# COMMENTS ON PREVENTION OF SYSTEM DEADLOCKS

Richard C. Holt

Technical Report
No. 70-50
January 1970

Department of Computer Science
Cornell University
Ithaca, New York 14850

# COMMENTS ON PREVENTION OF SYSTEM DEADLOCKS

Richard C. Holt

Department of Computer Science
Cornell University
Ithaca, New York 14850

ABSTRACT

Habermann's method of deadlock prevention is discussed, where deadlock is defined as a system state from which resource allocations to certain processes is not possible. It is shown that the scheduler may introduce deadlocks which Habermann's method does not prevent. Effective deadlock is defined as the situation where certain processes do not receive their resource requests. It is shown that deadlock prevention does not imply effective deadlock prevention. A method of effective deadlock prevention is given.

KEY WORDS AND PHRASES: multiprogramming, time-sharing, scheduling, resource allocation, deadlock, interlock, deadly embrace, knotting.

CR CATEGORIES: 3.72, 4.32, 6.20.

## 0.0 Introduction.

A. N. Habermann has made an important contribution to the theory of computer system design by presenting a method for preventing deadlock [1]. Deadlock, as Habermann uses the term, means that the resources of the system have been allocated among certain processes in such a way that it is impossible to grant additional requests to these processes. (Preempting resources from processes is not allowed.) Notice that although it may be possible for a process's request for resource to be granted (i.e., the system is not in a deadlock), this does not necessarily imply that the request will be granted. Let us define effective deadlock to be the situation where some process's request is never granted. An effective deadlock can occur when the system is in a deadlock and, thus, it is not possible to grant the request. On the other hand, an effective deadlock can occur when, although it is possible for a process to receive its request, the request is never granted. It is the purpose of this note:

(a) to show that the scheduler may introduce new deadlocks which Habermann's method does not prevent;

(b) to show that effective deadlock can occur even when deadlock is not possible; and

(c) to present a technique which prevents effective deadlock as well as deadlock.

## 1.0 Habermann's Method for Preventing Deadlock

We are concerned with computer systems in which each process must claim the maximum of the resources the process will require, and each process will eventually release all resources allocated to it. Habermann defines a safe state as a state in a system from which there exists a sequence of resource allocations and releases such that all processes can be granted their requests. (See formal definitions in [1].) Such a sequence is called a safe sequence. Habermann shows that if the scheduler only grants safe requests — requests which when granted leave the system in a safe state — it will be possible to grant all subsequent requests. In his article [1] Habermann states, "The algorithms [for deciding if a state is safe] decide only whether or not granting a request can produce a deadlock, so assignment rules (according to priority rules, for instance) can be implemented freely." The obvious conclusion would seem to be that any scheduler which grants only safe requests will avoid all deadlocks and will grant all requests. This conclusion is not warranted, as will be shown below.

The correct conclusion is that if a scheduler only grants safe requests then:

> The system will never reach a state where, regardless of what the scheduler does, certain requests cannot be granted.

The burden of seeing that a particular request is granted still

rests with the scheduler. Granting only safe requests assures that if the scheduler takes the right action, then it can always grant any request.

## 2.0 Deadlocks Introduced by the Scheduler.

Consider a system which consists of (a) two types of resources, $R_1$ and $R_2$, each containing one unit, and (b) three processes, $P_1$, $P_2$, and $P_3$. Deadlock will occur in this system if process $P_1$ holds $R_1$ and requests $R_2$ while process $P_2$ holds $R_2$ and requests $R_1$. A straightforward way to schedule the resources in this system is to use a FIFO queue for each resource. (This scheduling scheme can be implemented by OS/360 ENQ and DEQ primitives [2]. J.E. Murphy discusses deadlock in systems with FIFO queues [3].) Now suppose we seek to avoid deadlock by restricting the scheduler to grant only safe requests, without violating the FIFO rule. Unfortunately, deadlock is possible in this new scheme, as will be shown by an example. Assume $P_1$ and $P_2$ claim both $R_1$ and $R_2$ while $P_3$ claims only $R_2$. We initially observe the system while $P_1$ holds $R_1$ and $P_3$ holds $R_2$. Next $P_2$ requests $R_2$, then $P_1$ requests $R_2$, and finally, $P_3$ releases $R_2$. In Habermann's notation:

$a = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ (total resources),

$B = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$ (process claims),

$C_0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ (initial allocations), and

$C_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ (allocations after $P_3$ releases $R_2$).

Since $R_2$ was requested first by $P_2$, by the FIFO rule $R_2$ must be allocated to $P_2$ before $P_1$. However, if $R_2$ is allocated to $P_2$, the system will not be in a safe state. Thus the scheduler will not allocate $R_2$ until granting $R_2$ to $P_2$ will result in a safe state. But this will not happen until $P_1$ releases $R_1$, and $P_1$ cannot release $R_1$ until $R_2$ is granted to $P_1$. Hence, although the system was always in a safe state, it has become impossible for $P_1$ or $P_2$ to receive their requests. This simple example of deadlock results because the scheduler imposed an order on the allocations, thereby blocking the safe request for $R_2$ by $P_1$. Similar cases of deadlock can occur when a priority rule causes a safe request to be blocked.

Whenever the scheduler has the power to block safe requests, one must verify that the scheduler has not introduced new deadlock states. This type of deadlock can be avoided by weakening the meaning of FIFO (or priority) so that when the first process in a queue cannot be granted a request, the

scheduler passes down the queue, granting those requests which are safe.

## 3.0 The Problem of Effective Deadlock

An obvious condition, call it the expediency condition, to impose on a scheduler is to require it to continue granting requests as long as the requests are safe. It is interesting that certain systems will necessarily have effective deadlock if the expediency condition is imposed. Consider a system which consists of (a) one type of resource containing two units and (b) three processes, $P_1$, $P_2$, and $P_3$. Processe $P_1$ and $P_2$ claim one unit of the resource and process $P_3$ claims both units of the resource. In the initial state of the system one unit of the resource is allocated to the first process. In Habermann's notation

$$\begin{aligned} a &= (2) && \text{(total resources)}, \\ B &= (1\ 1\ 2) && \text{(process claims), and} \\ C_0 &= (1\ 0\ 0) && \text{(initial allocations).} \end{aligned}$$

For this system there exists an infinite sequence of allocatio such that the third process is never allocated resources:

$$\begin{aligned} C_0 &= (1\ 0\ 0), \\ C_1 &= (1\ 1\ 0), \\ C_2 &= (0\ 1\ 0), \\ C_3 &= (1\ 1\ 0), \\ C_4 &= C_0 . \end{aligned}$$

(If the system contains a fixed set of processes and each process halts after a finite time, as is the case in Habermann's thesis [4], then no such infinite sequence can occur. Effective deadlock is a problem only in systems which run for a "long" period of time.)

Suppose that in the above sequence, each time process $P_1$ holds no resources, $P_1$ requests a resource before $P_2$ releases its resource, and each time $P_2$ holds no resources, $P_2$ requests a resource before $P_1$ releases its resource. If the priority of process $P_3$ is lower than the priorities of $P_1$ and $P_2$ then $P_3$ will never be granted a request. If process $P_3$ requests both units of the resource and the expediency condition is used, then $P_3$ will never receive its request regardless of its priority. Here $P_3$ is the victim of an effective deadlock because at most one unit of the resource is available, and the scheduler will continue forever granting requests for one unit of the resource to processes $P_1$ and $P_2$. In this example, although deadlock is prevented, effective deadlock is possible.

The above example of effective deadlock comes close to being realized in some multiprogramming systems. Let us suppose that the user of a certain multiprogramming system must request either one or two 100K byte blocks of core when submitting a job, and that the system has a total of two such blocks to allocate. A user job must be granted its request before beginning execution and after a finite execution time,

it must finish and release its allocated blocks. If the system has many jobs requesting only one block, and if the scheduler uses the expediency condition, i.e., if the scheduler allocates a block to a waiting job whenever possible, then a job requesti two blocks may need to wait forever before the blocks are simultaneously available.

It might be argued that the situation just described is unrealistic because in a multiprogramming system job arrivals are separated by a random interarrival time. At some point in time, continues the argument, a long interarrival time will occur, the jobs with small core requirements will finally finish, and any waiting jobs with large core requirements will finally execute. For a user of Cornell's Clasp/OS/360 operating system who has a low priority, large core job, this argument is largely academic because the mean time to system cold start (when the job queue is lost) is comparable to the mean time to wait for the required long interarrival time.

In 1966 D.E. Knuth pointed out the danger of effective deadlock in a system where deadlock is not possible [5,6]. In 1965 E.W. Dijkstra had given a solution to the critical section problem which guarantees that only one process at a time gains entry to a critical section, and that if several processes are competing for entry, at least one process gains entry. Such a system can never deadlock in that it is possible for all processes to gain entry to the critical section.

However, as Knuth showed, if certain timing conditions persist, a particular process may never gain entry.

Dijkstra's solution is an example of a system which only grants safe requests and uses the expediency condition. Any such system avoids deadlock in that the following criterion is satisfied:

> The system will never reach a state where, regardless of what the processes do, certain requests cannot be granted.

A different condition, call it the eventuality condition, which can be imposed on a scheduler is to allow the scheduler to block a safe request, but only for a finite time. It can be shown that any system which grants only safe requests and uses the eventuality condition will also satisfy the above criterion. The eventuality condition offers the advantage, as we shall see, of giving the scheduler the power to prevent all effective deadlocks by temporarily blocking safe requests.

## 4.0 Preventing Effective Deadlock

In many systems, especially where resource utilization is low, the statistical arrival of resource requests and releases will prevent effective deadlock. In some cases effective deadlock is not detrimental. For example, a low priority process may be designed to compile system statistics when the system is relatively idle. If this process is never active, no harm is done. On the other hand, effective deadlock

can be quite expensive when the permanently blocked process has been allocated valuable resources. In systems where effective deadlock is possible and harmful, steps should be taken to prevent it.

A technique for preventing effective deadlock will now be described. The system is augmented by two 1 by n arrays called t and u . When process $P_i$ makes a request for resources, element $t_i$ is set to the time of the request. When process $P_i$ is granted a request, element $t_i$ is set to some special value, say -1 . The time, call it waittime, that process $P_i$ has been waiting for an ungranted request can be calculated at time now in the following manner:

```
if t_i = -1 then waittime := 0
            else waittime := now - t_i .
```

Element $u_i$ gives the maximum time process $P_i$ must wait for a request before the scheduler will activate a special strategy to ensure that process $P_i$ will receive its request. The scheduler is free to grant safe requests by any set of rules, but periodically it must examine the arrays t and u to see if any process, call it process $P_i$ , has waited beyond its maximum time. If so, the scheduler must activate the following strategy:

(a) A safe sequence which consists of process $P_i$ and all processes with non-zero allocations is found. Ideally, process $P_i$ should be near the beginning of the sequence.

(b) Requests are granted only to the first process in the sequence until that process has released all resources allocated to it. Then requests are granted only to the next process in the sequence until that process has released all resource allocated to it, and so on. This continues until enough resources are available so that the request by process $P_i$ is safe. Process $P_i$ is then granted its request.

(c) Each process other than process $P_i$ is examined to see if its maximum waiting time has been exceeded. If so, the process which has waited longest beyond its maximum waiting time is designated process $P_i$ and this strategy is repeated by returning to (a).

It can be proved that this technique always prevents effective deadlock. Notice that this technique does not violate the eventuality condition, but may violate the expediency condition in step (b) to force the granting of the request to process $P_i$.

## 5.0 Conclusion.

It is the purpose of this note to clarify the method of prevention of system deadlock which was proposed by Habermann. The system designer is warned that a scheduler which can block safe requests can introduce new deadlocks, and that even though deadlock is prevented, certain processes may be permanently blocked.

Acknowledgments. The author is grateful to Alan Shaw for pointing out the example of Dijkstra and Knuth, and for offering many valuable suggestions. Nelson Weiderman and Tom Wilcox also offered many helpful comments.

REFERENCES

1. Habermann, A.N. Prevention of system deadlocks. Comm. ACM 12, 7 (July 1969), 373-377, 385.

2. IBM System/360 Operating System Supervisor and Data Management Services. IBM Form No. C28-6646.

3. Murphy, J.E. Resource allocation with interlock detection in a multi-task system. Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33, Pt. 2, Thompson Book Co., Washington, D.C., pp.1169-1176.

4. Habermann, A.N. On the harmonious cooperation of abstract machines. Thesis, Mathematics Department, Technological University, Eindhoven, The Netherlands, 1967.

5. Knuth, Donald E. Additional comments on a problem in concurrent programming control. Comm. ACM 9, 5 (May 1966), 321-322.

6. Dijkstra, E.W. Solution of a problem in concurrent programming control. Comm. ACM 8, 9 (September 1965), 569.