and R.H. Perrott, Academic Press, New York, 1972, pp. 61 - 71.

[Hoa73] C. A. R. Hoare
"A Structured Paging System"
Computer Journal, vol. 16, no. 3, 1973, pp. 209 - 215.

[Hoa74] C. A. R. Hoare
"Monitors: An Operating System Structuring Concept"
CACM, vol. 17, no. 10, Oct. 1974, pp. 549 - 557. Corrigendum, vol. 18, no. 2, Feb. 1975, p. 95.

[Hoa78] C. A. R. Hoare
"Communicating Sequential Processes"
CACM, vol. 21, no. 8, August 1978, pp. 666 - 677. Corrigendum, vol. 21, no. 11, Nov. 1978, p. 958.

[Hol72] Richard C. Holt
"Some Deadlock Properties of Computer Systems"
Computing Surveys, vol. 4, no. 3, Sept. 1972, pp. 179 - 196.

[How76a] John H. Howard
"Signaling in Monitors"
Proceedings of the Second International Conference on Software Engineering, 1976, pp. 47 - 52.

[How76b] John H. Howard
"Proving Monitors"
CACM, vol. 19, no. 5, May 1976, pp. 273 - 279.

[Kat78] Howard 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," _Proceeding of the Tenth Annual ACM Symposium on Theory of Computing_, May 1978, pp. 86 - 88.

[Kel76] Robert M. Keller
"Formal Verification of Parallel Programs"
CACM, vol. 19, no. 7, July 1976, pp. 371 - 384.

[Kel78] Robert M. Keller
"Sentinels: A Concept for Multiprocess Coordination"
University of Utah, Dept. of Computer Science, Technical Report UUCS-78-104, 1978.

[Kes77a] J. L. W. Kessels
"An Alternative to Event Queues for Synchronization in Monitors"
CACM, vol. 20, no. 7, July 1977, pp. 500 - 503.

[Kes77b] J. L. W. Kessels
"A Conceptual Framework for a Nonprocedural Programming Language"
CACM, vol. 20, no. 12, Dec. 1977, pp. 906 - 913.

[KM79] J. L. W. Kessels and A. J. Martin
"Two Implementations of the Conditional Critical Region Using a Split Binary Semaphore"
Information Processing Letters, vol. 8, no. 2, Feb. 1979, pp. 67 - 71.

[KS78] Richard B. Kieburtz and Abraham Silberschatz
"Capability Managers"
IEEE Transactions on Software Engineering, vol. SE-4, no. 6, Nov. 1978, pp. 467 - 477.

[Knu66] Donald E. Knuth
"Additional Comments on a Problem in Concurrent Programming Control"
CACM, vol. 9, no. 5, May 1966, pp. 321 - 322.

[Kob77] Kojiro Kobayashi
"The Firing Squad Synchronization Problem for Two-Dimensional Arrays"
Information and Control, vol. 34, no. 3, July 1977, pp. 177 - 197.

[Kob78] Kojiro Kobayashi
"The Firing Squad Synchronization Problem for a Class of Polyautomata Networks"
Journal of Computer and System Sciences, vol. 17, no. 3, Dec. 1978, pp. 300 - 318.

[KuL79] H. T. Kung and Philip L. Lehman
"Concurrent Manipulation of Binary Search Trees"
Carnegie-Mellon University, Computer Science Dept., Report CMU-CS-79-145, Sept. 1979.

[KuR79] H. T. Kung and John T. Robinson
"On Optimistic Methods for Concurrency Control"
Carnegie-Mellon University, Computer Science Dept., Report CMU-CS-79-149, Oct. 1979.

[KuS77] H. T. Kung and S. W. Song
"An Efficient Parallel Garbage Collection System and its Correctness Proof"
Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science, 1977, pp. 120 - 131.

[Lam74] Leslie Lamport
"A New Solution of Dijkstra's Concurrent Programming Problem"
CACM, vol. 17, no. 8, Aug. 1974, pp. 453 - 455.

[Lam76] Leslie Lamport
"The Synchronization of Independent Processes"
Acta Informatica, vol. 7, 1976, pp. 15 - 34.

[Lam77a] Leslie Lamport
"Proving the Correctness of Multiprocess Programs"
IEEE Transactions on Software Engineering, vol. SE-3, no. 2, March 1977, pp. 125 - 143.

[Lam77b] Leslie Lamport
"Concurrent Reading and Writing"
CACM, vol. 20, no. 11, Nov. 1977, pp. 806 - 811.

[Lam68] Butler W. Lampson
"A Scheduling Philosophy for Multiprocessing Systems"
CACM, vol. 11, no. 5, May 1978, pp. 347 - 360.

[LC75] P. E. Lauer and R. H. Campbell
"Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes"
Acta Informatica, vol. 5, 1975, pp. 297 - 332. Addenda and Corrigenda, vol. 7, 1976, p. 325.

[LTS79] P. E. Lauer, P. R. Torrigiani, and M. W. Shields
"COSY - A System Specification Language Based on Paths and Processes"
Acta Informatica, vol. 12, 1979, pp. 109 - 158.

[Lau75] Soren Lauesen
"A Large Semaphore Based Operating System"
CACM, vol. 18, no. 7, July 1975, pp. 377 - 389.

[Lav78] Mark S. Lavenberg
"Synthesis of Synchronization Code for Data Abstractions"
Ph.D. Dissertation. MIT/LCS/TR-203, June 1978.

[Lev72] Karl N. Levitt
"The Application of Program-Proving Techniques to the Verification of Synchronization Processes"
AFIPS Conference Proceedings, vol. 41, part I, Fall Joint Computer Conference, AFIPS Press, Montvale, N.J., 1972, pp. 33 - 47.

[Lip73] Richard J. Lipton
"On Synchronization Primitive Systems"
Yale University, Dept. of Computer Science, Technical Report #22, 1973.

[Lip74] Richard J. Lipton
"Limitations of Sychronization Primitives with Conditional Branching and Global Variables"
Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, April - May 1974, pp. 230 - 241.

[LMS77] Richard J. Lipton, Raymond E. Miller, and Lawrence Snyder
"Synchronization and Computing Capabilities of Linear Asynchronous Structures"
Journal of Computer and System Sciences, vol. 14, no. 1, Feb. 1977, pp. 49 - 72.

[LSZ74] Richard J. Lipton, Lawrence Snyder, and Yechezkel Zalcstein
"A Comparative Study of Models of Parallel Computation"
Proceedings of the 15th Annual IEEE Symposium on Switching and Automata Theory, Oct. 1974, pp. 145 - 155.

[LSZ75] Richard J. Lipton, Lawrence Snyder, and Yechezkel Zalcstein
"Evaluation Criteria for Process Synchronization"
Proceedings of the IEEE Sagamore Conference on Parallel Processing, 1975, pp. 245 - 250.

[Lis72] Barbara H. Liskov
"The Design of the Venus Operating System"
CACM, vol. 15, no. 3, March 1972, pp. 144 - 149.

[MS78] Raymond E. Miller and Lawrence Snyder
"Multiple Access to B-Trees"
Proceedings of a Conference on Information Science and Systems, March 1978.

[Moo64] Edward F. Moore
"The Firing Squad Synchronization Problem"
In Sequential Machines: Selected Papers, ed. E. F. Moore, Addison-Wesley: Reading, Mass., 1964, pp. 213 - 214.

[ML68] F. R. Moore and G. G. Langdon
"A Generalized Firing Squad Problem"
Information and Control, vol. 12, 1968, pp. 212 - 220.

[Mor79] Joseph M. Morris
"A Starvation-Free Solution to the Mutual Exclusion Problem"
Info. Processing Letters, vol. 8, no. 2, Feb. 1979, pp. 76 - 80.

[New75] Glen Newton
"Proving Properties of Interacting Processes"
Acta Informatica, vol. 4, 1975, pp. 117 - 126.

[Opl65] Ascher Opler
"Procedure-Oriented Language Statements to Facilitate Parallel Processing"
CACM, vol. 8, no. 5, May 1965, pp. 306 - 307.

[Owi77] Susan Owicki
"Specifications and Proofs for Abstract Data Types in Concurrent Programs"
Stanford University, Computer Science Dept., Report STAN-CS-77-607, April 1977.

[OG76a] Susan Owicki and David Gries
"An Axiomatic Proof Technique for Parallel Programs"
Acta Informatica, vol. 6, 1979, pp. 319 - 340.

[OG76b] Susan Owicki and David Gries
"Verifying Properties of Parallel Programs: An Axiomatic Proof"
CACM, vol. 19, no. 5, May 1976, pp. 279 - 285.

[Par75] David L. Parnas
"On a Solution to the Cigarette Smoker's Problem (without conditional statements)"
CACM, vol. 18, no. 3, March 1975, pp. 181 - 183.

[PS75] D. L. Parnas and D. P. Siewiorek
"Use of the Concept of Transparency in the Design of Hierarchically Structured Systems"
CACM, vol. 18, no. 7, July 1975, pp. 401 - 408.

[Pat71] Suhas S. Patil
"Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes"
Project MAC, Computational Structures Group Memo 57, MIT, Feb. 1971.

[Pet75] Odd Pettersen
"Synchronization of Concurrent Processes"
Stanford University, Computer Science Dept., Report No. STAN-CS-75-502, 1975.

[Pre75] Leon Presser
"Multiprogramming Coordination"
Computing Surveys, vol. 7, no. 1, March 1975, pp. 21 - 44.

[Ram77] N. Ramsperger
"Concurrent Access to Data"
Acta Informatica, vol. 8, 1977, pp. 325 - 334.

[RK79] David P. Reed and Rajendra K. Kanodia
"Synchronization with Eventcounts and Sequencers"
CACM, vol. 22, no. 2, Feb. 1979, pp. 115 - 123.

[Wal72] David C. Walden
"A System for Interprocess Communication in a Resource Sharing Computer Network"
CACM, vol. 15, no. 4, April 1972, pp. 221 - 230.

[Wei66] Allen Weingarten
"The Eschenbach Drum Scheme"
CACM, vol. 9, no. 7, July 1966, pp. 509 - 512.

[Wir77a] Niklaus Wirth
"Modula: A Language for Modular Multiprogramming"
Software - Practice and Experience, vol. 7, no. 1, Jan. - Feb. 1977, pp. 3 -35.

[Wir77b] Niklaus Wirth
"The Use of Modula"
Software - Practice and Experience, vol. 7, no. 1, Jan. - Feb. 1977, pp. 37 - 65.

[Wir77c] Niklaus Wirth
"Toward a Discipline of Real-Time Programming"
CACM, vol. 20, no. 8, Aug. 1977, pp. 577 - 583.

[Wod72] P. L. Woden
"Still Another Tool for Synchronizing Cooperating Processes"
Carnegie-Mellon University, Computer Science Technical Report, Aug. 1972.

[Wur77] Harald Wurges
"Comment on 'Error Resynchronization in Producer Consumer Systems'"
Information Processing Letters, vol. 6, no. 3, June 1977, pp. 87 - 90.

[Yon77] Akinori Yonezawa
"Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics"
Ph.D. Dissertation. MIT/LCS/TR-191, Dec. 1977.