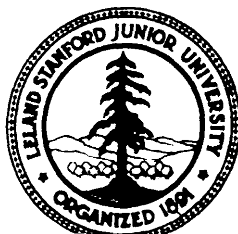
COMMUNICATING SEMAPHORES (#)

BY

HARRY J. SAAL
WILLIAM E. RIDDLE

STAN-CS-71-202
FEBRUARY, 1971

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



CS 202

Communicating Semaphores (#)

by: Harry J. Saal*
Stanford Linear Accelerator Center, and
Computer Science Department
Stanford University
Stanford, California 94305
U.S.A.

and

William E. Riddle*
Computer Science Department
Stanford University
Stanford, California 94305
U.S.A.

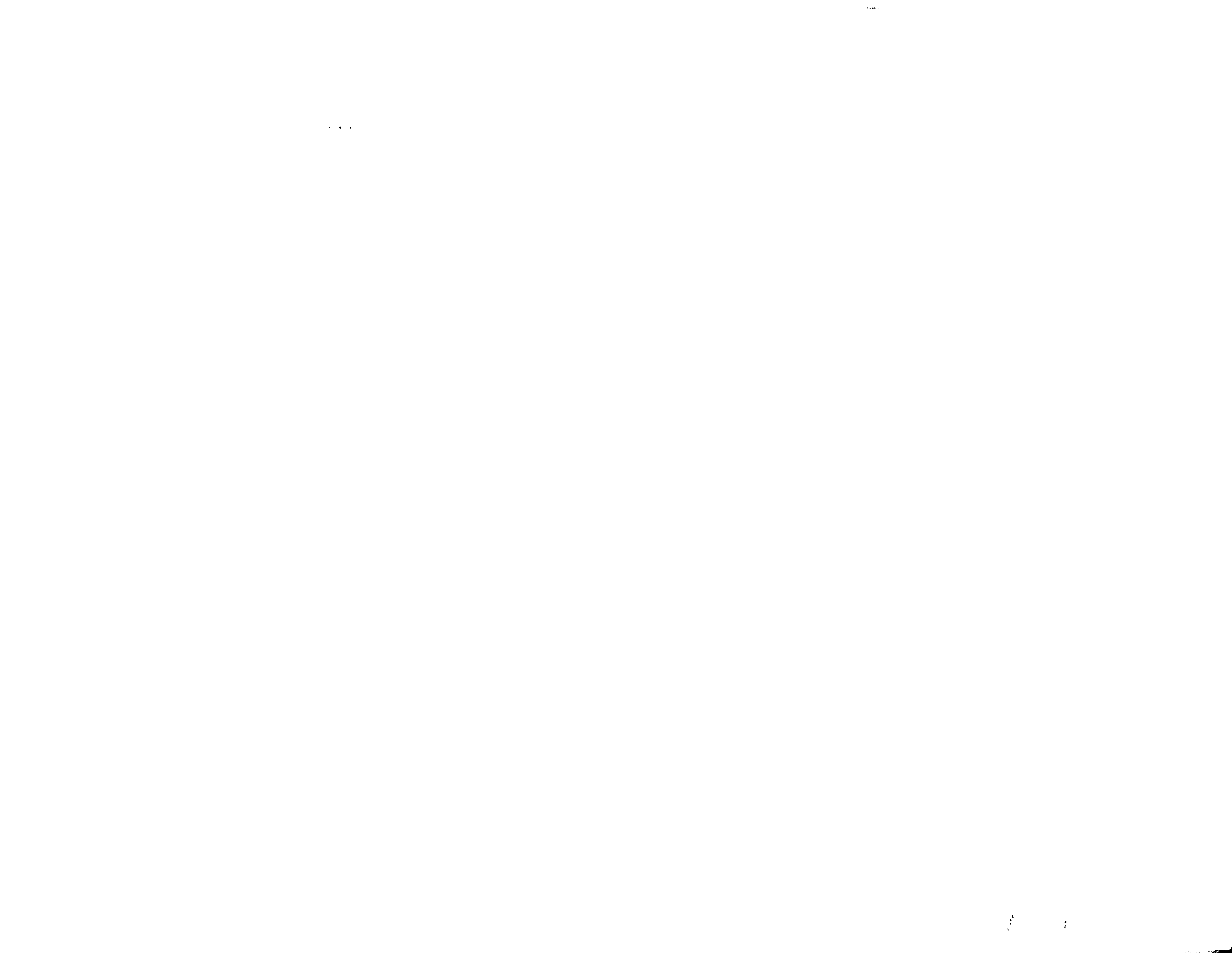
Present address: Department of Computer and Communication Science
University of Michigan
Ann Arbor, Michigan 48104
U.S.A.

* Supported by AEC Contract No. AT(04-3)-515, and
AEC Contract No. AT(04-3)-326

(#) Submitted to IFIPS Congress 1971

Abstract:

This paper describes two extensions to the semaphore operators originally introduced by Dijkstra. These extensions can be used to reduce: 1) the number of semaphore references; 2) the time spent in critical sections; and 3) the number of distinct semaphores required for proper synchronization without greatly increasing the time required for semaphore operations. Communicating semaphores may be utilized not only for synchronization but also for message switching, resource allocation from pools and as general queueing mechanisms.



Introduction

The introduction of semaphore operations by Dijkstra(1) in 1965 was motivated by the necessity of providing for correct interactions among a set of independent asynchronous processes which share data. For example, if a process is updating information in a common table, there exist moments when this updating is only partially completed. If, at this point in time, another process were to attempt to update this common table, it might destroy the partially updated information or otherwise behave incorrectly.

Some form of cooperation among the processes is required, and Dijkstra clearly demonstrated the benefits of using semaphore operations to provide the mutual exclusion over other mechanisms utilizing tests on shared variables.

Figure 1. illustrates this use of semaphores. The variable access-table is considered to take on integer values and is initialized to one prior to starting process 1 and process 2. The P operation, when issued by a process, decreases its argument by one and allows the process to continue execution (i.e. accesses the table) if the result is non-negative. If the result is negative, the process issuing the P is suspended until the value of its argument is increased towards zero. The V operation always increases the value of its argument by one and can never suspend the process issuing it. If its argument was originally negative when the V was issued, one suspended process waiting on this semaphore is enabled for execution. Each of these operations is considered indivisible; there is a form of mutual exclusion within the semaphore mechanism which ensures that one semaphore operation is completed prior to initiating a second*.

---

* We observe that this is a stronger statement than that which is actually required for proper behavior. We need only insure that no two semaphore operations on the same semaphore variable be in progress. There arise cases where a number of semaphore operations should be in progress, e.g. multiprocessing situations, or (as we shall see later) if one semaphore operation takes longer than some (real-time) constraint permits.

What mechanisms are required to implement semaphore operations? The V operation requires little work if its argument is non-negative. But, if it is negative, we must provide for the dequeueing of one

element from a list of suspended processes (and enable the running of that process). Also, some mechanism must be provided to return the now empty queue element to a pool*.

---

* The semaphore operation itself need not perform this task. A supervisory function might do it, but in any case the work must be done.

Whenever the semaphore variable becomes negative, the P operation requires the addition onto a queue of an element which has access to the information required to restart the suspended process. Figure 2 demonstrates two situations which may arise from P and V operations.

Producer-Consumer Relationship

Semaphores may be used to synchronize otherwise completely asynchronous processes at certain points during their execution. In particular, a large variety of process-pairs within a supervisory system may be viewed as having a producer-consumer relationship. The producer is generating information at some rate; the consumer processes this information at a rate which may be faster or slower than the producer, but the consumer is constrained not to get "ahead" o'f the producer. The producer may also be constrained never to get more than some number of messages ahead of the consumer*.

---

* Dijkstra(1) credits C.S. Sholten with discovering this use of semaphores for bounded buffering.

Figure 3. illustrates this producer-consumer relationship.*

---

* The terms "producer" and "consumer" apply only to the relationship between two processes and with respect to some object being transferred. One process in Figure 3 is producing filled buffers and is consuming empty ones. The other process is producing empty buffers and is consuming filled ones. This symmetrical relationship is not the case in general as later examples will demonstrate. A single process is often a consumer with respect to a second process, and a

producer with respect to a third.

We contrast the use of semaphores in producer-consumer relations; i.e., where a process performs P and V operations on different semaphores, from mutual exclusion, where the same semaphore is used.

There are many other examples of the producer-consumer relationship within a multiprogramming system. Table I lists a variety of typical cases.

Many-to-one and One-to-many Relationships

In a one-to-one producer-consumer relationship, semaphores as previously described, generally are sufficient for providing proper synchronization without requiring mutual exclusion semaphores. Consider the situation shown in Figure 4 where a number of producer-consumer pairs are taking buffers from a common pool. Pairs of processes (P1,C1),...,(Pn,Cn) are in a one-to-one producer-consumer relationship with respect to information in the line buffer lists. The processes (Cl,. . .Cn) and the pool B are in a many-to-one producer-consumer relationship with respect to buffers which are no longer in use. The pool B and the processes (P1,...,Pn) are in a one-to-many producer-consumer relationship with respect to buffers available for use.

The coordination required by the producer-consumer situation depicted in Figure 4. requires at least four types of semaphores. One semaphore variable, filled(i), serves to indicate the number of filled buffers in the shared line buffer list and a P operation on it causes the activity of the consumer to be suspended whenever there are no line buffers to be processed. Another, free, indicates the number of free buffers in the pool and causes the operation of a producer to be suspended whenever it requires a buffer but there is none in the pool. The final two semaphores, access-oool and access-list(i), serve to effect the constraint that only one producer or consumer process at a time may manipulate the linkages in the lists of free and filled buffers. The semaphores free and access-oool are peculiar to the pool of buffers and are used by all of the producer and consumer processes. The semaphores filled(i) and access-list(i) however, are unique to producer-consumer pal ri, and are established and initialized at the time that the pair is established. A fifth semaphore, empty(i), which is also unique to producer-consumer pair i, may be used to control how far the producer may get ahead of the consumer and hence, the maximum number of buffers that the pair may be utilizing at any given time.

Figure 5. indicates a flow diagram of the operation of one producer and consumer pair when the coordination is effected by the use of these five semaphores. Observe that 2+3N semaphores are required for N producer-consumer pa i rs.

There are several problems with the use of semaphores in effecting the required coordination for this situation. First, the P and V operations are indivisible and hence present a potential bottleneck in the operation of the system. Second, if we assume a first-come-first-serve discipline of the associated queue we will not al low the producers' requests for buffers to be recognized in any order other than that inherent in the time-sequence in which the producers issued P operations (although a more complex program, with several more semaphores and hence even more bottleneck potential, could be used). In particular, the priority of a producer's right or need to obtain a buffer from the pool cannot be expressed directly. Third, there is no provision for the producer directly to affect the order-in which the consumer processes the messages and hence it cannot cause a message to be processed before other messages that the consumer has not yet processed. Fourth, translation of this flow diagram into code would result in a program, the intent of which is not immediately obvious from a perusal of the code, except by someone thoroughly familiar with the operation of semaphores or by the aid of other documentat ion.

The rest of this paper develops a mechanism for the solution of these problems which are inherent in the use of semaphores. A formalism is developed for the succinct specification and efficient implementation of:

- message switching and communication in general
- synchronization of the use of shared resources
- coordination of asynchronous processes
- resource al location

whi ch, at the same time,

- requires fewer references to semaphores (by reducing the need for mutual exclusion) thereby reducing the potential bottlenecks,
- reduces the number of different semaphores required, adding eff icfency and clarity to the programming,
- allows more equitable service to be guaranteed to the processes which may be made to wait as a result of a P operation,

- allows a process which issues a P operation to specify its priority with respect to other processes which may be waiting on the same semaphore,
- allows a process which issues a V operation to affect the, decision as to which waiting process should be restarted next.

Communicating Semaphores

We observe that in the example of Figure 5 , when we determine the existence of a free buffer in the pool (using free), we have not received its identification (or address). We use a second semaphore (access-pool) to surround a critical section where we extract its address and unlink it from the pool. The semaphore free has allocated a buffer (since free is immediately decreased), but it has not provided specific information on which buffer we should use. Communicating semaphores (csems) operate similarly to Dijkstra's semaphores; however, a Send operation, in addition to having the semantics of granting permission to proceed, may pass a message to the process issuing a Receive operation. These extended operations are denoted by Receive. and Send to distinguish them from the usual P and V operations.*

---

* These operations produce the same result as conventional P and V operations if null messages are passed. Wirth (2) independently observed that a message exchange system sending null messages can be used to effect the same synchronization that P and V provide.

Syntactically, we may use the Send operation on the left hand side of an assignment statement; i.e., Send(sem)= message. The value of sem will be increased by one, and, if sem is initially non-negative, the message (or a pointer to it) will be linked to the queue associated with sem. If sem is negative, a process suspended on the sem queue will be enabled to run, and the message will be passed to this process. The Receive operation appears on the right hand side of an assignment statement; i.e., message = Receive(sem). When executed, sem is decreased, and if it becomes negative, the process is suspended as for a P operation. If sem is non-negative, a message is taken from the sem queue and assigned to message as its value. If the process is suspended, when it is re-enabled it will receive a message from the Send operation which caused it to awaken.

Figure 6. illustrates the nature of the queues that may be associated with sem. Figure 6b. is identical to the case of Figure 2b. However, Figure 6a. shows information queued up whenever an excess of messages (or permission) exists.

Does this extension require any additional mechanism for the Receive and Send operations? In large part it does not since the queueing of messages by Send is similar to the queueing of processes by a P operation. The only change is returning a value to a process issuing a Receive when that process is enabled to run. (It does take more time however to enqueue a message rather that simply increasing the value of sem.)

Have we reduced the number of semaphores and critical sections required? Figure 7. demonstrates a csem solution to the example given previously. We still have the global semaphore free and the private semaphores empty(i) and filled(i) for each pair i, but the global access-pool and local access-list(T) are not needed. We now require N semaphores and N+1 csems instead of 2+3N semaphores as in the previous solution. Nor do we have any critical sections remaining. In order to initialize the system, the creator of the buffer pool must set up the semaphore free by initializing free to zero, and then performing Send(free)= bufferl, Send(free)= buffer2,..., for example. For each pair, filled(i) is initially zero, and empty(i) is set to some value appropriate for producer-consumer pair i.

Csems with priorities

Csems, as formulated so far, have attacked the bottleneck problem by reducing the need for mutual exclusion and thereby the number of references to semaphores. However, nothing has been done to allow the processes which issue the Receive and Send operations to affect the priority with which messages are transmitted or received.

For example, many different processes may be transmitting messages to a process which prints them on a console typewriter. We may choose to have high priority messages appear as soon as possible, whereas other messages are to be typed in the order they are received.

In another application we may have several output printers: one is reserved for listings of less than some maximum size; otherwise we want the shortest listing to be printed next on the first available device. The second extension to semaphores permits the solution of these problems in a straightforward and transparent manner.

To permit the specification of a priority for a Receive operation, the queue is made into a ranked queue, and the syntax of the Receive operation is changed to allow the specification of a second parameter to indicate the rank with which the return address should be linked in the queue for this semaphore variable. The specification of a priority with which a message is to be transmitted via Send is provided in an analogous manner*.

---

* It should be pointed out that semaphores without priorities still require the implementer to choose a queueing discipline for messages and processes. Most natural would be FIFO, although LIFO may have some utility in situations where a priority interrupt nesting mechanism is in force.

The manner of handling the queueing of Receive or Send elements (when required) is according to the relative rank of the entries, and FIFO for entries of equal rank. These operations must be indivisible and the most efficient implementation of the queue will be a linked list since we may have to insert an element into the middle of the list*.

---

* In general, we expect to find fairly short queues associated with semaphore variables. In this case, we can simply scan down the list until we have found the point for insertion and complete the Receive or Send operation. In those cases where the list is very long (which can be immediately seen from the value of sem), a more sophisticated search mechanism may be desirable. On the other hand, we may interrupt any operation on a semaphore as long as we complete it prior to initiating a different operation on the same semaphore. One bit in the head of the semaphore queue would be sufficient to mark "operations in progress".

We have not described how the ranks affect the matching and dispatching of Receive's and Send's queue elements. One choice might be to simply have the highest Receive entry match the highest Send entry. In this case, the queues that arise will look like those in

Figure 6a. or 6b. (with rank information appended to each element).

A more interesting approach is to match only when the rank of a Receive entry is greater than or eaual to the rank of a Send entry. This algorithm for Receive and Send operations with conditional matching is shown in Figure 8. it is now possible that both process and message entries will be presentin a semaphore queue, since the ranks of the processes may be lower than the ranks of the messages. This requires maintaining two lists or using some interlacing technique using an additional bit on each queue element.

We can now achieve the behavior suggested earlier for multiple output writers, i.e., even though there are (long) messages available, a printer may remain idle until shorter messages arrive.

Figure 9. shows an example of many-to-many synchronization and allocation using conditionally matched csems. Each program produces output destined for a printer. The text (or the name of the disk file containing it) is passed to a printer via a Send operation on a global semaphore printer. The priority shown in this example is the actual 1inecount, but some other measure could be used as well. There are two printer processes called Exoress and Normal. The shortest text in the printer queue will be selected whenever a printer is free; however, no output longer than 2000 lines wi 11 be printed by Exoress. Short jobs will also appear on Normal whenever it is free. output longer than 100 thousand lines will be suspended until a new printer process is started to accept such longer jobs. User programs cannot determine the properties of the printer processes. If the normal printer is out of service, an operator may change the rank of the Receive operation in the Exoress process to accept more work if desired. Converse1 y, an operator might lower it if very sha t Exoress jobs have to wait too long for a printer.

Other features

There are several other functions related to csems which are quite useful in the construction of a multiprogramming system. We propose a HOLD operation which suspends further matching of Receive and Send queue entries. This enables global control (as by a human operator) of the system behavior. The details of HOLD are straightforward and will be given elsewhere. We do have to consider what to do with a process that tries to HOLD a queue which is already held, etc!

The operations which a human operator would have to perform may

be easily permitted by another extension. A CONVERT would hold a queue and return a copy of the queue structure in a form suitable for inspection and manipulation (e.g. on a scope at the operator's console). Modifications of the structure, ranks, etc. of an unusual nature could now be performed. The semaphore queue would then be RECONVERTed and RELEASEd in order to permit the standard algorithms to proceed.

Problems not yet discussed

There are a variety of questions which have not been discussed in this paper. We have not described the means of creating new semaphores and establishing initial communication (sharing of common semaphores) between processes, nor have we discussed where semaphore queue elements are taken from or what to do should a process go amuck and generate extremely many queue entries (or how to prevent it from doing so). These issues will be discussed in a separate paper.

The csems as described do not prevent their misuse by creating deadly embrace situations. They are not intended to resolve that form of conflict. However, since by using csems we require fewer semaphores and fewer critical sections, we will enhance programming correctness and clarity, thus avoiding some blunders. Similarly, csems do not provide any functions which could not be constructed in principle from conventional semaphores. We assume that the construction of the Receive and Send operations is well optimized and either microcoded or hardwired into our computer system. By making use of these operations for a wide variety of applications we take advantage of a highly efficient and guaranteed mechanism. Reproducing these same general queue mechanisms in many places throughout a large operating system is subject to error and differences in programmer ability. We do not believe the solution to the construction of large programs and systems lies in the exclusive use of a set of completely non-redundant atomic operations; rather, we need more powerful and generally applicable primitives which are well implemented at the hardware or firmware level.

Summary

A csem is a data structure composed of two ranked lists of entries that represent, respectively, processes which are waiting to receive messages and are waiting to be enabled, and messages that are available for transmission. The csems effect buffered communication between asynchronously operating processes for the purposes of synchronization, coordination, and allocation as well as for normal

message switching. The algorithms for controlling the transmission are non-interruptible primitive instructions, the operation of which can be modified by the processes completing the communication by specifying a rank which affects: 1) the order in which the requests are satisfied; and 2) whether or not a given message will satisfy a given request.

Csems help to reduce the amount of time processes are executing critical sections, reduce the total number of semaphore operations performed, and reduce the number of semaphores required. These reductions produce greater program clarity. Utilization of csems for the diverse applications previously described may also. help to increase program correctness by providing more appropriate, well-debugged and efficient primitives.

Acknowlednementg

The authors' wish to thank many members of the SLAC Computation Group for their interest and comments on this work.

References

(1) E. W. Dijkstra, 'Cooperating Sequential Processes', September 1965, (EWD123), reprinted in *Programming Languages', ed. F. Genuys, Academic Press (1968)

(2) N. Wirth, 'On Multiprogramming, Machine Coding, and Computer Organization', CACM 12, p. 489 (Sept. 1969)

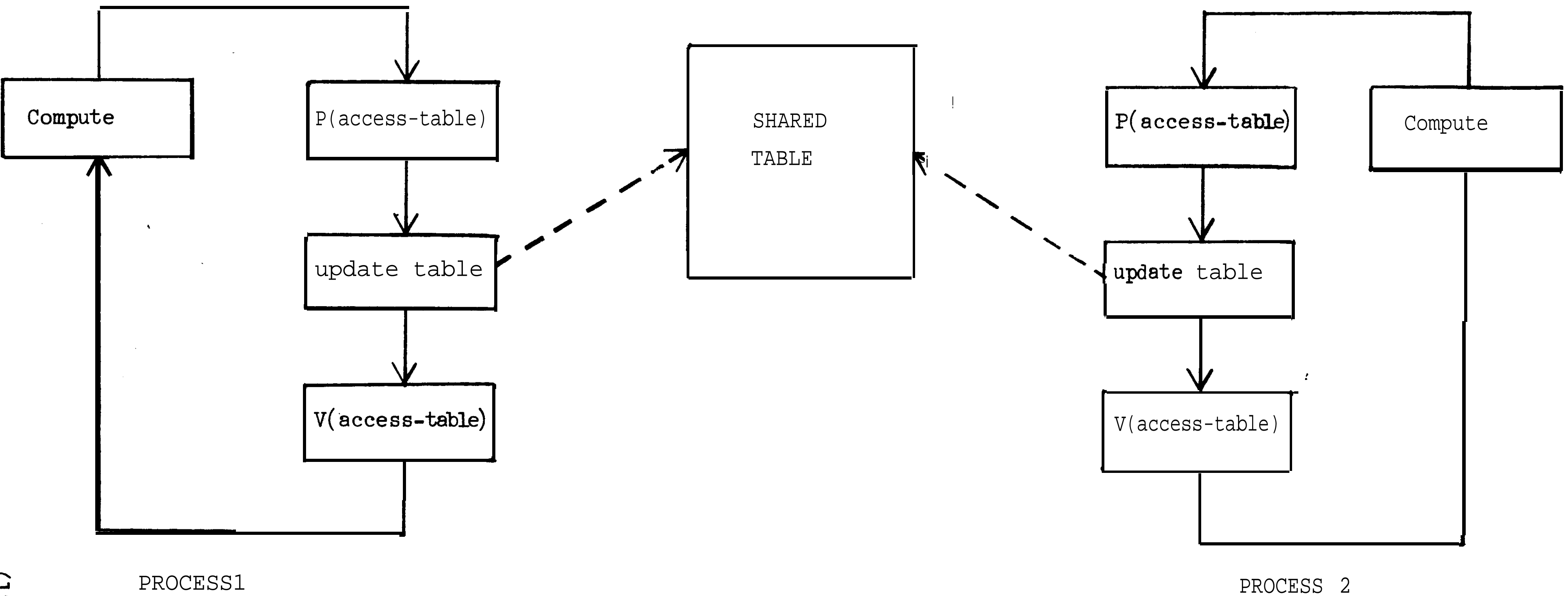
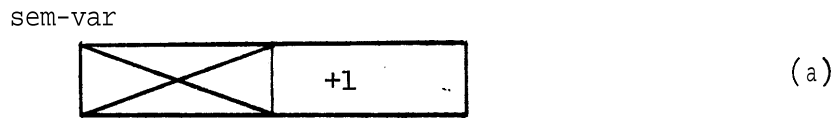


Figure 1: Semaphore used for mutual exclusion



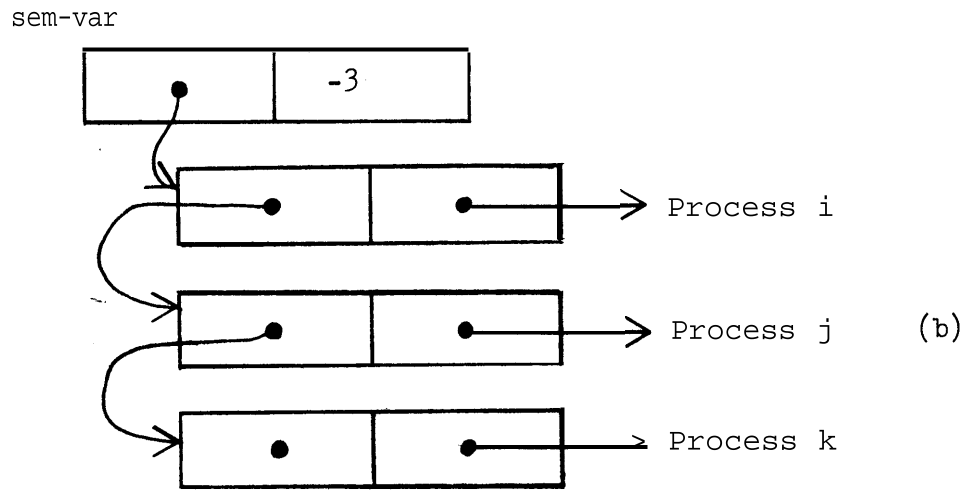


Figure 2: (a) No processes waiting

(b) Three processes suspended

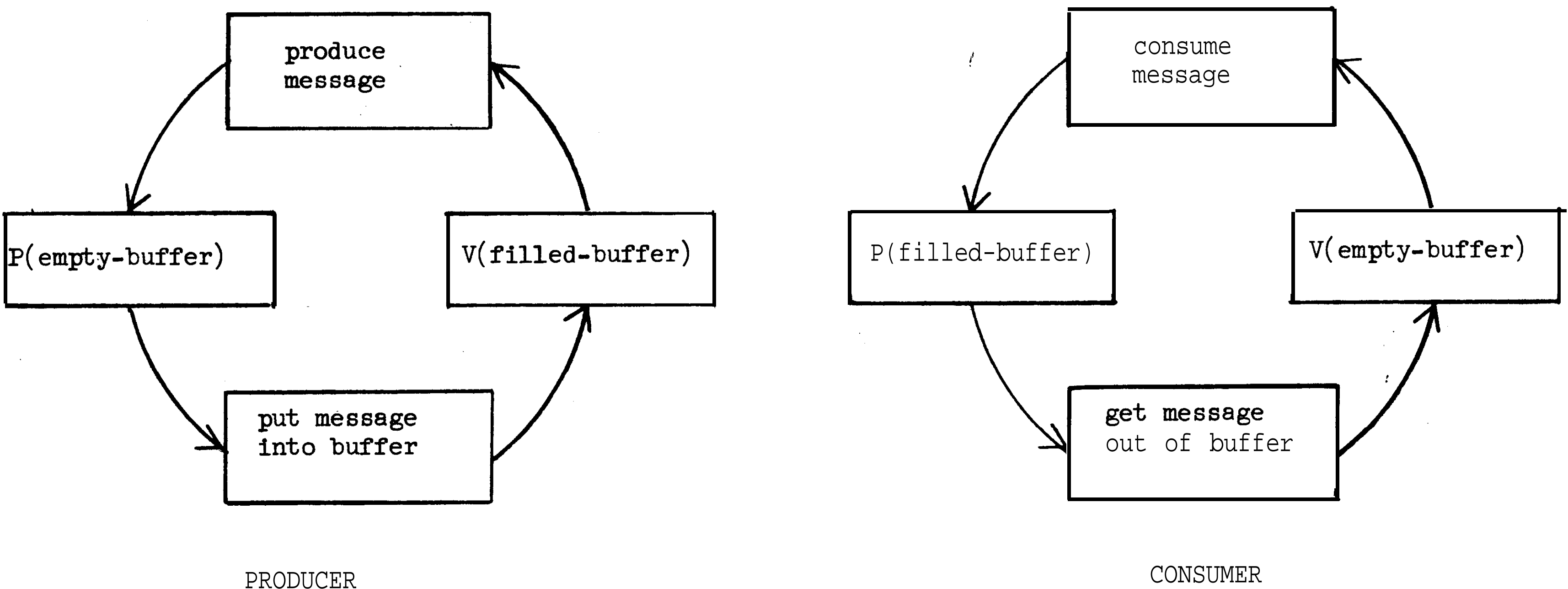


Figure 3: Producer-consumer synchronization sharing a three element ring buffer

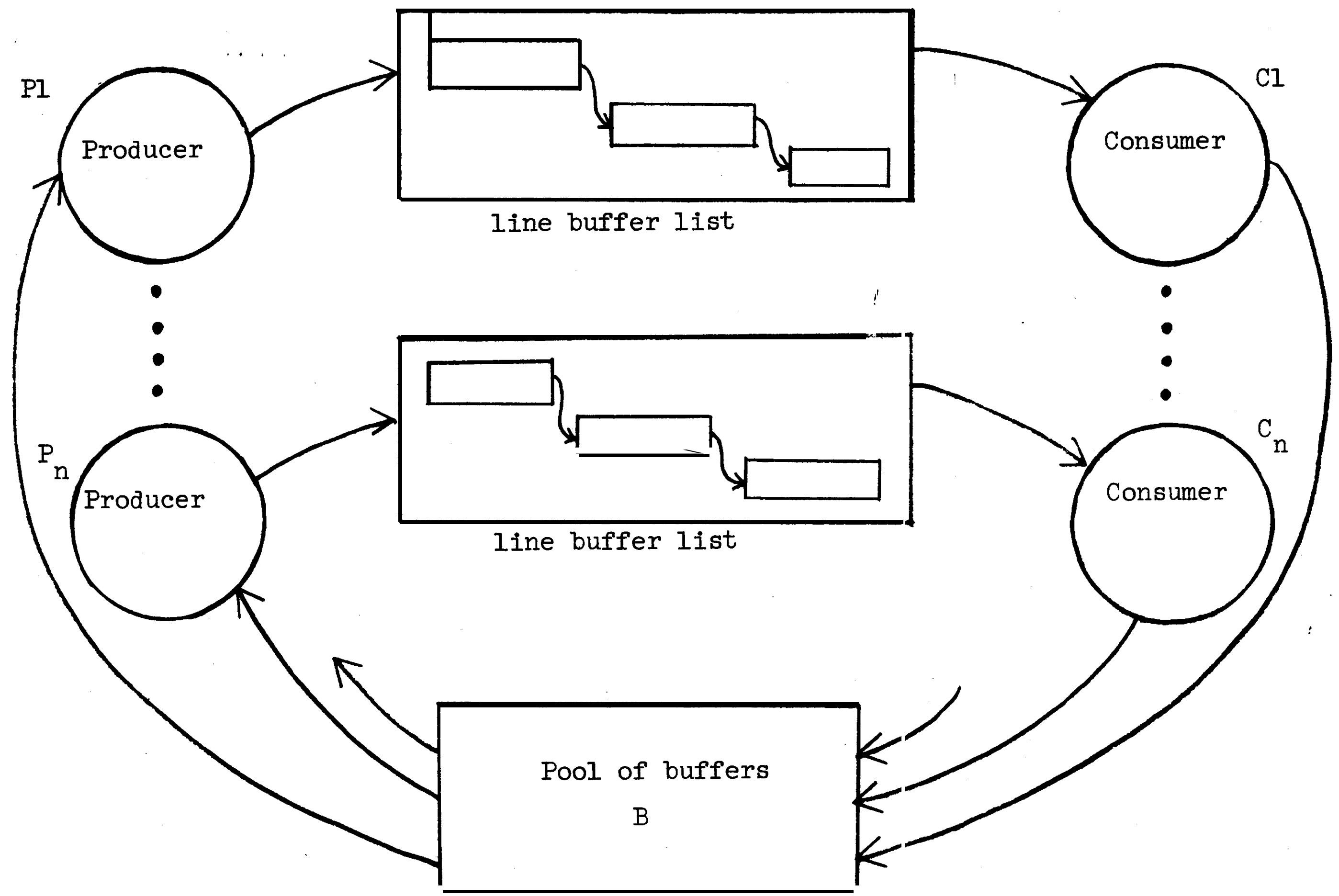
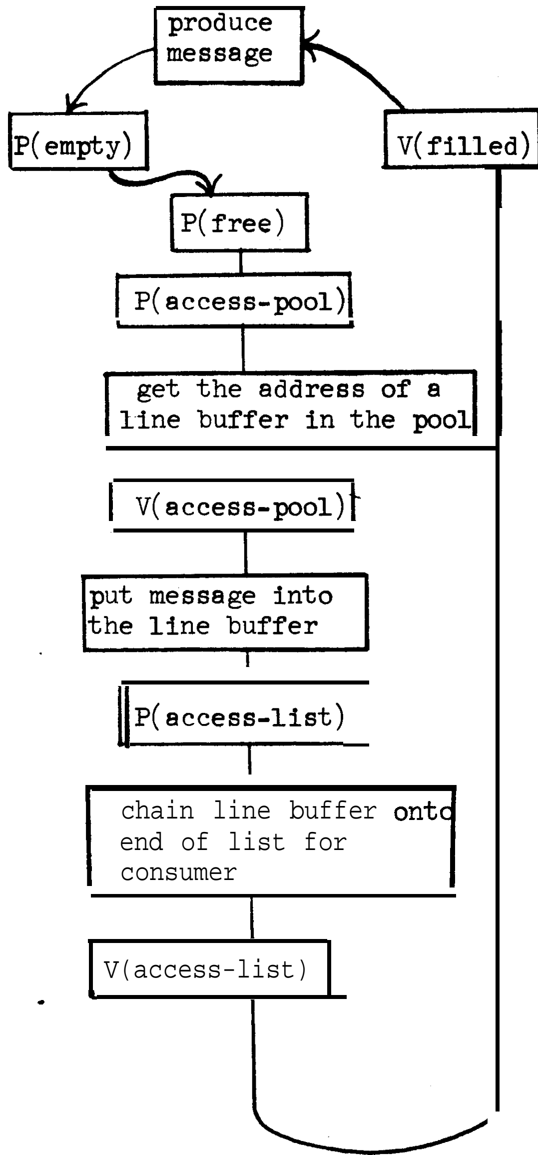
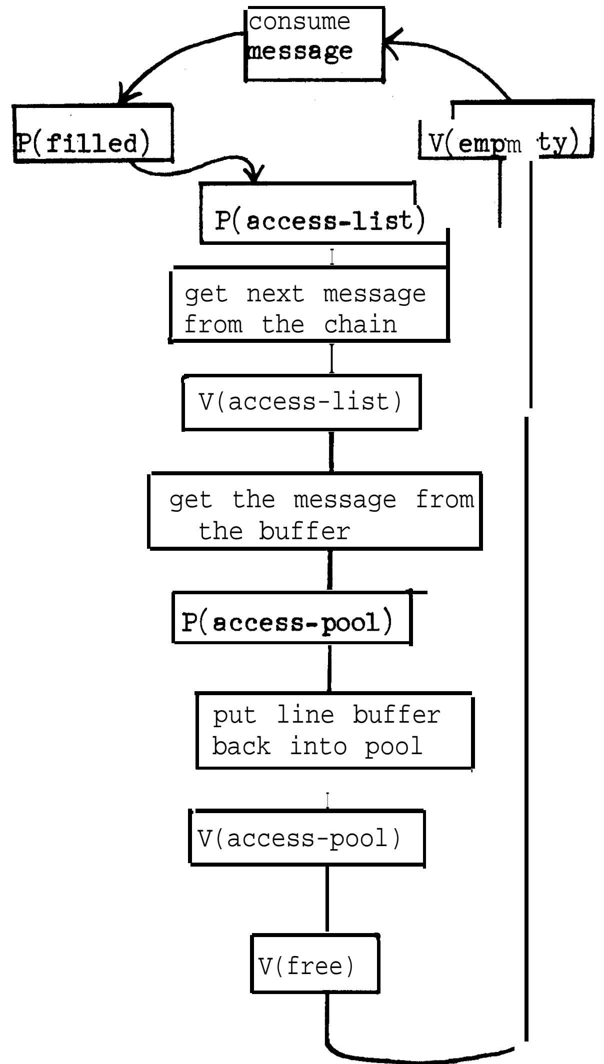


Figure 4: Producer-consumer pairs sharing a common pool of buffers

initially
- empty = N
- filled = 0
- access-list = 1



PRODUCER



CONSUMER

Figure 5: Producer-consumer problem with a buffer pool and limited use of-buffers

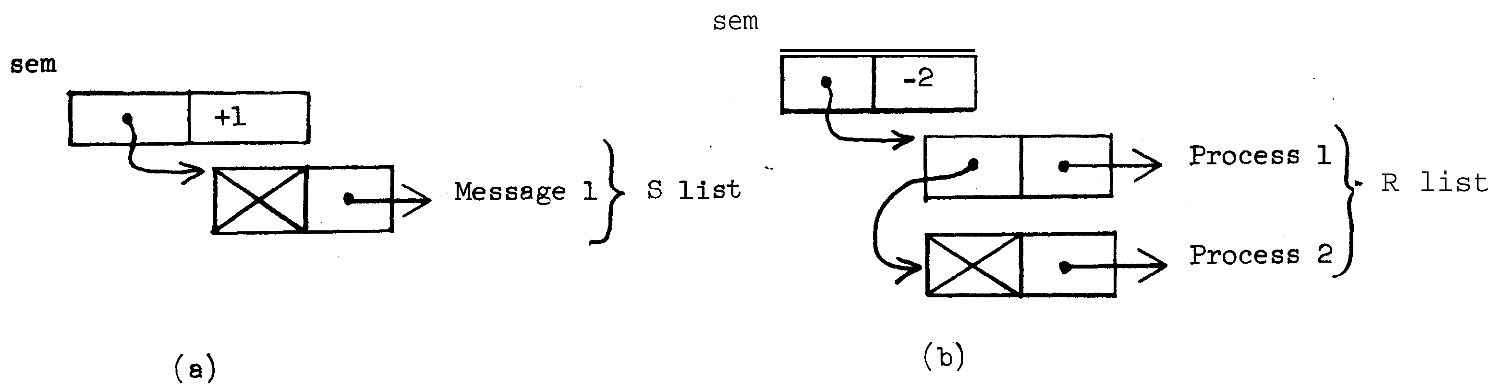


Figure 6: (a) Queue associated with sem when more Send than Receive operations have been performed

(b) The converse situation

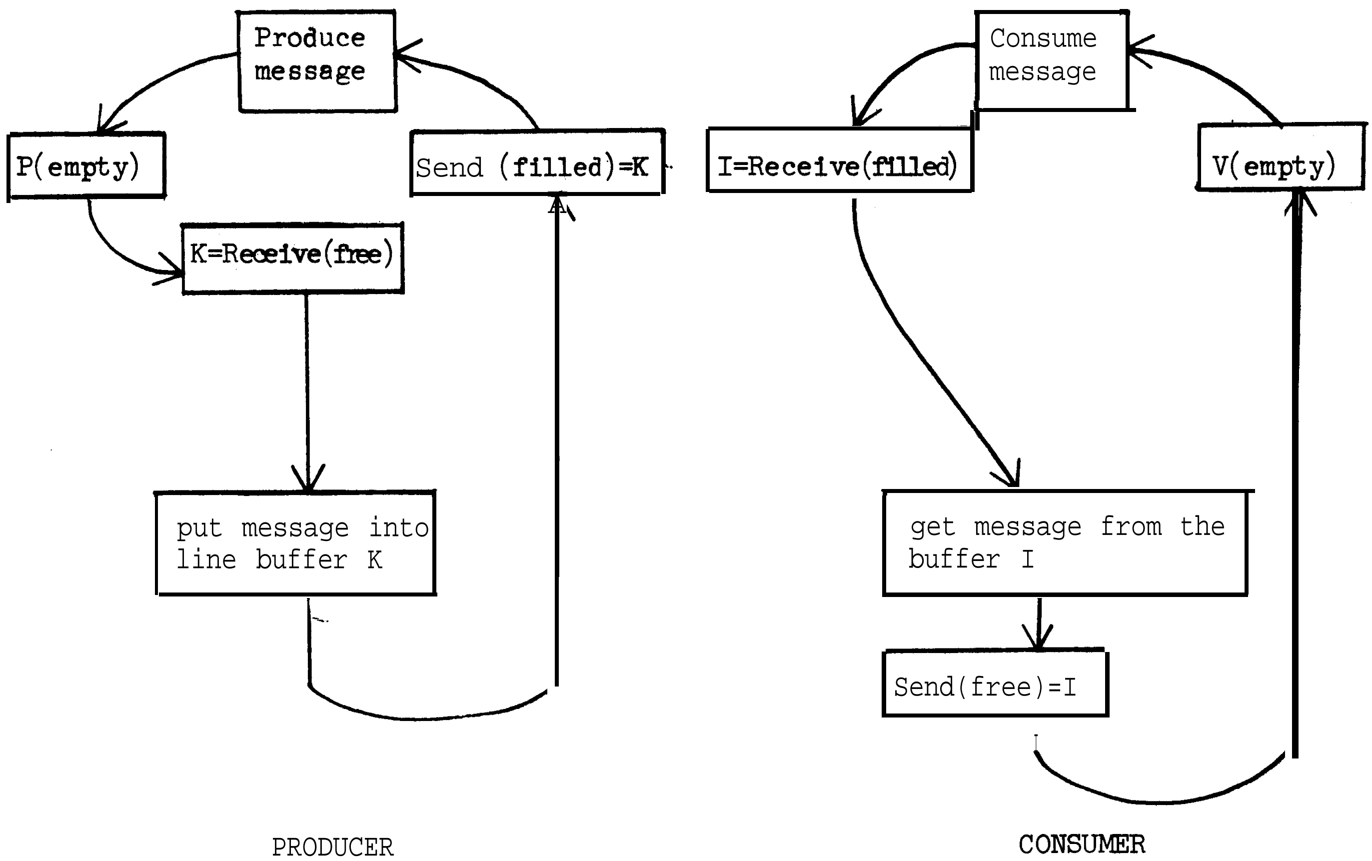


Figure 7: Solution to example of Figure 5 using csems

Receive operation

```
if S list is empty
      then insert return information into R list according
           to its rank and suspend the process
      else if rank of incoming Receive ≥ rank of first
                  S list element
           then pass message from the S list to the process
                  issuing the Receive and delete the
                  S list element
           else insert return information into R list
                  according to its rank and suspend the process
```

Send operation

```
if R list is empty
      then insert message into S list according to -its
           rank
      else if rank of incoming Send ≤ rank of first
           then pass the message to the process at
                  the top of the R list and awaken it
           else insert message into S list accordingto
                  its rank
```

Figure 8: Communicating Semaphore Algorithms with Conditional Matching

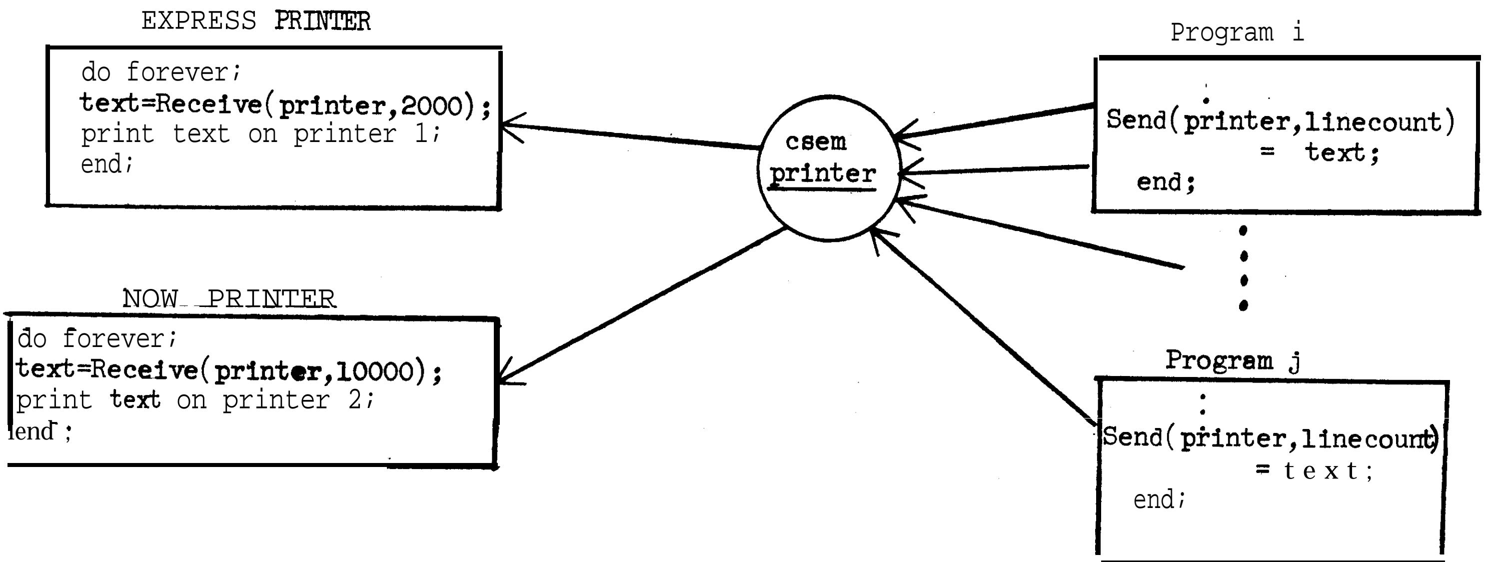


Figure 9: Many programs producing output for an express or normal printer.

Producer	Consumer	Information Exchanged
a program	output device	output text
last one to use a buffer	next 'process to need 'a buffer	pooled buffers
CPU	program	CPU cycles
program	CPU	program instructions
scanner	parser	modified program text
one phase of a multiphase algorithm	the next phase	intermediate data
terminals	multiaccess service programs	text strings or input
resource allocator	process	resource name or description

Table I: Several producer-consumer relationships in a Multiprogrammed System