SCHEDULING DEPENDENT TASKS FROM AN INFINITE STREAM IN SYSTEMS WITH NONPREEMPTIBLE RESOURCES

Wojciech Cellary

Institute of Control Engineering
Technical Univerity of Poznan
60-965 Poznan , POLAND

ABSTRACT. An infinite stream of independent jobs composed of sets of dependent tasks, which are being fed into a uniprocessor computer system with nonpreemptible resources is considered. A joint approach is presented, to optimization of a given system performance measure, namely mean flow time of jobs, and the solution of the system performance failure problems, namely the determinacy of the set of tasks, deadlock and permanent blocking. For deadlock avoidance the approach is applied which radically reduces overhead involved without losing the benefit of improved resource utilization.

1. INTRODUCTION

In this paper we will consider an infinite stream of jobs which are being fed into a system. The jobs are independent of each other, i.e. there are no precedence constraints put on the order of their execution. However, we will allow every job to be a set of dependent tasks. Thus, from the operating system point of view, we deal with the infinite stream of dependent tasks.

As is well known, this is an important assumption commonly met in practice. First, usually programs associated with a complex job are prepared in the form of subroutines which can be executed in parallel with some precedence constraints. Second, it is possible to detect in a continuous program, some program blocks which can be executed in parallel. This problem was considered in [1,2] . Of course, in general,

the division of a job into tasks improve system efficiency.

We will consider a uniprocessor computer system with many units of nonpreemptible resources of one kind. As is known, in such systems two general problems must be taken into account. The first one is the optimization of a given system performance measure like for example mean flow time. The second one is the solution of the system performance failure problems, namely the determinacy of the set of tasks, deadlock, and permanent blocking.

Commonly, these two general problems are solved separately. On the one hand, there are some approaches to the solution of the particular system performance failure problems [1,2,6,7,8,9,10] which do not take into account any system performance measure. On the other hand, there exist well known algorithms for scheduling tasks on processors tending to optimize a given system performance measure, which either do not take into account any additional resources, ar allow the usage of additional resources under such restrictions that the problem of system performance failures does not arise. These analyses only admit a hierarchical operating system structure. At the lower level, nonpreemptible resources are allocated with regard to system performance failures. Then, at the higher level, tasks with granted additional resource requests are scheduled to optimize a given system performance measure. Let us note, that such approach concerns only the optimal allocation of processor time but not the usage of additional resources.

However, in the last year some papers have appeared dealing with joint approach to the solution of the system performance failure problem and the optimization of a given system performance measure [3,4,5] , that is approaches to all system performance failures which explicitely take into account a given system performance measure. In particular, in [5] algorithms for task scheduling in systems with many nonpreemptible resources of one kind were considered, when an infinite stream of independent tasks is being fed into the system; these algorithms tend to minimize mean flow time. For the solution of the deadlock problem, they use a new approach to deadlock avoidance presented first in [8] and modified in [5] . This paper develops the results obtained in [5], since it concerns stream of dependent tasks, and the minimization of the mean flow time of jobs instead of tasks. As we have mentioned, such model is more realistic in practice and moreover, allows system efficiency to be improved.

In Section 2 of this paper we select approaches to the solution of the peformance failure problems.

In the third Section, some definitions are introduced, and tests used in the deadlock avoidance method are presented. These tests remain generally the same as those presented in [5] , but their presentation is necessary for other parts of this paper.

In the next Section we distinguish all situations which can occur in the system from the resource allocation point of view, and present algorithms for their servicing. The last Section contains some conclusions.

2. SELECTED APPROACHES TO THE SOLUTION OF SYSTEM PERFORMANCE FAILURE PROBLEMS

In this section we will select approaches to the solution of system performance failure problems. Let us start with the problem of the determinacy of the set of tasks.

As is known, the set of tasks is called non-determinate if the results produced by independent tasks depend on the speed and order in which these tasks are executed [7] . The determinacy problem concerns the cases, when independent tasks read from and write to common storage locations. In our system this problem concerns tasks composing one job, since we asume /which is usually done in practice/ that the storage locations reserved by any two jobs are mutually exclusive of each other. The solution of this problem consists in the introduction of the additional, proper precedence constraints among tasks. Algorithms determining which precedence constraints should be added in the set of tasks are presented in [7] . As we have mentioned, in our approach, we assume arbitrary precedence constraints among tasks composing a job. Thus, we allow for the solution of the determinacy problem.

Let us pass now to the deadlock problem. As is well known, deadlock is the system state in which the progress of some tasks is blocked indefitely because each task holds nonpreemptible resources that must be acquired by others in order to proceed. From three general approaches to deadlocks, detection and recovery, prevention, and avoidance, we have selected the avoidance approach since it is characterized by the highest system throughput. Moreover, a cost /i.e., overhead involved/ paid in this approach, which is relatively high can be significantly reduced [8] .

The main idea of deadlock avoidance consists in the application of a so called safety test to examine, on the basis of prior knowledge of the task resource claim, i.e. the strict upper bound on the resource

requests of a task, whether the granting of a given request involves deadlock danger or not. For this examination a so called safe sequence of tasks is created. The number of resources free at any moment /i.e., not allocated to any task/ must be enough to enable the remaining requests of the first task in the safe sequence to be granted. The sum of the resources free and allocated to the first task must be enough for granting the remaining requests of the second task in the safe sequence, and so on. It is proved [9] that if a safe sequence of tasks can be created then no tasks are deadlocked. The safety test consists in simulating request granting and an attempt to create a safe sequence containing a task that requests resources, in order to answer the question whether this request can be really granted or not.

The large overhead involved in avoidance methods results from the necessity to apply the safety test in every case of resource request and almost every case of resource release, and from the number of tasks making up every safe sequence. A way of significantly reducing overhead without losing the benefit of improved resource utilization is the application of the necessary condition for deadlock as an admission test. Such an approach was presented in [8] . The admission test precludes many unsafe resource allocation states and as a result, significantly reduces the number of applications of the safety test. A valuable property of the admission test is that it depends entirely on claims which are constant, and not on numbers of resources allocated to tasks which vary. Thus,application of this test is reduced to the moment when a new task enters the system or a task is completed. The safety and admission tests presented in [8] concern testing resource allocation states. They were modified in [5] to test state transition, and thus testing is reduced to comparison of only two numbers. These tests will be presented in the next section.

The last but not least system peformance failure problem is permanent blocking. As is well known, this problem concerns tasks whose resource requests will never be granted because of a steady stream of requests from other tasks which are always granted before those of a blocked task. In our strategy for solving this problem we will consider the blocking of whole jobs. We asume knowledge of the "blocked-free" time for every job and periodically examine whether the time elapsed since the arrival of a job exceed its blocked-free time. If so, the minimum number of resources is reserved, for the completion of every task composing the blocked job. This approach to the permanent blocking problem may be considered also as an optimization of the secondary system performance measure, namely lateness after blocked-free time.

3. BASIC DEFINITIONS

Every task T in the system will be characterized by the following.

1°. Claim - C(T) - which represents the strict upper bound on resources used simultaneously;

2°. Rank - R(T) - which represents the difference between the claim of task T and the number of resources currently allocated to T;

3°. Priority - $\pi(T)$ - calculated as follows.
Let J(T) be a set of tasks composing a job which contains task T, let S(T) be the set of successors of task T, and let $\tau(T)$ denotes the remaining performance time of task T, i.e., the run-time needed by task T to completion.
Then the priority of task T

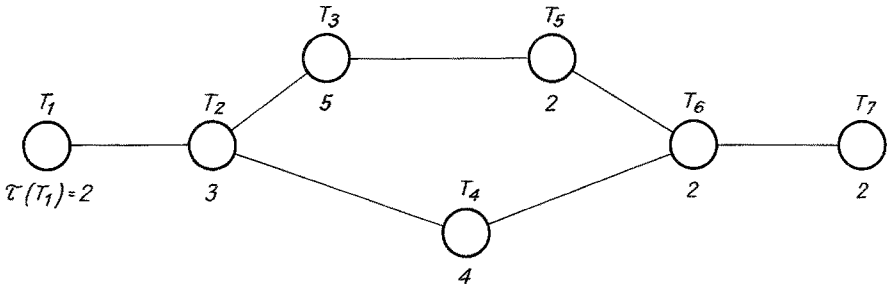
$$\pi(T) = \sum_{T_i \in S(T)} \tau(T_i) + \tau(T) - 10 \times \sum_{T_i \in J(T)} \tau(T_i) \quad .$$

Let us briefly discuss this formula.

The component $- 10 \times \sum_{T_i \in J(T)} \tau(T_i)$ concerns the priorities of jobs, i.e. it decides about the precedence of the choice in the case of tasks that belong to different jobs. In general, the job with the shortest remaining performance time will be taken first for servicing, so that the number of jobs residing in the system will be minimized as well as the mean flow time of jobs.

On the contrary, the component $\sum_{T_i \in S(T)} \tau(T_i) + \tau(T)$ of priority $\pi(T)$ decides about the precedence of the choice in the case of tasks composing the same job, since the values of remaining component are equal for all tasks composing a job. In this case the higher priority is associated with the task whose completion allows, in accordance with precedence constraints, for the starting of the subset of tasks with the greatest, global performance time. An example of task priorities in a job is presented in Fig. 1.

Summarizing, every task in the system will be characterized by its claim, rank and priority. Moreover, we assume that every job in the system is characterized by a blocked-free time which is a time elapsed from job arrival, after which the job is treated as permanently blocked.



point in time	T_i	$-10 \times \sum_{T_j \in J(T_i)} \tau(T_j)$	$\sum_{T_j \in S(T_i)} \tau(T_j)+\tau(T_i)$	$\Pi(T_i)$
job arrival	T_1	-200	+20	-180
completion of T_1	T_2	-180	+18	-162
completion of T_2	T_3	-150	+11	-139
	T_4	-150	+8	-142
completion of T_3	T_4	-100	+8	-92
	T_5	-100	+6	-94
completion of T_4	T_3	-110	+11	-99
completion of T_5	T_6	-40	+4	-36
completion of T_6	T_7	-20	+2	-18

Fig. 1. An example of task priorities.

Now let us divide tasks residing in the system into certain classes.

A task which has at least one resource allocated to it will be denoted by a competitor. A task all of whose predecessors are completed, and which are passed successfully through the admission test, but holds no resources will be denoted by a candidate. Candidates and those competitors whose last resource request did not pass successfully through the safety test will be denoted by waiters. Finally, tasks all of whose predecessors are completed, but which did not pass successfully through the admission test will be denoted by potential candidates.

Let us pass now to the safety and admission tests [5] . First, let us define the promotion of a task T,which consists in allocation a resource to task T and decrementing its rank by one. The allocation state of a task can be described by the number of promotions starting from the fictitious initial state in which no resources are allocated to it /i.e., R(T) = C(T) / up to its current rank. Let the number of resources be equal to t. The allocation state of the system can be stored in a vector denoted by $\bar{x}$, where x_k is the total number of promotions of all competitors from rank = k to rank = k-1 , /k=1,...,t/.

As an example let us consider the following system:

$$t = 6 \qquad \text{CLAIM} = (3,4,5) \qquad \text{RANK} = (2,1,4)$$

then

$$\bar{x} = (0,1,2,1,1,0) \ ,$$

since task T_1 was promoted from rank = 3 to rank = 2; task T_2 from rank = 4 to rank = 3, from rank = 3 to rank = 2, and from rank = 2 to rank = 1; and task T_3 was promoted from rank = 5 to rank = 4.

For the formulation of the safety and admission tests let us define the following vectors:

$$\bar{t} = (t,t-1,t-2,\ldots,1) \ ;$$

$$\bar{q} = (q_1,q_2,\ldots,q_t) \ ,$$

where

$$q_k = t - k + 1 - \sum_{j=k}^{t} x_j \ ;$$

$$\bar{s} = (s_1,s_2,\ldots,s_t) \ ,$$

where

$$s_k = t - k + 1 - \sum_{j=k}^{t} y_j \quad ,$$

and y_j is the number of tasks whose claim = j ;

and pointers :

$$I_q = \begin{cases} \min\limits_{q_k=0} \{k\} & \text{, if there exists such a k that } q_k = 0 \text{ ;} \\ t+1 & \text{, otherwise ;} \end{cases}$$

$$I_s = \begin{cases} \min\limits_{s_k=0} \{k\} & \text{, if there exists such a k that } s_k = 0 \text{ ;} \\ t+1 & \text{, otherwise.} \end{cases}$$

In our example :

$$\bar{t} = (6,5,4,3,2,1) ,$$
$$\bar{q} = (1,0,0,1,1,1) ,$$
$$\bar{s} = (3,2,1,1,1,1) ,$$
$$I_q = 2 ,$$
$$I_s = 7 .$$

The safety and admission tests can be formulated as follows [5] .

Theorem 1. Let the initial state be safe. A state transition is safe if and only if

$$R(T) < I_q$$

where R T is the rank of a task which request a resource.

Theorem 2. Let the initial state be safe. If the state remains safe after the addition of a task T to the set of candidates, then

$$C(T) < I_s$$

where C(T) is the claim of task T.

4. TASK SCHEDULING AND RESOURCE ALLOCATION ALGORITHMS

In this section we present algorithms for all the situations of resource allocation in the system, that require specific servicing. These algorithms, in reference to task scheduling, are generalizations of those in [5] .

Situation 1 : The occurence of a request from a competitor or the arrival of a new candidate.

In this situation we must examine whether the request can be granted safely or not. As result from Theorem 1 this examination consists in the comparison of only two numbers R(T) and I_q . If it is successfull, the resource is allocated, and the following algorithm is performed to calculate the new values of $\bar{q}$, I_q and R(T).

Algorithm 1

```
I_q, i := 1;
while  i ≤ R(T)  do
  begin
    q_i := q_i - 1;
    i := i + 1
  end;
while  q_{I_q} > 0  and  I_q ≤ t  do  I_q := I_q + 1;
R(T) := R(T) - 1;
```

As an example let us consider the system state:

t=6 CLAIM = (4,2,6) RANK = (3,1,4)

then

$$\bar{t} = (6,5,4,3,2,1),$$
$$\bar{q} = (2,1,1,0,0,0),$$
$$I_q = 4 .$$

In accordance with Theorem 1, we can grant only the request of either task T_1 or T_2 . The allocation of a resource to task T_3 could cause deadlock, since even after the completion of T_2, there may be

not enough resource to complete either T_1 or T_3 . After granting the request of T_1 we obtain:

$$\begin{aligned} &RANK = (2,1,4)\,, \\ &\bar{q} = (1,0,0,0,0,0)\,, \\ &I_q = 2. \end{aligned}$$

If a request cannot be granted safely, the competitor is included in the set of waiters and must wait for the release of resources.

The arrival of a new candidate is equivalent to its first request, and as a result to a request from a competitor.

Situation 2 : The release of resources.

In this situation, first, the new resource alocation state must be calculated. We use the following algorithm.

Algorithm 2

Let A be the number of released resources.

```
i := 1;
while  i ≤ R T   do
  begin
    q_i := q_i + A;
    i := i + 1
  end;
I_q, R T  := R T  + A;
while  A > 0  do
  begin
    q_i := q_i + A;
    i := i + 1;
    A := A - 1
  end;
while q_{I_q} > 0  and  I_q ≤ t  do  I_q  := I_q + 1;
```

For example let us consider the release of one resource /A=1/ by task T_3 in the following system:

t=6 CLAIM = (4,2,6) RANK = (3,1,4)

$$\bar{q} = (2,1,1,0,0,0) ,$$
$$I_q = 4 .$$

After the execution of Algorithm 2 we obtain :

$$RANK = (3,1,5) ,$$
$$\bar{q} = (3,2,2,1,1,0) ,$$
$$I_q = 6 .$$

After updating the resource allocation state we should try to grant the requsts of waiters, which up to now could not be granted because of the state transition safety test.

Algorithm 3

1°. Search for waiters with ranks less than I_q. If no such task can be found, stop the algorithm.
2°. Choose the task with maximum $\pi(T)$ and grant its request /use Algorithm 1/.
3°. Repeat from step 1°.

In the first step of Algorithm 3, we search for tasks whose requests can be granted safely. The searching procedure can be improved if we keep the set of waiters ordered in ascending rank order. Let us note that this algorithm is applied only for waiters and not all the tasks whose precedence constraints are fulfilled. Thus, the number of examined tasks is greatly reduced. Moreover, it is easy to prove that this algorithm will have to repeated at most as many times as the number of released resources.

Situation 3 : The arrival of a new potential candidate.

In this situation we must check whether the new potential candidate can be moved to the set of candidates or not, using the admission test. Moreover, if the new potential candidate did not pass successfully through the admission test, we will try to replace a candidate, with an appropriate claim and lower priority, in the set of candidates by a new potential candidate. Such replacement is beneficial for mean flow time, and is always possible since candidates do not have resources allocated to them.

For servicing of this situation, we will formulate two admission tests: the first calculated for the set of competitors $/\bar{s}$, $I_s/$ and the second for both competitors and candidates $/\bar{s}^*$, $I_s*/$. The use of these tests will be presented in Algorithm 5. However, first let us formulate an algorithm used for updating values of $\bar{s}$ and I_s after addition/deletion of a task to/from the set of competitors as well as $\bar{s}^*$ and I_s* after the addition/deletion of task to/from the set either of competitors or candidates.

Let T be the new potential candidate.

Algorithm 4

Boolean variable AD equal to true denotes addition, false - - deletion;

I_s, i := 1;
while i $\leqslant$ C T do
  begin
    if (AD) then $s_i := s_i - 1$ else $s_i := s_i + 1$
  end;
while $s_{I_s} > 0$ and $I_s \leqslant t$ do $I_s := I_s + 1$;

The algorithm for the admission of a new potential candidate is as follows.

Algorithm 5

1°. If $C(T) < I_{s*}$, move T to the set of candidates and use Algorithm 4 for the calculation of the new values of $\bar{s}*$ and I_{s*} ; then stop the algorithm.
2°. If $C(T) \geqslant I_s$ stop the algorithm.
3°. From among candidates with claims greater than or equal to C T find task T_r with priority $\pi(T_r)$ at the minimum. If no such task can be found, or $\pi(T_r) \geqslant \pi(T)$ then stop the algorithm; otherwise replace T_r by T in the set of candidates and update the values of $\bar{s}*$ and I_{s*} using Algorithm 6. Then stop the algorithm.

Algorithm 6

```
i := C(T) + 1;
while i ⩽ C(T) do
  begin
    s*_i := s*_i + 1;
    i := i + 1
  end;
I_s* := i;
while s_{I_s*} > 0 do I_s* := I_s* + 1;
```

Situation 4 : The completion of a competitor.

In this situation we should:

1°. Update values of $\bar{q}$, I_q, $\bar{s}$, I_s, $\bar{s}^*$, I_{s^*} using Algorithms 2 and 4.

2°. Try to move a potential candidate to the set of candidates /it can be proved that there will be at most one such potential candidate/ - using Algorithm 7.

3°. Add the tasks whose precedence constraints are now fulfilled to the set of potential candidates - using Algorithms for servicing Situation 3.

4°. Try to allocate resources released by the completed competitor using Algorithms for servicing Situation 2.

Algorithm 7

1°. Search for a potential candidate T with $C(T) < I_{s^*}$ and with $\pi(T)$ at the maximum. If no such task can be found, stop the algorithm.

2°. Move T to the set of candidates and use Algorithm 4 for calculating new values of $\bar{s}^*$ and I_{s^*}. Then, stop the algorithm.

Situation 5 : The detection of a permanently blocked job.

In this situation we must activate a special strategy of resource allocation to grant the requests of tasks composing the blocked job. This strategy consists in the division of system resources into two

parts: the one part ensures the completion of a blocked job, and the remaining resources are used for servicing requests from other jobs. The servicing of the blocked job and remaining jobs is independent, and performed in accordance with Situations 1 to 4 but for a reduced number of resources.

The first problem in Situation 5 which can be distinguish is the determination of the number of resources necessary for the blocked job. It can be shown that this number is equal to the maximum rank of tasks which are not yet being completed, composing the permanently blocked job. Let us note that the rank of a task which is not a competitor is equal to its claim. This number should be updated after the completion of every task composing the blocked job, since it should be at the minimum.

The next step of the permanent blocking prevention strategy consists in waiting for the completion of all competitors. Then, as was mentioned above, the resources are divided and servicing of the blocked job is performed independently.

It could seem that the completion of all competitors should not be necessary. However, we want to preclude a situation in which the remaining after division resources are allocated to tasks which cannot complete because of the lack of resources, and thus block these resources during permanently blocked job servicing.

Let us note that the strategy presented here is less conservative than Holt's well known strategy [10] , since it allows the parallel servicing of the blocked and non-blocked jobs.

5. CONCLUSIONS

The two general problems of task scheduling in systems with non-preemptible resources: first, the optimization of a given system performance measure, and second the solution of the system performance failures problems /determinacy of the set of tasks, deadlock and permanent blocking/, are commonly solved separately. The algorithms presented in this paper represent a joint approach to both of these problems, which may lead to better results, being obtained. They deal with the case of infinite stream of dependent tasks and thus concern a wide class of practical situations. The algorithms are presented in the form which allows for direct implementation in operating systems.

REFERENCES

1. Baer, J.L., A survey of some theoretical aspects of multiprocessing, Computing Surveys vol. 5, No 1, 1973.
2. Bernstein, A.J., Analysis of programs for parallel processing, IEEE Trans. Comp. vol EC-15, 1966, No 5.
3. Cellary, W., On resource allocation policies in uniprocessor systems with nonpreemptible resources, MTA SZTAKI Tanulmanyok 69, 1977.
4. Cellary, W., Resource allocation strategies in computer systems with nonpreemptible resources, Foundations of Control Engineering, vol. 2, No 3, 1977.
5. Cellary, W., Task scheduling in systems with nonpreemptible resources, in: H. Beilner and E. Gelenbe, Medelling and Performance Evaluation of Computer Systems, /Proc. of the III International Symposium/, North Holland Publishing Co., 1977.
6. Coffman, E.G., Jr., M.J. Elphick, A. Shoshani, System deadlocks, Computing Surveys vol. 2, No 3, 1971.
7. Coffman, E.G., Jr., P.J. Denning, Operating Systems Theory, Prentice Hall, Englewood Cliffs, N.J., 1973.
8. Habermann, A.N., A new approach to avoidance of system deadlocks, Revue Francaise d'Automatique, Informatique et Recherche Opérationelle 9. sept B-3, 1975
9. Habermann, A.N., Prevention of system deadlocks, Comm. ACM vol 12, No 7, 1969.
10. Holt, R.C., Comments on prevention of system deadlocks, Comm. ACM, vol. 14, No 1, 1971.