T H E   U N I V E R S I T Y   O F   M I C H I G A N
COLLEGE OF LITERATURE, SCIENCE, AND THE ARTS
Computer and Communication Sciences Department

AN ALGORITHM FOR DISTRIBUTED SCHEDULING WITH IMPLICIT SYNCHRONIZATION

Morton D. Hoffman

The Logic of Computers Group
and
Reliable Software Systems Group

(RSSM 19)

October 1975

Technical Report No. ~~188~~ 190

Supported by:

National Science Foundation
Grant No. DCR71-01997 to the Logic of Computers Group,
Computer and Communication Sciences Department

AN ALGORITHM FOR DISTRIBUTED SCHEDULING WITH IMPLICIT SYNCHRONIZATION

Morton D. Hoffman

ABSTRACT

An algorithm for distributed scheduling of tasks on the processors and devices of a computing system is presented and its correctness is proved. The algorithm is predicated on a computing system in which each device on the system is regarded as an autonomous processor responsible for its own scheduling decisions. The algorithm precludes the need for explicit synchronization among the system processors or for an interrupt facility. Inter-processor task switching is performed through a queueing mechanism.

INTRODUCTION

Processor and input-output device scheduling on computer systems has traditionally been based on the concept of the central processing unit which has full control of and responsibility over all activity in the system. The central processing unit is the only device in a computer system with the capability of exercising control over other processors. It is the only processor in the system with control paths to other devices (for I/O instructions) and from other devices (for interrupts), and it is the only processor with a sufficiently sophisticated order code to make scheduling decisions.

The scheme has been quite successful. However, as devices in a computing system become more complex in design, greater in number, and more diverse in form, the centralized control of the system has become awkward. In addition, multiprocessor computer systems have raised the question of which central processor is to be responsible for system scheduling. One response to these difficulties is the distribution of control more broadly through the computing system. Front-end computers now quite commonly handle complex communication links to present the central processor with a simple input stream. Computer system architectures have been developed in which smaller general purpose processors supplement the central processing unit, and perform many of the scheduling and ancillary tasks on its behalf [1,2]. Other computer systems have been developed in which a large community of mini-computers are organized about a shared main memory [3,4]. Concepts of distributed control are particularly significant in mini-computer design, since mini-computers are being designed with interfaces which often allow several distinct processors to be easily and economically interfaced in a variety of ways.

This paper is predicated on a concept of a computing system with distributed control, in which each device in the system is viewed as a processor, responsible for scheduling its own work [5]. Each processor (or device) must choose a task for execution from available tasks, execute it, and pass it to the next processor the task requires. To do this each processor must have a sufficient order code to execute a scheduling algorithm and to fetch a task for execution. Interaction (for the purpose of processor scheduling) is accomplished through shared queues of task control blocks. Thus, no network of control paths among the processors is needed. Since each processor performs its own scheduling, interrupts can be eliminated as a mechanism for control of the asynchronously operating processors.

This paper presents an algorithm which, when executed by each processor on the system, will manage the execution of tasks, scheduling them on various processors of the system (as needed) with a minimum of interference among the processors due to scheduling activity. The correctness of the algorithm will be proven.

INTER-PROCESSOR PROTOCOL

The architecture being proposed permits, and in fact requires, that the scheduling of tasks on the system processors be distributed through the system; that is, all processors, including I/O devices must schedule their own work. I propose that this be accomplished through the following basic algorithm (see figure 1).

Each processor will operate in a simple loop, fetching a Task Control Block (TCB) from its own work queue, executing the task until the task requests another processor, and inserting the TCB onto the requested processor's work queue. This is, in essence, the basic algorithm used by current systems for round-robin CPU scheduling, except that I have extended it to permit the TCB to be inserted onto a different processor's work queue. Thus, the work queue is analogous to the CPU queue of a traditional operating system, and, if the processor is a CPU, it is identical.

Each processor on the system will own a work queue, and will normally be the only processor permitted to remove a TCB from its work queue. Any processor may add to the work queue of any other processor (or even to its own). This forms the basis for most inter-processor communication.

For this queue manipulation we need to establish a protocol which will satisfy several requirements. Since all processors must execute it, it must employ a rudimentary instruction set. The protocol must avoid logical difficulties which might arise from several processors manipulating the same work queue simultaneously. Finally, it must minimize the interference among processors. It is also extremely desirable that the algorithm be simple enough so that its correctness may be readily established.

The algorithm I propose for this purpose is presented in flowchart form in figure 2 and in (fictitious) assembly language form in figure 3.

The algorithm requires some explanation. First, some assumptions: Each TCB begins with the sequence: link pointer (to the next TCB) followed by a pointer to the user program for this task. A null pointer (representing the end of a queue) is represented by the number zero (0). A minus one (-1) as a pointer represents a condition where the proper address has not yet been set, and may be found in the Z register of the processor which set the pointer. Finally, when the user enters the scheduling routine (at RETURN), he will leave the address of the next processor his task requires in register X, and the address of his own next instruction in register Y. The work queue described above is in reality divided into two queues. On one of these, WORKQ, TCB's are inserted at the top by any processor, and it is emptied by the owning processor zeroing WORKQPTR, where the address of the top TCB is stored. The top TCB is always the most recently inserted. The other queue, PVTQ, consists of TCB's that have been waiting for the processor longer than those on the previous queue. PVTQPTR is the pointer to the first TCB in this queue. At the top of this queue is the oldest TCB, and the top TCB is removed for running when the processor schedules a task. When this queue empties, it is renewed by the owning processor which takes the WORKQ and reorders it backwards (by reversing its pointers). It takes the top TCB to run, and places the remainder of the queue onto PVTQPTR. Thus, if we view the two queues as one, starting from the top of PVTQ to its bottom, followed by the TCB's in WORKQ from the bottom to its top, the TCB's encountered represent the logical work queue from the oldest TCB to the newest.

The code proposed uses a simple instruction set with very little arithmetic capability. Two instructions used may not be obvious: INC (increment) merely increments a register. SWP (swap) exchanges the value in a register with that in a word of memory, during which time no other processor is allowed access to the specified memory location. This latter instruction is at the heart of the algorithm, since it eliminates the need for inter-processor synchronization. It accomplishes this by doing a queue insertion or removal without the possibility of examination or modification of the queue in an intermediate state.

One should be aware that this is intentionally a very limited algorithm. Simplicity has been preserved with some effort in order to establish an algorithm which can be shown practical for a very simple processor. Otherwise, the system would be impractical on economic grounds. Note that the processor is not required to have any complex addressing or any arithmetic capability.

The scheduling provided by the algorithm is strictly FIFO over each processor (of which round-robin is a special case). One can extend this to any other scheduling algorithm, by having a processor always empty the WORKQ before scheduling any task. Then with all the TCB's on his PVTQ, he may manipulate them and choose among them as he sees fit. This would, however, involve a penalty in increased processing time and complexity of the algorithm.

CORRECTNESS OF INTER-PROCESSOR PROTOCOL

I will now prove the correctness of the scheduling algorithm. The proof will depend on two observations. First, no processor may add to the work queue (WORKQ) of another processor except at its top. Second, if we regard a -1 as an indirect pointer (in a logical sense), then at no time is the structure of the work queue incomplete or broken. Further, the logical structure is never changed except for insertions at the top, and by complete removal of the work queue by the owning processor.

The proof below refers to the algorithm in figure 3, as applied to the data structure of figure 5. Figure 4 represents the logical structure which is elaborated by the physical structure in figure 5. Note that a_1 is the address of the most recently inserted TCB, and a_n is the address of the least recently inserted TCB (the oldest) and the one which should be scheduled next.

The following notation will be used in the proof: x, y, z will represent the values held in registers X, Y, Z respectively. a_i represents the address of the ith TCB, and b_i represents the address of its user's code. $c(a_i)$ will be used to represent the value of the contents stored at address a_i, or more generally, c( ) will be used to specify the contents of the address given inside the parenthesis. When referring to the values of variables at a given step, the values at the end of the step will be implied, unless otherwise stated.

Consider the beginning of the algorithm. We have:

STEP	RESULTS	COMMENTS
1	$y=a_n$	0 iff PVTQ is empty (null pointer)
2	$y=a_n$	Branch on non-empty queue (PVTQ)

Now, assume a is zero so that we execute step 3 next. The case $a_n \neq 0$ will be considered later (in which case $y \neq 0$ and the branch to step 16 is taken). At step 3, y=0. Therefore, when the swap is performed, it empties the work queue by zeroing its pointer. Thus:

3	$y=a_1$ c(WORKQPTR)=0	
4	$y=a_1$	Branch back iff WORKQ null

If WORKQ and PVTQ are both empty, then there is no work to be done, and the processor loops here waiting for a TCB to arrive at WORKQ. No TCB can arrive on PVTQ, since no other processor can reference PVTQ. Thus the loop between steps 3 and 4 will cause the processor to wait only until there is work for it to do. Now, the processor will reach step 5 only with $y=a_1 \neq 0$. Note that after step 3, further changes to the work queue just obtained will not affect the algorithm, since other processors no longer have access to it. Further additions to the work queue will be to a new queue established at step 3 as empty by the insertion of 0 into WORKQPTR.

STEP	RESULTS	COMMENTS
5	$x=a_2$ or 0 or -1, $y=a_1$	x=0 iff a_1 is the last element on queue, x=-1 iff another processor is inserting a new value for a_1
6	$x=a_2$ or 0 or -1, $y=a_1$	Branch iff x=-1 (loop)
7	$x=a_2$ or 0, $y=a_1$	Branch iff x=0

Step 6 causes the processor to loop (back to step 5) if x=-1. This is a special flag indicating that the processor inserting this TCB into the queue has not completed the linkage and the value of the pointer has not yet been set properly. Since this flag is preset by the other processor (at step 28) before the TCB is linked into the queue (at step 30), the flag will be there until the pointer is properly set. Steps 5 and 6 will be repeatedly executed, until the flag is cleared by the insertion of the address a_2. Thus, at step 7, x can only have values of a_2 or 0. A value of zero signals the end of the queue, and implies that the queue is only one TCB long. In this case step 7 causes a branch to step 18, which situation will be examined later. Assume that step 8 is to be executed after step 7 (and therefore $x=a_2$).

With the processor about to execute step 8 of the algorithm, it has performed the tests to guarantee that PVTQ is empty, and WORKQ holds at least two TCB's. Moreover, it has emptied the WORKQ by setting WORKQPTR to 0, and must reverse the order of the TCB's on the queue it has obtained in order to set up PVTQ from it. The last TCB on the current queue, however, will not be put onto PVTQ, since this TCB is the candidate for immediate execution. Note that the queue links are now as indicated in figure 4 (i.e. $c(a_i) = a_{i+1}$ for $i=1,2,\dots,n-1$). With the links reversed, TCB_1 will be at the bottom of the list, and therefore, steps 8 and 9 set $c(a_1) = 0$.

STEP	RESULTS	COMMENTS
8	$z=0$, $x=a_2$, $y=a_1$	
9	$z=0$, $x=a_2$, $y=a_1$ and $c(a_1)=0$	End of PVTQ now flagged

Steps 10 through 15 represent the algorithm for reversing the pointers in the queue. It will be executed once for each pointer to be reversed, or n-1 times. It is entered each time with $y=a_i$ and $x=a_{i+1}$. After step 9, this invariant is satisfied with $i=1$.

before 10	$x=a_{i+1}$, $y=a_i$	
10	$z=a_{i+2}$, -1, or 0 $x=a_{i+1}$, $y=a_i$	-1 is not set flag, 0 signals end of queue, loop back on not set flag
11	$z=a_{i+2}$, -1, or 0 $x=a_{i+1}$, $y=a_i$	
12	$c(a_{i+1})=a_i$, $x=a_{i+1}$, $y=a_i$, $z=a_{i+2}$	Reverse pointer between a_i and a_{i+1}
13	$y=a_{i+1}$, $x=a_{i+1}$, $z=a_{i+2}$	Set up for next loop
14	$x=a_{i+2}$, $y=a_{i+1}$	
15	$x=a_{i+2}$, $y=a_{i+1}$	Loop back iff $x \neq 0$

In the above steps, we have not explicitly noted the three possibilities for $c(a_{i+1})$ fetched in step 10. Just as was the case at step 5, step 10 may fetch a -1, in which case steps 10 and 11 are repeated until the flag is cleared, or it may fetch a 0, which is detected at step 15. This represents (signals) the end of the queue, and is the termination condition for the loop (steps 10-15). Note that if $x \neq 0$, then the loop will repeat. The condition we set entering step 10 was $y=a_i$, $x=a_{i+1}$. These conditions are now satisfied with the value of i increased by 1. Thus the loop will re-execute for the succeeding TCB in the queue. By the structure of the queue, $c(a_i) = a_{i+1} \neq 0$ for $i=1,\ldots,n-1$, and $c(a_i) = 0$ for $i=n$. Thus the loop condition will be satisfied for $i=1,\ldots,n-1$, and after the n-1st execution, the condition will fail. This means that step 12 will reverse the pointers between a_i and a_{i+1} for $i=1,\ldots,n-1$, or more simply, all the pointers on the queue will be reversed.

The algorithm will reach step 16 in either of two circumstances. Either the step before (15) will have failed to branch, since $x=0$ (thus $y=a_n$, the top of the reversed list), or step 2 will have branched to step 16, having found a_n in the top of PVTQ already (thus $y=a_n$). In both cases, $y=a_n$, the address of a TCB which heads a queue of TCB's which runs in the order oldest at the top and newest at the bottom. The nth TCB (the top of the queue) will now be removed and executed, and the remainder of the queue will be saved as PVTQ by inserting $c(a_n) = a_{n-1}$ into PVTQPTR. Proceeding:

STEP	RESULTS	COMMENTS
16	$x=a_{n-1}$, $y=a_n$	Get second TCB on reversed list

STEP	RESULTS	COMMENTS
17	c(PVTQPTR) = a_{n-1} $x=a_{n-1}$, $y=a_n$	Establish PVTQ

Step 18 is executed after step 17, or, if WORKQ had a single TCB on it, after step 7. In either case, PVTQ is properly set up at this point in the algorithm. In the former case, it was set up by steps 16 and 17. In the latter case, there had only been a single item on WORKQ, and it has been selected to run. Thus there are no TCB's left to put on PVTQ and PVTQPTR should therefore be set to zero. However, PVTQPTR must already be zero, since it was examined by steps 1 and 2 and found to be zero (or the branch from step 2 to step 16 would have set up the former case). Thus, the queues are maintained in order for this processor, and the remainder of the algorithm is concerned with the execution of the TCB whose address is now in register Y, and following that, its insertion on another (or possibly the same) processor's work queue.

Since the TCB whose address is in register Y has been removed from the queues of waiting work, this has had the effect of reducing n by 1, and this TCB's address should no longer be referred to as a_n. Let us call its address $a_\#$ and the address of this TCB's user code $b_\#$ for consistency. Further, we will no longer have need to refer to this processor's queues, but will need to refer to some processor's queue. Let the variables corresponding to this other processor's queue and WORKQPTR be distinguished by italicizing their names. Also, for convenience let c(ACTIVE_TCB) be denoted by at. Continuing at step 18

18	at = $a_\#$, y = $a_\#$	

STEP	RESULTS	COMMENTS
19	$y=a_{\#}+1$, $at=a_{\#}$	Pointer to $b_{\#}$
20	$y=b_{\#}$, $at=a_{\#}$	$b_{\#}$ is address of user code
21	x=A(step 23) $y=b_{\#}$, $at=a_{\#}$	Give user return address
22	x=A(step 23) $y=b_{\#}$, $at=a_{\#}$	Branch into user code

At this point, the user is put into execution, and is provided a return address in register X. A return by the user to step 23 will be assumed, with register X set to the address of *WORKQPTR* for the processor at which the user desires to execute next, and register Y set to the address of the user code to be executed on the next processor. Call this address b_+ since it will be used to replace $b_{\#}$. Therefore, step 23 will be executed with x = A(*WORKQPTR*), and $y=b_+$.

23	$z=a_{\#}$, x = A(*WORKQPTR*), $y=b_+$	
24	$z=a_{\#}+1$, x = A(*WORKQPTR*), $y=b_+$	Get pointer to $b_{\#}$
25	$c(a_{\#}+1) = b_+$, x = A(*WORKQPTR*)	Replace $b_{\#}$ by b_+
26	$z=a_{\#}$, x = A(*WORKQPTR*)	
27	$y=-1$, $z=a_{\#}$, x = A(*WORKQPTR*)	
28	$c(a_{\#}) = -1$, x = A(*WORKQPTR*) $z=a_{\#}$	Flag pointer as not set
29	$y=a_{\#}$, $z=a_{\#}$ x = A(*WORKQPTR*)	

STEP	RESULTS	COMMENTS
30	$z=\alpha_1$ or 0	
	c(*WORKQPTR*) = $a_\#$	Link into *WORKQ*
	$y = a_\#$	
31	$c(a_\#) = \alpha_1$	Finish linking

At step 25, the new user code address was inserted into the TCB. The actual linking of the TCB into another processor's *WORKQ* occurs in steps 30 and 31. Before step 30, the only information about *WORKQ* that this processor has is the address of the *WORKQPTR*. Since that address is static, there can be no interference with the other processor before step 30. Also, the first work of the TCB is already preset to -1 so that the other processor will be held up if it tries to get the address of a TCB following the one to be linked into its *WORKQ*. After step 30 (before step 31), the *WORKQ* includes the TCB from this processor as its first entry and is logically complete if the -1 at location $a_\#$ is regarded as an indirect pointer through the Z register of the processor we are examining. Note that while it is physically incomplete, the owning processor (the processor owning the work queue affected), will not be affected by the incompleteness, except to wait when it encouters the -1. Finally step 31 completes the linking of the TCB, and the owning process may proceed if it is waiting, after at most a very short wait. Step 32 restarts the scheduling algorithm from its beginning.

The effect of one circuit through the scheduling algorithm has been the decrease of n by 1 (the running of one TCB's task, the oldest task), and the increase of another processor's work queue by the addition of one TCB. Thus the algorithm is complete and correct.

CONCLUSION

This paper has presented a scheduling algorithm requisite for the proper functioning of a computer architecture for distributed control. The algorithm is based on queue mediated interaction among the processors and devices in the system. A proof of the correctness of this algorithm has been given.

The work presented in this paper is part of a larger project whose goal is to complete the design of a computing system employing distributed control, and to demonstrate its practicality. To complete the design, issues of protection, efficiency, reliability, and expandability will be considered in further research. Finally, the range of applicability of a computer system of this type will be examined, and the application of these ideas to computer networks will be considered.

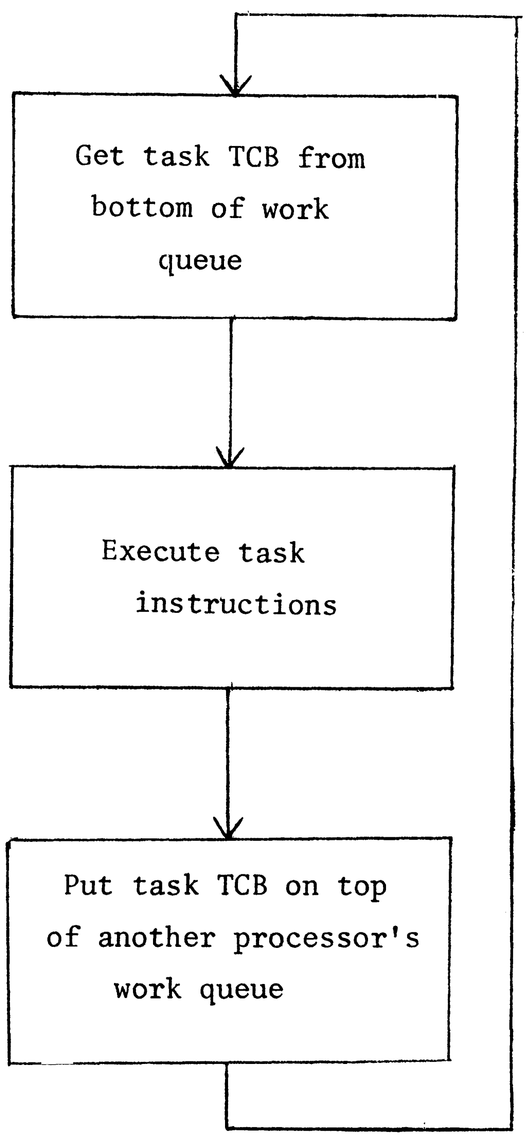

Get task TCB from bottom of work queue
Execute task instructions
Put task TCB on top of another processor's work queue

Figure 1

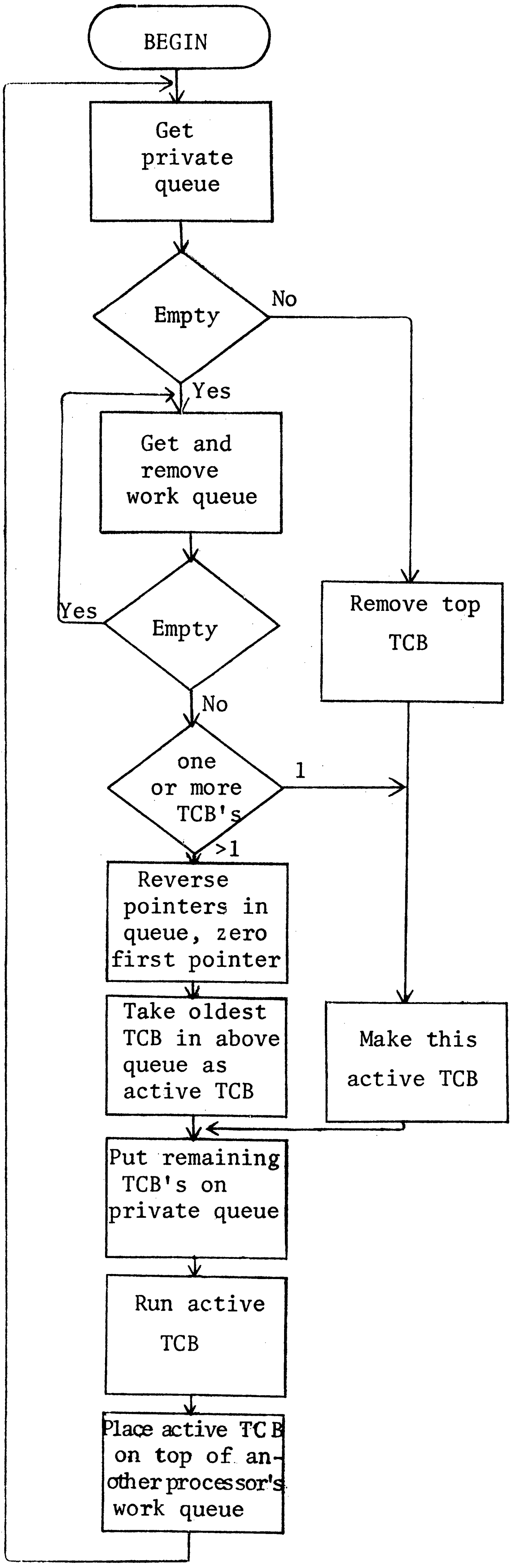
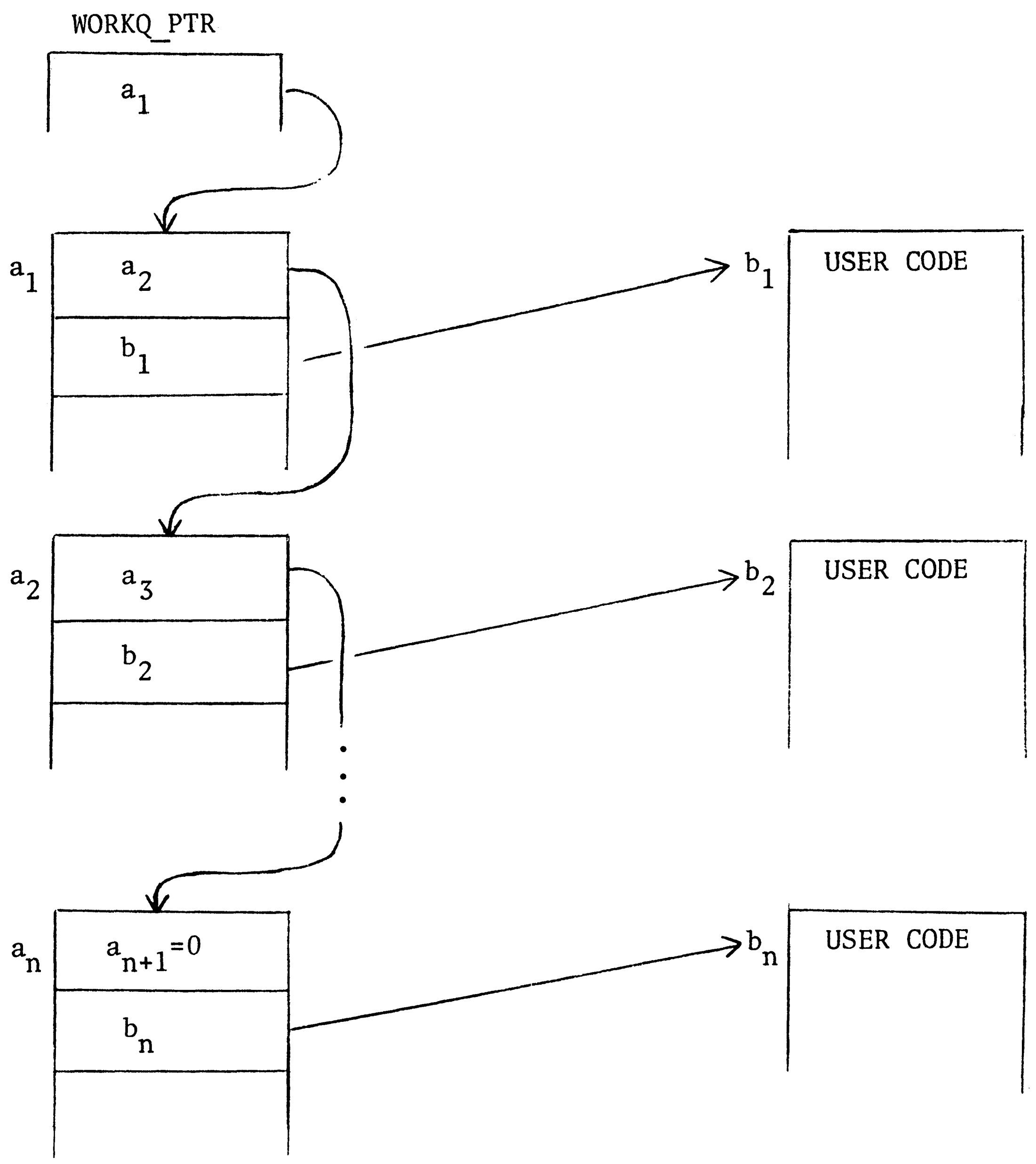

BEGIN
Get private queue
Empty
No
Yes
Get and remove work queue
Empty
Yes
No
Remove top TCB
one or more TCB's
1
>1
Reverse pointers in queue, zero first pointer
Take oldest TCB in above queue as active TCB
Make this active TCB
Put remaining TCB's on private queue
Run active TCB
Place active TCB on top of another processor's work queue

Figure 2

```
 1.  START    LD  Y   PVTQPTR     Get pointer to private queue
 2.           BNZ Y   RUNTOP      Non-empty → run top TCB
 3.           SWP Y   WORKQPTR    Get fresh list to search
 4.           BZ  Y   *-1         Empty → wait for it to get TCB(s)
 5.           LD  X   (Y)         Get pointer to second TCB on queue
 6.           BN  X   *-1         -1 → not set yet
 7.           BZ  X   RUNONE      0 → only 1 TCB on queue
 8.           LD  Z   ZERO
 9.           STO Z   (Y)         Zero pointer to mark end of reversed queue
10.  RELOAD   LD  Z   (X)         Get next TCB pointer
11.           BN  Z   *-1         Negative → address not set yet, wait
12.           STO Y   (X)         Insert reversed pointer
13.           LR  Y,X
14.           LR  X,Z
15.           BNZ X   RELOAD      ≠0 → continue down queue
16.  RUNTOP   LD  X   (Y)         Get pointer to second TCB on queue
17.           STO X   PVTQPTR     Establish second TCB as top of queue
18.  RUNONE   STO Y   ACTIVE_TCB  Save address of TCB
19.           INC Y               Get pointer to user code
20.           LD  Y   (Y)
21.           LD  X   A(RETURN)   Give user address to return
22.           B       (Y)         Branch into user code
23.  RETURN   LD  Z   ACTIVE_TCB  Get TCB of user just executed
24.           INC Z
25.           STO Y   (Z)         Save address of user code in TCB
26.           LD  Z   ACTIVE_TCB
27.           LD  Y   MIN_ONE
28.           STO Y   (Z)         Flag pointer in TCB as not set
29.           LR  Y,Z
30.           SWP Z   (X)         Swap TCB with other processor's WORKQPTR
31.           STO Z   (Y)         Complete WORKQ linkage
32.           B       START       And restart

     ZERO     DC      0
     MIN_ONE  DC      -1
```

Figure 3



WORKQ_PTR
a_1
a_1
a_2
b_1
b_1
USER CODE
a_2
a_3
b_2
b_2
USER CODE
a_n
$a_{n+1}=0$
b_n
b_n
USER CODE

Figure 4

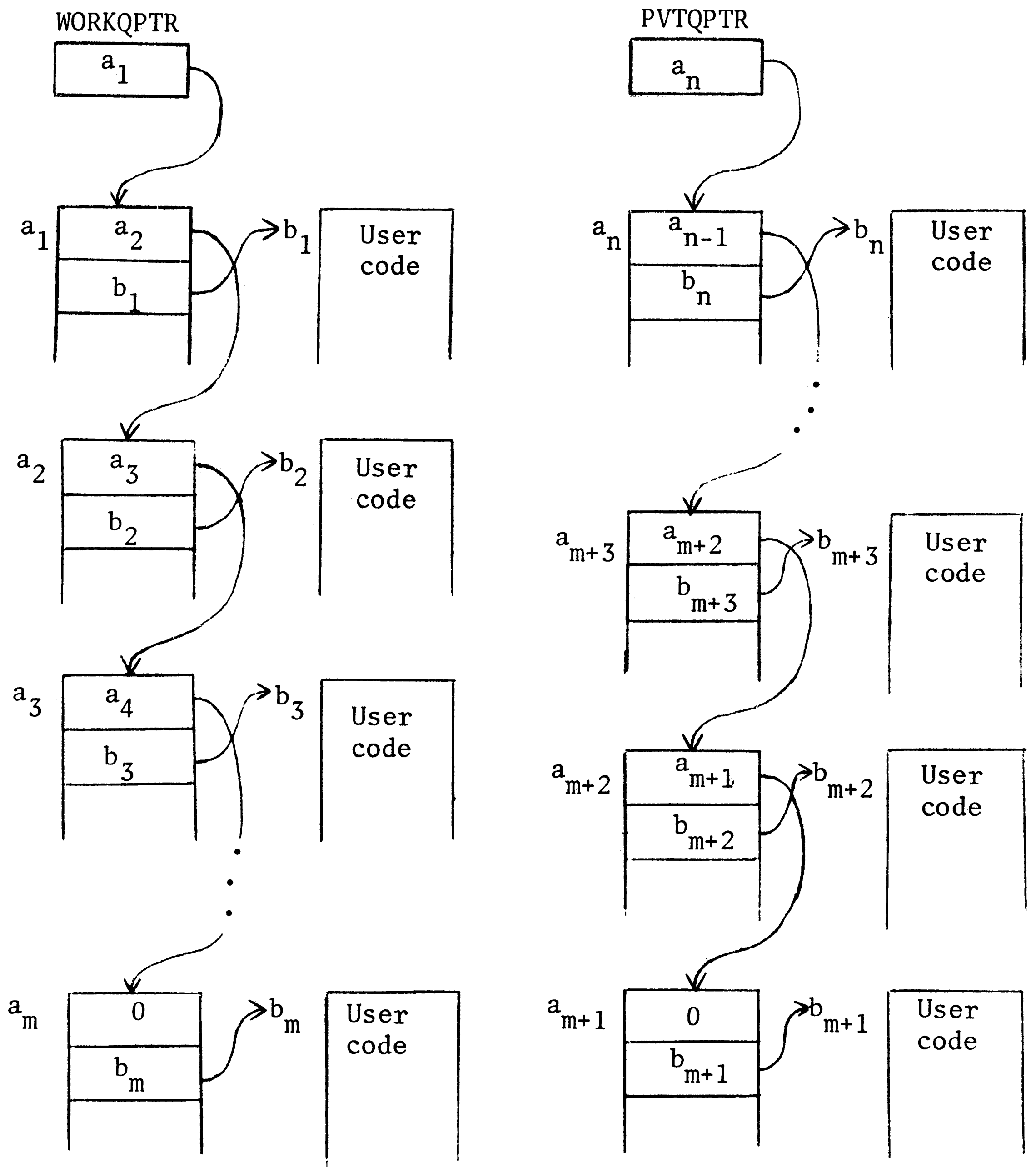

WORKQPTR
a_1
a_1
a_2
b_1
b_1
User code
a_2
a_3
b_2
b_2
User code
a_3
a_4
b_3
b_3
User code
a_m
0
b_m
b_m
User code
PVTQPTR
a_n
a_n
a_{n-1}
b_n
b_n
User code
a_{m+3}
a_{m+2}
b_{m+3}
b_{m+3}
User code
a_{m+2}
a_{m+1}
b_{m+2}
b_{m+2}
User code
a_{m+1}
0
b_{m+1}
b_{m+1}
User code

Figure 5

REFERENCES

[1] IBM System/360 Attached Support Processor (ASP) System Description H20-0223

[2] Thornton, James E., Parallel Operation in the Control Data 6600, *Proceedings* of AFIPS, 1964, FJCC, pt. 2, *V. 26*, pp. 33-40.

[3] Wulf, W.A. and Bell, C.G., C.mmp--a multi-mini-processor. *Proceedings* AFIPS 1972, FJCC, *V. 41*, pp. 765-777.

[4] Wulf, W.A. *et al.*, HYDRA: The Kernel of a Multiprocessor Operating System, *Comm. ACM*, *V. 17*, 6 (June 1974), pp. 337-345.

[5] Hoffman, M., Proposal for a Computer Architecture Based on Queue Mediated Interaction. The University of Michigan Department of Computer and Communication Sciences RSSM/14.