A NOTE ON STARVATION-CONTROL POLICIES

by

Hari Madduri
Raphael Finkel

Computer Sciences Technical Report #564

November 1984

A Note on Starvation-Control Policies

Hari Madduri and Raphael Finkel
Computer Sciences Department
University of Wisconsin
Madison, WI 53706.

ABSTRACT

The Banker's algorithm for resource allocation prevents deadlocks. It needs additional policies to ensure that processes with unsatisfiable requests are not starved (blocked forever). Several policies and their relative merits are presented.

1. Introduction

Dijkstra's Banker's algorithm for resource allocation [1] guarantees freedom from deadlock by refusing requests for resources that lead to an unsafe state. A safe state is one from which we can find a safe sequence of job completions. All other states are unsafe. A process making a request that would lead to an unsafe state is blocked. The blocked processes wait their turn in a *blocked List*.

Whenever resources are released the resource allocator tries to unblock such processes. In the absence of a fair unblocking policy it is possible for some processes to remain blocked for an arbitrarily long period. This situation is called *Starvation* or *Indefinite Postponement*. The original Banker's algorithm did not address the issue of fairness with regard to unblocking.

A straightforward strategy such as "unblock the first blocked job first" may not work for the following reasons.

- Assume that a process requests a large number of resources (very close to the total number of resources) and is blocked since resources are in use by other processes. The blocked process may be the last in all possible safe sequences for the current allocation state, in which case allowing that process alone to unblock is unsafe and can lead to deadlock.
- Assume that we relax this strategy slightly and allow other small requests to be satisfied in the hope that those processes will eventually release more resources so as to satisfy the first blocked process. This strategy can lead to starvation of the first blocked process if there is a continuous stream of new jobs with smaller requirements. (This situation is called *effective deadlock* by Holt [2].)

This paper presents and evaluates several policies for unblocking that avoid starvation while preserving the deadlock-avoidance property of the Banker's algorithm. Any kind of starvation-prevention policy reduces the level of concurrency. The better ones sacrifice concurrency to a lesser extent.

2. Policies

We present below several policies that may be used for starvation control.

2.1. First Fit

This is the most straightforward policy. Whenever resources become available, the queue of blocked requests is scanned in arrival order and resources are granted to any process that can be safely unblocked. Partial allocation to processes that cannot be unblocked is not allowed even if safe. This scheme is not starvation-free, but it is included in the study to measure the relative performance of other schemes.

2.2. Guaranteed Progress

This scheme tries to service the blocked requests in a first-come first-served fashion. However, it does not insist on satisfying only the first process in the blocked processes list. It concentrates on satisfying the first process' request while allowing other blocked processes with satisfiable requests to be granted. This policy is achieved as follows.

(1) Calculate the minimum number of resources needed to complete all the processes (hereafter referred as *Minimum Need*. The *Minimum Need* concept and algorithms to calculate it are discussed in Madduri and Finkel [3].)

(2) Allocate any resources on hand in excess of *Minimum Need* to the first process in the blocked list, even if the resources are insufficient to unblock it. (Such allocation is safe, by the definition of Minimum need.) Since this first process acquires a partial allocation every time resources are released, it should make progress and eventually have its request satisfied completely.

The guaranteed progress policy has been suggested by Holt [2] and Fontao [4] for preventing starvation. However, as pointed out by Holt [5] and Rossi & Fontao, [6] partial allocation does not in fact guarantee freedom from starvation. There exist allocation states for which even though

resources are released, partial allocation is not safe, so no progress is made.

2.3. Guaranteed Completion

This scheme does not give special attention to any specific process, but rather tries to recognize starvation and then take corrective measures.

Once starvation is detected (as discussed in Section 3) the guaranteed completion method imposes a ban on new processes until starvation disappears. During the ban active processes run as usual. After some of them terminate, the resource allocator gets sufficient resources to safely unblock the remaining ones. Both when the ban is on and when the ban is off, we unblock processes according to the first-fit policy. Since no new processes are holding up resources and the allocations are always safe, all processes that are currently active are guaranteed to finish. This method is essentially a flow-control technique. It is sensitive to the way control is imposed and lifted. If control is lifted too early then starvation is possible. If it is not lifted early enough then it affects the smooth flow of jobs.

Holt's fix [5] to the partial-allocation problem of guaranteed progress is similar to this completion policy. His solution effectively prevents all new jobs from acquiring resources. However, it is more conservative than ours because it denies requests from processes with zero holdings whether they are new or old; our policy denies requests only from the new processes.

2.4. Other policies

An obvious way to avoid starvation is to reserve all resources the starved process needs. The algorithm used by the resource allocator in the operating system Boss 2 for RC4000 [7] calculates a safe sequence of job completions and then allocates resources according to the safe sequence order. If a process in the middle of the safe sequence asks for resources, then the request is granted only after preallocating resources to all the preceding processes in the safe sequence. This scheme certainly avoids starvation, but it is too conservative since it depends on preallocation and a specific

safe sequence.

Starvation-control strategies for the Dining Philosophers Problem [8] have been proposed by Dijkstra [9] that make use of allowance counters for each philosopher. An allowance counter indicates the maximum number of times a philosopher might allow others to eat. Every time a hungry philosopher misses its chance to eat its counter is decremented. Since the scheduler has to make sure that no counter ever becomes negative, every philosopher gets a chance to eat in a finite number of attempts. Improved strategies based again on allowance counters have been suggested [10, 11] that are less restrictive than Dijkstra's and permit a continuous stream of philosopher processes. Although these strategies prevent starvation, these are specific to the dining-philosophers problem and cannot be generalized. Our present problem deals with an arbitrary number of processes concurrently executing with varying resource requirements. In contrast, the dining-philosophers problem deals with a fixed number of processes always requesting the same amount of resources (two forks).

3. Detecting Starvation

As we experimented with various unblocking policies we found that the guaranteed-completion method works better than others. As mentioned earlier it is a flow-control technique. Its effectiveness depends mostly on how soon the control is imposed and how quickly it is withdrawn.

The most important issue is to recognize starvation. As soon as the resource allocator recognizes starvation it can impose flow control (ban new processes) and allow starved jobs to make progress. What are the criteria to declare a job starved? The most obvious definition is based on the age of requests. When the age exceeds a preset threshold, the corresponding process is defined as starved. The age itself can be measured in many ways.

(1) In the Dining Philosophers problem, the number of times a hungry philosopher is overtaken by other philosophers can be taken as the age.

(2) The actual duration of wait can be taken as the age (though it is not a good measure in a multiprogramming environment).

(3) Processes can be given sequence numbers when they enter the system or when they make requests, and periodically the resource allocator can check whether any process with an old sequence number is still waiting. (This is the criterion used in our simulations.)

The decision whether to consider flow control is based on the length of the blocked queue of requests. When this length exceeds a threshold the ages of blocked processes are examined, and if there are very old processes (again there is a threshold for calling a job very old), then flow control is imposed. Flow control should be turned off as soon as signs of starvation disappear. For turning flow control off we can use the same age as the criterion. It is desirable to have some lag between the point of turn-on and the point of turn-off so that the flow control is not turned on and off too frequently.

4. Simulation Results

We ran some simulations to measure the performance of the three main unblocking policies First Fit, Guaranteed Progress and Guaranteed Completion mentioned in section 2. The following are the assumptions of our simulation model.

A single processor with a multiprogramming operating system is assumed. The time taken by operating system functions is negligible. Job arrivals follow an exponential distribution with mean arrival time equal to 12.5 minutes. The service time follows an exponential distribution with a mean service time of 10 minutes. The total simulation time is 12500 minutes. To get steady-state values all statistics are reset after the first 2500 minutes. (Thus the results shown in the following tables are for a simulated time of 10000 minutes.) There is no limit on the number of processes that can run concurrently (provided they are not blocked for want of resources). During the lifetime of a process it has an average of five interesting events, which can be requesting resources, releasing resources

and the job completion event. An average of five events is achieved as follows: The inter-event duration is exponentially distributed with a mean equal one fifth of process' lifetime. The type of the event is chosen randomly. For a request event the amount is uniformly distributed over the process' outstanding claim. For a release event the release amount is uniformly distributed over the amount currently held by the process. The claims of processes are uniformly distributed over the total number of resources in the computer system.

For each policy simulated we measured the request waiting times and the length of blocked list. In the following table the last row refers to the Guaranteed-Completion scheme. The parameters *Queuelength* (blocked-list length) and *Agelimit* (process age) are used to adjust flow control. When the length of the blocked list exceeds Queuelength the algorithm checks to see if there are any processes in the blocked list with age greater than Agelimit. If so, a ban on new processes is imposed. The ban is lifted when there are no processes in the blocked List with age greater than Agelimit. For the following simulation, Queuelength is 5 and Agelimit is 15. A process' age is equal to the process id of the latest process minus its own id. (This age determines when the ban on new processes is imposed and lifted).

The three policies are compared in the following table.

Algorithm Used	Request-queue size		Request waiting time		
	Mean	Max	Max	Variance	Mean
FirstFit	**0.77**	**8.0**	**313.5**	**1238**	**22.09**
Progress	**0.94**	**9.0**	**227.6**	**1218**	**24.31**
Completion	**0.71**	**8.0**	**216.2**	**1113**	**20.66**

Varying Queuelength (with Agelimit = 15) in the guaranteed-completion algorithm gave the following results:

Queuelength	flow control engaged time	Request queue		Request waiting time		
		Mean	Max	Max	Variance	Mean
0	1157	0.63	7.0	195.1	851	20.08
1	1157	0.63	7.0	195.1	851	20.08
3	1114	0.65	6.0	155.4	734	19.08
4	830	0.65	6.0	194.4	917	20.53
5	567	0.71	8.0	216.2	1113	20.66
6	406	0.79	8.0	216.9	1210	23.40
7	241	0.78	9.0	209.6	1051	21.35

Varying Agelimit yielded the following results. (For this simulation Queuelength was set to 3)

Agelimit	flow control engaged time	Req Waiting time			Level of MultiProg	
		Max	var	Mean	Max	Mean
4	2077	170.4	363.1	17.07	14	2.409
5	1736	126.7	397.8	15.89	13	2.490
6	1663	140.1	536.0	18.14	13	2.479
8	1816	159.0	658.8	19.80	15	2.538
10	1894	122.7	529.2	20.74	17	2.554
12	1553	133.3	621.6	19.80	17	2.558
15	1114	155.4	734.1	19.08	13	2.467
20	697	210.0	1019.0	20.81	14	2.397

The duration of flow control is affected by the value of Agelimit. For small values of Agelimit (around 4), flow control is imposed very often, and hence the flow-control duration is longer than at a moderate value (about 6). For slightly larger values of Agelimit (say 10) the duration is again longer because it takes longer to lift the control. For very large values of Agelimit (say 20) the duration is much smaller, because flow control is imposed rarely, and unblocking is mostly handled by the default First-Fit policy.

The average wait of blocked processes is also affected by the value of Agelimit. When Agelimit is small, the average wait tends to be small because starvation is recognized sooner, but when it is too small the average can go up because of the batching effect of longer flow control. Consider, for example, the cases of Agelimit = 4 and Agelimit = 5 in the above table. We would expect the case

of Agelimit = 4 to have a smaller average waiting time, but because of the longer flow-control duration (2077 as against 1736) there is a batching effect, and this effect is responsible for higher average wait time. For smaller values of Agelimit the two effects of flow control, namely, recognizing starvation sooner and the batching effect act in opposite directions. At higher values of Agelimit, starvation is not recognized sooner, and even after recognizing it, flow control is not lifted early enough to prevent batching.

5. Conclusions And Future Work

Several policies for starvation control are presented and their relative merits are compared. Some of kind of flow-control technique that sacrifices concurrency only to a limited extent seems to work well. The effectiveness of this technique depends on how the presence or the absence of starvation is recognized. We use a two-step procedure to recognize starvation; first, we watch the length of blocked queue, and when it exceeds a preset value we look for the presence of old processes (as defined by Agelimit) to declare starvation. But recognizing starvation is still an important issue, because what we might consider as starved situation depends on the particular job mix we are dealing with. So we need a method that ensures fairness to both short and long jobs.

Although our guaranteed-completion method seems to have performed well, we think its effectiveness can be increased by considering several other factors in recognizing starvation. For example, the extent to which a process is starved can be measured by a *starvation index*, which is a function of the following.

- The request amount. The greater the request amount the longer the wait may be.
- The maximum claim. Similar to request amount.
- The age of the request. The time that elapsed since the request was made.
- The priority of the process. The higher the proirity the shorter the wait may be.

- The age of the process itself.

Flow control can be imposed when the starvation index for a waiting job exceeds a given threshold.

6. References

[1] E.W. Dijkstra, "Cooperating Sequential Processes," pp. 103-110 in *Programming Languages*, Academic Press, New York (1968) F. Genuys, ed.

[2] R.C. Holt, "Comments on prevention of system deadlocks," *Communications of the ACM 14*, 1, pp. 36-38 (Jan 1971).

[3] H. Madduri and R. Finkel, "Extension of Banker's algorithm for resource allocation in a distributed operating system," *Information Processing Letters 19*, 1, pp. 1-8 (July 1984).

[4] R.O. Fontao, "A concurrent algorithm for avoiding deadlocks in multiprocess multiple resource systems," *Operating Systems Review 6*, 1,2, pp. 72-79 (1972).

[5] D.L. Parnas and A.N. Habermann, "Comment on deadlock prevention (with reply by R.C. Holt)," *Communications of the ACM 15*, 9, pp. 840-841 (Sept 1972).

[6] D.E. Rossi and R.O. Fontao, "A parallel algorithm for deadlock-free resource allocation," *Revista Telegrafica Electronica (Argentina). In Spanish.*, 824, pp. 1062-8 (Nov 1981).

[7] S. Lauesen, "Job scheduling guaranteing Reasonable turn-around times," *Acta Informatica by Springer-Verlag 2*, 1, pp. 1-11 (1973).

[8] E.W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica 2*, 1, pp. 115-138 (1971).

[9] E.W. Dijkstra, "A class of allocation strategies inducing bounded delays only," *AFIPS Spring Joint Computer Conference*, pp. 933-936 (1972).

[10] J.J. Cocu and R.E. Devillers, "On a class of allocation strategies inducing bounded delays only," *The Computer Journal 25*, 1, pp. 52-55 (1982).

[11] P.J. Courtois and J. Georges, "On starvation prevention," *RAIRO Informatique/Computer Science 11*, 2, pp. 127-141 (1977).