+Supported in part by U.S. Army Research Office under Grant No. DAHC04-72-0001.

++Supported in part by Office of Naval Research under contract N00014-75-C-0752.

On the Complexity of Resource Managers

by

Richard J. Lipton[+] and Lawrence Snyder[++]

Research Report #64

Abstract

When requests for resources cannot be granted they must be placed into "wait queues." The complexity of managing these pending requests and servicing them when resources become available is investigated. Both upper and lower bounds are obtained.

1. *Introduction*

Within an operating system a common decision is: should a particular request for a resource (real or abstract) be granted? The criterion by which one decides whether or not to grant a request has been carefully studied: Knuth [1] studies, for example, the question of requests for memory, while Habermann [2] studies the question of requests for an abstract resource (represented as a vector.) However, when the request cannot be granted what happens? This is the central question of this paper.

More exactly let us imagine a stream of requests arriving one at a time to some resource manager. If a request can be granted, then his job is simple: he just grants the request. On the other hand, if the request cannot be granted he has two basic strategies. First, he can "forget" about the ungranted request and begin to process the next one. This has the consequence of forcing "busy waiting" [3]. If some request is not granted, then the process making that request is forced to repeatedly make its request. As is well known, this is wasteful of processor time. Second, he can "remember" the ungranted request and then go on to process the next one. By remember we mean that the manager will in the future "release" this request, i.e. he will grant it as soon as the request can be satisfied.

A resource manager has then two main tasks: updating and selecting.

When a request cannot be granted he must update his state in the correct way. Later on he must be able to find what requests can now be satisfied quickly. The complexity of a resource manager must, therefore, include both the cost of updating his state and the cost of selecting the next request to be granted. For example, if a resource manager simply keeps a linear list of those ungranted requests, then updating is trivial. On the other hand, selecting would require a complete scan of the linear list. Our interest is in improving upon this naive solution and in evaluating the amount of additional improvement that might be expected.

This paper is organized into four sections. In section 2 the basic definitions of resource managers are given. In section 3 upper bounds, i.e. algorithms, are given for these resource managers. In section 4 lower bounds for these algorithms are given. These lower bounds show that our algorithms are optimal - at least for a certain reasonable class of algorithms.

2. *Definitions*

Suppose we have a set of processes making requests for resources such that at any point in the execution, at most n requests are pending due to the unavailability of sufficient resources to satisfy any of the requests. It is assumed that all pending requests are "reasonable" in the sense that if the resources of the system were not currently in use, any of the pending requests could be satisfied. Thus, the pending requests are given by a set P, of ordered pairs,

$$P = \{ \langle t_1, q_1 \rangle, \ldots, \langle t_n, q_n \rangle \}$$

where t_i is the time (monotonically increasing) of a request for a quantum q_i of the resource. In addition, we suppose that a quantum, Q, has just been released by one or more of the processes. The general task, then, will be to choose a $<t_i, q_i>$ from P such that $q_i \leq Q$. The way in which the choice is made depends upon the type of resource and the service policy.

We recognize three classes of resource requests.

Definition 2.1: A request is a time-space/fixed-unit (TSF) request if

$$q_i \in S = \{s_1, \ldots, s_r\}, \quad 1 \leq i \leq n.$$

TSF requests can be thought to represent memory requests in a system where space is allocated in one of r fixed size blocks given by the set S. Alternatively, the requests can be considered to be pending processes waiting on a semaphore where the synchronization primitive is of the "PV chunk" [4] variety and the "chunks" are known, *a priori*, to be of sizes $s_1, \ldots, s_r$. When the "chunks" are not known *a priori* or the memory is allocated in units of arbitrary size, we have the TSV type of resource request.

Definition 2.2: A request is a time-space/variable-unit (TSV) request if the $q_i \in \mathbb{N}$, the natural numbers, $1 \leq i \leq n$.

A third type of resource request is motivated by the PV multiple type of synchronization primitive [4] as well as various types of binary resource assignment.

Definition 2.3: A request is a multiple binary attribute (MBA) request if

$$q_i \in \mathbb{B} = \{0,1\}^r, \quad 1 \leq i \leq n.$$

Thus, MBA requests are r-bit binary numbers with each bit position representing a specific semaphore or resource. Of course, a request q_i is satisfied by the release, Q, (i.e. $q_i \leq Q$) provided Q covers[†] q_i.

The two service policies of interest are first fit (FF), grant $(t_i,q_i) \in P$ with $q_i \leq Q$ and with minimum t_i and best fit (BF) grant $(t_i,q_i) \in P$ with $q_i \leq Q$ and maximal possible q_i (maximal for $q_i \in \mathbb{N}$ translates to largest, for $q_i \in \{0,1\}^r$ it translates to with most number of 1's)[††] where for MBA requests, a maximal q_i has the greatest number of positions set.

3. *Upper Bounds*

The question to be investigated in this section is: for each type of resource request and each service policy, how rapidly can an element from P be chosen? The complexity criterion will be execution time as a function of the number n of pending requests. Since the management process is assumed to be on-going, there are as already stated, two parts to the complexity question:

insertion - the time required to add a new request $<t,q>$ to the pending requests data structure

searching - the time required to find the appropriate request, given a release of quantum Q, and to delete the request from the data structure.

The worst case bounds per request for insertion and searching are summarized in tables 1 and 2, where c means time is bounded by some constant independent

---

† x covers y if whenever x_i is 0 it follows that y_i is 0.

†† Break ties here by using FF.

of n.

	TSF	TSV	MBA
BF	c	$0^{\dagger}(\log n)$	c
FF	c	$0(\log n)$	c

Table 1. Upper bounds on insertion, per request

	TSF	TSV	MBA
BF	c	$0(\log n)$	c
FF	c	$0(\log n)$	c

Table 2. Upper bounds on searching, per request

The following informal descriptions of the algorithms should be sufficient to establish the validity of the results in tables 1 and 2.

TSF-FF: Recall that there are r quanta. Keep r queues of requests in ascending time order within each queue. Insertion requires appending to the appropriate queue. Searching requires finding the minimum among the heads of those queues containing requests for quanta less than or equal to the release Q. The constant is clearly dependent upon r.

TSF-BF: Same organization as for FF. Searching generally requires only the fetch of the head element of the appropriate queue, although in the worst case (queue for $s_2, \ldots, s_r$ empty, $Q = s_r$), reference to empty queues will cause the constant to depend upon r. A linked list organization appears to avoid reference to empty queues at the expense of causing

† $O(f(n))$ denotes some function $g(n)$ with $g(n) \leq Af(n)$ for some constant A.

the insertion constant to depend upon r.

The following result for TSV are our main results:

TSV-FF: Keep the pending requests in an AVL [5] tree such that the symmetric order is the ascending time order of the requests and the root node of any subtree contains an auxiliary cell indicating the minimum quantum request for any request in the left subtree of the root. Insertion into AVL trees is $O(\log n)$. Given a release Q, the search proceeds as usual for AVL trees with the branch decision based first on whether the left subtree, then the root and finally (the default) the right subtree satisfies the request. Search and deletion times are both $O(\log n)$ [5].

TSV-BF: Keep the pending requests in an AVL tree such that the symmetric order of the tree is the quantum size of the requests in ascending order, breaking ties with ascending time order. The search proceeds in the normal manner.

Remark: The TSV problem is the dual of the FF and BF space allocation problems where the q_i are interpreted as available free space and the release Q is interpreted as the size of the allocation request. Since the allocation problems can be solved efficiently (in worst case) using AVL trees, the observation of duality leads to a symmetric solution to the combined problem: suppose there are two AVL trees, a free space tree (fs) of unallocated memory blocks and a pending request tree (pr) of unsatisfied requests (none of which fit in the currently available free space). Then figure 1 shows a flow chart that, given an allocation request, satisfies it from fs (if possible) or inserts it

in pr or, alternatively, given a release, services a pending request from pr (if possible) or inserts it into fs.

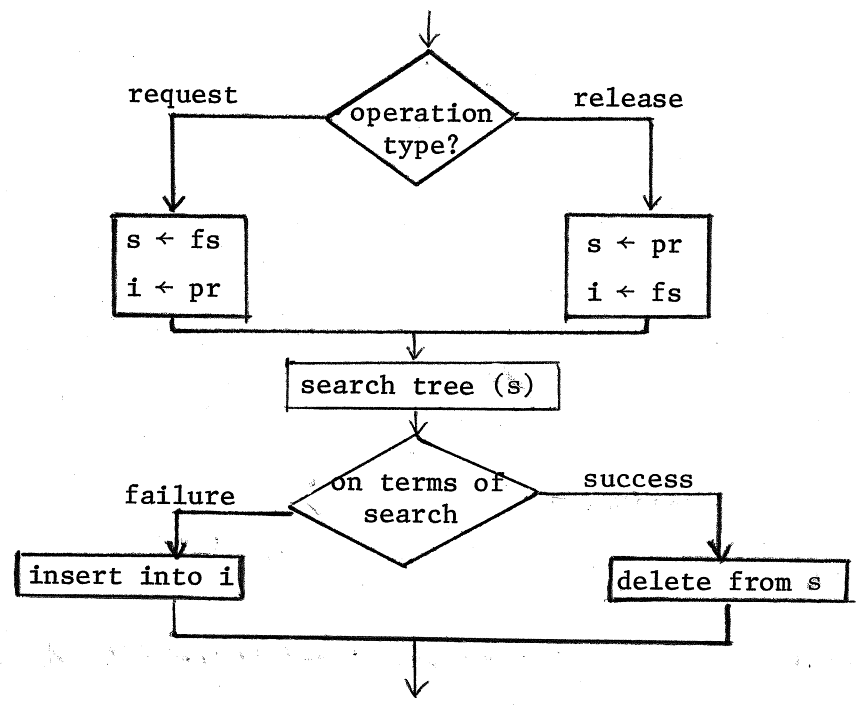

Figure 1. Duality of the allocation and management problems

MBA-FF: Keep the pending requests in $2^r - 1$ linked list queues, one for each nonzero bit configuration. Insertion is constant. Let U be the currently unused positions and Q the release $(U \wedge Q \equiv 0)$. Searching is performed by finding the minimum time among all heads of the queue for $U \vee Q$ together with those queues found by removing one or more bits from $U \vee Q$. Search time is constant, with respect to the number of pending requests although it is proportional to 2^r.

MBA-BF: Keep the pending requests in 2^r-1 linked list queues, one for each nonzero bit configuration. Searching begins with queue $U \vee Q$. If

it is empty, then all queue heads defined by $U \vee Q$ with one bit removed are tested (ties are broken by time order). If these queues are all empty, consider all queue heads with two bits removed, etc. The search time is independent of the number of pending requests, but as before, is related to 2^r.

Before proceeding further, it is well to analyze just how practical the various algorithms are in the context of criteria other than the number of pending requests. The TSF algorithms for small r use storage efficiently and their execution times are bounded by small constants. Of course, as r becomes large, the situation approaches that of TSV and at some point it may be preferred.

The TSV algorithms employ AVL trees and, although their asymptotic behavior is $O(\log n)$, the difficulty in performing AVL manipulations suggests that alternatives might be sought. Queues and stacks are more easily manipulated structures and it may be that algorithms based on these data structures can be found which do beat the AVL implementations for all practical cases. But, as the next section will show, the asymptotic behavior of $O(\log n)$ will not be improved upon!

Similarly, the MBA requires the maintenance of a large number, 2^r, of queues; an unreasonable assumption for even modest values of r. Once again we shall see that substantial improvement along these lines will not be possible.

4. *Lower Bounds*

In order to study the complexity of implementing resource managers, it will be necessary to have a very general view of what a resource usage is. We shall characterize resource usage by the way in which it reorders requests. More precisely, let

$$1 \ldots m \underset{R}{\rightarrow} \pi_1 \ldots \pi_m$$

represent m resource events (requests and releases) for resource type R in the time order in which they arrive at the manager and the resulting order in which they are serviced, $\pi_1 \ldots \pi_m$. For example

$$1\ 2\ 3\ 4\ 5\ 6 \underset{\text{TSV}}{\rightarrow} 3\ 2\ 5\ 1\ 6\ 4$$

is a possible behavior for a TSV resource. We interpret the activity as

event 1: take 2 units
event 2: take 1 unit
event 3: release 1 unit
event 4: take 3 units
event 5: release 2 units
event 6: release 3 units

and the space available is initially zero. It is only necessary to study those behaviors $\pi_1 \ldots \pi_m$ that can actually occur, in which case we say R realizes $\pi_1 \ldots \pi_m$*.

---

* We assume a FF policy in this section. A similar analysis could be carried out for BF.

The permutation model of resource requests allows a simple and compact way of defining and studying the complexity of various resource managers. The basic theme - to be developed in the remainder of this section - is that the complexity of a resource manager is directly proportional to the number of permutations that the resource type can realize. Thus, in order to determine the complexity required for the management of a new resource, one need only determine the number of permutations that resource type can realize.

Note also that the permutation model describes the "global" behavior of resource usage and thus both pending requests and release are being considered. To identify the pending requests, we have

Definition: Let $\pi_1 \dots \pi_m$ be a permutation. Then π_i is pending provided for some $j < i$, $\pi_i < \pi_j$.

Intuitively, of course, π_i is pending if at some point in the system it is saved and subsequently $(\pi_j > \pi_i)$ released by some π_j. For example, in $\pi = 3\ 2\ 5\ 1\ 6\ 4$ the pending requests are π_2, π_4 and π_6.

Definition: Suppose $\pi = \pi_1 \dots \pi_m$ is a permutation. Then π is simple provided no two adjacent π_i and π_{i+1} are both pending.

A permutation can fail to be a simple permutation in essentially two ways. First, a release could be "large" and release more than one pending request. For example,

$$3\ 1\ 2$$

where

event 1: take 1
event 2: take 2
event 3: release 3

Secondly, there can be a "cascading" effect in the case where resources can be both seized and released. Thus

3 1 2

could be interpreted as

event 1: take 1 unit of A return 1 unit of B
event 2: take 1 unit of B
event 3: return 1 unit of A

where A and B are initially zero.* It is interesting to note that there are two interpretations for 3 1 2 while 3 2 1 has only one, cascade, because of the FIFO assumption on time.

The restriction to simple permutations is made to avoid such subtleties. The next lemma indicates, however, that not too much is lost by this restriction.

Lemma: There are at least $(\lfloor m/2 \rfloor)!$ simple permutations of $1 \ldots m$.

Proof: Let $t = \lfloor m/2 \rfloor$ ($\lfloor x \rfloor$ = greatest integer $\leq x$.) Consider the permutation

$$\pi = t+1\ x_1\ t+2\ x_2 \ldots t+t\ x_t$$

where $x_1 \ldots x_t$ is some permutation of $1 \ldots t$. Clearly, only the x_i are pending; hence π is simple. Moreover, there are $t!$ such permutations. □

---

* We have not previously considered this type of resource (which corresponds to a Vector Replacement System [6]).

The construction of the lemma actually represents the basic situation of section 3 where the $x_1 \ldots x_t$ are pending (and have been inserted in the data structure) and then the releases $t+1 \ldots t+t$ cause the corresponding x_i to be found in the search. The next theorem states that any permutation of the pending elements can be realized.

Theorem 1: A TSV resource can realize any simple permutation.

Thus, any TSV can realize at least $(\lfloor m/2 \rfloor)!$ permutations of $1 \ldots m$.

Proof: Let π be a simple permutation. Let $r_1 \ldots r_k$ be the places where pending requests get released. By definition there are releases at places $s_1 \ldots s_k$. Now let r_i be "take i units" and s_i be "release i units." When there are no units initially available, it is clear that this is a TSV behavior and thus π can be realized by a TSV.

It is interesting to note that the "power" of TSV resources come from the ability to vary the amounts. Considering TSF, it is easily seen that it can only realize 2^{cm} permutations. The key to this observation is the fact that there are only A possible distinct types of requests (A is some constant). Thus, there can be at most A^m realized permutations of $1 \ldots m$.

We now proceed to the question raised at the end of section 3, that is, can a manager of a TSV resource implement the possible TSV behaviors by manipulating a fixed structure of stacks, queues and (for completeness) deques? Such a strategy is a common - if not the standard - way to manage resources. Pending requests are stored in these data structures until they are chosen for service.

The problem is easily answered if we view the manager as a <u>switchyard</u> in the sense of Tarjan [7]. A switchyard is an acyclic directed graph with a unique source (the manager's input events) and a unique sink (the manager's service permutation). Each vertex is either a stack, queue or deque. A switchyard can realize $\pi = \pi_1 \ldots \pi_m$ provided $1 \ldots m$ are placed in the source and after some sequence of "moves" $\pi_1 \ldots \pi_m$ appears at the sink. A "move" is the placement of an element from one vertex (data structure) to another vertex by movement along an edge. Tarjan then argues that there are at most

$$(4e)^{vm}$$

possible move sequences where e is the number of edges, v the number of vertices (data structures) and m is the length of the input. This leads us to conclude

<u>Theorem 2</u>: No "switchyard" manager can manage a TSV resource for all arbitrarily large input sequences.

<u>Proof</u>: By theorem 1 a TSV resource can realize any simple permutation and from the lemma there are at least $(\lfloor m/2 \rfloor)!$ such permutations. By Tarjan's result there are at most $(4e)^{vm}$ move sequences possible for the manager. However, for large m, $(\lfloor m/2 \rfloor)! > (4e)^{vm}$ and thus some TSV behaviors cannot be realized. □

The theorem appears to imply that the simple mechanisms of queues etc. are at fault. This is not the case as we now argue. Let us hypothesize that the manager is given the entire sequence of input elements and then produces the entire output permutation. This is strongly suggestive of sorting, and,

indeed, an argument from sorting theory [5] implies that $0(t \log t)$ time will be required by the manager where $t = (\lfloor m/2 \rfloor)!$ To see this, we assume that at each "step" the manager makes a binary decision based on the input. These decisions can be abstracted by a binary tree with the output permutations at the leaves. The maximum depth of the tree indicates the worst case number of steps required to arrive at the appropriate output permutation. By the lemma and theorem 1, a TSV resource must correspond to a tree with at least $(\lfloor m/2 \rfloor)!$ leaves. But such a binary tree will have depth $t \log t$. Hence

Theorem 3: Any manager for a TSV resource requires $0(t \log t)$ time to manage m input events, where $t = (\lfloor m/2 \rfloor)$.

Turning our attention to MBA resources we observe the following:

Theorem 4: An MBA resource can realize at least $\binom{r}{\lfloor r/2 \rfloor}!$ simple permutations of $1 \ldots 2\binom{r}{\lfloor r/2 \rfloor}$, where r is the number of attributes.

Proof: Let $t = \binom{r}{\lfloor r/2 \rfloor}$, the binomial coefficient, and consider the permutation

$$\pi = t+1 \; x_1 \; t+2 \; x_2 \ldots t+t \; x_t$$

where $x_1 \ldots x_t$ is a permutation of $1 \ldots t$. There are $t!$ such permutations and π is simple. Suppose all attributes are 0 initially. Let w_i, $1 \le i \le t$ be the t r-bit attribute sequences with $\lfloor r/2 \rfloor$ ones which are incomparable, i.e. $i \neq j$ implies w_i does not cover w_j and w_j does not cover w_i. Now, each x_i will request resources w_i and each

$m+1$ will release w_i. It follows from the incomparability of the w_i that π can be realized by an MBA resource. □

Consequently, unless r is quite small, the complexity of servicing MBA requests is quite high, even though it may be independent of n. Analysis similar to that of theorems 2 and 3 can be carried out to establish that management of MBA systems is, indeed, quite difficult. The conclusion to be drawn is that systems providing MBA resources should limit the number of attributes severely if reasonably efficient management of these resources is to be expected.

References

[1] D.E. Knuth. *The Art of Computer Programming*, Vol. 1. Addison Wesley: Reading, Mass., 1968.

[2] A.N. Habermann. Synchronization of Communicating Processes. CACM 15(3), 1972.

[3] S.E. Madnick and J.J. Donovan. *Operating Systems*. McGraw-Hill: New York, 1974.

[4] R.J. Lipton, L. Snyder and Y. Zalcstein. A Comparative Study of Models of Parallel Computation. Proc. of the 15th annual IEEE Symposium on Switching and Automata Theory, 1974.

[5] D.E. Knuth. *The Art of Computer Programming*, Vol. III. Addison Wesley: Reading, Mass., 1973.

[6] R.M. Keller. On maximally parallel schemata. Proc. of the 11th annual IEEE Symposium on Switching and Automata Theory, 1974.

[7] R. Tarjan. Sorting Using Networks of Queues and Stacks. JACM 19(2), 1972.