

Acknowledgments. Many people have contributed to COKO III. A particularly large contribution was made by Ed Hafer while an undergraduate at UCD. He wrote a graphic display interactive program, with debugging aids, supplied a special FORTRAN debugging package, and helped to improve the literary content of the chess program itself. We also wish to thank Ross Brown of the University of California at Davis and John McCarthy of the Stanford Artificial Intelligence Laboratory for allowing use of their computer facilities.

Received September 1970, revised June 1972

References

1. Scott, J.J. A chess playing program. *Machine Intelligence IV*. American Elsevier, New York, 1969.
2. Botvinnik, M.M. *Computers Chess and Long-Range Planning*. Springer-Verlag, New York, 1969.
3. Cooper, Dennis W. Heuristic tree searching in the game of chess. Working paper, 1969.
4. Kozdrowicki, E.W. A practical application of machine learning: use of learning in an interpreter for a tree searching language. *Proc. IEEE Systems Science & Cybernetics Conf.*, San Francisco, Oct. 1968, pp. 250-257.
5. Kozdrowicki, E.W. An adaptive tree pruning system: a language for programming heuristic tree searches. *Proc. ACM 23rd Nat. Conf.*, 1968.
6. Huberman, Barbara J. A program to play chess end games. Ph.D. Thesis, Stanford U., Stanford (also Technical Report CS106-August) 1968.
7. Baylor, George W. and Simon, Herbert A. A chess mating combinations program. *Proc. AFIPS 1966 SJCC*, AFIPS Press, Montvale, N.J., pp. 431-447.
8. Samuel, A.L. Some studies in machine learning using the game of checkers. II-Recent progress. *IBM J.* (Nov. 1967).
9. Reinfeld, Fred. *Improving Your Chess*. Barnes & Noble, New York, 1955.
10. De Groot, Adriaan. *Thought and Choice in Chess*, Basic Books, New York, 1966.
11. Greenblatt, R.D., Eastlake, D.E., and Crocker, S.D. The Greenblatt chess program, *Proc. AFIPS 1967 FJCC*, Vol. 31, AFIPS Press, Montvale, N.J., pp. 801-810.
12. Gardner, Martin. Mathematical games. *Sci. Am.* 206, (Mar. 1962) 138-144.
13. Shannon, C.E. Programming a digital computer for playing chess. *Phil. Mag.* 41, (Mar. 1950) 256-275.
14. Feigenbaum, Edward A., and Feldman, Julian, Eds., *Computers and Thought*, McGraw-Hill, New York, 1963.
15. Good, I.J. A five-year plan for automatic chess. In *Machine Intelligence 2*, Oliver and Boyd, Edinburgh, 1967.
16. Bell, A.G. How to program a computer to play legal chess. *The Computer J.* 13, 2 (May 1970).
17. Fine, Reuben. *The Psychology of the Chess Player*. Dover, New York, 1967.
18. Levy, D.N.L. Computer chess—a case study on the CDC 6600. *Machine Intelligence VI*. American Elsevier, New York, 1971.
19. Slagle, James R. *Artificial Intelligence—The Heuristic Programming Approach*. McGraw-Hill, New York, 1971.
20. Nilsson, Niels J. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, 1971.
21. Bellman, R. On the application of dynamic programming to the determination of optimal play in chess and checkers. *Proc. National Academy of Sciences*, 53 (1965).
22. Bellman, R. Stratification and control of large systems with application to chess and checkers. *Information Sciences*, 1 (Dec. 1968).
23. Slagle, James, and Bursky, Philip. Experiments with a multi-purpose, theorem-proving heuristic program. *J. ACM*, 15, 1 (Jan. 1968) 85-99.
24. Slagle, James, and Koniver, Deena. Finding resolution proofs and using duplicate goals in AND-OR trees. Heuristics Lab., Div. of Computer Research and Technology, National Institutes of Health, Bethesda, Md., 1970.

Operating
Systems

C. Weissman
Editor

Mixed Solutions for the Deadlock Problem

John H. Howard Jr.
University of Texas

Mixtures of detection, avoidance, and prevention provide more effective and practical solutions to the deadlock problem than any one of these alone. The individual techniques can be tailored for subproblems of resource allocation and still operate together to prevent deadlocks. This paper presents a method, based on the concept of the hierarchical operating system, for constructing appropriate mixtures and suggests appropriate subsystems for the most frequently occurring resource allocation problems

Key Words and Phrases: deadlocks, resource allocation, operating systems, multiprogramming, hierarchical systems

CR Categories: 4.30, 4.32

Introduction

A deadlock occurs in an operating system when each of several processes holds some of the system's resources and cannot proceed until it gets resources already assigned to the others. For example, there might be two processes, each holding half the system's memory and each needing to use two-thirds. Even if the processes are individually correct and would run to completion in the absence of the others, the combina-

Copyright © 1973, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This research was supported in part by NSF Grant GJ-1084. Author's address: Department of Computer Sciences, University of Texas, Austin, TX 78712.

Communications
of
the ACM

July 1973
Volume 16
Number 7

tion of processes creates a permanent stoppage of system processing.

Current work on the deadlock problem has been ably surveyed by Coffman, Elphick, and Shoshani [1]. Adopting their terminology, we have three basic approaches: detection, avoidance, and prevention. None of these alone is appropriate for the entire spectrum of resource allocation problems encountered in operating systems. This paper presents a method for combining the basic approaches and thus allowing the selection of the optimal one for each class of resources in a system. The method is based on the hierarchical structure of operating systems, which manifests itself in resource ordering.

The discussion is organized as follows. After an initial section defining terms and stating assumptions, the three basic approaches and their limitations are discussed. Then the mixed technique is defined, an informal proof of validity is given, and finally an example of an appropriate mixture is presented.

Definitions and Assumptions

In considering the deadlock problem we are interested only in the allocation of resources to processes. Thus we ignore much programming detail and consider a process to be simply a sequence of requests and releases of resources. Processes are dynamic in the sense that one process may create another. Processes may intercommunicate, using messages which are themselves resources created by the sending processes and consumed by the receiving processes. In this case a process which is expected to create a message must be treated as if it were holding the resource.

It is important to note that the *only* reason a process is blocked is that it is waiting for a resource. This assumption is made to separate deadlock considerations from other system errors such as infinite loops in an individual process. It is assumed that if a process is given its resources then it will finish and release them in a finite number of steps. System stoppages due to the failure of an individual process are not considered to be deadlocks. Potentially faulty processes such as user jobs may be dealt with by the imposition of time limits and forced termination.

Some resources, such as main memory, come in groups of functionally identical units. The transactions (requests and releases) for these resources must specify a quantity. It is assumed that no process requests a larger total quantity than exists in the system.

Finally, it should be noted that even in the absence of a deadlock, some process may be blocked for an indefinitely long time. For example, if priority scheduling is used, then a low-priority process may be permanently locked out by an oversupply of high-priority processes competing for the same resource [6]. This is a fundamental characteristic of priority scheduling and is not a deadlock.

Preemption

Properly speaking, preemption is only part of a solution to the deadlock problem. Since it occurs in both detection and prevention strategies, it merits separate discussion. Preemption is the capture of a resource from a process without action by the process. Since the process presumably still needs the resource, it cannot continue. Rather it must be abandoned, restarted from the beginning, or forced to rerequest and thus wait for the preempted resource.

Both abandonment and restarting imply the loss of the time spent by the process before the preemption occurred. Thus these forms of preemption are inherently wasteful, and can be used only when they are not expected to occur often. A more severe problem is that the preempted resource may be a memory device containing valuable data which has been partially rewritten. Abandonment implies the loss of such data, and restarting implies the necessity for reconstructing the original memory state, which may be extremely costly if possible at all.

Resumption, on the other hand, does not lose processing time already expended. If the preempted resource has memory, then its contents must be protected from alteration until the process recovers the resource. This is usually done by saving the contents in some backup storage (another resource to be included in the deadlock analysis) and reloading them before continuing the process. Typical examples are: (1) swapping, in which the preempted resource is main memory and the backup is a drum or other mass storage; and (2) cpu interrupts, in which the preempted resource is the cpu and the backup is the core memory used to save the processor status and registers. Resumption is useful only when the information to be saved is well defined and the overhead needed to save and restore it is acceptably small relative to the frequency with which preemptions occur.

Three Basic Approaches

Detection

Detection is the periodic use of an algorithm which inspects the current resource allocations and outstanding (unsatisfied) requests, to produce an indication of whether a deadlock currently exists and, if so, what processes and resources are involved. Such algorithms are described by Holt [2] and Shoshani and Coffman [3].

Detection of a deadlock does not solve it. Rather, if a deadlock is found, the system must break it by preempting some of the resources involved in the deadlock. Thus detection involves not only the overhead of the detection algorithm but also the losses inherent in preemption. It is useful primarily when a resumption mechanism such as swapping is already present in the system. It is difficult to apply to resources such as tape drives, for which preemption involves possibly unacceptable overhead.

A second problem with detection is that it takes no action until a deadlock actually occurs. This may well mean that a blocked process will hold a resource for a long period of time without being able to use it. On the other hand, preemptions are minimized since only the absolutely necessary ones occur.

Avoidance

An avoidance algorithm projects detection into the future in order to keep the system from committing itself to an allocation which will eventually lead to a deadlock. Habermann's algorithm is the best example of avoidance [4]. Like all avoidance algorithms, it must be provided with information about future resource requirements for each process. In Habermann's algorithm, this knowledge is in the form of an upper bound on the quantity of each resource group that is more than will ever be used by each process. A state is safe if there is a process which can be run to completion using only the unallocated resources plus those already allocated to it, such that the state thus obtained is (recursively) safe. The resource allocator avoids deadlocks by testing each possible allocation and making only those which lead to safe states.

Avoidance does not need preemption, but does require knowledge of the future. It will use resources inefficiently if the upper bounds are too liberal. Thus the need for good upper bounds is a technical difficulty with the avoidance approach.

A second difficulty arises if the system is heavily loaded and thus is running with most of its resources allocated. In this case the allocator will see relatively few safe allocations among the outstanding requests, so many processes will be blocked for long periods while holding valuable resources. This and similar situations have been described by Habermann [5] and by Holt [6].

Prevention

Prevention is the process of structuring the system so that deadlocking requests will never occur. It differs from avoidance in that no run-time testing of potential allocations need be performed. As pointed out in [1], a necessary condition for the existence of a deadlock is that there is a circular chain of processes, each of which holds exclusive and non-preemptable control of some resource and each of which is requesting the resource held by the next process in the circular chain. This situation can be prevented in several ways as described by Havender [7] and Howard [8], including preemption, requesting all resources at once, and resource ordering.

Preemption can be used to prevent deadlock as follows. Whenever a process's request for some resource cannot be satisfied immediately, other resources held by the process are preempted. A typical example is the use of main memory. If a process requests more memory than is available, then it is completely swapped out. In other words, the memory already owned by the process is preempted. The process is not swapped back in until

the entire larger quantity of memory is available. In contrast to the use of preemption in detection schemes, this technique errs by preempting more often than is really necessary.

Requesting all resources at once is a simple and effective way of preventing deadlocks, but has the obvious difficulty that processes may hold resources for extended periods during which they are not needed. This technique works well for processes such as input/output drivers which perform a single burst of activity, but it fails for processes such as user jobs with fluctuating requirements.

Resource Ordering. Resource ordering is a more sophisticated form of prevention, and is of special interest because it provides the framework for the mixed methods to be described later. In a resource-ordered system, the many kinds of resources, both reusable and consumable, are grouped into ordered classes C_1, \dots, C_k . If a process holds a resource of class C_j , then it may request a resource of class C_i only if $i > j$. Adherence to the ordering rule prevents deadlocks by making circular chains of blockages impossible. A major advantage of resource ordering is that it can be enforced by a compile-time check, a virtue touched upon by Hoare [9].

The difficulty with resource ordering is that it restricts the allowable sequences of requests by processes. If it is applied to user jobs, then there must be a checking and enforcement mechanism, and the users must be educated as to the order. The lack of flexibility in request sequences can lead to a process requesting and holding a resource unnecessarily early. A specially severe example of this is main memory, for which there is a fluctuating need. The resource ordering rule disallows requests for extension of the quantity of such a resource. Thus the maximum quantity ever to be needed must be requested and held from the beginning.

On the positive side, resource ordering and the other prevention techniques solve the deadlock problem completely in the system design and do not need runtime computation. This not only eliminates overhead but also makes the use of priority or other scheduling much easier. If first-in-first-out queuing is used, then resource ordering solves not only the deadlock problem but also the permanent blockage problem since every request can be satisfied after a delay bounded by the finite amount of work preceding the request in the queue. (This presumes that the first request in the queue is the first to be satisfied, even if the queue contains a smaller request for which resources are available.)

Combined Approach

Hierarchical Structure

Many well-designed operating systems have a hierarchical structure [10], that is, a series of layers of software each of which modifies and extends the capabilities

provided by the underlying layer. A "primitive operation" of a higher layer is implemented by creation of a process in a lower layer, which performs the necessary actions and returns some result.

In general the higher-level process must wait for a message signaling completion. This affects the deadlock mechanism since the message must be treated as a resource. In the case of resource ordering, the effect is that the higher-level process must not hold any resources needed by the lower-level processes it invokes. This constraint provides a "natural" ordering of resources in a hierarchical system. Resources used at the lower levels must appear later in the ordering.

Mixtures

Whether or not a system has an explicit hierarchical structure, it should be possible to divide resources into classes along hierarchical lines so that a natural ordering between classes exists. Any such division can be used as the basis for a mixed solution for the deadlock problem. Resource ordering is applied to the classes. Within each class the most appropriate technique can be used, considering only the resources in that class.

Such a mixed system will not get into a deadlock for the following reason. Consider a hypothetical deadlock. It cannot involve resources in more than one class, because if so some process in the circular chain of deadlocked processes is violating the class ordering rule. But it cannot involve resources in a single class because the algorithm for that class has prevented or avoided such a deadlock. Hence the hypothetical deadlock cannot exist.

Example. This example is an idealization of the system in use at the University of Texas [8] on a CDC 6600. It contains resources in the following classes: (1) space in the swapping store; (2) assignable devices, such as tape drives, and other job resources, such as access to files within the file system; (3) central memory for user jobs; (4) internal resources, such as memory for system transient overlays, and channel and controller access.

Note that the cpu is not considered to be a resource since it is preemptable via interrupts and thus can be shared by many processes.

A typical mixed deadlock solution for such a system might order the above classes as shown, and apply the following individual approaches to the classes.

1. *Swapping space.* Probably the most practical method is preallocation of the maximum space needed by each process. Avoidance could also be used since it does not require preemption, which is impossible due to the lack of backup for the swapping store.
2. *Job resources.* Avoidance is the most reasonable solution as it is usually possible to deduce a considerable amount of knowledge of a job's future from its control cards. If this is impossible, then resource ordering can be used. Detection and preemption are undesirable due to the possibility of partially-rewritten files.

3. *Central memory.* Prevention using preemption is the logical choice under the assumption that the system already contains swapping. A job is swapped out whenever it requests more space than is available. Detection is also a possibility, but is undesirable since it ties up memory with blocked jobs.

4. *Internal system resources.* Prevention, usually in the form of resource ordering, is the best choice. Transactions occur so frequently that overhead in the allocator itself must be minimized. Prevention does so by requiring no run-time choices among pending requests. It also allows the use of appropriately chosen service disciplines without the necessity of explicitly dealing with deadlocks. A natural ordering is usually present due to the hierarchical nature of internal system structure.

Conclusion

Various deadlock solutions may be mixed within the framework of resource ordering, which corresponds closely to the structures already present in hierarchical operating systems. Although no one approach is useful in all applications, mixtures should be able to handle the problems usually encountered in operating systems. The practical advantages of mixed methods will almost certainly outweigh the theoretical desirability of using a single method throughout.

Acknowledgment. The author wishes to thank Dr. James C. Browne for his advice and encouragement during the development of this paper and the research it reports. The referees' suggestions, which contributed substantially to the paper's clarity, are also acknowledged with thanks.

Received March 1971; revised October 1972

References

1. Coffman, E.G., Elphick, M.J., and Shoshani, A. System deadlocks. *Computing Surveys* 3, 2 (Feb. 1971), 67-78.
2. Holt, R.C. On deadlock in computer systems. Ph.D. Diss. Dept. of Computer Science, Cornell U., Ithaca, N.Y., Jan. 1971.
3. Shoshani, A., and Coffman, E.G. Prevention, detection, and recovery from system deadlocks. *Proc. 4th Ann. Princeton Conf. on Information Science and Systems*, Mar. 1970.
4. Habermann, A.N. Prevention of system deadlocks. *Comm. ACM* 12, 7 (July 1969), 373-377; 385.
5. Habermann, A.N. Comments made at the Third Annual Symposium on Operating System Principles, Palo Alto, Calif., Oct. 1971.
6. Holt, R.C. Comments on prevention of system deadlocks. *Comm. ACM* 14, 1 (Jan. 1971), 36-38.
7. Havender, J.W. Avoiding deadlock in multi-tasking systems. *IBM Syst. J.* 2 (1968), 74-84.
8. Howard, J.H. The coordination of multiple processes in computer operating systems. Ph.D. Diss. Department of Computer Sciences, U. of Texas, Austin, Tex., Dec. 1970.
9. Hoare, C.A.R. Towards a theory of parallel programming. In *International Seminar on Operating Systems Techniques*. Academic Press, New York 1972.
10. Dijkstra, E.W. The structure of the THE-multiprogramming system. *Comm. ACM* 11, 5 (May 1968), 341-346.