A note on deadlocks

By Z. LABORCZI

Introduction

A set of processes uses resources of several types concurrently. We assume that there is only a limited number of resources from each type. The number of resources can be characterized by a vector **t**, where t_i is the total number of resources of type i.

Another constraint is that the processes cannot be forced to release resources they currently use.

If we know nothing about the behaviour of the processes, then the only possible way for scheduling the processes is the strictly sequential ordering. Real concurrency could not be allowed because each concurrent process may request all the resources at any time, and this request cannot be fullfilled if resources are allocated to other processes.

Therefore, we must have information on the behaviour of the processes in form of some kind of limitation the processes comply with.

One possible limitation among others [4] is the following: on entering the system, a process p has to announce a vector $\mathbf{goal}(p)$ declaring that it will not use more than $goal_i(p)$ resources from the i-th resource type. In order to be able to satisfy other requests, processes are not allowed to work forever, that is, if we place $\mathbf{goal}(p)$ resources at p's disposal and wait, p will terminate in finite time and return all the resources allocated to it. When p starts, it usually does not need all $\mathbf{goal}(p)$ resources immediately, and if we want to describe the current state of p, we have to introduce the vector $\mathbf{alloc}(p)$ which tells us how many resources have been allocated to p. It is clear that $\mathbf{alloc}(p) \leqq \mathbf{goal}(p)$, and the difference

$$\mathbf{need}(p) = \mathbf{goal}(p) - \mathbf{alloc}(p)$$

shows how many resources p still needs in order to complete. We may assume that there is no process in the system with $\mathbf{need}(p) = \mathbf{0}$ for if that is the case, we wait until p completes and continue the examination of the system only after the completion.

1. Graphical representation of processes

We describe a graphical representation of competing processes, which will turn out to be very useful later. The current need of a process may be represented by a point of the l-dimensional space, where l is the number of different resource types. A process p starts from **goal**(p) and currently stays at **need**(p), thus it is very natural to think of p as an arrow from **goal**(p) to **need**(p).

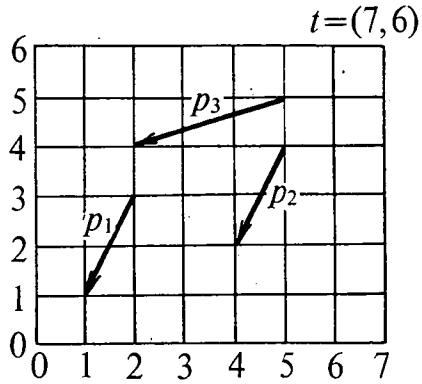

Figure 1

Figure 1 visualizes three processes competing for resources of two different types.

The number of resources in the system equals (7, 6), furthermore,

goal$(p_1)=(2, 3)$,
need$(p_1)=(1, 1)$,
goal$(p_2)=(5, 4)$,
need$(p_2)=(4, 2)$,
goal$(p_3)=(5, 5)$,
need$(p_3)=(2, 4)$.

This kind of representation is applicable only if $l=2$. For greater values of l one needs to be highly imaginative.

2. The definition of deadlock

Informally speaking, a system of concurrent processes is in deadlock, if there is no guarante that every process can complete. In other words the system is free of deadlocks if all the processes can finish, even if they request all their **needs** immediately. A formal definition of the latter assertion is the following:

Definition. The set of processes π is said to be free of deadlock (or deadlock-free) if there exists a permutation $p_1, p_2, \ldots, p_k$ of the processes in π such that

$$\mathbf{need}(p_i) \leqq \mathbf{t} - \sum_{j=i}^{k} \mathbf{alloc}(p_j)$$

for $i=1, 2, \ldots, k$.

This inequality means that if $p_1, p_2, \ldots, p_{i-1}$ have completed and returned the resources they used, then the **need** of p_i does not exceed the amount of the currently available free resources.

The following theorem is sometimes stated as another definition of the deadlock.

Theorem 1. The set of processes π contains a deadlock (or is in deadlock) if and only if there exists a nonempty subset π' of π such that the following inequality holds for every p in π':

$$\mathbf{need}(p) \not\leqq \mathbf{free}(\pi') \tag{1}$$

where $\mathbf{free}(\pi')=\mathbf{t}-\sum_{q \text{ in } \pi'} \mathbf{alloc}(q)$ is the amount of the resources currently not used by the processes in π'.

3. A condition and an algorithm

The definition of the deadlock and Theorem 1 speak about permutations and subsets of the process set. It is desirable to find a necessary and sufficient condition in which these notions do not occur, or in other words in which every process is mentioned only once, and not as a member of a permutation or a subset. This is accomplished by Theorem 2.

Theorem 2. Let $\mathbf{n}$ be a vector describing some amount of resources and $\mathbf{n}<\mathbf{t}$ ($\mathbf{t}$ is the total number of resources)[1]. A set of processes π is deadlock-free if and only if for every such $\mathbf{n}$ we have:

$$\sum_{\substack{p \text{ in } \pi \text{ and} \\ \mathbf{need}(p) \not\leqq \mathbf{n}}} \mathbf{alloc}(p) \not\geqq \mathbf{t}-\mathbf{n}. \tag{2}$$

For a deadlock, we can not only state that we can find an $\mathbf{n}$ such that $\geqq$ holds in (2) instead of $\not\geqq$, but we can replace $\geqq$ by $=$, that is the following assertion holds: π is in deadlock if and only if there is an $\mathbf{n}<\mathbf{t}$ such that

$$\sum_{\substack{p \text{ in } \pi \text{ and} \\ \mathbf{need}(p) \not\leqq \mathbf{n}}} \mathbf{alloc}(p) = \mathbf{t}-\mathbf{n} \tag{3}$$

Proof. It is sufficient to prove that

I. if π is deadlock-free, then (2) holds for each $\mathbf{n}$;

II. if π contains a deadlock, then (3) holds for at least one $\mathbf{n}$.

Proof of I. Let π be deadlock-free and $\mathbf{n}<\mathbf{t}$. We define

$$\Phi(\mathbf{n}) = \{q \text{ in } \pi | \mathbf{need}(p) \not\geqq \mathbf{n}\}.$$

$\Phi(\mathbf{n})$ contains exactly those processes for which (2) forms a sum, so if $\Phi(\mathbf{n})$ is empty, (2) is true.

Otherwise we apply Theorem 1 for $\Phi(\mathbf{n})$ and find a p in $\Phi(\mathbf{n})$ for which

$$\mathbf{need}(p) \leqq \mathbf{t} - \sum_{q \text{ in } \Phi(\mathbf{n})} \mathbf{alloc}(q) \tag{4}$$

For p as a member of $\Phi(\mathbf{n})$ we have also $\mathbf{need}(p) \not\leqq \mathbf{n}$, and replacing the left hand side of this inequality by the right hand side of (4) we get (2).

Proof of II. Let π be in deadlock and let π' be a maximal subset of π which statisfies Theorem 1, i.e., if we put a new element to π', (1) will not be true. We prove that (3) holds for

$$\mathbf{n} = \mathbf{t} - \sum_{q \text{ in } \pi'} \mathbf{alloc}(q)$$

by showing that $\Phi(\mathbf{n})=\pi'$.

Assuming that p is in $\Phi(\mathbf{n})$, we recall the definition of Φ and $\mathbf{n}$

$$\mathbf{need}(p) \not\geqq \mathbf{t} - \sum_{q \text{ in } \pi'} \mathbf{alloc}(q).$$

[1] $\mathbf{n}<\mathbf{t}$ means that $\leqq$ holds for every component and $<$ holds for at least one component.

It is now easy to verify that (1) holds for the subset $\pi' \cup \{p\}$ and as π' is a maximal subset in deadlock, p is in π'.

Starting from the other end, assume that p is in π'. From (1) we get

$$\mathbf{need}(p) \not\leqq \mathbf{t} - \sum_{q \text{ in } \pi'} \mathbf{alloc}(q),$$

that is $\mathbf{need}(p) \not\leqq \mathbf{n}$ and this means that p is in $\Phi(\mathbf{n})$.

After completing the proof of Theorem 2, we append the following remark to the last step of the proof:

At the very end we showed that π' is a subset of $\Phi(\mathbf{n})$. This means that

$$\mathbf{t} - \mathbf{n} = \sum_{p \text{ in } \pi'} \mathbf{alloc}(p) \leqq \sum_{p \text{ in } \Phi(\mathbf{n})} \mathbf{alloc}(p),$$

and this is exactly the negation of (2). If π happens to be a maximal subset, then the equality will hold.

If we write the formula of Theorem 2 in the following way

$$\mathbf{t} - \sum_{p \text{ in } \Phi(\mathbf{n})} \mathbf{alloc}(p) \not\leqq \mathbf{n} \tag{5}$$

we might formulate the meaning of Theorem 2 in terms of the arrows introduced earlier. Choosing an $\mathbf{n}$ we select the members of π which (as arrows) lie outside the rectangle consisting of the points less than or equal to $\mathbf{n}$. This set is $\Phi(\mathbf{n})$. (5) states that if we start at $\mathbf{t}$ and decrease our coordinates by $\mathbf{alloc}(p)$ for each p in $\Phi(\mathbf{n})$, we eventually reach a point lying still outside the rectangle.

For a deadlock state there must be an $\mathbf{n}$ such that the resultant point is not only within the rectangle but is identical to $\mathbf{n}$.

Exploiting these facts we may devise an algorithm to decide whether a set of processes is in deadlock or not. The algorithm runs as follows:

I. We define the function $\mathbf{f}$:

$$\mathbf{f}(\mathbf{n}) = \mathbf{t} - \mathbf{n}$$

for every $\mathbf{n} < \mathbf{t}$.

II. For every p in π, decrease the value of $\mathbf{f}$ by $\mathbf{alloc}(p)$ in the points of its domain for which

$$\mathbf{n} \not\geqq \mathbf{need}(p) \quad \text{and} \quad \mathbf{n} \leqq \mathbf{t} - \mathbf{alloc}(p) \tag{6}$$

III. Test after each decrease, whether the new value of $\mathbf{f}$ is 0. If so, we have a deadlock situation, otherwise if no 0 occured while performing II the system is free of deadlocks.

Condition (6) in step II needs some explanation. Requiring $\mathbf{n} \not\geqq \mathbf{need}(p)$ guarantees that p is in $\Phi(\mathbf{n})$. However, not all such $\mathbf{n}$-s have to be taken into consideration, because the decrease of $\mathbf{f}$ may result in 0 only for $\mathbf{n}$-s for which $\mathbf{n} \leqq \mathbf{t} - \mathbf{alloc}(p)$ is also true.

The small circles on Figure 2 indicate the points for which $\mathbf{f}$ has to be decreased in connection with p_3. For the point $\mathbf{n} = (3, 3)$ we have initially $\mathbf{f}(\mathbf{n}) = (4, 3)$. On executing step II for p_2 and p_3, $f(\mathbf{n})$ becomes 0, so these two processes are in deadlock independently of the existence of p_1.

The algorithm as described above detects deadlock within a system. A slight

modification makes it capable of handling the deadlock avoidance problem. Let us assume that π is deadlock-free and a process p requests some more resources. If we update **f(n)** by performing step II and step III for p only, it can be decided whether the fulfilling of the new request leads to a deadlock state.

Unfortunately this algorithm is not suitable for being incorporated into a real system, because it requires a great amount of space and time. Let us assume that we have 256 memory pages and 4—4 peripheral devices from two different types as resources at our disposal. In this case **t** becomes (256, 4, 4) and the number of **n**-s is $(256+1)*(4+1)*(4+1)-1=6424$.

Therefore we need almost 100,000 bits for representing the function **f**, and we have mentioned nothing about the time needed to update such amount of information. Therefore, we conclude that the general algorithms for detecting and avoiding deadlock [2, 3] have to be applied in the general case.

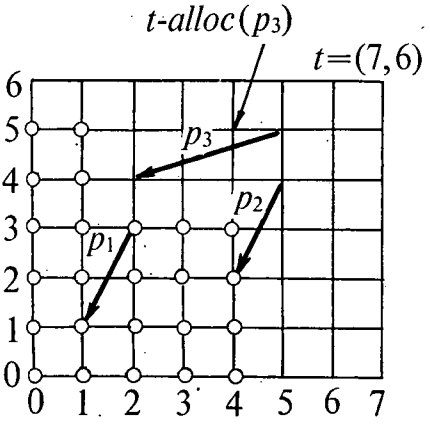

Figure 2

In one dimension, that is for one resource type, however, everything becomes very simple. There are no vectors, thus we may replace $\not\leq$ and $\not\geq$ by $>$ and $<$ respectively. In addition **f** becomes a scalar to scalar function. The updating of f is also less complicated: for a process p, f has to be decreased in those n-s where $n<need(p)$ holds. In fact we have arrived at a modified version of Habermann's theorem on detecting deadlocks for one resource type [1], thus this paper generalizes his results.

Conclusion

On investigating whether a simple theorem on detecting deadlock with one resource type can be generalized to more resource types, we found that the answer is yes, but the new theorem still needs further works to develop a practically usable algorithm.

Acknowledgement. I am indebted to Mr. J. Somogyi for many fruitful discussions on the subject of this paper and related areas.

Abstract

According to a paper by A. N. Habermann there is a simple necessary and sufficient condition whether a set of processes utilizing a limited number of resources of the same type is in deadlock. We show that this condition may be generalized to the case of more than one resource types. This result can be illustrated graphically in a clear way. The question of constructing algorithms on the basis of the extended condition is considered as well.

RESEARCH INSTITUTE FOR
APPLIED COMPUTER SCIENCES
H—1536 BUDAPEST, HUNGARY
P. O. BOX 227.

References

[1] HABERMANN, A. N., A new approach to avoidance of system deadlocks, Operating Systems, Proceeding of an International Symposium, *Lecture Notes in Computer Science*, v. 16, 1974.
[2] HABERMANN, A. N., Efficient deadlock avoidance algorithms, Unpublished paper, Intended to present at the Winter School, Budapest, 1977.
[3] HOLT, R. C., Some deadlock properties of computer systems, *Operating System Review*, v. 6, June 1972.
[4] DEVILLERS, R., Game interpretation of the deadlock avoidance problem, *Comm. ACM*, v. 20, No. 10, 1977.

(Received April 14, 1978)