ON DEADLOCK IN OPERATING SYSTEMS

By

Z. GIDÓFALVI

Department of Mathematics of the Faculty of Electrical Engineering
Technical University, Budapest

Received June 20, 1978
Presented by Prof. Dr. O. KIS

Introduction

The operating systems of computers have a double task. These have partly to guarantee the optimum use of hardware resources, and partly to control and supervise user's processes.

The operating systems consist of modules. Each module is responsible for a well-defined and rather restricted work. This structure makes the systematic and manageable design of system possible. Starting from the user's requirements and from the hardware available, the subtasks and the implementing system modules can be constructed. The modules can be classified according to their work: there are interrupt handler routines receiving the hardware interrupts and invoking the selected routine associated with the interrupt type; process management routines performing the creation, control and termination of processes; resource management routines dealing with the allocation and administration of system resources; file system routines handling the programs and data on the secondary storage and making the access to files possible, i.e. organizing the information transfer between the main and auxiliary storage; and a part of service programs, which perform the fundamental services. Translators and program products are not considered to belong to the precedings, because their work is rather in the user line.

Recently the modules of operating systems are written in re-entrant codes to perform their simultaneous use by more than one system or user processes.

Servicing the user processes raises many requirements which far exceed the work of a module, demanding the co-operation between several routines. This co-operation is realized in non-deterministic manner, because the system simultaneously services several user processes. So it is easy to realize, that some processes come to a deadlock situation because of the requests issued to the others, hence this state cannot be modified without external interference. A deadlock situation especially with system modules is not desirable, so prevention of this problem must already thought in design.

The system hs ato be examined whether a deadlock state may ever arise ıo not, with other words: can the system work servicing all imaginable user processes?

The system model will be examined for methods of prevention and a dynamic control method suggested.

The static system model

The complete system including the user processes becomes uniform with synchronization signals. These signals may be hardware interrupts, requests for various services or data and the corresponding answers. These have to be considered as message-like, consumable resources. They are resources, because they can cause the blocked state of processes and consumable, because the synchronization signal, sent out say by process p_1 to activate p_2, is consumed by p_2, so the signal is never more available. In the above example p_1 is the producer, p_2 is the consumer of the resource according to our terminology. p_1 issues a request to p_2 (releases a unit of the resource) with the instruction RELEASE (resource type); likewise p_2 consumes this unit with the instruction REQUEST (resource type), if it is possible. Otherwise p_2 gets blocked by the instruction REQUEST (wait for the request from p_1). As the system programs are in blocked state until used, they can be considered in our model starting with an instruction REQUEST and waiting until the desired resource unit will be created. The re-enterable modules can be simultaneously used by several processes, so they have to wait always at the instruction REQUEST for a new request. This fact,..., as it wil be seen later,..., causes — though manageable — difficulties.

Our model considers static states, where the number of simultaneously handled processes is constant. This model — with some supplements — is also suitable for describing the dynamic behaviours of the system, to be discussed in the next chapter. Moreover, the producers and consumers of all resources are supposed to be *a priori* known.

Let $P = \{p_1, \ldots, p_i, \ldots, p_n\}$ the set of processes staying simultaneously in the system in a static state and $R = \{r_1, \ldots, r_j, \ldots, r_m\}$ the set of resources produced by P. The connection between the system elements can be represented by the consumable resource graph. Its nodes consist of processes and resources $N = P \cup R$, the (p_i, r_j)-type edges represent requests (process p_i requests a unit of resource r_j, marked with an arrow from p_i to r_j), the (r_j, p_i)-type ones represent producer relations (p_i is the producer of r_j, marked with an arrow from r_j to p_i). Moreover each $r_j \in R$ has a non-negative integer t_j, which means the number of available units. The state of the graph can be changed by the following three operations:

1. *Request*: if p_i is not blocked, it can request the elements of $R^* \subseteq R$ (R^* is *a priori* known), thus the graph is amplified by the associated request edges.
2. *Production*: if p_i is not blocked, it can produce the elements of $R^+ \subseteq R$ (R^+ is *a priori* known), according to this the graph is amplified by the proper producer edges. If the producer relation has already been marked, the last amplification is unnecessary, because the producer edges are never removed from the graph. Furthermore all corresponding unit counters are raised with the number of produced units.
3. *Consumption:* if p_i has outstanding requests and the wanted resources are simultaneously available, p_i can consume them. The desired units are available, if $t_j \geq |(p_i, r_j)|$ for all $r_j \in R^*(|(p_i, r_j)|$ means the number of request edges from p_i to r_j). Thereby the corresponding edges are removed from the graph, and the t_j's are decreased accordingly.

This model is useful for describing any state of the system. Our purpose is to eliminate even the possibility of deadlock. Therefore first it has to be examined whether or not a given state is deadlocked. A state is deadlocked if there are certain processes deadlocked in this state, i.e. there is no way to activate them. Let T be a state of the system to be find out whether it is deadlocked or not. Let us try to terminate the processes in all possible orders, considering the fact that p_i can only be terminated if it is not blocked, that is, p_i has no outstanding requests. If there is at least one order in which all processes can be terminated, T can be said not to be a deadlock state. In that case the processes are terminated by uniprograming, but this operation cannot lead to false results, because the terminating order is a possible set of state changing. The transitions can also be demonstrated on the graph. This activity is called the graph reduction. The graph corresponding to the T system state can be reduced by p_i, if p_i is not blocked, that is, first the requests of p_i are satisfied then released units are enough to satisfy all outstanding requests to the resources produced by p_i. Meanwhile the edges connected with p_i are removed from the graph. On the basis of the above it can be stated: if the graph representing the state T can be completely reduced, i.e. there exist at least one order, T is not deadlocked.

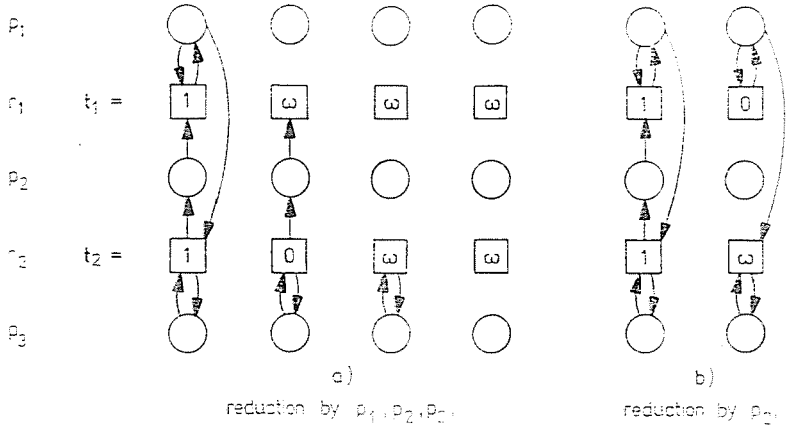
If only safe states are wanted in our system, the so-called claim-limited state and the corresponding graph have to be examined. This state has the following properties:

1. all resources have zero available units,
2. there is only one edge directed from p_i to all $r_j \in R^*$,
3. there is only one edge directed to p_i from all $r_j \in R^+$.

The next statement is easy to prove:
All states in a consumable resource system in which producers and consumers are known are safe if and only if the claim-limited consumable resource graph is completely reducible.

Proof: First assume that all states are safe, so the claim-limited state is safe, too, and the claim-limited graph is completely reducible. On the other hand, assume that the claim-limited graph is completely reducible with an order $p_{q1}, \ldots p_{qn}$. If the claim-limited graph can be reduced by p_{q1}, then any other state can be, too, because p_{q1} cannot have requests otherwise the claim-limited graph cannot be reduced by p_{q1}. Likewise if the claim-limited graph can be reduced by p_{q2}, then any other state can be, too, because p_{q2} can only request the units of p_{q1}. Continuing in this manner it can be shown that all states are safe, because all of them can be reduced by $p_{q1}, \ldots . p_{qn}$, in order.

In connection with the reducibility a question arises: in which order have the reductions to be made? It is easy to see that the reduction of a general state is too difficult, because the only sure fact is: if there exists at least one order, in which the reduction is possible, then the original state was not deadlocked. But trying all possible orders is a very time-consuming work. The next simple example illustrates the difficulties in the reduction. The same graph is reducible in the figure *a)* and irreducible in the figure *b)* because the resource demands of p_1 cannot be satisfied (ω represents the number of resource units left).



In this respect the claim-limited state and the coresponding graph are characteristic ones, because the order of reduction leads to the same state. It is easy to see, because at the beginning of the reduction every resource has

zero available units and any of the reductions deletes the same edges from the graph.

After these it is expedient to examine two questions. The first relates to the claim-limited graph. From the proof of reduction theorem the necessity of at least one process — having no request, only producer edges in the graph — is seen. Generally it is realized, because the operator is always a resource producer. The other question is to fit the hardware interrupts into our model. It is useful to treat the interrupt handlers as the producer of hardware interrupts, so these signals also arise from processes. These processes guarantee also the necessary condition of deadlock prevention, because they have no requests.

Summarizing the precedings it can be stated that the complete reducibility of the claim-limited resource graph guarantees the safe states in the system, when producers and consumers of resources are known *a priori*.

The dynamic system model

As mentioned in the previous chapter, the dynamic model does not essentially differ from its static equivalent. The difference is only that the new claim-limited graph has to be built in all cases where the number of simultaneously serviced processes has been changed, and the reduction has to be made once more. To build up the graph two problems have to be examined in details:

1. should all created processes appear in the claim-limited graph?
2. how to handle the re-enterable system modules?

The first question arises in realizing, that there is one system module from the created ones, not to be activated in a static state, i.e. none of the created processes would produce the desired resource. If this inactive process occurred on the claim-limited graph, our model would not correctly reflect the reality, because the presence of this module is irrelevant in the system, but the claim-limited graph becomes irreducible by this one (its requests cannot be satisfied). Accordingly the process p_i must not be taken up on the claim-limited graph, if there exists at least one resource requested by p_i without producers in the system.

Attention has been called to the re-enterable system modules in the previous section. They always ought to have requests representing the fact that they are simultaneously usable by some processes. This would mean their permanent blocked state, which is against reality. Therefore every process requesting one of the re-enterable modules is assumed to get a new copy of this, so a module appears in several independent copies on the claim-limited graph.

The two questions can be joined in the next one: how many copies of system modules are needed on the claim-limited graph? So many copies are

required, as many processes can activate the module, i.e. as many processes produce resources to activate the module.

Let us survey the process of changes. First, the system moduls are created, then the user ones. The created processes have to be administered, of course. Therefore it is useful to handle a graph containing all created processes and all resources produced by them. Let us call attention to this graph, that is not equivalent to the claim-limited graph, but it has to be constructed from the administrator graph.

Creating a new process, first it is taken up on the administrator graph, then the complete graph is reviewed to find which modules and how many copies of them are required. (Attention: the new process may produce resources to activate such ones, which exist only on the administrator graph, so these must be taken up on the claim-limited graph, too.) This is followed by the attempt to reduce the new claim-limited graph. In case of success, the safe states are guaranteed until the next change. Otherwise there are two options. Either the entry of the new process into the system is delayed (accordingly the edges directed to and from it are removed from the graphs), or the possibility of deadlock is risked. At the last choice a much more difficult task has to be accomplished than in the former case, i.e. to try reducing the resource graph (not the claim-limited graph!) after all requests and consumptions (both can cause deadlock, see [1] 235—236). On the basis of the previous section it is a very time-consuming task. If the resource graph is not completely reducible, then the original state was a deadlock state, and this situation has to be recovered. The only way to perform this is to terminate certain processes, which can destroy the deadlock most economically.

At termination of processes all edges from and to them have to be removed from the administrator graph and enough resources left to accomplish the further requests directed to them. In such a case the claim-limited graph needs not to be built and reduced again, because the process termination only improves the situation.

It is expedient to entrust the administration of processes and the reduction of the claim-limited graph to system modules. For this purpose the data structure of the graphs has to be defined. The matrix representation is suitable for the administrator graph, handling two matrices representing the requests and the producer relations. The list structure is better for the claim-limited graph, where three sets of lists are handled:

— the resources requested by p_i are linked to p_i
— the resources produced by p_i are linked to p_i
— the processes requesting r_j are linked to r_j.

Summarizing, the prevention of deadlock, the realization of safe states are easy to do. In creating a new process it can be decided whether or not the new process enters into the system.

Summary

Deadlock situations in operating systems are dealt with, restricted to those caused by message-like resources. On the basis of our model, created for this problem, the methods of system programing to prevent deadlock are examined. The requirement of prevention may seem to be too rigorous, but otherwise the deadlock detection and recovery would not be economical, because of the great time and space requirements of the realizing algorithm. Our results can be composed into an algorithm excluding deadlock situations in any dynamically varying system.

References

1. SHAW A. C.: The logical design of operating systems Prentice-Hall Inc. 1974.
2. HOLT. R. C.: On deadlock in computer systems Ph. D. thesis, Cornell Univ., Ithaca, N. Y. (Jan. 1971).

Zoltán GIDÓFALVI H-1521 Budapest