Project MAC *July 18, 1975*

Computer Systems Research Division *Request for Comments No.* 85

Two Algorithms for Process Synchronization without Software Locks

by Raj Kanodia

1.0 Software Locks

In a modern multi-programming computer system it is often necessary to allow independent, concurrently executing processes to share the same data base. If processes are allowed to modify a shared data base, then the data base may be temporarily in an inconsistent state while being modified by some process. In general, it is necessary that the processes accessing such a data base be synchronized so that a process will not access a data base which is in an inconsistent state. Usually this synchronization is provided by means of semaphores or software locks. Before a process is allowed to modify a data base it must set a software lock associated with the data base. This lock is such that, once set, it can not be reset by any other process until "released" or unlocked by the locking process. Having set the lock, a process performs the necessary operations on the data base, and then releases the data base by unlocking the associated lock. If a process's request to set a lock is denied because some other process has the lock set (and is working on the data base), then the requesting process must wait until the other process releases the data base by unlocking.

1.1 Disadvantages of Software Locks

Though widely used, the locking method just described, has several disadvantages. As we mentioned above, if a data base is already locked then other processes that want to access the data base must wait until the locking process has released the data base. If for some reason the locking process is slowed down while the lock is set then other processes must wait longer. Normally, a real time delay in a multi-programming system does not present a serious problem. However, if a shared data base represents a critical system resource then a real time delay in its use by processes may reduce its effectiveness and degrade the system performance considerably. Two examples will clarify this point: first, consider the data base associated with a multiplexed input-output channel. A tape multiplexer channel through which i/o operations can be performed on several tape drives represents such a channel. The use of this channel among several processes can be synchronized by means of

---

a software lock. If for some reason, a process is delayed while it has this lock set, then use of the i/o channel is denied to other processes thus reducing the effectiveness of the channel. Another example follows. In Multics, the page control system operates with a global lock. This implies that two simultaneous page faults can not be processed simultaneously even though there might be two processors available on the system. It is estimated that in the Multics system at M.I.T., which is normally run with two processors, approximately five percent of the processing capacity is lost in simply waiting on the page control data base lock.

A process can slow down due to several reasons. For example, a process may have exhausted its CPU time quantum, in which case it must wait until it is allocated another CPU time quantum. To prevent this from happenning, most operations on shared data bases are performed in a privileged environment in which a process can not be pre-empted even if it has used up its CPU time quantum. This can introduce a fair amount of complexity in the system scheduler which must now cope with another mechanism. Often it is the case that a process is protected from pre-emption while executing in the system supervisor (ring-zero in Multics), and therefore all shared data base modification operations are simply pushed into the supervisor thus increasing its size and complexity; all of which has an adverse effect on its certifiability. Another point that needs to be emphasized here is that even though a process may be fully prepared to perform some operation on a shared data base, it may have to wait to do so (and thus be unfairly penalized) simply because some other process has locked the data base and is unable to complete its operation in a short time.

While a small real time delay may, at best, simply reduce the effectiveness of the resource utilization, a degenerate case may deny the use of resources forever or at least until the next system crash or shutdown. Consider what will happen if a process were to die with a lock set on a critical resource. There are several reasons why a process may terminate. It may have exhausted its CPU resources, or it might have taken an irrecoverable fault. A process might also be terminated due to externally induced reasons. In general these problems are solved by providing a privileged environment and ensuring that all data base modifications are performed within this environment. The environment guarantees that the process will not be terminated due to lack of resources or externally induced signals while executing in the critical data base modification routines. Frequently the routines to modify critical data bases have complicated abort handlers to bring the data base into a consistent state should the data base modification operation be terminated in the middle due to a hardware or software failure.

The problem is further compounded by the interaction of interrupt mechanism with the locking mechanism. If a process can be interrupted while modifying a data base, then the interrupt handlers must not be allowed to access the locked data base. If the interrupt handler simply followed the strategy of waiting for the data base to be released, we would have a dead-lock situation. Fairly complicated mechanisms are needed to prevent the occurrence of such dead-locks. One must either mask interrupts for the duration of the execution of the data base modification operation or one must program the interrupt handler very carefully to deposit requests for actions in a queue to be examined at the time of release of the data base.

1.2 Short-term Locking

In a paper published in 1972, Easton describes a different approach to the design of algorithms which operate on the shared data [Ea72]. His technique, in brief, is:

> "to provide each shared object with a version number, to remember the version number prior to making a decision about modifying the object, and then, when the actual modification takes place, to compare the version number with the remembered one, to change the version number, and to perform the modification, all in a single instant of time. The technique is somewhat inelegant, in that a process may be forced to repeat the work it has already done."

In this technique, the actual data modification operation, or the critical section is very brief and therefore can be performed in an uninterruptible privileged state. Since the critical section is rather short in duration, processes waiting for their turn to perform an operation on shared data need not give up the processor (which requires involvement of the supervisor in the form of block and wait primitives), instead they loop on a test-and-set instruction to guarantee mutual exclusion. Essentially, long term locks have been replaced by short term locks, and the supervisor involvement is minimized. One must still make use of a privileged state to perform the data base modification and a lock to guarantee mutual exclusion. The main disadvantage of the version number technique is that occasionally a process will not succeed in performing a data base operation because some other process has modified the data base and the version numbers do not match. The process will have to repeat the entire operation. However, this repetition should not be confused with the familiar race conditions in the switching circuits in which no useful work gets done at all. In the case of this technique, if a process fails to perform an operation, it is only because some other process has succeeded.

1.3 A hardware provided single-instruction lock

If we carry the short term locking philosophy to extreme and let the hardware provide a special instruction which would check the version number, perform the intended data modification, and change the version number, all in a single instant of time and at the same time guarantee mutual exclusion, then - at least for those operations on shared data that can be efficiently programmed using the special instruction - there will be no need for (1) a supervisor provided privileged state, and (2) the software locks to guarantee mutual exclusion. Such an instruction offers many advantages over usual software locking techniques. First, since no interruption or pre-emption is permitted during the execution of this instruction there is no need for a supervisor provided privileged state. Presumably the instruction takes only a short time to execute and therefore the second benefit is that no process can cause other processes to be delayed indefinitely. The third advantage of using this instruction is that there is no need of software locks to ensure mutual exclusion. If two processes try to update a data base at the same time, only one of them will succeed and the other process will be automatically deferred for the duration of the instruction. And since this instruction is uninterruptible, there is no need to mask the interrupts. The present hardware base of the Multics system, Honeywell 68/80 provides such an instruction though with a rather severe limitation - the version number and the data to be changed must be confined to a single memory word. At the first glance it would seem that such an instruction would be useless for all practical purposes since most operations on shared data require changing more than a word. However, as we shall see shortly, there are some important shared data operations that sit at the heart of any operating system and that can be programmed in terms of this special instruction. We shall look at two specific algorithms on message queueing and assignment of resources. Both algorithms provide primitives that can be executed in parallel by a number of processes on shared data. Anyone familiar with the working of operating systems will immediately recognize their importance. These algorithms are presented as procedures which, once certified, may be used by the operating system as well as users on a uniform basis in a variety of situations. At the cost of occasional wasteful repetition, we have removed all of the previously discussed disadvantages associated with the software locking. It is not claimed that these disadvantages would not exist in any system that used these two algorithms, rather, from among several reasons that introduce complexity in a system, two have been removed. Next, we briefly examine the special instruction. The rest of the paper is devoted to the description of the algorithms followed by their applications.

1.4 The 68/80 stacq instruction

The 68/80 stacq (Store A Conditional on Q) machine instruction is a read-alter-rewrite (RAR) class instruction that changes the content of a single word memory operand if the following condition is met: If the content of the operand word is equal to the Q register then the A register is stored into the operand and the zero indicator is set on. Otherwise the operand is left unchanged and the zero indicator is set off. The RAR instructions have the property that while a processor is executing an instruction of this type, all other RAR instructions directed to the same system controller (and executing on other processors) are delayed. Since RAR instructions do not lock out non-RAR instructions, it is essential that all write operations on memory words that might be operands of RAR instructions, be performed with RAR instructions. When describing the algorithms that we mentioned before, we will use a PL/1 procedure abstraction of the stacq instruction in order to keep the unessential machine specific detail out of our way. The first argument to this procedure is the name of the memory word whose content need to be examined and replaced, the second argument is the expected content of this word, and the third argument is the value to be placed in the word if stacq succeeds. The procedure returns a single bit value of "1"b on success and "0"b on failure.

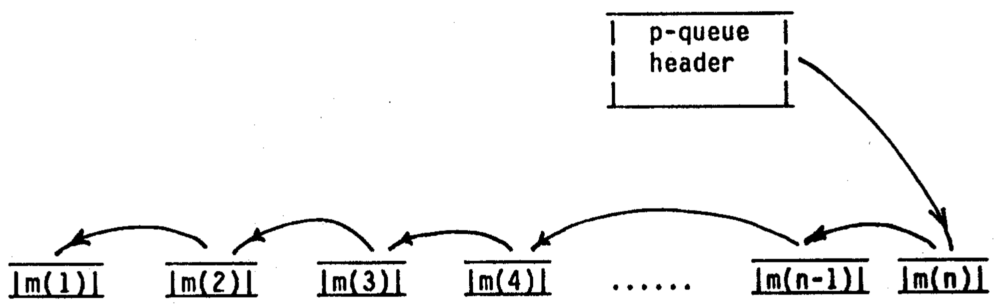
2.0 A multiple-producer single-consumer queueing algorithm

Consider a system consisting of several processes, called the producers, that are generating messages to be sent to a single process, called the consumer. This form of communication among processes can be programmed using a shared queue. We define two operations on this queue, called ENQUEUE and DEQUEUE. Whenever a producer has generated a new message and needs to send it to the consumer process, the producer appends the message at the end of the queue by performing the ENQUEUE operation. The consumer process performs the DEQUEUE operation to remove the oldest message (if any) from the queue. The messages are removed from the queue in the order of their arrival and processed accordingly. In general, to ensure consistency, the processes must be synchronized such that: (1) a producer does not perform the ENQUEUE operation while another process is performing either one of the operations on the same queue, and (2) the consumer does not execute a DEQUEUE while a producer is performing the ENQUEUE on the same queue. We present here an algorithm to manipulate such a queue which uses the stacq instruction and requires no explicit synchronization other than that implied by the stacq instruction.

The algorithm follows. Let the multiple-producer, single-consumer queue (called MPSC queue, hereafter) consist of a p-queue (for producer queue) and a c-queue (for consumer queue). Let p-queue consist of a header (called pq) and a number of messages m(1), m(2), m(3), ..., m(n) such that message m(i+1) is more recent than m(i). (*See figure 1*). When the p-queue is empty, the header pq is empty; otherwise pq points to the most recent message in the p-queue (i.e. pq points to m(n)). Associated with each message m(i), let there be a back pointer m(i).bp and a forward pointer m(i).fp such that when the messages are in p-queue, m(i+1).bp points to m(i) and m(1).bp is empty. The forward pointers are not used in the p-queue and we shall describe their role later when discussing the c-queue. Now, there are two operations on the p-queue that can be programmed in terms of stacq without any software locks or synchronization; these operations are:

1 - PQ$ADD_MESSAGE: Add a message to the p-queue
2 - PQ$REMOVE_CHAIN: Remove the entire chain of messages from the p-queue.

A producer process can add a message to the p-queue at any time by using the first operation. The consumer process can remove an entire chain of messages from the p-queue by the second operation and then save this chain in the c-queue such that all messages in the c-queue are in proper order. Since only one process, namely the consumer, is using the consumer queue there is no synchronization problem in the processing of the c-queue. Notice that the chain of messages that the consumer process removes from the p-queue can be built into a symmetrical linked list where only the forward pointers need to be filled in.



Messages arrive in the sequence m(1), m(2), m(3), ..., m(n).

- A p-queue -

Figure 1

PL/1 procedures for operations on the p-queue are provided in *figure 2*. A step by step explanation follows. The operation PQ$ADD_MESSAGE is performed in the following steps:

1 - Copy the pointer from the p-queue header into a local area.[1] The pointer either points to the most recent message in the p-queue or is null.

2 - Assign this pointer to the back pointer of the new message.

3 - If the p-queue header hasn't changed in the meantime then change it to point to the new message. This step is performed by a single stacq instruction. If the stacq succeeds, the operation is complete, otherwise we go back to the step 1 and start the operation all over again. If this operation is succesful the p-queue header will point to the new message.

[1] A local area is private to a process and not used by other processes. In Multics, the stack segment is the local area for a process.

```
PQ$ADD_MESSAGE: procedure (pq, message);

declare  pq pointer unaligned parameter;
declare 1 message aligned parameter,
        2 fp pointer unaligned,
        2 bp pointer unaligned,
        2 text bit (*);

declare  lastmp pointer unaligned automatic;
declare  stacq ext entry (pointer, pointer unaligned, pointer unaligned)
         returns (bit (1));

step_1:   lastmp = pq;
step_2:   message.bp = lastmp;
step_3:   if ^stacq (addr (pq), lastmp, addr (message))
          then goto step_1;
          return;
     end PQ$ADD_MESSAGE;


PQ$REMOVE_CHAIN: procedure (pq) returns (pointer);

declare  pq pointer unaligned parameter;

declare  chainp pointer unaligned automatic;

declare  stacq ext entry (pointer, pointer unaligned, pointer unaligned)
         returns (bit (1));

step_1:   chainp = pq;
step_2:   if ^stacq (addr (pq), chainp, null ())
          then goto step_1;
          return (chainp);
     end PQ$REMOVE_CHAIN;
```

NOTE: It is assumed that the internal representation of PL/1 "pointer unaligned" data type occupies a single memory word.

<u>*Figure*</u> *2*

The operation PQ$REMOVE_CHAIN is performed in the following steps:

1 - Copy the pointer from the p-queue header into a local area.

2 - If the p-queue header hasn't changed then change its value to null. Again this step is performed by a single stacq instruction. If it succeeds the copied value of the pointer points to the chain of messages. If the stacq fails we go back to the step 1 and try again.

Using these two operations it becomes possible to construct the ENQUEUE and DEQUEUE operations on the MPSC queue.

In this algorithm we have assumed that each message has the same unique address in the producer and consumer processes. Since each message has a unique address, adding a message to the p-queue will necessarily change the value of the p-queue header which is essentially a change in the version number of the p-queue. However, it should be noted that even though the message addresses correspond to version numbers, it is still possible to reuse message addresses. Let us examine the algorithm PQ$ADD_MESSAGE. Suppose that while a producer process is executing PQ$ADD_MESSAGE between the steps 2 and 3, the p-queue is changed by other processes such that the p-queue header contains the same address, that is, even though the p-queue has changed, its version number has not changed.[1] In this situation, process executing PQ$ADD_MESSAGE at step 3 will not be able to detect the change in the p-queue and therefore it will succeed in appending a new message to the p-queue. However, structure of the p-queue is still correct and the operation remains well defined. The algorithm to remove an entire chain of messages from the p-queue also works in a similar fashion and produces correct result. However in a single consumer system, while the consumer is executing the algorithm to remove an entire chain of messages from the p-queue, the p-queue can not change such that the version numbers remain the same.

Despite the apparently non-terminating loops in both operations on the p-queue, notice that an attempt to perform any one of the operations can fail only if some other process has succeeded in altering the p-queue. For a single-producer, single-consumer system if the first attempt to add a message to the p-queue fails then the second attempt is guaranteed to succeed since the first failure could have resulted only from the consumer having emptied the p-queue. The p-queue has another interesting property which is that to add a message to the p-queue the producer process does not need to touch the old messages in the p-queue. Both these properties make the single-producer, single-consumer version of this queue particularly useful for use by the interrupt handlers for the input devices. In most operating systems, interrupt handlers execute in severely constrained environments. In Multics, interrupt handlers can neither take page faults nor can they wait indefinitely for a lock to become free. The inability to take page faults implies that every part of

[1] This type of change may occur if after step 2 and before step 3 of an add operation, the consumer-process has processed the p-queue, freed the space occupied by the message at the top of the p-queue, and some other process has used the same space for another message which is now sitting at the top of the p-queue.

the virtual memory that can be referenced by the interrupt handler must be constrained to the primary memory. And, since the invocation of an interrupt handler must be completed in a short period of time, an interrupt handler is usually programmed such that if it encounters a lock already set by some other process, it quickly returns rather than wait for the lock to become free and then it becomes the responsibility of the locking process to complete the action initiated by the interrupt handler. This often requires extremely complicated synchronization mechanisms between the interrupt handlers and processes that share data with the interrupt handlers. Using the algorithm presented here, the interrupt handlers can ENQUEUE new messages into a shared queue for some consumer process in a finite number of operations without having to wait indefinitely for a lock and secondly, the old messages are freed from being constrained to the primary memory.

3.0 An algorithm to synchronize resource assignment

In this section we present another algorithm for one of the most common synchronization problems that occur in any modern multi-programmed computer system, namely, the problem of synchronizing the use of shared objects and facilities among processes that might be executing in parallel. A tape controller, controlling several tape drives, is a shared object that should be in use by at most one process at any time. The synchronization problem in using the tape controller is that of ensuring that not more than one process uses the tape controller at any given time. The tape controller problem is a restricted case of a more general problem that occurs when there are more than one functionally equivalent objects to be shared among several processes and for proper operation it is required that the use of the shared objects satisfy certain rules. One common rule is that an object may not be in use by more than one process at any given time. To enforce this rule, computer systems require that a process can use a shared object only after the object in question has been assigned to that process and the assignment algorithms ensure that no shared object is assigned to more than one process at a time. In general, we have a pool or a set of functionally equivalent objects and a process may request that it be assigned one, any one of these objects. The synchronization problem consists of ensuring that an object is not assigned to more than one process at any given time. All the tape drives in the example of the tape controller represent such a pool. In a paging system the pool of free memory blocks represents such a pool of resources. The ENQUEUE and DEQUEUE operations on the multiple-producer, single-consumer queue may also be viewed as shared objects. However, while the use of the DEQUEUE operation is governed by the rule that only one instance of it should exist at any time, the ENQUEUE operation may be in use by many processes at the same time. Each instance of the ENQUEUE or DEQUEUE must still be in use by only one process at any given time.

It needs to be emphasized here that we are not trying to answer the question whether a process should be allowed access to a particular resource or not - that is the larger problem of resource allocation - we are merely interested in enforcing a fundamental requirement of any resource allocation policy. This requirement is a manifest of the nature of objects and states that no object should be in use by more than one process at any given time. We are assuming that a simple rule of not attempting to use objects that have not been assigned is either voluntarily observed by the processes themselves or enforced by the system in some manner. At the same time we are seeking a mechanism that can be uniformly applied to either the system defined objects or user specified objects.

Suppose we are given a resource set R, consisting of a fixed and finite number of functionally equivalent objects. Even though these objects are functionally equivalent, we associate a unique identifier with each object which makes it possible to distinguish among them. Without any loss of generality, we can assign the numbers 0, 1, 2, ..., n-1 to these objects. Thus resource set R = {r(0), r(1), r(2), ..., r(n-1)}. We define the following two operations on R:

1 - ASSIGN
2 - RELEASE

The ASSIGN primitive is invoked by a process seeking assignment of a resource from R. This primitive returns two arguments, the first of which is a boolean value indicating whether the assignment was performed or not, and the second argument returns the identifier of the assigned resource if the assignment was performed. Once a resource has been assigned, it becomes unavailable for further assignment until released. A process may release a resource by invoking the RELEASE primitive whose only argument is the identifier of the resource to be released.

In order that same ASSIGN and RELEASE primitives may be used on different resource sets, we add the name of the resource set as an extra argument to each of these primitives, whereupon the primitives look as follows:

1 - ASSIGN (resource_set, assigned_sw, resource_id)
2 - RELEASE (resource_set, resource_id)

The "assigned_sw" and "resource_id" in the ASSIGN primitive are return or output arguments and all other arguments are input arguments.

3.1 A simple resource assignment algorithm

Consider an array R(0:n-1) associated with the resource set R containing n objects. Let each element of this array be a single memory word such that the stacq instruction can operate on it. Let R(i) = 0 indicate that the resource object r(i) has not been assigned to any process and is available for assignment. Then a succesful execution of the stacq instruction with the second argument (the expected value) equal to zero and the third argument equal to some non-zero value on the element R(i) may be defined to be the succesful assignment of object r(i) to the process on whose behalf the stacq was executed. The ASSIGN primitive simply consists of cycling through the array R(0:n-1) with stacq operations (as defined above) on successive elements until one stacq operation succeeds. Notice that it does not matter on which element the ASSIGN primitive starts. RELEASE (R, i) merely consists of setting R(i) to zero.

The most important characteristic of this algorithm is its simplicity and there are situations for which this algorithm is quite suitable. However, the following disadvantages make it less attractive as a general resource assignment algorithm:

1 - If none of the objects in the resource set R is available, the ASSIGN primitive will cycle through the array indefinitely thus wasting processor time.
2 - The processor time taken by the ASSIGN primitive will depend upon the number of objects, n, in the resource set R. It is to be expected that larger the value of n, greater will be the time taken by the ASSIGN primitive.

There are methods of solving the first problem, but only at the cost of added complexity. The second problem is intrinsic to the algorithm and not much can be done about it.

3.2 An efficient but impractical resource assignment algorithm

In this section, we present an algorithm that is efficient in the sense that the time taken to execute the ASSIGN and RELEASE primitives does not grow with the size of the resource set, but somewhat impractical in the sense that its implementation requires an unbounded array. Despite its impracticality, this algorithm is presented here as a stepping stone to the final resource assignment algorithm described in the next section of this paper.

Let R(0:*) be an infinite length array, each element of which is a single memory word capable of storing an integer and hence a resource-id. Let IN and OUT each be a single memory word also capable of storing an integer value that indexes into the array. Let each element of the array be initialized to an integer value of minus one (-1). The choice of -1 is somewhat arbitrary; it merely needs to be an integer value that can not be assigned to a resource-id. Let IN and OUT be initialized to zeroes. Roughly speaking IN points to the array element available for storing the next resource-id and OUT points to the array element where the next resource-id can be picked up from, and all the elements between IN and OUT contain identifiers of resources that are available for assignment. The RELEASE primitive consists of storing the resource-id into array element pointed to by IN and advancing IN to the next element. The ASSIGN primitive consists of grabbing a resource (if any) from the array element pointed to by OUT and advancing OUT to the next array element. If we view the array as extending from left to right then initially IN and OUT are positioned at the left-most element of an empty array and each RELEASE operation moves IN by one to the right, and each ASSIGN operation moves OUT by one to the right. Now we assert that the algorithms presented in *figure 3*, correctly implement the ASSIGN and RELEASE primitives as defined previously. An explanation of the steps involved is provided next. First, the RELEASE procedure. In step 1, we copy the value of IN into a local variable "in". The null step 2 is provided solely to facilitate comparison with a modified version of this algorithm to be presented later. In step 3 we attempt to store the resource-id into the array element being pointed to by "in". However, it is possible that some other process has already used up this word, in which case our attempt will fail. After completion of step 3, regardless of success or failure, the array element pointed to by "in", has been used up and IN needs to be advanced to the next element of the array. This is accomplished by step 4. Step 4 is crucial in the effective use of this algorithm. If we were to rely upon the process that used up the array element to advance the IN pointer to the next element, then we would have a situation where if a process stopped between steps 3 and 4, every other process trying to perform the RELEASE operation would have to wait for the stopped process to restart and complete the step 4. In step 5 we examine whether we succeeded in releasing the resource or not, and if not, we start the operation all over again.

The ASSIGN procedure: In step 2, we are trying to get a resource, but it is entirely possible that there are no more resources, in which case we return to the caller with an indication of failure (step 3). If, however, we had succeeded in getting a resource-id in step 2 then we must ensure that no other

```
RELEASE: procedure (resource_set, resource_id);

declare 1 resource_set aligned parameter,
          2 (IN, OUT) fixed binary (35),
          2 R (* /* 0:infinity */) fixed binary (35);
declare  resource_id fixed binary parameter;

declare (word_value) fixed binary (35) automatic;
declare  in fixed binary (35) automatic;
declare (SUCCESS, ignore) bit (1) automatic;

declare  stacq ext entry (pointer, fixed bin (35), fixed bin (35))
         returns (bit (1));

step_1:   in = IN;
step_2:
step_3:   SUCCESS = stacq (addr (R (in)), -1, (resource_id));
step_4:   ignore = stacq (addr (IN), in, in+1);
step_5:   if ^SUCCESS then goto step_1;
          return;
     end RELEASE;


ASSIGN: procedure (resource_set, assigned_sw, resource_id);

declare 1 resource_set aligned parameter,
          2 (IN, OUT) fixed binary (35),
          2 R (* /* 0:infinity */) fixed binary (35);
declare  resource_id fixed binary parameter;
declare  assigned_sw bit (1) parameter;

declare (word_value) fixed binary (35) automatic;
declare  out fixed binary (35) automatic;

declare  stacq ext entry (pointer, fixed bin (35), fixed bin (35))
         returns (bit (1));

step_1:   out = OUT;
step_2:   word_value = R (out);
step_3:   if word_value = -1 then do;
               assigned_sw = "0"b;
               return;
          end;
step_4:   assigned_sw = stacq (addr (OUT), out, out+1);
step_5:   if ^assigned_sw then goto step_1;
step_6:   resource_id = word_value;
          return;
     end ASSIGN;
```

<u>Figure</u> 3

process takes the same resource-id. This is accomplished by step 4 which attempts to move the OUT pointer one element beyond the current element. Whichever process succeeds in moving OUT from "out" to "out"+1 has the resource in R(out) assigned to it. If we fail to accomplish it, then clearly some other process has grabbed the resource and we try again from start.

If the resource-set contains a maximum of n objects, then at most n elements of the array may be in use at any given time. This corresponds to the situation of all resource objects being available for assignment. Using this fact we can modify this algorithm to use a finite length array rather than an arbitrarily large, unbounded array. The algorithm, thus modified, is described next.

3.3 An efficient resource assignment algorithm

Rather than view the array R as a straight line extending from left to right, let us view it as a spiral staircase starting from the ground and going vertically up. We view each step of this spiral staircase as corresponding to an element of the array. We define those steps that correspond to the elements containing identifiers of available resource objects to be "active" and others as "non-active". Thus if the resource-set contains n objects then there can be at most n active steps at any given time. Now, let a rotation of 360 degrees correspond exactly to the ascent of N steps where N is an integer greater than or equal to n. Then an orthogonal projection of the spiral staircase on any horizontal plane is a circle and each step is mapped onto a piece of the pie formed by the angle 360/N degrees. Furthermore, we can divide all steps into N equivalent classes such that any two members of an equivalent class are projected on the same piece of the pie. Also, any two members of the same class must be at least N steps apart (because N is greater than or equal to n). Since at any given time all active steps must be on a contiguous section of the staircase and there can be no more than n active steps, projection of all active steps must be on a contiguous section of the circle. This suggests that it should be possible to describe the state of the staircase by a circle as far as the active steps are concerned. We can accomplish this if, with each piece of the circle, we keep a measure of how many 360 degree revolutions have been completed to reach the corresponding active step on the spiral staircase. For step number S (the first step being number zero), this quantity is the quotient that results when S is divided by N. Since the projection circle and its pieces can be represented by a single dimensional array R(0:N-1) of length N, we have succeeded in representing the infinite length array by a finite length array. However, each element of the finite length array must store two quantities: (1) the resource-id of the object, and (2) the number of circles transcribed by the corresponding element in the infinite array. Next we describe a method of fitting both these quantities into a single memory word.

If we identify resource objects by integers 0, 1, ..., n-1 (as we did before), then we can combine the circle number and resource-id into an integer capable of being stored in a single memory word by the following PL/1 computation:

```
C = divide (S, N);          /* Compute the circle number */
V = C*N +resource_id;       /* Compute the combined value */
```

Given a value V, the circle number, resource-id, and step number (corresponding element of the infinite length array) can all be computed by the following PL/1 computation:

```
C = divide (V, N);              /* Compute circle number */
resource_id = mod (V, N);       /* Compute resource-id */
S = C*N + index;                /*Compute step number */
where index is the number of element on the finite length array
from which the value V is obtained.
```

In the above formulae, the function "mod" returns the remainder and function "divide" returns the quotient. Complete PL/1 procedures for these algorithms are presented in *figures 4 and 5*. These algorithms are quite similar to those described in the section 3.2 for an infinite length array. In the RELEASE primitive *(figure 4)*, the most significant difference occurs in steps 2 and 3. IN points to an element in the infinite length array and a corresponding stair in the spiral. Before we attempt to store the new resource-id into the finite length array element pointed to by IN, we must make sure that this element represents a stair lower than the one being pointed to by IN. We do this by computing the number of the stair being represented by the value contained in the finite length array element and comparing it with IN. The ASSIGN primitive of this section *(figure 5)* differs from that of section 3.2 in a similar respect in steps 2 and 3.

Initialization of the finite length array differs from the unbounded case in that it is the circle number portion of the array elements which is initialized to minus one rather than the complete element. Initial values of IN and OUT are zeroes. The PL/1 entry INIT of procedure RELEASE *(figure 4)* will properly initialize a resource-set.

```
RELEASE: procedure (resource_set, resource_id);

declare 1 resource_set aligned parameter,
          2 N fixed binary,
          2 (IN, OUT) fixed binary (35),
          2 R (* /* 0:N-1 */) fixed binary (35);
declare  resource_id fixed binary parameter;

declare (word_value, word_circle, word_step) fixed binary (35) automatic;
declare (in, in_index, in_circle) fixed binary (35) automatic;
declare (SUCCESS, ignore) bit (1) automatic;

declare  stacq ext entry (pointer, fixed bin (35), fixed bin (35))
         returns (bit (1));

step_1:    in = IN;
           in_index = mod (in, N);
           in_circle = divide (in, N, 35);
step_2:    word_value = R (in_index);
           word_circle = divide (word_value, N, 35);
           word_step = word_circle*N + in_index;
step_3:    if word_step < in
           then SUCCESS = stacq (addr (R (in_index)),
                          word_value, in_circle*N+resource_id);
           else SUCCESS = "0"b;
step_4:    ignore = stacq (addr (IN), in, in+1);
step_5:    if ^SUCCESS then goto step_1;
           return;


INIT:      entry (resource_set, size);

declare  size fixed binary parameter;

           N = size;
           do in_index = 0 to N-1;
                R (in_index) = -1*N;
           end;

           IN, OUT = 0;
           return;

     end RELEASE;
```

<u>*Figure*</u> *4*

```
ASSIGN: procedure (resource_set, assigned_sw, resource_id);

declare 1 resource_set aligned parameter,
        2 N fixed binary,
        2 (IN, OUT) fixed binary (35),
        2 R (* /* 0:N-1 */) fixed binary (35);
declare  resource_id fixed binary parameter;
declare  assigned_sw bit (1) parameter;

declare (word_value, word_circle, word_step) fixed binary (35) automatic;
declare (out, out_index) fixed binary (35) automatic;

declare  stacq ext entry (pointer, fixed bin (35), fixed bin (35))
         returns (bit (1));

step_1:   out = OUT;
          out_index = mod (out, N);
step_2:   word_value = R (out_index);
          word_circle = divide (word_value, N, 35);
          word_step = word_circle*N + out_index;
step_3:   if word_step < out then do;
               assigned_sw = "0"b;
               return;
          end;
step_4:   assigned_sw = stacq (addr (OUT), out, out+1);
step_5:   if ^assigned_sw then goto step_1;
          resource_id = mod (word_value, N);
          return;

     end ASSIGN;
```

Figure 5

4.0 Some Comments on the MPSC-queue and Resource Assignment Algorithms

The most important property of these algorithms is that the objects manipulated by them (i.e. MPSC queue and the resource-set) remain in a consistent state at all times and do not have any locks associated with them. This implies that even if a process is stopped forever while executing in the middle of the primitives used to manipulate these objects, other processes may continue to use these shared objects without having to wait for completion of the primitive initiated by the stopped process. We discussed the benefits of this property in *section 1.3*.

The main disadvantage of these algorithms is the possibility of endless repetition. In practice, however, it is not likely to be a serious problem other than occasional wasteful repetition.[1]

As we discussed in *section 2*, a single-producer, single-consumer version of the MPSC queue has the property that the producer is guaranteed to succeed in appending a message to the queue in at most two attempts. This property makes this queue particularly useful for interrupt handlers for input devices.

One limitation of the resource-assignment algorithm of *section 3.3*, not yet discussed, is that the number of times the RELEASE operation (and ASSIGN operation) can be performed for any resource-set is limited by the size of the memory word on which the atomic operation stacq acts. For a 36 bit word length, one can perform 34,359,738,367 RELEASE operations and an equal number of ASSIGN operations on any given resource-set. Assuming a rate of one RELEASE operation per millisecond, it will take 397 days before the number of permissible operations is exhausted. Therefore, in a practical sense, this limitation does not cause any problems in a machine of adequate word length.

---

[1] While executing one of these algorithms, a process will loop endlessly only if during each failure, some other process (or a combination of several processes) succeeds in performing an operation on the same data base. This, of course, implies that the sequence of succesful operations (carried out by other process or processes) is in exact synchrony with the sequence of failures. There are two reasons why we believe that this synchrony can not continue indefinitely. First, the looping process is continually executing the basic loop of the algorithm and doing nothing else whereas the succesful process is not only executing the basic loop but, presulmably, carrying out some other operations as well. One would therefore, expect that the processes will eventually get out of this synchronization. Secondly, in a multi-programmed system, any sequence of operations (whether successes or failures) would eventually be interrupted by the action of the scheduler whose job it is to multiplex the CPUs among several processes.

5.0 Applications

In this section, we shall briefly discuss applications of these algorithms. In *section 2.0*, we discussed an interesting application of the MPSC queue (multiple-producer, single-consumer) to interrupt handlers for input devices. This queue is applicable to output devices in a similar manner. Processes wishing to transmit data on a shared output channel can queue their data into a p-queue, which can be processed by the process controlling the shared channel.

Whenever a shared storage area consists of a fixed number of fixed size storage blocks, the resource assignment algorithm can be used for allocating and freeing of these blocks. Thus, this algorithm is applicable to assignment of entries in tables, assignment of message space in mail segments etc.

Since the MPSC queue can be used for transmition of messages among processes and the resource assignment algorithm can be used for allocation of space, it should be possible to construct almost any kind of inter-process communication system using these two algorithm. Such a system will have no software locks and none of the associated problems that we discussed in *section 1.1*. In its most simple form, Multics ipc_ event channels are nothing but a MPSC queue. In the next paragraph we shall briefly discuss construction of a mail system.

In mailing systems, where the recipient provides the space for storage of messages, there are two synchronization problems:

1 - allocation of space among senders and freeing up of space by the recipient.
2 - queueing of messages by senders and removal of messages by the recipient.

The first problem can be solved by the resource assignment algorithm if we divide the mail segment into fixed size blocks. The second problem can be solved by using the MPSC queue.

References

[Ea72] "Process Synchronization Without Long-term Interlock" by William B. Easton, (ACM) Operating Systems Review, vol. 6, no. 1,2, pp 95-100 (June 1972)