Acta Informatica 2, 1—11 (1973)
© by Springer-Verlag 1973

Job Scheduling Guaranteing Reasonable Turn-Around Times

Søren Lauesen

Received September 3, 1972

Summary. This report describes the algorithm for job scheduling and resource allocation used in the operating system Boss 2 for RC 4000. Most resources in the system are nonpreemptible, which causes the usual deadlock problems. The algorithm gives modest jobs a short turn-around time and more greedy jobs a correspondingly larger turn-around time. All jobs are guaranteed a finite turn-around time even if an infinite stream of other jobs is fed to the system (i.e. Holt's permanent blocking [7] is prevented). An estimate of the expected finishing time is computed when the job is enrolled. The estimate is updated continuously and is available from all terminals. The algorithm is a modification of the Banker's Algorithm described by Habermann [5]. It pays high attention to turnaround time and less attention to resource utilization.

1. Introduction

The Boss 2 operating system for RC 4000 handles batch jobs, on-line editing, remote job entry, time sharing jobs, and process control jobs [8–10]. It is implemented for the RC 4000 computer manufactured by Regnecentralen, Copenhagen. The system works under a modified and extended version of the monitor described in [1, 6].

A job running under Boss 2 may use the following types of resources: Disc space, drum space, tape stations, special devices (punch, process control device, etc.), core store, CPU-time, and certain buffers and catalog entries.

Boss allocates the available resources among the jobs enrolled for the moment according to a strategy to be explained in the sequel. The resources are *exclusive* in the sense that a resource allocated to one job cannot be allocated to another job simultaneously. In some cases, access rights to files should be considered a resource too [2]. However, in Boss such rights may be allocated to any number of jobs at the same time, leaving the control of the sharing to the jobs. Hence we disregard this "resource" in the scheduling of exclusive resources.

Resources may be classified as preemtible, temporary, or permanent in the following way:

Preemptible resources may be allocated to a job and withdrawn again at any time without acknowledgement from the job. In Boss core store and CPU-time are handled as preemptible resources. The core store of the job may be withdrawn by means of swapping, the CPU-time is constantly multiplexed between the jobs and the operating systems.

Temporary resources allocated to a job may only be withdrawn at job termination or when the job orders it explicitly. The bulk of the resources are handled in this way. Some of them (e.g. tape stations) might be handled as preemptible resources, but this is not done because of the overhead involved.

Permanent resources are allocated to projects or users by the computer staff. Unused permanent resources cannot be borrowed by a job from another project, and thus the allocation algorithm is straightforward. The permanent resources comprise part of the disc and part of the file catalog.

The main problem in the resource allocation is to handle the temporary resources in such a way that *deadly embrace is avoided* and *reasonable turn-around times are obtained for all jobs*. Temporary resources are to some extent reserved and released dynamically by the the jobs, which improves the possibilities for resource utilization. Typically, a job needs very few resources for a long initial period (spooling of input files, waiting for bulk of resources) and for a long terminal period (spooling of output files).

The deadly embrace problem arises because a job may refuse to release it's temporary resources until it has reserved further resources. In order to solve the problem at all, we need a predefined set of *claims* for the job, i.e. a strict upper limit on the resources demanded by the job. The deadly embrace problem may then be put as follows: A request exceeding the claims is refused. Any other request for temporary resources must be granted in a finite time.

In the literature the problem has occurred in two versions:

Habermann's deadly embrace problem [5]: The request must be granted in a finite time after stopping the input stream of jobs (corresponding to close down at night).

Holt's permanent blocking problem [7]: The request must be granted in a finite time even if a steady stream of input jobs is submitted.

The Boss algorithm solves the problem in the stricter Holt version, and in such a way that modest jobs will have a short turn-around time, greedy jobs a correspondingly longer turn-around time. An estimate of the finishing time is computed when a job is enrolled, and it is updated whenever changes in the schedule occur. The latest estimate is available from any terminal. It is rather precise in practice, with a tendency to be somewhat pessimistic.

In Sections 2 to 4 below we explain our concepts of a reasonable turnaround time. In Sections 5 to 7 we describe the algorithm to avoid Holt's permanent blocking and estimate the turn-around times.

2. Job Priorities

To each job j is associated a priority function which defines the job priority $p_j(t)$ at time t. We assume that the user is interested in the *final priority* only, i.e. $p_j(t)$ at $t =$ job termination. The larger the final priority, the more dissatisfied is the user. One part of the scheduling problem may now be formulated as the following bottleneck problem: *Keep the largest final priority as low as possible.*

In principle any increasing function might be used as a priority, but for the sake of simplicity Boss uses functions of this form:

$$p_j(t) = \begin{cases} (t - a_j)\, u_j / b_j & \text{for } t - a_j < M_j \\ (t - a_j)\,\text{large} & \text{for } t - a_j \geqq M_j. \end{cases}$$

Here, a_j is the arrival time of the job, b_j is the expected run time of the job, and u_j is a constant associated with the user in question. The constant *large* is larger than all values of u_j/b_j. M_j specifies a maximum turn-around time and may be stated by the user within certain limits (most users have a very high lower limit on M_j).

The first part of the function $(t - a_j < M_j)$ defines the final priority in all normal cases. It expresses that dissatisfaction is proportional to turn-around and inversely proportional to expected run time.

Assume that all jobs in a certain period terminate with nearly the same final priority. The priority functions chosen for Boss will then make the turn-around times nearly proportional to the run time of the job (except when M_j becomes significant). This reflects our wish to encourage short jobs by means of the best payment to programmers: short turn-around times.

3. Optimal Job Sequence

Assume that the system at a given moment "now" has to execute n jobs with these remaining run times:

$$r_1\, r_2 \dots r_n.$$

Suppose for a moment that we want to execute these jobs strictly sequentially. The following algorithm determines the *priority sequence*, which is the optimal sequence of execution in the sense of making the largest final priority as low as possible:

Step 1. Determine the time T when the last job is finished:

$$T = \text{now} + r_1 + r_2 + \cdots + r_n$$

Step 2. Choose the job j with the lowest value of $p_j(T)$ and let this job be executed last. Disregard the job in the rest of the algorithm.

Step 3. Repeat from step 1, working on the remaining jobs until none are left.

Fig. 1 illustrates this algorithm. Below we prove that the priority sequence determined by the algorithm is optimal under these conditions:

1. All priority functions are increasing.
2. All job sequences are possible.
3. All sequences have the same total execution time $r_1 + r_2 + \cdots + r_n$.
4. All run times are known in advance.

Only condition 1 is completely fulfilled in practice.

Condition 2 is not fulfilled if some job has reserved temporary resources already, as this may prevent jobs wanting the same resources from being first in the sequence. Section 4 explains how the effect of this may be reduced, section 5 explains how a more realistic sequence is computed.

Condition 3 is not completely fulfilled, especially not if multiprogramming is used. Boss executes the first few jobs in the sequence simultaneously in core, hoping in this way to reduce the total execution time. Unfortunately, such multiprogramming may increase the total execution time instead, for instance if

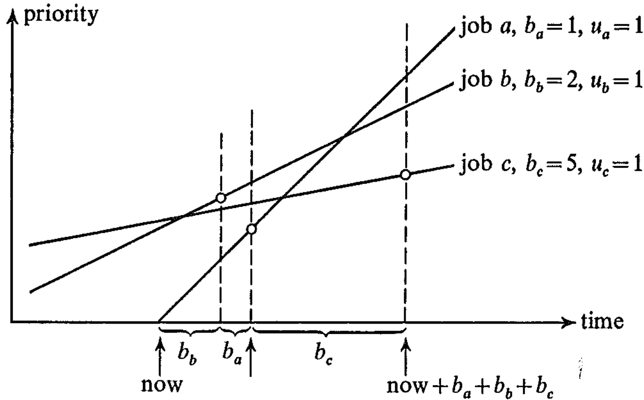

Fig. 1. The priority functions for three jobs a, b, and c are shown. The jobs have remaining run times b_a, b_b, and b_c and thus the latest job is completed at time now $+b_a+b_b+b_c$. The job to run last is determined as the job with the lowest priority at that moment. The algorithm is then repeated for the remaining jobs

two jobs use the same disc simultaneously, each job working on one cylinder only. Nevertheless, it is generally believed that multiprogramming on average reduces the execution time by a certain factor. If this is true, condition 3 becomes reasonable if all r_j are multiplied by this factor. Section 9 elaborates on the subject.

Condition 4 assumes that all jobs have a correct run time specification, but in practice we only have an upper limit. To compensate for this we compute a new schedule whenever a job is finished. The run time specification does not include waiting time for teletype *i/o*, for operator to mount tapes, etc. Various precautions are taken to ensure that such waiting does not wreck the schedule. For instance the user must specify (implicitly or explicitly) the total resources on backing store needed for spooling of teletype output.

The proof of the algorithm goes as follows: Assume that $j_1 j_2 \ldots j_n$ is the priority sequence computed. Let j_a be the job with the largest final priority and let T_a be the termination time of j_a. Now consider some other job sequence. If j_a terminates later than T_a, condition 1 implies that this sequence will have a larger maximal final priority. If j_a terminates before T_a, at least one of the jobs $j_1 j_2 \ldots j_{a-1}$ will terminate at T_a or later (because of condition 3). But because of step 2 of the algorithm, this job will have a final priority $\geqq p_{j_a}(T_a)$. Thus, no sequence is better than $j_1 j_2 \ldots j_n$.

4. Dispersion Bounds

A typical situation preventing a short turn-around time for a modest job is this: Assume that job_2, job_3, ... are long running jobs, which have reserved nearly all temporary resources. Now the short job_1 enters the system, and according to the priority sequence it should run first. But as insufficient resources are left, job_1 has to await the completion of some long job.

We have chosen the following solution to this problem: Let M_t be the set of all jobs with a run time larger than t. For each type of resource r let the total

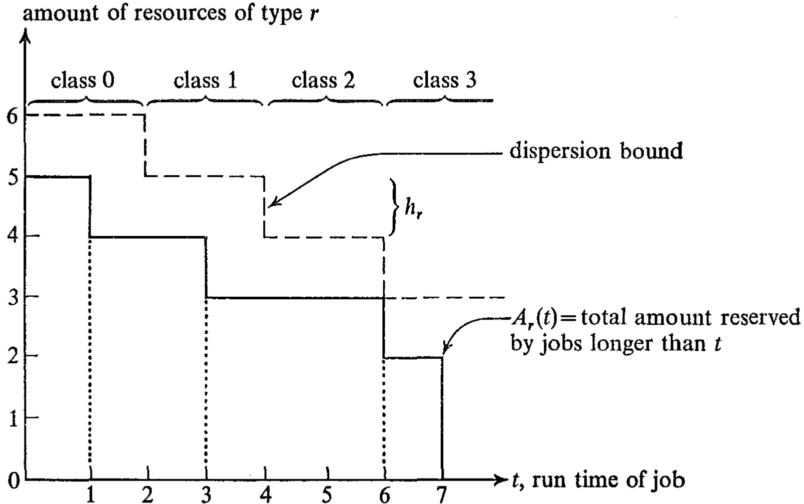

Fig. 2. Illustration of dispersion bound and time classes. $A_r(t)$ corresponds in this case to 4 jobs with run times 1, 3, 6, 7, and having reserved 1, 1, 1, 2 units of resource r

amount reserved by jobs in M_t be $A_r(t)$. A_r is a decreasing function with an appearance as in Fig. 2.

The resource allocation introduces an upper boundary function on A_r. The boundary function prevents long running jobs from spreading their possessions over too many resources, and hence we call the function a *dispersion bound*. In Boss we use a step function with 4 levels (dashed in Fig. 2). The step height is called h_r. The step function also classifies the jobs according to run time in four classes, class 0 being short jobs and class 3 very long jobs.

Now, consider a job from class 0 with a demand of resources $\leq h_r$. Resources occupied by jobs from class 1 to 3 cannot effect our job, which may be executed according to the pirority sequence except for delays caused by other jobs from class 0. If it had a demand less than $2h_r$, delays from class 0 and class 1 jobs would be possible. Similar results hold for jobs in other time classes.

Thus the dispersion bounds define our notion of modest and greedy jobs: A modest job has short run time *and* low claims of temporary resources. The goal of the resource allocation is now to ensure that modest jobs will have short turn-around time.

A side-effect of the dispersion bounds is that jobs with a long run time may claim a fraction of the resources only, while very short jobs may claim the entire computer.

5. Feasible Job Sequence

Any job requesting a preemptible resource (e.g. core store) will get it unless a preceding job in the priority sequence uses it for the moment. This means that if the jobs used preemptible resources only, they might be executed in the optimal sequence.

The following algorithm computes a *feasible job sequence* which takes into account the temporary resources and which is as close as possible to the optimal sequence. The feasible sequence is the basis for granting requests for temporary resources and for computing the expected finishing time reported to the user.

Step 1. Let the *potential resources* be the temporary resources free for the moment. Let the feasible job sequence be an initially empty list.

Step 2. Search the jobs in priority sequence and find the first job for which the claims may be fulfilled by means of the potential resources.

Step 3. Let this job be the next in the feasible job sequence. Add the resources held by this job to the potential resources (thus simulating the completion of the job). Disregard the job in the rest of the algorithm.

Step 4. Repeat from step 2 until no jobs are left.

This algorithm is the Banker's algorithm for determining whether a situation is safe [5]. In step 2 we might imagine that no job can have its claims fulfilled. This would be Habermann's Deadly Embrace; but if just one feasible sequence exists, the algorithm will find it (proof in [5]). However, at least one feasible sequence exists because of the way requests are granted (Section 6) and because no job may have claims exceeding the total resources of the system (strictly: the dispersion bounds).

It should be obvious that the sequence in fact is feasible, as the algorithm just simulates that the jobs are executed one by one, each job releasing all of its temporary resources when it terminates. An estimated finishing time for a job is computed as the sum of the run times for the job and the jobs preceding it in the feasible sequence. If no jobs are enrolled later, this estimate is pessimistic because many jobs terminate earlier than expected. Jobs enrolled later may cause the estimate to be optimistic, but the user may at any time ask the system for the latest estimate.

The algorithm works on resource vectors with one component to each type of resource. The operations on such vectors are addition (step 3) and comparison (step 2), defined straightforward like this:

$$\boldsymbol{a}+\boldsymbol{b}=(a_1+b_1, \ldots, a_n+b_n)$$

$$\boldsymbol{a}\leqq\boldsymbol{b}\Leftrightarrow\forall i\ (a_i\leqq b_i).$$

However, because of the dispersion bounds and the time classes of Section 4, Boss works with matrices of resources. Each time class corresponds to a row of the matrix, the row containing a resource vector as above:

$$A=(a_{ij}),\quad i=0, 1, 2, 3 \text{ (time classes)},\quad j=1, 2, \ldots, n.$$

When a job in time class c holds resources r, they will be treated as this matrix:

$$A=(a_{ij}),\quad \text{where } a_{ij}=r_j \quad \text{for} \quad 0< i\leqq c,\quad a_{ij}=0 \quad \text{for} \quad c< i\leqq 3.$$

Addition and comparison is now done element by element like this:

$$A+B=(a_{ij}+b_{ij}),\quad A\leqq B\Leftrightarrow\forall ij\ (a_{ij}\leqq b_{ij}).$$

This means that resources of a job in time class c are considered borrowed from the resource pools of time classes $0, 1, \dots c$. The matrix describing the total available set of resources has a row corresponding to each of the four plateaus of the dispersion bound (Fig. 2).

6. Granting Requests

When the feasible sequence has been determined, Boss grants requests and allocates resources in this way:

Step 1. Let the *available resources* be the temporary resources free for the moment.

Step 2. Examine the next (first) job in the feasible sequence: If the job requests resources now and if the available resources are sufficient, grant the request and reduce the free resources and the available resources correspondingly.

Step 3. If the job has not yet requested all resources claimed by it, then reduce the available resources by the amount not yet requested.

Step 4. Repeat from step 2 until all jobs in the sequence have been examined.

This algorithm clearly avoids Habermann's Deadly Embrace, because a job will have its request granted only if the preceding jobs can reserve all resources claimed by them (step 3). Thus the sequence is still feasible, so that the algorithm of Section 5 will work properly the next time. If a new job with legal claims is enrolled, it will always be possible to make a feasible sequence by extending the present sequence by the new job.

Step 3 of the algorithm—which holds back all resources claimed by the job—is unnecessary strict if only Habermann's Deadly Embrace is to be avoided. It would be possible to grant more low priority requests if step 3 reduced the available resources only by the amount needed to prevent Deadly Embrace. However, the Boss algorithm is designed to avoid Holt's permanent blocking under certain reasonable conditions to be explained below. We tried for a long time to find a less strict algorithm which also avoided permanent blocking and which had a simple uniform appearance like the one above—but in vain. In Section 7 we show such a promising but wrong attempt and a correct solution with improved resource utilization.

The algorithm as it stands pays greater attention to justice in turn-around time than to efficient resource utilization. A useful property of the algorithm is that the sequence stays feasible after reservations of resources, so that a new feasible sequence must be computed only after changes in claims (i.e. when jobs leave or enter the system).

We will now prove that *jobs with a sufficiently high priority are executed sooner or later*. Precisely we will prove this:

Theorem 1. A job J which after time T precedes all other jobs in the priority sequence will eventually terminate.

Proof. Though J precedes all other jobs in the priority sequence, a non-empty set $P(t)$ of jobs may precede J in the feasible sequence. The set $P(t)$ will contain

some jobs which hold resources claimed by J. Let $Q(t)$ be the remaining jobs i.e. those which follow J in the feasible sequence.

Because of step 3 of the granting algorithm, jobs in Q will not be granted resources claimed by J or P-jobs. Because the sequence is feasible, P-jobs will either execute to the end or move to the Q-set. Q-jobs will never have to move to the P-set. When all P-jobs have disappeared, J will run to the end.

Next, we will prove that the priority functions of section 2 prevent Holt's permanent blocking. Assume that a set of jobs are never executed. Let J be the earliest submitted job in the set. Wait until all jobs enrolled prior to J are completed and wait further—if necessary—until the moment $a_J + M_J$. From then on the priority functions will cause the conditions of theorem 1 to be fulfilled, which causes a contradiction.

However, this result is not satisfactory as we wish to keep M_j very large. The first part of the priority function $(t - a_j < M_j)$ is the important one in practice and yet does not enter the proof. At present I search for a proof with weaker conditions on the priority functions, perhaps something like all functions $\to\infty$ and all functions have a common bound on their steepness. This must be combined with restrictions on the total execution time of all jobs enrolled for the moment. A better safe estimate of the completion time will also needed.

7. A Wrong Granting Algorithm, and an Improved One

For a few days we believed that the following granting algorithm would prevent permanent blocking. It resembles the algorithm of Section 6, but in step 3 it holds back only the resources wanted by the job for the moment and resources needed to prevent Deadly Embrace:

Step 1. Let the *available resources* be the temporary resources free for the moment, and let the *released resources* be 0.

Step 2. Exactly as in Section 6.

Step 3. Reduce the available resources by the amount wanted by the job now, but not granted. Cover the resources claimed by the job but not yet requested in the following way: Use part of the released resources first. If insufficient, then use part of the available resources. Increase the set of released resources by the resources claimed by the job, thus simulating the completion of the job.

Step 4. Repeat from step 2 until all jobs in the sequence have been examined.

Obviously, this algorithm utilizes more of the free resources. The reader is invited to construct a counter-example which exhibits the permanent blocking (a hint may be found in Holt's reply in [4]).

This wrong algorithm may be mixed with the algorithm in section 6 and yield a correct algorithm with improved resource utilization. All what is necessary is to follow section 6 up to and including the first job in the priority sequence, and the algorithm above for the remaining jobs in the feasible sequence. That permanent blocking is still prevented follows from the proof of theorem 1 which works

without modifications. Step 3 may be relaxed even more, as the wanted resources need not be deduced from the available resources. Furthermore, the strict part from section 6 need only be invoked when the priority of the first job in the priority sequence becomes critical in some sense.

8. Overload

C. A. R. Hoare has drawn my attention to the overload problem occurring when the Banker's algorithm is utilized fully: So many resources are granted away that very few feasible sequences remain, and as a consequence the jobs will be executed sequentially one by one.

Hoare proposed that a limit was put on the sum of the claims of the jobs with nonzero allocations. Another solution is to define that at any time N jobs should be able to have all their claims fulfilled simultaneously. This would make it likely that several jobs could reside in the core store simultaneously and enjoy the multiprogramming. The Banker's Algorithm is a special case of this—employing $N=1$.

The implementation with $N>1$ could be built into the algorithms above. For instance, determination of the feasible sequence (Section 5) could be modified in step 3 so that the resources held by the job were not added to the potential resources. Instead, resources, of a job scheduled earlier (if any) would be added. Other minor changes are needed to assure that the actual degree of concurrency is between 1 and N, and as close to N as possible.

The overload problem has not been felt under Boss because of the dispersion bounds and the strict algorithm of Section 6.

9. Improvement of the Turn-around Prediction

In the preceding sections we have argued as if the job execution was sequential. This may be ok for batch processing—even with several jobs executing in core store simultaneously, but if several time sharing jobs are executed in parallel, they will complete much sooner than stated by the sum of their run times (measured as teletype time). In general this is the case when jobs are bound by different processors (i.e. peripheral devices, in this case the teletypes). Batch jobs are usually bound by the common processors: CPU, drum, disc. Thus they fulfil condition 3 of Section 3 more closely.

In fact the prediction algorithm of Section 5 will give poor results when time sharing jobs are present, as these are often in the beginning of the feasible sequence. The calamity is cured if the feasible sequence is used to simulate more accurately the future job execution, in this way computing better estimates. Such a simulation is implemented in Boss and is based on the granting algorithm and two expected run times for each job: the net run time stating the demand for the common processors and the gross run time which includes waiting time for teletype, operator actions, etc. The simulation assumes that the net run time parts of the jobs are executed sequentially and the gross run times may overlap. Resources are simulated to be released after completion of the gross run time.

10. Temporary Resources not Released

For practical reasons, some resources are not released when a job terminates. Instead they are transferred to a special purpose job which is expected to release them later. In Boss this is done with backing store areas which are to be printed after the job termination (a kind of spooling), and with accounting information which is to be collected and processed later.

These details are simulated in the algorithm for computing the feasible sequence. As a by-product we get reasonable moments for starting the special purpose jobs, i.e. as late as possible, but before deadly embrace and other influences on the normal execution occurs.

11. Implementation and Evaluation

The scheduling algorithm was designed and implemented by the author during 1971. The algorithm occupies about 2000 instructions of which the 1000 implement what has been described here.

The Boss 2 operating system was designed and implemented in the period 1970 to 1972 with an effort of 8 man-years, and it consists of 25 000 instructions. The system was released to costumers in August 1972, but two installations had used it experimentally since April 1972. All installations needed help to trim the dispersion bound, but after that the resource allocation performed satisfactorily and we have had no complaints about turn-around time—not even from heavily loaded installations where the common processors (CPU and disc) are busy 22 hours a day serving jobs (overhead disregarded). I believe that the main reason for this is the estimated finishing times. Users are at ease when they know what to expect and they can better stand a long turn-around time.

At present we develop programs to measure the resource utilization and the accuracy of turn-around prediction, but results are not yet available.

The resource allocation algorithm is executed about 4 times a job. The run time of the algorithm depends upon the number of jobs in the queue, for 20 jobs it is about 30 ms (one machine instruction is about 4 microsec). This should be compared to the basic time to execute a job: about 15 jobs can be executed a minute, each job consisting of translation and execution of a small program.

Acknowledgements. Inspirating discussions with Karolyi Simonyi in 1966 led me into the algorithm for computing the optimal job sequence. The proof of this algorithm is due to Per Mondrup. Collaboration and fruitful discussions with Jørn Jensen and Per Mondrup in 1971 are a main source of my construction of the allocation algorithm. I would like to thank C. A. R. Hoare for many valuable suggestions, which also caused me to improve the strict algorithm of Section 6.

References

1. Andersen, P. L.: Monitor 3, RCSL No: 31-D109, Copenhagen, Regnecentralen, 1972.
2. Bernstein, A. J., Shoshani, A.: Synchronization in a parallel-accessed data base, Comm ACM **12**, 604–607 (1969).

3. Dijkstra, E. W.: A class of allocation strategies inducing bounded delays only, 1972 Spring Joint Computer Conference, p. 933–936, AFIPS Press.
4. Habermann, Y. N., Parnas, D. L.: Comment on deadlock prevention method with a reply by R. C. Holt), Comm ACM **15**, 840–841 (1972).
5. Habermann, A. N.: Prevention of system deadlocks, Comm ACM **12**, 373–377, 385 (1969).
6. Hansen, P. B.: The nucleus of a multiprogramming system, Comm ACM **13**, 238–241, 250 (1970).
7. Holt, R. C.: Comments on prevention of system deadlocks, Comm ACM **14**, 36–38 (1971).
8. Lauesen, S.: Boss 2, User's Manual, RCSL No: 31-D108, Copenhagen, Regnecentralen, 1972.
9. Lauesen, S.: Boss 2, Operator's Manual, RCSL No: 31-D123, Copenhagen, Regnecentralen, 1972.
10. Lauesen, S.: Boss 2, Installation and Maintenance, RCSL No: 31-D191, Copenhagen, Regnecentralen 1972.

Søren Lauesen
A/S Regnecentralen
Falkoner alle 1
2000 Copenhagen F
Denmark

Universitätsdruckerei H. Stürtz AG Würzburg
Printed in Germany