SIAM J. COMPUT.
Vol. 18, No. 1, pp. 152-165, February 1989

010

MINIMAL THRESHOLD SEPARATORS AND MEMORY REQUIREMENTS FOR SYNCHRONIZATION*

EDWARD T. ORDMAN†

Abstract. Suppose that in a system of asynchronous parallel processes, certain pairs of processes mutually exclude one another (must not be in their critical sections simultaneously). This situation is modeled by a graph in which each process is represented by a vertex and each mutually excluding pair is represented by an edge. Henderson and Zalcstein have observed that if this graph is a threshold graph, then mutual exclusion can be managed by simple entrance and exit protocols using **PV**-chunk operations on a single shared variable whose possible values range from zero to t, the minimal threshold separator number of the graph. A new expression is given for this separator t of a threshold graph in terms of the normal decomposition of the threshold graph given by Zalcstein and Henderson. It is shown that $t+1$ values would be needed in the shared variable even if the mutual exclusion were being managed by the Fischer-Lynch test-and-set operator, which is considerably less restrictive than **PV**-chunk.

Key words. mutual exclusion, threshold graphs, synchronization primitives, test-and-set, PV-chunk

AMS(MOS) subject classifications. primary 68Q10; secondary 68R10, 05C70

1. Introduction. Concurrent processing by several asynchronous processes presents control problems that have been widely studied [1], [2], [4]-[6], [9], [11], [14]. It may be, for instance, that due to a need to access shared resources, such as a printer or shared data, certain events must be prevented from happening simultaneously in two (or more) processes. One approach is to have a designated section of code in each process identified as a **critical section** and to have the processes execute a joint algorithm that controls access to the critical sections. This algorithm is typically represented within each process by two protocols: the **entry protocol**, which is a section of code executed by a process before it is admitted to its critical section (and in which it may loop for some time, if the shared resource is in use by other processes); and the **exit protocol**, which is executed when the process leaves its critical section and makes the shared resource available to other processes. See [2] for a more precise discussion.

The processes may communicate by sending and receiving messages or by manipulation of one or more **shared variables**. A large number of ways of accessing shared variables have been studied, such as elementary read and write operations [5], **P** and **V** operations [6], **PV**-chunk operations [9], and test-and-set operations [2], [14]. In this paper we will implicitly be using the model of critical sections and of test-and-set operations laid out in [2], which provided one of the principal motivations for this paper. One of our goals is to compare the **PV**-chunk and test-and-set operations in a certain context; we describe them further in § 4.

Many of the papers cited above study a form of the mutual exclusion problem in which only one process can be in a critical section at one time. In fact, there are problems of interest in which more than one process can be in a critical section at a time. An early example of such a problem in the literature is called the **dining philosophers problem** (see [10] and some earlier references cited therein); a very practical problem of this type is the **readers-and-writers problem** (see [17] and the references therein) in which several processes may be allowed to read a data item simultaneously, but a process may change the item (write it) only at a time when no other process is

* Received by the editors January 30, 1985; accepted for publication (in revised form) April 12, 1988. This work was partially supported by National Science Foundation grant DCR-8503922.
† Department of Mathematical Sciences, Memphis State University, Memphis, Tennessee 38152.

accessing it. A set of processes, some of which may not enter critical sections simultaneously, is sometimes called a generalized dining philosophers problem; one study of such problems is given in [12]. A second principal motivation for this paper was the hope that the sort of analysis done in [2] could be extended to generalized dining philosophers problems. Only a small start is made on that here: we discuss memory requirements for mutual exclusion (although not lockout prevention, starvation avoidance, etc.) for a limited class of problems, which do include some forms of readers and writers problems. We find that, for this very limited goal, **PV**-chunk operations are as powerful as test-and-set operations, although they appear not to be as memory-efficient in some other cases.

Some generalized dining philosophers problems correspond in a natural way to graphs. Suppose each vertex of a graph represents a process, and two vertices are **adjacent** (connected by an edge) if and only if the two corresponding processes cannot be executing their critical sections simultaneously. In this way each finite undirected graph without self-loops (we describe this more formally below) corresponds to a generalized dining philosophers problem. On the other hand, not every generalized dining philosophers problem corresponds to a graph. Consider four processes, each of which requires two tape drives, in an environment where four tape drives are available. Clearly any two processes can (if all goes well) obtain two drives each and proceed, so the corresponding "graph" would have no edges. However, three processes can never acquire enough tape drives (and enter their critical sections) at once, so a hypergraph would be required to model this situation. For some further discussion of this analogy see [9], [13]. We will be considering mutual exclusion in systems of processes represented by a certain class of graphs, the threshold graphs. Graph theory background is given in the next section.

2. Graph theory preliminaries. By a **graph** $G=(V, E)$ we mean a finite set of vertices V together with a finite set of edges E, each of which is a different unordered pair of distinct vertices: that is, a finite undirected graph without self-loops or parallel edges. If a and b are vertices we say that they are **adjacent** if (a, b) (or equivalently (b, a)) is an edge of G; we also say that this edge **connects** a and b.

Threshold graphs were introduced in [3] and have been studied extensively [9], [11], [13], [15], [16]; we will rely very heavily on [9]. A graph $G=(V, E)$ is a **threshold graph** if there is an integer t called the **threshold** (or sometimes the **separator**), and with each vertex x in V is associated a nonnegative integer label $a(x)$ such that a subset S of N is **stable** (no two nodes in it are connected by an edge in E) if and only if the sum of the $a(s)$ for all s in S is less than or equal to t. (We will call such a labeling, including knowing t, a **threshold labeling**). A great many other characterizations of threshold graphs are known (see, for example, [3], [9], [13]); some that we need are recalled at the start of § 4.

A graph is a threshold graph if and only if it has a threshold labeling. The labeling, however, is not unique. In [16] Orlin has given an algorithm for determining the **minimal separator** t (and an associated labeling) that will work for a given threshold graph. In § 4 we use a slight modification of the **normal form** for a threshold graph (implied in Corollary 1B, [3, p. 151] described in detail and named in [9]) to give a more closed form for the minimal value of t, and to see that this minimal value of t is important in determining the minimum amount of shared memory to do mutual exclusion.

In § 5 we study a mutual exclusion problem (Example 5.2) for a generalized dining philosophers problem that corresponds, in the sense described in § 1, to a threshold graph. The connection between threshold graphs and **PV**-chunk operations has already been discussed in [9]. We find that the minimal separator t is a measure of the amount of memory needed to enforce mutual exclusion (without lockout prevention) in a system of processes represented by a threshold graph, using **PV**-chunk operations; we further see that at least this much memory is required, even if test-and-set operations are used instead. We see that test-and-set is, in cases other than threshold graphs, less demanding of memory than **PV**-chunk. We also look at a simple application to concurrent accesses of data bases.

3. The minimal separator of a threshold graph. We review some graph theory terminology. If $G=(V, E)$ is a graph and V' is a subset of V, the **subgraph** of G **induced** by V' is the graph whose vertices are the vertices of V' and whose edges are all edges of E which connect points in V'. A graph is a **clique** if every two points in it are adjacent; a subgraph of another graph G is a clique if it is a clique considered as a graph by itself, and is a maximal clique in G if it is not contained in any larger subgraph of G which is a clique. The clique with n vertices is denoted K_n.

The **degree** of a vertex is the number of vertices adjacent to it. We call a vertex **isolated** if it has degree zero, and **nonisolated** otherwise. We will call a vertex **dominating** (this is *not* a standard notation) if it is adjacent to every nonisolated vertex. By the **neighborhood** $N(x)$ of a vertex x we mean the set of vertices adjacent to x, together with x itself.

By way of making these terms clearer, we restate a well-known fact [3] about threshold graphs: no threshold graph may have as an induced subgraph any of the graphs shown in Fig. 1. These are the path on four vertices, P_4; the cycle on four vertices, C_4; and the union of two disjoint edges, $2K_2$ (note that a single edge is a clique on two points, K_2).

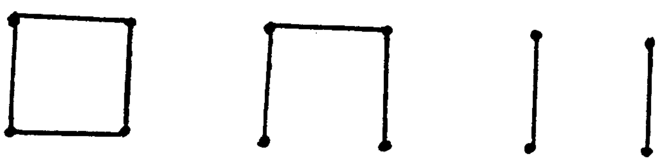


FIG. 1. *Graphs C_4, P_4, and $2K_2$.*

To prove that none of these graphs can be an induced subgraph of a threshold graph G, suppose that one of them is and label the four vertices w, x, y, and z. Suppose also that (w, z) and (x, y) are edges, but that (w, y) and (x, z) are not (e.g., label the vertices in each graph in Fig. 1 clockwise from upper left.) We will consider the labelings from a threshold labeling of G. Clearly, $a(w)+a(z)>t$ and $a(x)+a(y)>t$, since these pairs of vertices induce edges; clearly, $a(w)+a(y)\leq t$ and $a(x)+a(z)\leq t$, since these pairs do not. Adding the two pairs of inequalities produces a contradiction.

We also recall that an induced subgraph of a threshold graph must be a threshold graph; one simply restricts the threshold labeling to the subset of vertices and retains the same separator t.

Given a set of vertices V there may be various labelings $a(x)$ of the vertices and various separators t associated with them that lead to the same threshold graph G. For a given G, we want to determine the smallest possible t. This has been done in [16], which gives an algorithm to compute the smallest possible t; we approach the problem slightly differently to get t in a more explicit form we can apply later in the paper. To this end, we need to recall and modify slightly the definition of "normal form" of a threshold graph given in [9].

LEMMA 3.1. *Let G be a threshold graph with an associated threshold labeling. Let x be a vertex with a label as large as any other label in G. Then x is a dominating vertex of G.*

Proof. Let z be any nonisolated node. Then it is connected to some node y and $a(z)+a(y)>t$. But $a(x)\geqq a(y)$, so $a(z)+a(x)>t$ and z is adjacent to x. □

In [3] threshold graphs are characterized by the fact that the three graphs of Fig. 1 cannot occur as induced subgraphs. We will need an additional characterization.

THEOREM 3.2. *For a graph G, the following are equivalent:*

(a) *G is a threshold graph.*

(b) *Every induced subgraph of G (including G itself) has a dominating node.*

(c) *G does not have as an induced subgraph the graphs P_4, $2K_2$, or C_4.*

Proof. Items (a) and (c) have been proved to be equivalent in [3]. Item (a) implies (b) by Lemma 3.1, since every induced subgraph of a threshold graph is threshold. Item (b) implies (c), since none of P_4, $2K_2$, or C_4 has a dominating vertex. □

The fact that (b) implies (a) is already implicit in [3]. That paper also contains the following Corollary 1B. A graph $G=(V, E)$ is threshold if and only if there is a partition of V into disjoint sets A, B, and an ordering $a_1, a_2, \cdots, a_k$ of A such that no two vertices in A are adjacent; every two vertices in B are adjacent; and if $j\leqq k$, then $N(a_j)\subseteq N(a_k)$. This fact is developed considerably in [9]; we will expand on it further here.

Given a threshold graph, we construct the normal form as follows. Choose all isolated vertices and put them in class D_0; take all dominating vertices and put them in class C_1. The subgraph of G induced by the remaining vertices we call G_1. For each consecutive subgraph G_k, place the isolated vertices in D_k, and the dominating vertices in C_{k+1}. Continue until G_{n+1} is empty.

Note the following:

(a) No vertex in any D_k is connected to any other vertex of any D_j (including $k=j$).

(b) Every vertex of every C_k is connected to every vertex of every C_j (including $k=j$).

(c) Every vertex of D_k is connected to every vertex of C_j for $j\leqq k$, but to no other vertices.

We may need to rearrange the last sets slightly to guarantee that both C_n and D_n are nonempty and that both C_{n+1} and D_{n+1} are empty. We distinguish, temporarily, two cases. If C_{n+1} is empty, D_n is nonempty (in fact it has at least two vertices, since if there were only one it would have been in C_n) and C_n is nonempty (else the construction would have stopped sooner). In the second case, C_{n+1} is nonempty. Then C_{n+1} must contain at least two vertices; otherwise the one vertex would have been in D_n. In this case we arbitrarily choose one vertex of C_{n+1}, move it to D_{n+1} (previously empty), and increase n by one. Thus we also have C_n and D_n both nonempty, and proceed with further analysis.

It is easy to see that all C_k and D_k are nonempty for $1\leqq k\leqq n$, since before each removal there are dominating vertices by Theorem 3.2 and their removal must leave some vertices isolated or else each dominating vertex of the newly reduced graph would have been already dominating prior to the reduction.

We call the resulting decomposition of G into $(D_0, D_1, \cdots, D_n, C_1, C_2, \cdots, C_n)$ the **normal form** of G. It is unique except perhaps for the choice of one node when we combined the two cases above; we tolerate that since it simplifies calculations below.

In Fig. 2 we illustrate the normal form, and the labeling that will be introduced below. On the left we show the graph G; the right shows the same graph with the vertex in C_1 labeled 11, that in C_2 labeled 8, those in D_1 labeled 1, and those in D_2 labeled 4. For this graph and labeling, $t = 11$. To convince ourselves that lower labels will not work, note that (once r, s, and u are labeled 1 and the separator is t) vertex x must have label at most $t-3$ since x, r, s, u induce no edges; now x and w induce an edge so w and similarly z must have label at least 4. Since r, s, u, z, and w induce no edges, t must be at least 11. The reader may find it helpful to delete vertex w from G and find the normal form and a labeling.

THEOREM 3.3. *Let G be a threshold graph and $(D_0, D_1, \cdots, D_n, C_1, C_2, \cdots, C_n)$ its normal form. Let d_k denote the number of vertices in D_k. Then the vertices can be labeled so as to give a threshold labeling with separator t satisfying*

$$t+1 = \prod (d_k + 1), \qquad k = 1, \cdots, n.$$

Proof. Our labeling method is the same as that of [9] and [16]. For simplicity of formulas we define $g_i = \prod (d_j + 1) - 1$, where the product is for $j = 1, \cdots, i$; by definition $g_0 = 0$. Assign each element of D_0 the label 0 (they lie on no edges). Assign each element of D_1 the label 1; the total of all the labels assigned so far is d_1, which is equal to g_1. Assign each element of D_2 the label $d_1 + 1$; the total of the labels in D_2 is $d_2(d_1+1)$ and the grand total so far is g_2. We show by induction that when we assign each element of D_k the label $g_{k-1}+1$, the labels in D_k will total $d_k(g_{k-1}+1)$ and the sum of the labels to this point will be g_k. The induction step is to observe that $g_k + d_{k+1}(g_k+1) = g_{k+1}$, that is,

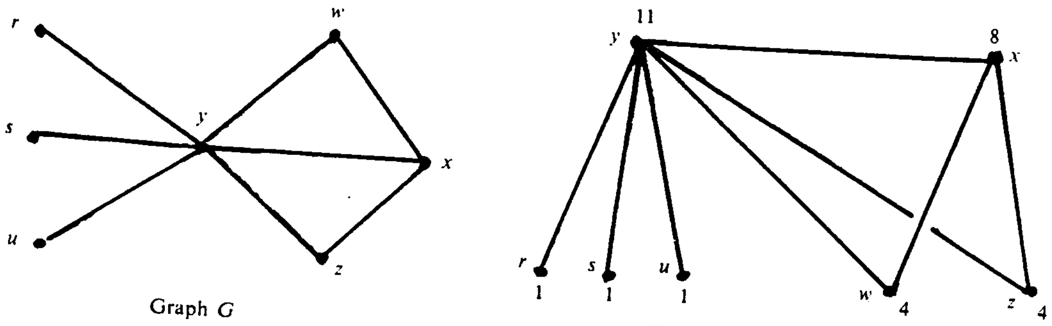
$$\begin{aligned} &\prod (d_j+1) - 1 + d_{k+1} \prod (d_j + 1) && (j = 1, \cdots, k) \\ &= (d_{k+1}+1) \prod (d_j+1) - 1 && (j = 1, \cdots, k) \\ &= \prod (d_j + 1) - 1, && (j = 1, \cdots, k+1) \end{aligned}$$

as desired.

Now we let $t = g_n$, and for each k we assign each element of C_k the label $g_n - g_{k-1}$. We must show that this gives a threhold labeling of G. Note the following:

(a) No two vertices of any two D_k are connected, since the labels in all the D_k total g_n.

(b) Any two vertices in any C_k are connected, since each such point has label at least $g_n - g_{n-1}$. This must exceed half of g_n since $g_n = (g_{n-1}+1)(d_n+1) - 1$ and d_n is at least 1.



Graph G

Graph G with threshold labelling

FIG. 2.

(c) To test if a vertex in D_k is connected to a vertex in C_j, note that their labels total $(g_{k-1}+1)+(g_n-g_{j-i})$. This exceeds t exactly if k exceeds j, as required.

This completes the proof of Theorem 2.3. □

It can be shown that this is the same labeling as that in [16]. We could then rely on the proof there that the resulting t is minimal. However, a distinctly different proof of minimality will follow from the results of § 4 below.

4. Process synchronization. Now we turn to the problem of managing asychronous processes. We are motivated primarily by the considerations in [9] and in [2]; the reader is referred to [2] for a more complete background and terminology. Suppose we have a number of processes, some of which conflict with each other (e.g., they demand access to the same resources that cannot be shared simultaneously by all processes wanting them). We connect this to the considerations above by supposing that each vertex of the graph G represents a process, and that two processes conflict precisely if there is an edge connecting those vertices.

For example, suppose there is a record in a file consisting of two fields, a name (the key) and an address. Suppose there are five transactions in the system, each wanting to locate the same record, and then carry out the following tasks:

Process A: Read the name (that is, locate the record and confirm that it exists, no further use of it).
Processes B and C: Read the address.
Process D: Change the address in the existing record.
Process E: Change the name (key) and address.

This is a slight generalization of a conventional readers-and-writers problem. Processes A, B, and C are "readers" and all can access the record at once and still produce consistent results. Process D can proceed simultaneously with Process A but not with B or C; Process E cannot proceed at the same time as any of the others. Drawing a graph with vertices A through E and drawing the appropriate edges yields the graph of Fig. 3(a); this graph is, in fact, a threshold graph, with a threshold labeling as shown in Fig. 3(b). This example is further expanded in § 5.

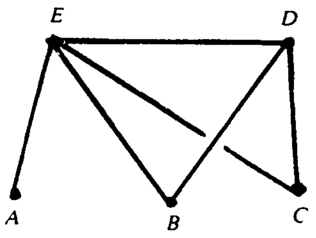


FIG. 3(a)

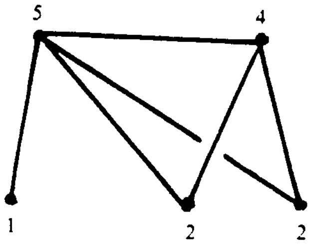


FIG. 3(b)

Suppose we are given a system of asynchronous processes and a set of pairs of these processes such that two processes in a pair cannot proceed simultaneously; i.e., members of each pair mutually exclude each other in the sense discussed in § 1 above and defined carefully in [2]. Each process has a piece of code called a "critical section"; we must provide entry and exit protocols for all processes in the system such that execution of these protocols will guarantee that two processes in a given pair will never be in their critical sections simultaneously. The graph G, with one vertex corresponding to each process and one edge for each mutually excluded pair of processes, will be called the **exclusion graph** of the system. (For other work with this graph, see [15].)

Our treatment is much weaker than that in [2] in that we do not consider lockout prevention or any sort of fairness condition (e.g., bounded waiting). However, the treatment in [2] assumes that only one process out of the N processes in the system can be in its critical section at once: that is, the exclusion graph is a clique. Here we deal with an exclusion graph that is a threshold graph, and any set of processes corresponding to a stable set of vertices in the graph can be in their critical sections at the same time.

For communication between the processes, we will assume that there are one or more shared variables $V_1, \cdots, V_k$, each of which can be accessed by more than one process, perhaps by all the processes. One thing we are seeking is bounds on the storage necessary for these variables. We denote the size of the set of values assumable by V_k by $|V_k|$; thus if V_k can assume values from 0 to N, $|V_k| = N+1$.

The processes access these shared variables only by specified operations called synchronization primitives (see [1], [2], [5], [6], [9], [11], [14] for some guides to this rather extensive literature). In this paper we will have occasion to use two distinct synchronization primitives: **PV**-chunk [9] and test-and-set [2], [14].

The syntax of the test-and-set operator, given in more detail in [2], may be summarized as follows: a test-and-set operator allows a process to test a shared variable until it reaches a fixed (set of) values and then perform certain actions, including resetting the shared variable to a value, which may be determined by the process using its knowledge of the shared variable value. The statement may be written as follows:

test V until $V = x_1$ or x_2 or $\cdots$ or x_n
then $V := \text{function}\,(V)$.

If V is not one of the indicated values then the statement is reexecuted from the beginning (busy waiting). As soon as V assumes one of the indicated values, function (V) is computed, V is set to the new value, and control passes to the next statement. (The computation and substitution is an atomic action; that is, if several processes are attempting to access V, only one at a time will actually change the value of V.) Note that $V := \text{function}\,(V)$ is an acceptable form of the test-and-set statement, since by implication it tests V first and sees that it has any one of its finitely many possible values. In each case, the function in function (V) is an arbitrary programmable function; it may take significant computation time or space. It is this feature that makes the general test-and-set operator both extremely powerful and difficult to implement efficiently.

A **PV-chunk operator** [9] can be implemented as a special case of a test-and-set operator but, being much more restricted, is usually written rather differently. Essentially, it restricts the test to testing for a certain one-sided inequality and the function to incrementing or decrementing by a (freely user-chosen) constant. The syntax we will use is the following:

Test V until $V \geqq c_1$ then $V := V - c_1$.

Note that c_1 can vary from one occurrence of this statement to another. V is initialized to some positive integer at the start and will never become negative; it can be increased by any process by executing the statement with c_1 negative, in which case the test condition is met automatically.

Variations of both test-and-set and **PV**-chunk can be specified in which a "failure" message is returned if the condition is not met, instead of busy waiting until it is met.

An operation very much like **PV**-chunk is available as a system call "SEMOP" in UNIX System V. The operation there increments or decrements a variable (or even several variables) by varying amounts as an atomic operation, provided that the change does not carry the variable(s) below 0 or above $2^{15}-1$. Thus, there is real motivation for finding methods that coordinate large time-shared or networked systems using **PV**-chunk which require a limited range for the shared variable(s). In addition, hardware experimenters, such as those designing the New York University Ultracomputer, have been experimenting with parallel hardware designed to perform N changes on a variable in less than time proportional to N (e.g., time proportional to $\log N$). The operations proposed seem not inconsistent in complexity with **PV**-chunk, but are surely less complex than the general test-and-set. Test-and-set requires that an individual process receive a value from the shared memory, compute, and return a value to shared memory; **PV**-chunk allows a value to be sent to special hardware serving the shared memory, which can do the calculation internally and needs to pass very little back to the calling process (in the model given here, the calling process sleeps until V is large enough, then V is decremented and the process awakens; in an alternative model, a single-bit message "fails" or "succeeds" and may be returned to the caller by the special hardware.

A principal result of [9] is that given a collection of processes and conflicts forming a threshold graph, conflict avoidance (mutual exclusion) can be achieved by using **PV**-chunk operations on a single shared variable with range 0 to t; each process can enter its critical section if and only if it can decrement V by an amount given by the label of its node in the graph. Further, the threshold graphs are precisely the graphs for which **PV**-chunk operations on a single shared variable will achieve conflict avoidance.

The results in [2] assume that all processes in the system are deterministic and impose a technical requirement "No Memory": each process knows nothing about the state of the system other than what is in the shared variable(s). That is, each time a process enters or leaves its critical section when the shared variable(s) have a particular value, it makes the same change in the shared variable(s). We do require this in the proof of Theorem 4.2 below.

Our main result in this section is that managing mutual exclusion in a system where the exclusion graph is a threshold graph, both test-and-set and **PV**-chunk operations require the same amount of shared memory and that is the amount determined by Theorem 3.3.

THEOREM 4.1 [9]. *Let a system of processes have the exclusion graph $G=(V,E)$, which is a threshold graph with separator t. Then there are entry and exit protocols that achieve the desired mutual exclusion by using* **PV***-chunk operations to access a single shared variable whose range includes the integers* 0 *to* t.

Proof [9]. It suffices to start the shared variable V with the value t. The entry protocol for a process whose corresponding vertex has label a is simply the following:

Test V until $V \geqq a$ then

$$V := V - a$$

and the exit protocol is simply

$$V := V + a$$

(as noted, the prefix "Test V until $V \geqq -a$" would add nothing). It is easy to see that a collection of processes can be in their critical sections at the same time if and only if their corresponding vertex labels total no more than t, i.e., if and only if there are no edges between their corresponding vertices. □

Clearly, the same theorem holds if the term **PV**-chunk is replaced by test-and-set, since **PV**-chunk is a special case of test-and-set. The main result of this section is that the range 0 to t in Theorem 4.1 is the best possible, even if we use several shared variables and the more general test-and-set operations.

THEOREM 4.2. *Let a system of processes have the exclusion graph $G=(V, E)$, which is a threshold graph with separator t as above. If there is a collection of entry and exit protocols that enforces the mutual exclusions in G, using test-and-set on a collection of shared variables $V_1, \cdots, V_k$ with $|V_j| = v_j$, then the number of different sets of values assumable by the V_j (and hence the product of the v_j) must be at least $t+1$.*

Proof. Suppose our synchronizing method stores adequate information in V_1 through V_k so that a process can determine from them if it can enter its critical section, and suppose the set of V_i assume no more than t values. We will obtain a contradiction.

Put G in normal form as in § 3, so that

$$t+1 = \prod (d_k+1), \qquad (k=1, \cdots, n).$$

We will select $t+1$ distinct collections R_p of vertices from the union of the D_k, $k=1, \cdots, n$, and arrange them in order such that we have the following:

(a) Any two R_p and R_q differ, but their intersections with each D_k have one containing or equal to the other.

(b) R_p and R_{p+1} differ by only one vertex.

We do this by a process suggestive of Gray codes for numbers of mixed base. First, order each D_k. For each sequence of integers $(a_1, a_2, \cdots, a_n)$ with $0 \leqq a_k \leqq d_k$, select a set consisting of the first a_k elements of D_k for each k. This yields $\prod (d_k+1) = t+1$ sets meeting the conditions of (a). We must now order them to obey condition (b). We do this by starting with the set determined by $(0, \cdots, 0)$, then going to $(1, 0, \cdots, 0)$, $(2, 0, \cdots, 0), \cdots, (d_1, 0, \cdots, 0)$. On "filling" each position change the next position by one and then step "down" through the previous cases: $(d_1, 1, \cdots, 0)$, $(d_1 - 1, 1, 0, \cdots, 0), \cdots, (1, 1, 0, \cdots, 0)$, $(0, 1, 0, \cdots, 0)$, and then $(0, 2, 0, \cdots, 0)$ and so on. The resulting list includes all $t+1$ sets and consecutive sets differ in just one element.

Start the system G with all processes outside their critical sections. Clearly, the first element of D_1 can enter its critical section; let it do so. Now go through the sequence of starts and stops (entries and exits of critical sections) dictated by the sequence R_p above: each element of D_1 starts (enters its critical section), the first element of D_2 starts, each element of D_1 stops (exits its critical section), and so on. Each step involves one element of a D_i entering or exiting its critical section. At each stage some change may be made in one or more of the V_k. There are $t+1$ stages (starting with no processes in critical sections and going through all the R_p). By hypothesis, the collection of V_k can assume only t distinct values, so there are two stages, R_p and R_q, of the above process when the V_k are in identical states. We must now get from this to a contradiction.

Suppose the set R_p of processes in critical sections is represented by the n-tuple $(a_1, \cdots, a_n)$ and the set R_q by $(b_1, \cdots, b_n)$. These must differ; without loss of generality suppose $b_j < a_j$ and $b_i = a_i$ for $i > j$. Now, one at a time, stop each process (if any) represented by b_{j+1} through b_n and the corresponding process in a_{j+1} through a_n. Note that in each case we make the same process in each group leave its critical section, and the V_i have the same values afterward. Next we stop elements of D_j, one at a time, in both sets of processes (the same element in each) and do this b_j times so that one set has no elements of D_j left in its critical section and the other has one or more still in critical sections; the values of the V_i are still the same. But now we have our contradiction; i.e., a process from C_j can enter its critical section with the remnant of the set R_q (where now no process in D_j or any later D_k is in its critical section) but cannot do so with the remnant of R_p, despite the fact that the V_k values are identical and no process in C_k can tell one situation from the other. This completes the proof of Theorem 4.2. □

Since this shows that $t+1$ values of shared variables are needed for a protocol using the test-and-set operation, it follows that at least $t+1$ values are needed using the more restrictive operation **PV**-chunk. From the fact that $\prod (d_k+1)$ values are needed we also obtain Corollary 4.3.

COROLLARY 4.3. *The value of t found in Theorem 3.3 is the smallest value of t allowing G to be labeled as a threshold graph.*

Proof. If the graph had a threshold labeling with a separator $s<t$, then by Theorem 4.1, mutual exclusion could be managed with a shared variable with $s+1$ distinct values. □

This is the minimality result of Orlin [16], proved in a quite different fashion. We also obtain a proof of a theorem on graph coverings [15].

COROLLARY 4.4. *Let G be a threshold graph and let $G_1, G_2, \cdots, G_n$ be subgraphs, which are threshold graphs and whose union covers G (i.e., includes all vertices and edges of G). Let G have separator t and let each G_k have separator t_k. Then $\prod (t_k+1) \geqq t+1$.*

Proof. The system of processes whose exclusion graph is G can have mutual exclusion enforced by enforcing it within each subgraph G_k. Each process will, in its entry protocol, attempt to decrement shared variables V_1 through V_n by the amount of its associated label in the respective G_k. While this approach is definitely lockout- and deadlock-prone, it does enforce the needed exclusions since every edge of G is in a G_k; hence the shared variables are capable of at least $t+1$ distinct values. (The tendency to deadlock can be overcome by having the set of **PV**-chunk operations be a single atomic action, as is the case in the UNIX System V implementation.)

We have now seen that for threshold graphs, $\prod (d_k+1)=t+1$ is a necessary and sufficient measure of the shared values needed to enforce mutual exclusion, both for **PV**-chunk and for test-and-set.

In the special case of a clique (all nodes connected), we have a threshold graph with (after adjustment for our normal form) one element in D_1, the rest in C_1, $t=1$, and a shared variable with two values is necessary and sufficient. This special case is Theorems 3.1 and 4.4 of [2].

5. Some examples. Section 4 shows that for processes whose conflict graph is a threshold graph, **PV**-chunk and test-and-set operations require similar storage to avoid conflicts. In Example 5.1 we show that when the conflict graph is not a threshold graph, test-and-set may require less storage.

Example 5.1. We consider a system S consisting of four processes A, B, C, and D (Fig. 4), such that each of A and C conflict with each of B and D. (This is the case $n=4$ of the famous dining philosophers problem. See, for instance, [10].) We can avoid conflict using test-and-set with one shared variable, values 0 through 3; to avoid

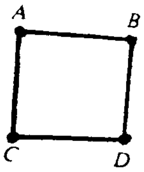


FIG. 4

conflict using **PV**-chunk requires at least two shared variables and four bits to store the shared variables.

We first give a solution for test-and-set. Let A and C each incorporate the entry protocol:

test V until $V=0$ or 1 then $V := 2 * V + 1$

(i.e., change 0 to 1 or change 1 to 3) and the following exit protocol:

$$V := (V-1)/2.$$

Let B and D have the following entry protocol:

test V until $V=0$ or 2 then $V := V/2+2$

(i.e., change 0 to 2 or change 2 to 3) and the following exit protocol:

$$V := 2 * (V-2).$$

The effect is that V is 0 if no process is in its critical section (or entry or exit protocols), 1 if A or C is in its critical section, 2 if B or D is, and 3 if both A and C or both B and D are in critical sections. This shows that four possible shared values (one variable, occupying two bits of storage) enable test-and-set to enforce mutual exclusion in this system.

It is harder to show how many values are needed to control this system using **PV**-chunk. We use a method developed more fully in [15]. Suppose there are shared variables $V_1, \cdots, V_n$ which control the system S. Each of A, B, C, and D, while executing its entry protocol to enter its critical section, changes one or more of $V_1, \cdots, V_n$; it fails to enter its critical section if some V_k cannot be sufficiently decremented at this time. Denote by a_k through d_k the decrements that A through D, respectively, apply to each V_k and by t_k the maximum permissible value of each V_k. Now for each k, the labels a_k through d_k and maximum t_k induce a graph on the four vertices (possibly with no edges) showing the processes prevented from running at once by that V_k; in [9] it has been shown that each graph induced in this fashion is a threshold graph. The square denoting the system S is the union of these graphs. Since the square is not itself a threshold graph, there must be at least two V_k's: **PV**-chunk operations cannot control S with just one shared variable.

Here is a simple solution using two shared variables, both initialized to 2 before the processes begin execution. Note that it does enforce the necessary mutual exclusion, is deadlock-free, but is not lockout-free and does not have bounded waiting.

A's Entry Protocol:	Test V_1 until $V_1 \geqq 2$ then $V_1 := V_1 - 2$;
A's Exit Protocol:	$V_1 := V_1 + 2$;
B's Entry Protocol:	Test V_1 until $V_1 \geqq 1$ then $V_1 := V_1 - 1$;
	Test V_2 until $V_2 \geqq 1$ then $V_2 := V_2 - 1$;
B's Exit Protocol:	$V_1 := V_1 + 1$;
	$V_2 := V_2 + 1$;
C's Entry Protocol:	Test V_2 until $V_2 \geqq 2$ then $V_2 := V_2 - 2$;
C's Exit Protocol:	$V_2 := V_2 + 2$;

D's Protocols are the same as B's.

In fact, the square can be a union of threshold graphs in very few ways: the threshold graphs that are subgraphs of the square are (i) the single edge, which we can take as having $t=1$ and each vertex labeled 1; and (ii) the union of two adjoining edges, which we can take as having the central vertex labeled 2, each end vertex 1, and $t = 2$. Thus the minimal sets of V_i associated with the square, up to some reasonable notion of equivalence, must be one of (i) V_1 and V_2, each with values 0 to 2, each controlling two edges; (ii) V_1 to V_4, each with values 0 to 1, each controlling one edge; or (iii) one variable with values 0 to 2, controlling two edges, and two variables with values 0 to 1, each controlling one edge. Since a variable with values 0 through 2 requires two bits to store the shared variables, this means at least two shared variables and at least four bits of shared memory are needed to control this system. This completes Example 5.1. Some arguments very similar in philosophy to those in the latter part of this example have been given in [7] and [8].

Example 5.2. Here we illustrate a natural way in which threshold graphs arise in accessing data base management systems. The reader-writer problem is a well-known problem concerning synchronization of accesses to a data base; I was motivated to think about it in this context by [17], and other references may be found therein. This example is an expansion on this problem in the spirit of the example at the start of § 4. In the standard problem, there are in a system several transactions wishing to access the same record: some (readers) want to read data from it without changing it; others (writers) want to change the data (we oversimplify here by not distinguishing update or read-and-write transactions from simple writes). Any number of readers can proceed at once; a writer cannot proceed unless it has exclusive use of the record (a lock on the record), since simultaneous writes might produce nonsense and a read overlapped with a write might return, e.g., a partially changed, not internally consistent, record.

Suppose a database record contains keys with several parts. For example, the fields in a record might be

```
1: OFFICE# 2: SALESMAN# 3: ACCOUNT# 4: AMOUNT
```

where the first three are keys, that is, jointly they uniquely identify the customer's record so we can find the amount due from the customer; this amount is stored in field 4, "AMOUNT." It is possible that one or more of the keys, as well as the amount due, might change because of, e.g., the customer moving to a region served by a different office, the salesman being replaced, or two customer firms merging. Thus, for each of the four fields, we can image a transaction to read the record as far as that field, or to write the record from that field onward.

For example, a transaction of type 2R (read fields to 2) would answer a query of the form "Does salesman 3057 serve any accounts at office 103?" and a 3W (write from field 3) would serve a transaction of the form "Open account number 566 for salesman 3057 in office 103." Note that, in fact, these two transactions could proceed at the same time, if it is understood that the second will fail if there is no salesman 3057 in office 103. The second transaction could not proceed simultaneously with a 2W ("delete all records for salesman 3057 in office 103") or a 4R ("what is the balance due from account 566, office 103, salesman 3057?").

Interestingly, a set of transactions consisting of transactions of types 1R, 1W, $\cdots$, 4R, 4W (and similarly for longer sets of multiple keys) has an exclusion graph that is a threshold graph. The transactions of type 1W, $\cdots$, 4W correspond to vertices in the sets $C_1, \cdots, C_4$, respectively, and those of type 1R, $\cdots$, 4R correspond to vertices in the sets $D_1, \cdots, D_4$ in the normal form of a threshold graph. (All writes conflict with each other; no reads conflict with each other; jR conflicts with kW if $j \geqq k$.)

Thus, if we have an upper bound on the number of transactions of each type which might appear in the system at one time, we could manage record locking—including the indicated partial record locking—with PV-chunk operations on a single shared variable. If we knew there would be at most four operations of each of the "read" types, the labels of the corresponding vertices would be 1 for 1R, 5 for 2R, 25 for 3R, 125 for 4R, and the separator would be $t=624$. Mutual exclusion could be managed using a single shared variable capable of assuming 625 distinct values.

Interestingly, changing the number of writers—the transactions that appear to require the most extensive locks—does not change these numbers; to reduce t we must reduce the number of readers, not the number of writers. This is consistent in a broad sense with the observations in [17] where the added memory to avoid delays is determined by the number of potential readers (there, only one writer is considered). By contrast, changing reader transactions from one subtype to another does change t: if there were at most two transactions of types 1R and 2R, and at most six each of types 3R and 4R, then we would have

$$t=(2+1)(2+1)(6+1)(6+1)-1=440.$$

6. Conclusions and additional problems. Earlier papers such as [2] present a careful analysis of the amount of shared memory required to solve the problem of mutual exclusion when all processes exclude all others. In real applications, it may be possible for some sets of processes, but not others, to enter critical sections simultaneously. A major step in this direction appears in [12], which bounds the delays occurring in the mutual exclusion algorithm by imposing a "locality" condition: particular processes are constrained to share resources only with a limited number of other "nearby" processes. It would be desirable to have efficient solutions, and bounds on possible solutions, for other more general cases. This paper considers systems of asynchronous parallel processes in which the desired mutual exclusions can be modeled by a threshold graph. These differ strongly from the cases concentrated on in [12], since threshold graphs always have a vertex that is adjacent to all other (nonisolated) vertices. In our limited case, simple mutual exclusion (without lockout prevention or other desirable features) can be managed by a single **PV**-chunk operation in the entry protocol preceding the critical section in each process, using a single shared variable with range from 0 to t, with t denoting the minimal separator of the corresponding threshold graph.

Establishing this requires giving a new algorithm for the minimal separator previously calculated by Orlin, and allowing the separator to be written as a product formula in terms of the numbers of vertices in certain classes in the graph (this could also be expressed in terms of vertex degrees or the size of maximal cliques; see also [15]).

For controlling mutual exclusion in this limited case, **PV**-chunk requires no more shared memory than the Lynch–Fischer test-and-set operation. An example suggests that for more general graphs, test-and-set will manage exclusion with fewer shared variables and fewer bits of shared variables than **PV**-chunk. A final example suggests that **PV**-chunk may, in fact, provide an efficient tool for certain kinds of partial-record locking and certain kinds of reader-writer management problems in data base systems.

This leaves a great many open problems. What are efficient ways of managing mutual exclusion situations modeled by more complex graphs than threshold graphs, or indeed, not modeled by graphs at all? (Surely the different form of graph models provided in [12] carry different information than the models here.) Can we find algorithms that incorporate lockout prevention or fairness as well as mutual exclusion, while still using **PV**-chunk operations or other operations that seem easier to implement in efficient hardware than the general test-and-set operation? Does test-and-set solve these other problems with significantly less memory, or significantly faster algorithms, than simpler synchronization primitives? A referee suggests that in some sense Theorem 4.2 does not appear to depend on the synchronization primitive used. The number

$t+1$ is in some sense a measure of the space needed to synchronize the processes represented in the graph, without regard to exact method. Can this be formalized? In Example 5.1, it appears that test-and-set needs less memory than **PV**-chunk; how can this be measured more systematically? Finally, is it connected to Lipton's concept of using one primitive to "simulate" another [11]?

Acknowledgments. The author had numerous helpful discussions with Y. Zalcstein. The referees suggested extensive revisions to improve clarity of the presentation; the remaining awkwardness is due to the author.

REFERENCES

[1] T. BLOOM, *Evaluating synchronization mechanisms*, in Proc. 7th Annual ACM Symposium on Operating Systems Principles, (ACM SIGOPS), Pacific Grove, CA, 1979, pp. 24-32.

[2] J. E. BURNS, P. JACKSON, N. A. LYNCH, M. J. FISCHER, AND G. L. PETERSON, *Data requirements for implementation of N-process mutual exclusion using a single shared variable*, J. Assoc. Comput. Mach., 29 (1982), pp. 183-205.

[3] V. CHVATAL AND P. HAMMER, *Aggregation of inequalities integer programming*, Ann. Disc. Math., 1 (1977), pp. 145-162.

[4] A. CREMERS AND T. HIBBARD, *Mutual exclusion of N processors using an O(N)-valued message variable*, in Lecture Notes in Computer Science 62, Springer-Verlag, Berlin, New York, 1978, pp. 165-176.

[5] E. W. DIJKSTRA, *Solution of a problem in concurrent programming control*, Comm. ACM, 8 (1965), p. 569.

[6] ———, *Cooperating sequential processes*, in Programming Languages, F. Genuys, ed., Academic Press, New York, 1968.

[7] K. ECKER AND S. ZAKS, *On a graph labeling problem*, Bericht Nr. 99, Gesellschaft für Mathematik und Datenverarbeitung MBH Bonn, Federal Republic of Germany, 1977.

[8] P. ERDÖS, A. W. GOODMAN, AND L. POSA, *The representation of a graph by set intersections*, Canad. J. Math., 18 (1967), pp. 106-112.

[9] P. B. HENDERSON AND Y. ZALCSTEIN, *A graph-theoretic characterization of the PV-chunk class of synchronization primitives*, SIAM J. Comput., 6 (1977), pp. 88-108.

[10] D. LEHMAN AND M. O. RABIN, *On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem*, in Proc. 8th Annual ACM Symposium on Principles of Programming Languages, 1981, pp. 133-138.

[11] R. LIPTON, L. SNYDER, AND Y. ZALCSTEIN, *A comparative study of models of parallel computation*, in Proc. 15th Annual Symposium on Switching and Automata Theory, New Orleans, LA, 1974, pp. 145-155.

[12] N. A. LYNCH, *Upper bounds for static resource allocation in a distributed system*, J. Computer System Sci., 23 (1981), pp. 254-278.

[13] M. C. GOLUMBIC, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[14] N. LYNCH AND M. FISCHER, *On describing the behavior and implementation of distributed systems*, Theoret. Comput. Sci., 13 (1981), pp. 17-43.

[15] E. T. ORDMAN, *Threshold coverings and resource allocation*, in Proc. 16th Southeastern International Conference on Graph Theory, Combinatorics, and Computing, Congr. Numer., 49 (1985), pp. 99-113.

[16] J. ORLIN, *The minimal integral separator of a threshold graph*, Ann. Discrete Math., 1 (1977), pp. 415-419.

[17] G. L. PETERSON, *Concurrent reading while writing*, ACM Trans. Programming Lang. Syst., 1 (1983), pp. 46-55.