Abstract:

The purpose of the paper is to describe a simulation of a multiprogrammed paging system, and to report on a continuing series of experiments concerning the problem of thrashing, i.e. of excessive paging due to over-enthusiastic multiprogramming of core storage.

Thrashing in a Multiprogrammed Paging System

by

A. Alderson
W. C. Lynch
and
B. Randell

TECHNICAL REPORT SERIES

Number 26
October, 1971

Series Editor: Dr. B. Shaw

1969 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
ALDERSON, Albert

Thrashing in a Multiprogrammed Paging System

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory 1971.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, No. 26)

12"

Added entries

LYNCH, William C
RANDELL, Brian


Suggested classmarks (primary classmark underlined)

Library of Congress:
Dewey (17th):
U.D.C.: 621.381952
681.322.02

Suggested keywords

DEMAND PAGING
MULTIPROGRAMMING
OPERATING SYSTEMS
THRASHING

Abstract

The purpose of the paper is to describe a simulation of a multiprogrammed paging system, and to report on a continuing series of experiments concerning the problem of thrashing, i.e. of excessive paging due to over-enthusiastic multiprogramming of core storage.

About the Author

Mr. A. Alderson is a Ph.D. student in the Computing Laboratory of the University of Newcastle-upon-Tyne.
Professor Lynch is associate Professor of Computer and Information Science, Case Western Reserve University, U.S.A., and during 1970-71 was a Visiting Professor in the Computing Laboratory of the University of Newcastle upon Tyne. Professor Randell has been Professor of Computing Science in the Computing Laboratory of the University of Newcastle upon Tyne since 1969, before which he was at the I.B.M. Watson Research Centre, U.S.A.
1. **Introduction**

One of the problems facing the designer of a paging system that uses multiprogramming in order to overlap processing with input/output activity is that of avoiding page thrashing [1,2]. If too many programs are allowed to compete for a share of working storage, they will be unable to obtain sufficient storage, and will suffer from very frequent page faulting. As a result the system will achieve very low performance, coupled with an excessive amount of page transfers.

It is not usually satisfactory to choose a constant level of multiprogramming, since such a level, if low enough to avoid any chance of thrashing occurring, will not necessarily give sufficient I/O overlap. On the other hand if the multiprogramming level is allowed to vary, unless care is taken the response of the system to the onset of thrashing will be to bring pages from yet further programs down into working storage, and thus accentuate the thrashing.

Techniques for dynamically controlling the level of multiprogramming have been described by Shils [3] and Denning [2]. Both techniques involve an explicit feedback mechanism. The 'Load Leveller' described by Shils takes decisions based on CPU utilization and paging rate, whereas the scheme outlined by Denning involves an attempt at obtaining an explicit estimate of the size of each job's working set. (In this usage the term 'working set' is intuitively the set of pages which a job requires in order to progress without undue paging.)

The purpose of this paper is to describe some of the results obtained in an investigation of the problem of thrashing, and in particular the experiments conducted on several simple algorithms for controlling multiprogramming level by means of implicit feedback mechanisms.

2. **The System Model**

The computing system that has been simulated consists of a processor, core storage and a paging drum. Each job is modelled as an alternating sequence of intervals of processing punctuated by page faults and of waiting (for unmodelled I/O activity). In our model we regard paging activity to be concerned with the drum and input/output to be concerned with disks. However, we do not model contention for I/O
devices nor do we require that any of the jobs' pages should be regarded as I/O buffers and so be required to be present in core during I/O activity. Most paging simulators keep track of each individual page, and are either capable of actually executing programs, or are driven by address traces previously gathered from such programs. Such simulators are extremely laborious even for modelling the behaviour of just a single program. Instead our model keeps track of only the numbers of pages that each job has, and uses appropriate probability distributions to simulate the status of the pages. The amount of processing that a job will achieve before it suffers a page fault is calculated from a probability function which has as a parameter the number of pages that the job currently has in core storage. This has proved to be a most convenient level of simulation, being far less demanding in terms of code, storage and execution time than the address trace level simulations, and has enabled us to perform a large number of experiments modelling lengthy periods of system operation.

The form of the probability distribution of the time to next page fault is based on published data, most notably that gathered on the M44/44X system [4], and attempts to model two distinct aspects of the behaviour of programs running in a paging environment. The first of these concerns the relationship that has been observed to hold between page fault rate and amount of core storage that a job is allowed to use. The fault rate remains at a quite reasonable level as the number of real core pages is reduced until a critical point is passed, when the fault rate rises very rapidly. Above this point the program is capable of retaining its 'working set' of pages in core storage, for perhaps quite lengthy periods of processing. However, the pages which constitute the working set can change during the execution of the program. The second aspect of program behaviour that we model is the gradual 'drift' of working set membership - abrupt changes, such as occur between different phases of a job, particularly if the job has been designed using an overlay structure, can be modelled by treating the phases as constituting distinct jobs.

The observed relationship between page fault rate and available core storage has been modelled in a perhaps oversimplified fashion by taking the probability that a given instruction causes a page fault
to be

$$-\frac{16^{\text{BCP}}}{\text{DWS}}$$

where BCP is the number of pages which the job has in core, and DWS is the working set size of the job. Even more arbitrarily, in the set of experiments described in this paper, we have assumed that the gradual drift of a job's working set is steady and involves the job completely changing its working set three times during the course of its execution. The appropriate probability here is

$$\frac{3\times \text{DWS}}{\text{CPUTIME} \times 1000}$$

where CPUTIME is the total CPU time, in milliseconds, required by the job, the factor 1000 converting this to instructions.

From the combination of these two factors we obtain that the expected length of processing time that a job will achieve before page fault is given by

$$m = \frac{1-k}{k \times 1000}$$

where

$$k = \frac{-16^{\text{BCP}}}{\text{DWS}} + \frac{3\times \text{DWS}}{\text{CPUTIME} \times 1000}$$

is the per instruction probability of a page fault being caused by a given instruction.

A given simulation experiment would involve one or more different classes of job. Each class of job would be represented by a job 'profile' indicating the size of working set and the amount of CPU and I/O time required. The parameters for individual jobs are obtained by using the parameters of the job profile as the means of appropriate probability distributions. The parameter DWS in the above formula is sampled from the uniform distribution $U(X-\frac{1}{2}X, X+\frac{1}{2}X)$, where $X$ is the value of the appropriate job profile parameter, and CPUTIME is sampled from the exponential distribution.

The simulator allows different drum organizations to be used. The scheme employed in the simulations described in this paper is a
sector-queued organization with priority ordering of the sector queues. We simulate the drum as being able to revolve every 17.5 msec., and capable of holding 4.5 pages on each track. The drum is considered to be laid out as in the Michigan Terminal System, where each track is divided into 9 sectors each of one half page. Two adjacent tracks may then be employed to record 9 pages, so that 1 page may be read in 2/9 of a physical drum revolution (1/9 of a 'logical' drum revolution) by splitting each complete page between two adjacent tracks in the appropriate manner.

A queue of requests is maintained for each of the 9 sectors and these queues are priority ordered. The maintenance of these queues is idealized in our model, in that we assume the supervisor to be capable of maintaining the queue instantaneously, so that re-ordering of a queue may take place up to the instant at which the drum read is to be performed. That is to say we ignore the time required to set up the channel programs. We make the further assumption that the system is aware of the completion of the page transfer from the instant at which the transfer is completed physically, whereas in a real system this 'posting' might not occur until the end of the logical drum revolution, or some other convenient time.

3. **Workload Experiments**

The simulation model described above has been used to conduct a series of experiments into the behaviour of a number of scheduling algorithms, which have been designed to minimize the loss of processor utilization that occurs when all jobs in core are held up waiting for either an I/O operation to be completed or a page fault to be serviced. The workload simulated in these experiments was an attempt to model a workload representative of that presented to the Michigan Terminal System at Newcastle. This model workload is composed of three distinct components.

a) Small jobs requiring a working set of 5-15 pages and of the order of 1 second of CPU time and 3 seconds of I/O. These are intended to represent interactive work such as editing and listing files and perhaps APL sessions. This type of work was estimated to demand 10% of the available CPU time.
b) Medium jobs requiring a working set of 15-45 pages and of the order of 20 seconds CPU time and 20 seconds of I/O.
These are intended to represent the compilations and runs of simple programs, which, due to the teaching work of the establishment, represent the bulk of the jobs presented to the system. These were estimated to demand 40% of the available CPU time.

c) Large jobs requiring a working set of 25-75 pages and of the order of 100 seconds of CPU time and 33 seconds of I/O.
These are intended to represent the CPU bound component of the workload which accrues from the research work, which was estimated to demand 25% of the available CPU time. No attempt was made to model the really large jobs often required by numerical work, since these are unlikely to be run during the normal daytime MTS session at Newcastle.
Overall, the load demands 75% of the available CPU time.

Figure 1 shows a graph of CPU utilization versus core size, obtained on this model workload by various algorithms, all of which contained provisions intended by their designers to prevent the occurrence of thrashing. The core size of 5000 pages may be thought of as an 'infinite' core, since with the above workload no page replacements are required with this amount of core available. This core size enables us to establish the maximum CPU utilization obtainable for the load presented since there is no page wait caused by contention for core. The results labelled SINGLE-PROGRAMMED are the utilizations obtained by running the jobs serially through the system and may be used as a basis for comparison of the various algorithms. This graph illustrates the fact that given enough core even the most ill-conceived algorithms can be made to perform satisfactorily!

In figure 2, we present a graph of CPU utilization against multiprogramming level for a fixed core size of 70 pages. The value of multiprogramming level is set as an initial parameter of the system and effectively limits the length of the scheduler queue. The scheduler queue is a priority ordered queue of all the jobs in the system, (jobs are detached only when they terminate) with flags set to mark whether the job is in I/O wait, page wait or requires the CPU.
A job is 'CPU-ready' if it is not in wait state. In the algorithms described in this paper, when the value of multiprogramming level has been set to X, the scheduler scans only the first X positions of the queue and allocates the CPU to the first CPU-ready job that it encounters. If there is no such job the system goes into CPU-idle state until one of the jobs in wait state becomes CPU-ready once more.

From the graph in figure 2 it can be seen that some algorithms have the very desirable property that their performance improves essentially monotonically with multiprogramming level. Despite this however some algorithms do not significantly improve upon uni-programming. At the other end of the spectrum, some algorithm can be seen to fail dismally at avoiding thrashing and very quickly become less efficient than uni-programming.

A further aid to insight into the behaviour of the various algorithms is the 'core map', an example of which is shown in figure 3. The core map gives a pictorial representation of the way in which the core is divided amongst the jobs in the mix. Each character represents a page of core, the code being the job number modulo 10. That is, each '1' represents a page belonging to job 1 or job 11 or job 21, etc. Since the codes are laid out according to the priority order of the jobs we can easily find from the context, to which job a certain page, represented in the core map, belongs. An '*' represents a free page frame. The contents of core may be sampled at any pre-set interval. We have found sampling every simulated second to be adequate.

Such maps have been our major tool in confirming or understanding the behaviour of each scheduling algorithm, thus leading on several occasions to the development of new algorithms. We now have reason to believe that the model workload is in fact more severe than the actual workload that the 360/67 is subjected to at Newcastle. In fact in some ways this excessive severity has been useful, as the core maps have shown us how the various algorithms have reacted in a wide variety of circumstances. However, in order to simplify and shorten the present paper, we will base our discussions on graphs and core maps obtained using a much simpler workload than that described above. Furthermore, we will concern ourselves with just three of the more basic algorithms from the set whose behaviour is illustrated in figures 1 and 2. The workload consists of a single job entering the system at time 0,
requiring a working set of a certain size, and equal amounts of CPU and I/O time. The I/O operations require a mean time of 100 msec., to complete and therefore a computation period of mean 100 msec., between I/O operations is required to give the chosen I/O to CPU ratio. All further jobs are identical in their requirements to the first and enter the system at time 1 second. From this we observe that the lowest CPU utilization which we would expect to obtain is 50%, since this is the utilization which the first job, running by itself, would attain. This figure may be depressed due to time spent in page wait during the initial period, when the job is being loaded by demand paging, and also when a page fault occurs because of the slow drift of working set.

4. Wharton's Algorithm

This algorithm, which led to the development of interest in the class of implicit algorithms, was proposed by R. M. Wharton [5], a student of J. J. Horning at Toronto.

An external priority structure is imposed upon the jobs submitted to the system, this priority ordering being fixed externally and not being modifiable by the scheduler. The scheduler allocates the CPU to the highest priority job waiting for the CPU.

On occurrence of a page fault, any free page frame is allocated, if such a page frame is available. When all of core has been allocated, the lowest priority job with pages in core, of priority less than or equal to the job causing the page fault, has a page replaced from amongst its in-core set according to some appropriate strategy. If no such job exists then the page request is denied and the requesting job cannot proceed until a higher priority job terminates its execution thus freeing pages of core.

This situation may arise because we assume that all pages are loaded on demand, there being no initial loading phase. A job which has no pages in core may then be scheduled but will page fault immediately. Such a job may then, because of its priority, find that there is no valid page replacement for it to make.

Page requests for pages to be transferred to core from the drum are serviced according to the same priority ordering as the
scheduler queue priority.

The philosophy behind Wharton's algorithm is to give the top priority job the service which it would obtain if it were running by itself in the system. Further jobs are then run as a background stream utilizing any resource not required by the top priority job. By this scheme the worst utilization that will occur is that utilization obtained by running the jobs serially through the system. We acknowledge that if we were in fact producing a uni-programming system we could introduce improvements and also that we are ignoring the interference caused by the supervisor dealing with the interrupts of the lower priority jobs. However these should not cause very large discrepancies and so the statement is fairly accurate.

The control upon the level of multiprogramming is obtained by the allocation of core storage. If a job has no core then it cannot affect the effective multiprogramming level of the system. Since a job may only obtain more core space by replacing pages of jobs of equal or lower priority, (we employ priority orders in which there is no repetition) the lowest priority job may only obtain core by utilizing free page frames. Also as the jobs of higher priority acquire more pages the lower priority jobs will be deleted from core. This is because their pages will be replaced by jobs of higher priority and since they will be unable to obtain page frames, they will be suspended until a job of higher priority terminates execution, thus freeing core.

We can observe the manner in which this occurs by studying the core map shown in figure 4. The contents of core were sampled at the end of each simulated second. The core map shown here is the first part of that for the simulation of Wharton's algorithm when we limit the multiprogramming level to 10, and all of the jobs are identical, each requiring a 20 page working set.

We can see after time 2 seconds that the core allocations of jobs 1, 2, 3, increase at the expense of job 4 until that job is deleted from core. Thus multiprogramming level has been 'implicitly' reduced. After this time we see that the allocations of jobs 1, 2 increase at the expense of job 3, which is the lowest priority job with pages of core.
It is interesting to examine the number of page frames jobs 1, 2, and 3 have at the time job 4 is deleted from the mix. Job 1 has 29, job 2 has 26 and job 3 has 25. We see then that the top priority jobs accumulate pages in core for which they no longer have a requirement, and unless a job becomes the lowest priority job in core, there is no mechanism included by which these pages can be removed until the job terminates. Thus, whilst we are ensuring that contention for core will not be allowed to depress CPU utilization, we are making poor utilization of our limiting resource. A particularly bad case of the working of this algorithm is that of the top priority job being an I/O bound job which is dumping a large area. It will require many different pages during its execution but each page will be required only for a very short period and will subsequently be superfluous to the progress of the job.

Let us now consider the graph of CPU utilization versus multiprogramming level (figure 5). The working set size was given the value 20(10)80 pages for a system in which 80 pages of core were available. The level of multiprogramming was varied from 1 (monoprogramming) up to 10.

We see for Wharton's algorithm that as the multiprogramming level increases from 1 for a particular working set size the CPU utilization increases to some maximum value which it then maintains with only slight variations. The interpretation of this behaviour is that Wharton's algorithm implicitly sets a value of the multiprogramming level and any further jobs in the mix are ignored. When the pre-set limit of multiprogramming level is less than the level at which Wharton's algorithm is capable of working then the CPU utilization will be below the maximum attainable. As we increase the pre-set limit the CPU utilization improves to the maximum.

5. Horning's Algorithm

We have observed in the discussion of Wharton's algorithm that the higher priority jobs retain pages which are superfluous to their progress. This algorithm, proposed by J. J. Horning, was derived from Wharton's algorithm and attempted to include a mechanism which would free this unproductive core storage.
As before, external priority structure is imposed upon the jobs submitted to the system, this priority ordering being fixed externally and not modifiable by the scheduler. The scheduler allocates the CPU to the highest priority job waiting for the CPU.

On occurrence of a page fault, any free page frame is allocated, if such a page frame is available. When all of core has been allocated, a job is chosen at random from amongst those which have real core pages, the probability of choosing a job being directly proportional to the number of page frames it has, and a page is replaced from amongst this job's page frames according to some appropriate scheme.

Page requests, for pages from the drum, are serviced according to the same priority ordering as the external priority ordering. The scheduler and drum organizations of Wharton's algorithm are again employed.

The reasoning that led to the Horning algorithm was as follows. If we superimpose graphs of drum reads versus multiprogramming and CPU utilization versus multiprogramming we would expect to obtain a figure like figure 6. Obviously that level of multiprogramming which will give us the maximum CPU utilization is the level at which we would like to run the system. However, this level is dependent upon the load at any time. We are, therefore, interested in a mechanism which will vary the effective level of multiprogramming so that we may obtain this optimum. The form of this control will be the introduction of jobs into, or removal of jobs from core. Thus when we are in area A of the graph we would like there to be a net drift of pages from high to low priority jobs, thus increasing the multiprogramming level, and from low to high priority jobs when in area B, thus decreasing the level. This should be achieved by the random page replacement policy.

The policy is biased towards 'stealing' pages from the jobs with the most pages, which are probably the highest priority jobs. When page demand is low, area A, the queues for drum service will be short, and high and low priority jobs will obtain essentially equal service. A net drift of pages from high to low priority jobs will result. When page demand is high, area B, the lower priority jobs will be blocked from obtaining drum service due to the rapid queuing
of the service requests of the higher priority jobs. The priority ordering of the drum queues will therefore bias the gain of pages to high priority jobs and so low priority jobs will eventually be deleted from core. As paging activity subsides then the service requests of the low priority jobs will become unblocked and these jobs will once more be able to gain page frames.

Let us now consider the core map, figure 7, which was obtained for Horning's algorithm with the multiprogramming level limited to 10 and all jobs identical, as for Wharton's algorithm, each requiring a 20 page working set. We observe that, in contradiction to the above reasoning, all of the jobs quickly obtain pages in core and that no job has the number of pages required for its working set. Also, the higher priority jobs obtain no greater share of core than do the lower priority jobs. For instance at time 6 seconds, job 1 has 6 pages whilst job 10, the lowest priority job, has 11 pages. We can explain the failure of this algorithm as follows.

The premise upon which this algorithm was founded is that when drum queues are short the core is not overpartitioned with the implication that the effective level of multiprogramming may be increased. A simple example is sufficient to display that this is not always true. Suppose that the top three jobs in the scheduler queue have a combined working set requirement greater than the number of pages in core, so that thrashing would ensue if avoiding action were not taken. The longest possible queue for drum service would contain just three elements which would be insufficient to block the requests of the third job. (Even if this were sufficient the request of the third job would be unblocked as soon as the higher priority jobs obtained their working set. Thus, the top priority jobs would have their working sets only for very short periods.) However with a sector-queued drum organization, this queue length is of low probability. Thus we see that the problem is caused because lightly loaded queues act like FIFO queues with the result that the competing jobs get almost equal service from the drum. This, coupled with the random page-steal strategy causes the core to become equally divided amongst the jobs on the scheduler queue.
We see also that the other blocking mechanism, the priority scheduler, is similarly ineffective. In a thrashing situation the CPU is grossly under-utilized and so all jobs in the scheduler queue will obtain all of the CPU they demand, which is very little. Thus the other method of blocking low priority jobs from gaining core pages, not scheduling them, is undermined. We see then that once thrashing has begun this algorithm will cause further degradation. Similarly, we see that if the CPU is not fully utilized the reaction of this algorithm is to introduce further jobs to utilize it, such a policy must eventually lead to thrashing.

Let us now consider the graph of CPU utilization versus multiprogramming level, figure 8, which was obtained with the same mixes as those for Wharton's algorithm.

We see that the CPU utilization increases up to some maximum before decreasing, as multiprogramming level is increased. The maxima for those job mixes where the working set size is at least 50 pages occur at level of multiprogramming = 1. It is sufficient to multiply the working set size of the jobs by the multiprogramming level at which the maximum occurs, to explain this. We observe that the maximum occurs at the multiprogramming level for which that product is closest to the core size, that is, at the highest level at which the core is not overcommitted. A continual degradation of CPU utilization is then seen as each further job is introduced into the mix, causing greater overcommitment of core.

6. Lynch's Algorithm

This algorithm, proposed by W. C. Lynch, is a more successful attempt at modifying Wharton's algorithm to deal with the problem of unrequired pages being accumulated in core. The proposal is to couple a 'drain' with the Wharton algorithm. The 'drain' consists of writing out of core one page every second logical revolution of the drum from

a) the job using the CPU at that time
b) all jobs in I/O wait at that time

and freeing those page frames.
The reasoning is that if a job has more page frames than it requires to contain its working set then we will gain page frames from it. If we were to steal a page which belonged to the top job's working set, then we would expect the job to page fault almost immediately and to re-acquire that page within half of a logical revolution of the drum on average. Thus, if we were to steal from that same job each time we applied the drain, we would only impair that job's performance by 25%, a loss we would hope to recoup by the improved performance of the background stream. We would also expect that the jobs most likely to be doing I/O or utilizing the CPU, would be those jobs which have at least their working set of pages in core. Thus we will, for the most part, be draining pages from the correct set of jobs.

One does not drain pages from a job which is in page wait, because page requests are not a function of the job's 'local' time (i.e. the CPU and I/O time the job demands from the system). Thus by deleting pages only from jobs which are in I/O wait or occupying the CPU, the extent to which a job may be delayed in the system is bounded. This is not so if we drain pages when a job is in page wait.

Let us now consider the core map for Lynch's algorithm for multiprogramming level = 10 and each job having a working set of 20 pages, (figure 9). We observe that the core is evenly divided amongst the top four jobs on the scheduler queue and that the number of pages which each of these jobs has is in the region of their working set size of 20 pages. The situations in which a job has less that its working set correspond to I/O waits during which the job was losing pages due to the drain and was unable to regain them until it began processing once again. We see then that the 'drain' is successful in ensuring that jobs do not claim more space than their working sets demand.

Now let us consider the graph of CPU utilization versus multiprogramming level (figure 10); we have omitted the results for working sets of 60 and 70 pages since they are very similar to those for 50 and 80 pages and only confuse the graph.

We noted when discussing the workload that when uni-programming we would expect to obtain a CPU utilization of 50%, further when
discussing Lynch's algorithm we observed that due to the drain we would expect to depress the utilization of the top priority job by no more than 25%. We see that when multiprogramming level = 1 we obtain a utilization of 38%, 75% of that obtainable when not employing the drain.

It is instructive to compare the results of Lynch's algorithm with those for Wharton's algorithm. We observe that for working sets of 20 and 40, in which an integral number of jobs each with their full working set just occupy the whole of core, Lynch's algorithm is superior to Wharton's algorithm. Also, in these cases the loss of CPU utilization due to the drain is very small. However, in all other cases Lynch's algorithm is inferior. This is because there is no significant gain to be made by squeezing only half of a job's working set into core and, moreover, in order to allow this we have drained pages from other jobs so depressing their CPU-utilization. Thus we cause a loss of overall CPU utilization by draining in these situations. We also see that in those cases in which the core would be under-utilized, (low multiprogramming level), draining also incurs a loss. This is because the drain depresses the CPU utilization of the jobs in core and no further jobs are available to utilize the free page frames.

Conclusion

The present paper attempts to document some of the findings of a continuing series of experiments concerning the problem of thrashing. Although much remains to be investigated we consider that we have considerably extended our knowledge of thrashing and also of some simple mechanisms which have achieved some measure of success in avoiding the problem. It is hoped to obtain analytical confirmation of some of these results in the near future.

It is our intention to attempt to provide further validation of the parameters of the simulator, in particular, those concerned with the modelling of paging behaviour based on the workload of the 360/67 at Newcastle. We also hope to investigate some further inter-related aspects of the scheduling of multiprogrammed paging systems.
One by-product of our work has been to increase our already keen awareness of the ease with which one can fall into the trap of developing an algorithm whose behaviour cannot in practice be successfully predicted. The reasoning that was used to justify the Horning algorithm was accepted by all of us until experiments showed how inadequate this reasoning was. Yet the typical operating system contains many algorithms whose behaviour is far more impenetrable than this comparatively simple scheduling algorithm.

Acknowledgements

The work described in this paper has benefited greatly from conversations with our colleagues, in particular Jim Horning of the University of Toronto, who spent the autumn term of 1970 at the University of Newcastle upon Tyne, and Hugh Lauer, who provided a great number of helpful criticisms and suggestions.
References

Demand Paging in Perspective.


The Load Leveller.

Program Behaviour In A Paging Environment.

and

An Evaluation of Program Performance on the M44/44X System.

Technical Report No. 23, January 1971, Department of Computer Science, University of Toronto, pp. 23-34.
Figure 1.

Various scheduling algorithms.

Variation of CPU utilization with core size,
for standard workload.

Figure 2.

Various scheduling algorithms.

Variation of CPU utilisation with multiprogramming level, for standard workload.

20th February 1971.
Wharton's Algorithm

Figure 4
WHARTON'S ALGORITHM.

VARIATION OF CPU UTILIZATION WITH MULTIPROGRAMMING LEVEL FOR VARIOUS WORKING SET SIZES:
1ST JULY 1971.
Superimposed Graphs of CPU Utilisation Against Multiprogramming Level and Number of Drum Reads Against Multiprogramming Level

Figure 6
HORNING'S ALGORITHM.

VARIATION OF CPU UTILISATION WITH MULTIPROGRAMMING LEVEL, FOR VARIOUS WORKING SET SIZES.

1ST JULY 1971.
LYNCH'S ALGORITHM

VARIATION OF CPU UTILISATION
WITH MULTIPROGRAMMING LEVEL
FOR VARIOUS WORKING SET SIZES.

1ST JULY 1971.