Algebra of Synchronization with Application to Deadlock and Semaphores

Ernesto Gomez

School of Computer Science and Engineering
California State University, San Bernardino
California,USA
ernesto@csusb.edu

Keith Schubert

School of Computer Science and Engineering
California State University, San Bernardino
California,USA
schubert@csusb.edu

Abstract*—Modern multiprocessor architectures have exacerbated problems of coordinating access to shared data, in particular as regards to the possibility of deadlock. For example semaphores, one of the most basic synchronization primitives, present difficulties. Djikstra defined semaphores to solve the problem of mutual exclusion. Practical implementation of the concept has, however, produced semaphores that are prone to deadlock, even while the original definition is theoretically free of it. This is not simply due to bad programming, but we have lacked a theory that allows us to understand the problem. We introduce a formal definition and new general theory of synchronization. We illustrate its applicability by deriving basic deadlock properties, to show where the problem lies with semaphores and also to guide us in finding some simple modifications to semaphores that greatly ameliorate the problem. We suggest some future directions for deadlock resolution that also avoid resource starvation.

***Keywords*-component; formatting; style; styling;**

I. Introduction

Concurrent programming has become increasingly relevant given the ubiquity of fast networks, multicore processors and highly parallel GPUs (Graphic Processing Units). In all but the most trivial cases, concurrent execution requires coordination between processes - that is, synchronization.

However we lack a comprehensive theory or even a standard definition of synchronization. For example, Kosaraju [14] admits that a precise definition of synchronization is lacking. Things are not much improved since then. Authors on operating systems [23], [22], [19], [18] tend to give examples of synchronization mechanisms, as do authors on parallel computing [7], [13], [17], but not a formal definition or theory. Apt and Olderog [2] establish a more theoretical approach to synchronization between parallel processes, focusing on the semantics of process waiting for conditions to become true. However they also limit themselves to giving specific examples of what these conditions might be. As a result there is little if any theoretical guidance for understanding synchronization issues. We here introduce a formal definition of synchronization and develop an algebraic theory that allows analysis of synchronization properties. To illustrate the usefulness of our theory, we will apply it in two cases. We will first use it to extend the well-known Coffman deadlock conditions [4]. We will further apply our theory to semaphores, one of the most used synchronization primitives. We here outline our theory of synchronization - a more detailed treatment with rigorous proofs of our claims is the subject of a forthcoming paper.

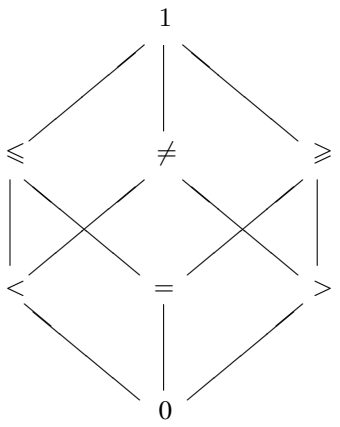
II. A Theory of Synchronization

We define a simple synchronization as a relation between the times at which two events occur, enforced by code. For example, a barrier is code that forces two events to occur at the same time; a mutual exclusion forces two events to occur at different times; a scheduler enforces an ordering of event times. It is evident (and easily provable) that synchronizations involving multiple events can be built up from simple two event synchronizations.

Note that synchronizations occur in real time; they are therefore defined in terms of an external real clock, independent of times as measured at each process. We take the conventional view of time as measured by clocks which report a single scalar number that is continuously increasing. A synchronization is then characterized by an ordered pair of numbers (t_1, t_2) and a relation between them, where t_1 corresponds to event 1 and t_2 to event 2. Since t_1 and t_2 are scalars, the set $\mathbb{R} = \{\geqslant, \leqslant, =, \neq, >, <\}$ exhausts all possible relations. We name the synchronization type after the relation in $\mathbb{R}$; so for example we may talk about an "equal" synchronization as the set of pairs (t_1, t_2) such that $t_1 = t_2$; that is, each synchronization type is given by a set of ordered pairs.

We now observe that the sets that characterize synchronizations are related to each other. For example the set of "$\leqslant$" pairs is the union of the sets "$<$" and "$=$"; the set "$>$" is the intersection of "$\neq$" and "$\geqslant$"; the sets "$=$" and "$\neq$" are complements of each other. If we add the set of all pairs, denoted by "1", and the empty set denoted by "0" to $\mathbb{R}$ we obtain the set $\mathbb{S} = \{1, \geqslant, \leqslant, =, \neq, >, <, 0\}$. We now see that $\mathbb{S}$ is closed under intersection, union and complement. We further observe that $\mathbb{S}$ is partially ordered by subset (superset). Under the subset relation, 1 is top, 0 is bottom, and we may graph the partial ordering as a cube. see Figure 1.

Figure 1. Partial ordering of $\mathbb{S}$



In fact, $\mathbb{S}$ together with union (meet,$\vee$) and intersection (join, $\wedge$) is a specific instance of a finite algebra of sets, and is therefore a Boolean algebra. (The proof that $\mathbb{S}$ is closed under union, intersection and complement and is therefore an algebra follows immediately from the fact that the set of relations in $\mathbb{S}$ is complete for numeric pairs).

Specifically, $< \mathbb{S}, \vee, \wedge >$ is isomorphic to the Boolean algebra $\mathbb{B}^3$, since it is an algebra of subsets with 8 elements [3], [12]. We are not claiming to introduce any new mathematics here; we are simply noting that an old and well developed area of mathematics can be applied to the definition of synchronization.

In general the times that appear in each synchronization pair are selected from an infinite set of numbers. However each synchronization in $< \mathbb{S}, \vee, \wedge >$ implies restrictions at least on one of the times in a pair; combinations of these restrictions can make it impossible to satisfy the specific synchronization, leading to deadlock. In algebraic terms, deadlock results when any combination of relations resolves to the relation 0 at any pair of events. It is easy to see how combinations of synchronizations acting on the same event pair can produce deadlock; simply enforce two synchronizations with empty intersection.

The more interesting and practical case results from a set of synchronizations involving multiple events; in this case the restrictions on a particular time t_0 may result from a combination of synchronizations with multiple other events $t_1, t_2, ..., t_n$, each of which may also have synchronizations with each other.

There are a number of ways of getting from the algebra of synchronization to a mathematics of a single event time; we have found it particularly useful to consider that each synchronization in $\mathbb{S}$ implies bounds on the times at which a specific event may occur.

We find that event times can be unbounded, be bounded from above, from below, or be bounded from above and below. We have therefore four boundedness classes:

- Unbounded, $\bar{B}$: No limit can be placed on how early or late an event time could occur.
- Bounded Above, B^+: A limit can be placed on how late an event time could occur but not on how early it could occur.
- Bounded Below, B^-: A limit can be placed on how early an event time could occur but not on how late it could occur.
- Bounded, B: Limits can be placed on both how early and how late an event time could occur.

We note that events restricted by $\neq$ and 1 are unbounded above and unbounded below ($\bar{B}$); relations $>$, $\geqslant$ both give bounded below(B^-) on the event time on the left, and above (B^+) on the event time on the right (the reverse applies to $<$ and $\leqslant$). $=$ and 0 are fully bounded - B (we consider that upper and lower bounds coincide in 0, excluding all possible times).

The set $\mathbb{L} = \{\bar{B}, B^+, B^-, B\}$ is complete, is an algebra under union, intersection and complement, and is therefore again a Boolean algebra. In this case $< \mathbb{L}, \wedge, \vee >$ is isomorphic to $\mathbb{B}^2$.

We observe, however, that there are two subalgebras of $\mathbb{S}$ that induce the algebra $\mathbb{L}$, as follows:

1) $\mathbb{L}_0 = \{\neq, >, <, 0\}$ such that $\neq$ is top and 0 is bottom.
2) $\mathbb{L}_1 = \{1, \geqslant, \leqslant, =\}$ such that 1 is top and $=$ is bottom.

(We may think of these as resulting from splitting the cube that represents $\mathbb{S}$ along the $=$ axis).

Note that 1 and $\neq$ do not establish upper or lower bounds, $>$ and $\geqslant$ (likewise $<$ and $\leqslant$) establish upper or lower bounds to times on the left or right of the relation, $=$ and 0 are bounded above and below (in the case of 0, which is the intersection of $>$ and $<$, there is no point that satisfies both boundaries).

An immediate conclusion is that deadlock (0) cannot occur if all relations are in $\mathbb{L}_1$, but it can in $\mathbb{L}_0$. Unfortunately, many if not most practical synchronization primitives appear to be in $\mathbb{L}_0$. Once we include any relation in $\mathbb{L}_0$ into a set of synchronizations, it becomes possible that an empty interval will result, which is deadlock.

III. Deadlock

Deadlock is a major problem in parallel programming. Substantial amounts of effort have been dedicated to deadlock detection, which would allow breaking the deadlock so processing can continue. We believe that the wrong problem is being addressed, due to application of deadlock criteria that depend on assumptions not appropriate to parallel execution.

Part of the problem is that there is no agreement on what is meant by deadlock. Tanenbaum [21] characterizes deadlock as a subset Σ of states such that there is no transition out of Σ, and there are no transitions in Σ which cause forward progress. Apt and Olderog, [2] similarly characterize deadlock, as a configuration that has no successor in the state transition relation. A more common characterization is found

in Lynch [15], who defines deadlock in terms of a circular dependence between processes. Elmasri and Navathe [6] give a similar definition, as do most database references of which we are aware.

Most authors, for example Silberschatz [18] and Tanenbaum [22] quote the results derived by E. G. Coffman, M. J. Elphick, and A. Shoshani's 1971 paper [4] which are:

1) Mutex
2) Hold and wait
3) No Preemption
4) Circular wait

We will define deadlock to be when any relation between events in an execution cannot be resolved and thus is in $\{0\}$. To see why we make this definition, we will first show how this definition relates to the Coffman conditions.

1) Mutex requires that equality cannot exist between events, and so a group of events are in $L_0 = \{\neq, <, >, 0\}$.
2) Hold and wait requires that the process itself will not change the timing constraint.
3) No Preemption requires that no other process can change the timing constraint. Essentially, 2 and 3 require a static timing relation.
4) Circular wait requires a chain of $<$ or $>$ which begins and ends with the same event, thus causing that event to be in $\{0\}$.

The Coffman conditions are thus one way (even the most common way) of achieving a deadlock, but they are not the only way. For instance, if a process becomes a zombie, or is waiting on a zombie process, it will deadlock because a zombie process is in $\{0\}$ by definition. There is no need for a circular wait in this case or even mutex in this case. Another case which results in deadlock is to write a program which will only run if it is sleeping, thus enforcing $A \neq A$ and thus $A \in \{0\}$.

Our definition also subsumes Tanenbaum's definition, as the requirement for no transitions out implies the event of being in Σ is in either $\{1\}$ or $\{0\}$, and the requirement of no progress implies it is in $\{0\}$. Apt and Oderog's definition only implies that some event is in $\{1\}$ or $\{0\}$, and thus includes things which are not deadlock, such as a program which asserts a power on signal. Such a program would not have a successor state but is still doing a useful action thus progressing, and so would not be considered as deadlocked, being in $\{1\}$. The remaining definitions essentially define deadlock as something that meets the Coffman conditions, and so also meets our definition.

Definition 3.1 (Deadlock): when any relation between events in an execution cannot be resolved and thus is in $\{0\}$.

This definition allows a clearer understanding of the problem with deadlock, even showing how adding semaphores to a program is dangerous by forcing the program into L_0, where we are a bad schedule away from deadlocking.

IV. Semaphores

Semaphores are an extremely useful construct in concurrent programming, used to implement critical sections and in general to keep processes from interfering with each other when accessing shared resources or data. Experience shows, however, that once we get past the simplest uses, in particular when multiple processes must acquire multiple semaphores, semaphores become hard to use. It is difficult in practice to understand the effect of all possible combinations of semaphore access by multiple processes, and deadlock becomes a possibility.

Djikstra's paper introducing the concept of semaphores [5] proved they were free from deadlock, but practical implementations have added complexity in the form of queues [19], [22]. This adds fairness but introduces the possibility of deadlock. Various other schemes have been proposed - for example "weak" queues and blocked sets [19], [20], multiple semaphore instructions [14], [16], [9], and other more complicated mutual exclusion primitives such as monitors [10], [11]. The most used mutual exclusion semaphore remains the queued semaphore, we suspect this is because of its simplicity and ease of hardware support. Even so, Tanenbaum remarks [23] that even a quarter century after their introduction, research into semaphores continues. This is still true, for example, Agarwal and Stoller [1] examines potential deadlock detection in programs using semaphores.

We will restrict our attention to semaphores as commonly implemented; that is the basic P and V primitives used for mutual exclusion, with a FIFO queue holding blocked processes to guarantee fairness [19].

An immediate challenge to our algebraic theory is the syntax of semaphores:

```
 P(s) // acquire semaphore s
          //    or wait in queue
{
   // critical section code block
}
V(s) // release semaphore s
```

A semaphore establishes an interval during which the $\neq$ relation holds (mutual exclusion), but we would like to identify a single event that we associate with a semaphore and here we apparently have two events - semaphore acquisition P and release V. Even so, mutual exclusions should not be dangerous. Even though they are in the subalgebra $\mathbb{L}_0$ and therefore admit the possibility of deadlock, any combination involving only union or intersection of $\neq$ still yields $\neq$, which is unbounded.

Problems are introduced by the queues we attach to semaphores to ensure fairness. Queues are an ordering, and thus a $>$ synchronization. Note that the intersection

$\{\neq\} \wedge \{>\} =>$; therefore a semaphore and queue is an $>$ ordering synchronization.

Semaphores are implemented as two instructions defining an interval. If a particular process acquires more than one semaphore, it is possible for the corresponding intervals to be nested or overlapped. What does $>$ mean in this situation? If we need to consider both P and V in analyzing the semaphore, then a semaphore P_i,V_i is a two point synchronization and does not fit in the algebra.

Fortunately, this does not happen, although the point is not clear in the literature. For example, recent editions of major Operating Systems texts such as Stallings [19] pg.227, and Silberschatz and Galvin [18] pg.204 point out a need to carefully analyze or nest the location of semaphore release instructions. It is possible to prove, by considering multiple cases of different combinations of process acquisition and release of semaphores, that in fact only the semaphore acquisition affects the order in which processes are blocked and released (as long as each semaphore acquisition is matched by a subsequent release in any order).

This fact is easily and elegantly proved in our theory, as follows. An alternate proof that does not rely on our theory is included in Appendix A for comparison.

Theorem 4.1: The boundedness class (in $\mathbb{L}$) of event sequences for processes synchronized by a set of semaphores depends only on the P instructions, not on the V instructions.

(i.e.: $\{P_1, P_2\} \times \{V_2, V_1\} := \{P_1, P_2\} \times \{V_1, V_2\}$)

Proof: Let $s_i = \{P_i\} \times \{V_i\}$ be a semaphore with P_i and V_i the usual acquire and release events. Now when a process tries to acquire the semaphore (P_i), it could be blocked or allowed to acquire. If it acquires the semaphore its boundedness class does not change as no new information on the boundedness class is learned, however if the process is blocked then the boundedness class has been intersected with $\neq$, if it is also queued then it is intersected with $<$. For any process that has the semaphore when another is blocked this causes its boundedness class with respect to the blocked process to be intersected with $\neq$. For a process that is queued before the new process becomes queued then based on the priority scheme the boundedness class will be changed for both. It does not matter how it changes (either intersected with $<$ or $>$), the key point is the P_i can and often will change the boundedness class of processes synchronized with a semaphore.

When a process has a semaphore, that semaphore does not provide any blocking to it. It can easily release the semaphore (V_i) and no process can block this. Some might protest that it might have to acquire another semaphore before it can release the current one, but that is a block on some P_j, not on V_i. It is thus apparent that V_i does not impose any restrictions on the boundedness class.

QED

Following the theorem, we can indeed analyze semaphores with queues as "$>$" synchronizations relating the P semaphore acquisition events, which determine process position in the semaphore queues. In the following theorem, we have generalized from semaphores to synchronizations which impose orderings $<$ or $>$ if in reverse order, which includes any semaphore with fairness conditions.

Theorem 4.2: In situations where only a common set of synchronizations, S with cardinality of 2 or greater, imposes orderings $<$ or $>$ if in reverse order, the following holds. There is a schedule that results in deadlock if and only if it is possible for processes to access elements of S in different orders.

Proof: (*If*) Assume there is a schedule which results in deadlock, thus we are in 0. The synchronizations can only induce $<$, or $>$ if they were in inverted order. It follows that the only way to obtain deadlock is $\{<\} \wedge \{>\} = 0$. Thus some process must acquire at least two synchronizations in the opposite order of some other process.

(*Only If*) Let q_1 and q_2 be two processes which share the set of synchronizations S. Since the cardinality of S is at least two, then there are at least two synchronizations $s_1 = P_1 \times V_1$ and $s_2 = P_2 \times V_2$ in S. Without loss of generality, consider the schedule, where q_1 seeks to acquire s_1 then s_2 and q_2 seeks to acquire s_2 then s_1. Thus due to q_1 we have $s_1 < s_2$, but from q_2 we have $s_1 > s_2$. We thus have $\{<\} \wedge \{>\} = 0$, and deadlock is possible.

Note that boundedness classes create necessary conditions on synchronizations. Sufficiency requires that q_1 acquire s_1 before q_2 and q_2 acquire s_2 before q_1. We only need to show existence, and since there is no atomicity between P_1 and P_2, consider the schedule where q_1 acquires s_1, then is interrupted, and q_2 woken, then it acquires s_2. Thus there exists at least one schedule, which results in deadlock if permutations of synchronization acquisition is allowed.

QED

That is, if any process acquires more than one semaphore, it is asserting $=$ (for the mutexes protected by each semaphore) and $\neq$ (the semaphores themselves) at the same time - this is the worst possible case, since the intersection $\{=\} \wedge \{\neq\}$ is empty! If another process also asserts both semaphores, binds them to $=$ at a different time, we can get deadlock. This is immediately evident from our algebra: Since mutual exclusion is in $\mathbb{L}_0$, anything that enforces it implies the possibility of deadlock.

The analysis of the problem with semaphores we have described is not dependent on the algebra of synchronization, although the algebra has guided us in it. The algebra, however, allows us to prove that the problem is inescapable - anything we do that involves mutual exclusion and multiple queues will introduce at least the possibility of deadlock.

We may here object that Djikstra proved both absence of deadlock and fairness properties for his semaphores in [5] However in Djikstra's scheme it is not possible for two different processes to assert P on the same semaphore. Therefore Djikstra semaphores are not affected by our deadlock proofs, but also are too restrictive for practical implementation.

A. How To Fix Semaphores

Djikstra's original concept of semaphores, in which semaphore owned by process and only owner could set, coupled with no queue or other structure, thus putting it in $\neq$ instead of $<$ or $>$, would not deadlock on its own, since multiple Djikstra semaphores would just be $\{\neq\} \wedge \{\neq\} = \neq$. Fairness considerations and other desirable system goals can impose conditions on the boundedness classes of the semaphores and result in forcing the system to null if the order of one semaphore acquisition is ever inverted (cycle). An immediate fix is to force acquisition of semaphores in a global order by a:

- runtime system like SOS [8].
- DLL linked to OS and compiler.

The fix that would work on current code without recompilation. Slightly more involved but superior fixes include:

- force process to release semaphores it holds if made to wait. This is unwieldy at the OS level, because if queues are length M and there are N semaphores, it is $O(N * M * M)$ (check all queues and compress as needed)
- bounce processes into random wait (like ethernet) and force semaphore release - better, since if there are no queues, work by OS is $O(N)$.

None of these are perfect, which underscores why there continues to be interest in semaphore research.

B. Alternatives

A deeper question we need to ask is should we fix semaphores. Semaphores are not just a construct that requires careful programming - once we are out of Djikstra's original concept, semaphores are in theory an inherently dangerous construct - and not even fully protective, since resources may be grabbed without checking semaphore.

A natural objection arises to not throw out a common tool that has been used so long due to flaws. In response we certainly acknowledge the great history of the semaphore, but note the flaws are fundamental to the construct. A semaphore either lacks fairness, can deadlock, or requires some global mitigation scheme (ordering, forced release, etc.) which violates the distributed nature of semaphores. There is no avoiding the possibility. Do we wish to continue to use a flawed concept due to history?

A second objection arises as to what alternatives exist. We note that with regard to resource access - processes don't actually need to grab resource, they just need access to a resource queue. Each process can have a locally owned queue that signals the OS when it has items in it, then OS can check the process queues, order the resource requests, and pass requests to global queue - this is like a queue of queues, or a multi-tailed queue. For normal resource access, reads are not an issue so you only need this for writes, though a combined system could be done if needed. This scheme has a number of implications:

- ALGEBRAIC IMPLICATIONS - since the multi-tailed queue is global and both ordered and maintained by the OS, it imposes a consistent ordering synchronization. Theorem 4.2 thus shows that such a multi-tailed queue cannot deadlock, since queues impose orderings $<$ or $>$ if in reverse order, but the globally ordered queue does not permit reverse order, so no schedule can deadlock. This corresponds to uniqueness of resource - semaphores are a distributed construct, no particular reason they should lead to any consistent ordering on a single shared resource.
- DESIGN PRINCIPLE - cardinality of control should match cardinality of resource - if single resource, access control should be single as well.
- DB - database needs to sequence reads, as well as writes - multi-tailed queues still work, but now they include writes. Note certain circumstances might require priority queues.
- REALTIME - requires hard allocation to meet time constraints, so the multi-tailed queue will not work, but this does not work under semaphores either.
- SCALABILITY - in a distributed system, each node handles its own queues which are global to the node. Only problem is inter-node requests.
- INTER-NODE - if all inter-node requests are handled as messages, then an SOS system message queue resolves all scheduling conflicts, see [8]. The only problem not handled by this is a data cycle.
- DATA CYCLES - can be broken by assuming values, converge over multiple iterations to consistent values (standard numeric technique). Problem only arises in inter-node message cycles, which could be detected by SOS-like system (since SOS knows the variables it is communicating - high level semantics interaction to runtime system). If OS can detect cycle, OS can schedule a thread for iterative solution.

A multi-tailed queue is thus a feasible solution to synchronizations in most areas without requiring semaphores.

V. Conclusions

We have introduced a formal definition and algebra of synchronization, which permits the analysis of when deadlock can occur. We followed with an analysis showing that the four traditional deadlock conditions follow from our algebra, but in fact describe a subset of the problem.

We then analyzed semaphores and shown why current implementations fail, while Djikstra's version does not. We then presented fixes and an alternative that will not deadlock.

We are preparing a larger monograph that expands the algebra of synchronization and its implications, with detailed formal derivations of all the results in this paper. We expand the concept to include data cycles and solutions to this problem, which were introduced briefly here and in [8].

Further work includes the possible application of the algebra of synchronization to optimization of synchronization code.

APPENDIX

The following proof of the first semaphore result (Theorem 4.1) does not rely on the algebra of synchronization we developed and is included to demonstrate not only the correctness of the first proof, but also its elegance and the related power of the theory. Recall that Theorem 4.1 states that the order of process release by a set of semaphores depends only on the P instructions, not on the V instructions.

Proof: We claim:

P1 ... Pn x Vi ... Vj is the same (semantically) for any permutation of Vi.

This is true because any process Q that reaches x has blocked all other processes attempting it at some semaphore Pi. Q executing x is always free to continue and release all semaphores Pi in P1 .. Pn, and allow all held processes to continue.

The held processes may be released in different order depending on the permutation of the Vi. There are three possibilities:

1) All processes need to execute P1 .. Pn in that sequence to reach x. In this case, all processes will be held in the queue at P1, because Q has blocked P1 and allowed no other process to get by. Held processes will be released in queue order, independent of the permutation of Vi.
2) Some processes execute a subsequence of the Pi. In that case, there may be a process R held at some Pk, k>1, while other processes T are held at P1. Assume this is the case. Then, either P1 is released before Pk, or Pk is released before P1. If Pk is released before P1, then R will hold the lock at Pk, and all processes T will be held there. If P1 is released first, then the processes T will either reach Pk before it is released or after. If before, then they will be queued behind R. If after, then R holds the lock Pk and all processes T will be queued at R. In all cases, R executes X before any process in T. By induction extends to any set of processes R held at any of the Pi.

   Therefore process execution order depends either on the queues or on the order of semaphores, and not on the permutation of Vi.
3) Some processes execute a permutation of the Pi to reach x.

   Consider first processes S and R, such that S is held at Ps and R is held at Pr. Process Q, which is executing x, holds all locks.

   R must execute Pr before Ps, and S must execute Ps before Pr for this to happen, else each process would have been held at a different Pi (because Q holds all the locks and has therefore blocked everybody). In fact, it must be the case that Pr is the first Pi in the sequence R executes, and Ps is the first Pi executed by S.

   Assume that S is released first. Now we have case 2, above: that is, the result is that R holds the lock at Pr and S is in the Pr queue. But in order to get here, S holds the Ps queue, which R will encounter after its release. Therefore we have deadlock because S is held in queue Pr, Q is held in Ps, and neither gets to execute x and the Vi afterwards.

   Same applies if R is released first, by symmetry. By induction, extends to N processes.

THEREFORE: The order of semaphore release does not affect the order of execution of the critical section by processes held in the semaphore => the permutation of Vi does not change the semantics of multiple semaphores.

QED

REFERENCES

[1] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 51–60, New York, NY, USA, 2006. ACM.

[2] Krzystof R. Apt and Ernst-Rudiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer, New York, 2nd ed. edition, 1997. Chapter 6.

[3] Garrett Birkhoff and Saunders Mac Lane. *Algebra*. American Mathematical Society, 3rd ed. edition, April 1999.

[4] E.G. Coffman, M.J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[5] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9), September 1965. The original semaphores paper - Describes a mechanism with shared arrays that implements mutual exclusion on a critical section for N processes. Proof of particular algorithm described.

[6] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, second edition, 1994.

[7] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[8] E. Gomez, Y. Karant, and K.E. Schubert. Preventing Deadlock with Dynamic Message Scheduling. In H. Selvaraj and V. Muthukumar, editors, *Proceedings of the* 18^{th} *International Conference on Systems Engineering*, pages 52–57, Los Alamitos, Ca, 2005. IEEE.

[9] Peter B. Henderson and Yechezkel Zalcstein. Synchronization problems solvable by generalized pv systems. *Journal of the ACM (JACM)*, 27(1), January 1980. An analysis of the kinds of problems solvable with semaphores. Shared memory is assumed.

[10] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10), October 1974. The original paper introducing monitors¿ Monitor is an abstract data structure, with internal data and methods. monitor does mutex by preventing more than one external program to enter at a time, blocks other attempts.

[11] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), August 1978. Shows that unbuffered message passing, rendezvous synchronization is sufficient for parallel computation.

[12] Nathan Jacobson. *Lectures in Abstract Algebra*. Springer-Verlag, 1951. Chapter VII.

[13] Harry F. Jordan and Gita Alaghband. *Fundamentals of Parallel Processing*. Prentice Hall, 2003.

[14] S. Rao Kosaraju. Limitations of dijkstra's semaphore primitives and petri nets. *ACM SIGOPS Operating Systems Review, Proceedings of the fourth ACM symposium on Operating system principles*, 7(4), January 1973. Admits that a precise definition of synchronization is lacking, does not attempt it. Assumes delays between instructions, uses blocked-queue definition for semaphores. Defines Petri net.

[15] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1995. Chapter 19.

[16] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2), February 1979. Proposes different synchronization mechanism.

[17] L. Ridgway Scott, Terry Clark, and Babak Bagheri. *Scientific Parallel Computing*. Princeton, 2005.

[18] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne Wylie. *Operating System Concepts*. Wiley, 7th ed. edition, 2005.

[19] William Stallings. *Operating Systems, Internals and Design Principles*. Prentice Hall, New Jersey, New Jersey, 5th ed. edition, 2005.

[20] Eugene W. Stark. Semaphore primitives and starvation-free mutual exclusion. *Journal of the ACM (JACM)*, 29(4), October 1982. Attempts a formal definition of two varieties of semaphore primitives: Discussion is standard counter, and binary semaphores.

[21] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, New Jersey, New Jersey, 1995.

[22] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, New Jersey, 2nd ed. edition, 2001.

[23] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems, Design and Implementation*. Prentice Hall, New Jersey, 3rd ed. edition, 1997.