PROJECT MAC February 10, 1973

Computer Systems Research Division Request for Comments No. 6

AN ALGORITHM FOR PREDICTING PROCESS MEMORY REQUIREMENTS

by David P. Reed

The following paper describes my ideas for improving the primary memory scheduling algorithm currently implemented in the Multics traffic controller. My MIT Bachelor's Thesis will consist of developing and experimenting with these ideas inside an experimental version of Multics. Comments and criticisms on the content of the paper will be appreciated.

An Algorithm for Predicting Process Memory Requirements

David P. Reed

I. Why try?

Effective assignment of primary memory resources to processes in a multiprogrammed, virtual memory computer system such as Multics requires that the degree of multiprogramming (1) be automatically controlled in order to prevent devotion of excessive resources to page fault processing. Control of multiprogramming is important from the operating system's point of view in order to maximize the effective use of the available hardware resources. Were this the only goal, however, the task would be much simpler; a very important requirement for a multiprogramming control is to allocate primary memory resources fairly among user processes. It is easy to ignore the variation in usage patterns among user programs and design a scheme which works for the average programs on a particular system, and performs poorly for exceptional classes of users. The exceptions which I am concerned with now are the ends of the memory usage scale: the small users, such as the SIP3 Experimental Calculator Service, which offers a cheap, interactive subset of BASIC to all

---

(1) "degree of multiprogramming", "level of multiprogramming", and "number of eligible processes" are used synonymously in this paper to mean the number of processes which the traffic controller has made eligible to run at any particular time. The set of processes which are eligible to run includes processes currently executing on a processor, processes waiting for an I/O event or a lock to be unlocked, and processes waiting for a processor, and can be characterized by the fact that their per process data bases such as the descriptor segment are wired down in core.

MIT students, and the large users, such as data-base management systems, automatic programming systems, and so on. A fair scheme for allocating primary memory resources should allocate primary memory to users in proportion to their need for primary memory, so that large users of memory will have enough memory to run with an acceptable level of overhead due to page faults. On the other hand, a small memory user should not be allocated more memory than is necessary to run him efficiently, for any excess resources might as well be assigned to others waiting in the scheduling queues to run.

In order to manage primary memory effectively, taking into account the variations among classes of usage, the traffic controller must be able to predict the memory requirements of processes and use this prediction in its scheduling decision. Unfortunately, until recently there was no easily tractable model of a process's memory reference behavior which could be invoked to assay a process's memory needs at a particular instant of time. Consequently, in the current design of the Multics primary memory scheduling algorithm, the control of multiprogramming is achieved by an algorithm which has been observed to show several anomalies (in particular, it performs very poorly for some programs with larger than average "working sets", and does not adapt to unusual system configurations, such as 256K of memory with 2 CPU's). I believe, however, that Saltzer's linear model

of demand paging performance (1) does lend itself to a simple, easily implemented algorithm for dynamic multiprogramming control. For my thesis, I intend to develop such an algorithm, and determine experimentally its performance.

II. Yardsticks

Before discussing particular algorithms to achieve the purposes hinted at above, it seems important to enumerate several criteria which I think must be met by any primary memory scheduling algorithm. First, the variables most important to heavily loaded interactive systems, response time and system throughout, must be optimized. Second, the algorithm must properly respond to the variation in memory requirements among different classes of users -- not penalizing large users excessively, and giving the small user his fair share of resources. It is under this criterion that the current Multics memory management algorithm seems to be deficient. Thirdly, the algorithm must be stable under fluctuating load conditions. Although the algorithm is primarily used to prevent thrashing, a condition occuring only under a significant load, light loads should not cause the algorithm to operate incorrectly, and transient loads due to interactive processes should not cause the algorithm to oscillate, or otherwise perform suboptimally. Finally, the parameters by which the algorithm is tuned should be

---

(1) Saltzer, J. H., A Simple Linear Model of Demand Paging Performance, Multics Repository Document M0131, MIT, Cambridge, Mass., 1972.

independent of system configuration as much as possible. In other words, once a system is tuned for one configuration, it should be correctly tuned for any other configuration. I am particularly interested in the configuration with respect to the size of primary memory and the number of CPU's. It is interesting to note that the current algorithm on Multics causes the following anomaly: if the parameters are set for a 1 CPU, 256K system, then the performance of a system such as a 2 CPU, 256K system is very poor. This probably means that the second CPU is causing thrashing, and should remain idle more of the time, since a second CPU should not cause degradation of system performance on the same amount of memory. (1)

III. Simplicity

I devote a separate heading to this topic simply (!) because it is worth emphasizing. It is all too easy to design algorithms which are more complex than our understanding of the problems they intend to solve. Since the difficulty in maintaining programs is primarily due to the time required to understand them, simplicity is an operational virtue.

In the context of a discussion of primary memory management algorithms, simplicity has several applications. Our understanding of programs' memory reference behavior is limited,

---

(1) Barring such effects as memory interference and data-base interference, both of which can be minimized by idling the additional CPU more frequently.

so it would be unjustifiable to assume that a complicated algorithm has any better chance of working well than a simple algorithm would. A simple algorithm is much easier to explain, thus allowing others who must understand the algorithm (such as external Multics sites who have need to tune their system for their own purposes) to achieve the necessary knowledge with a minimum of difficulty. Also, since the multiprogramming algorithm is a basic part of the operating system, a simple multiprogramming algorithm has the virtue of easier verification, an important consideration in a system striving to be proved correct.

IV. Basic Principles

For most of this paper, I will sketch the development of an algorithm based on Saltzer's linear model, which I hope will satisfy the above criteria, along with the criterion of simplicity. Towards the end of the paper, I will discuss the techniques which I intend to use to investigate the performance of the algorithm experimentally.

Before I begin describing the details of my proposed algorithm, I would first like to note the basic concepts of the current multiprogramming control mechanism. First of all, since we are primarily concerned with obtaining effective use of main memory, the level of multiprogramming depends directly on the size of primary memory available for paging (I will henceforth use the symbol M for this). The number of CPU's is not germane to this

discussion, since addition of a CPU has the effect of increasing the rate of references to memory, which effect will be factored out by considering time as an axis measured in memory references. (1) In order to characterize a process's memory requirements, the current algorithm attempts to compute a value, called the working set estimate, which indicates the amount of primary memory which must be available to that process to prevent excessive paging. This is basically computed from the size of a subset of the recent page exceptions (those which correspond to pages which have been referenced recently, according to the hardware and software reference bits).

The scheduler then makes enough processes eligible so that the sum of their computed working set estimates is as large as possible, but not exceeding the amount of primary memory available. An additional constraint is added which places lower and upper bounds on the number of eligible processes.

In practice, this algorithm has failed to be useful, as is shown by the fact that the computed working set estimate is multiplied by a fraction, called the working set factor, before summing the eligible processes' working set estimates, and that this fraction has in practice been set to a value which is too small to allow the working set estimates to influence the scheduling decision at

---

(1) Of course, I oversimplify here...in fact there are important effects such as sharing of pages and overloading of I/O channels which do come into play in determining the load on primary memory. For a first order theory, I will explicitly ignore these effects.

all (increasing the working set factor causes performance to degrade significantly -- indicating the algorithm does not calculate a working set estimate which is useful in scheduling).

This algorithm has been observed to have several deficiencies. The first is poor parameterization, which does not carry from one configuration to another. More importantly, the working set estimate has shown some obvious random behavior for certain programs with large working sets. The symptom of this is that the working set estimate remains very small even though the user's program is taking over 100 page faults per cpu second (a mean headway between page faults of several instructions). This has been observed by people in the LISP project, and by Rick Gumpertz in the Paris Multics.

Finally, the current algorithm fails on the criterion of simplicity. It bases its decision of whether to include a page in the working set on a complicated calculation based on 5 bits of information about each recent page fault. The time eligible is not taken into account, and page faults which are too frequent to be recorded in the page fault trace table are not taken into account. In addition, there are complicated interactions with other processes, the core allocation algorithm, and the system call, hcs_$reset_working_set, all of which modify the page usage bits which the working set algorithm depends on.

V. The Proposed Algorithm

I would like to try a somewhat different approach in synthesizing an algorithm here, with the hope that the resulting algorithm will be simpler, and more amenable to analysis and verification. I will also try to define a set of less ambiguous terms for describing concepts, since such terms as "working set estimate" are apt to be confusing, because they have many possible interpretations in current usage.

The following analysis is not intended to be mathematically precise; instead, it is intended only as an argument to justify the plausibility of the resulting algorithm.

To begin the discussion of my proposed algorithm, I would like to define a function, p(h,t), which is called the partition size of a process with respect to a particular mean headway between page faults (mhbpf, measured in memory references), h, at a particular time t. This partition size function is defined to be the amount of primary memory required for that process to run at time t in its execution with a specified local mhbpf, h. Given this function for all processes, a near-optimal multiprogramming level can be determined.

One way of preventing degradation of each process's performance due to thrashing is to place a lower bound on each process's mhbpf, h_min. Given h_min, a near optimal multiprogramming strategy is to schedule as many processes as possible under the constraint that the sum of their instantaneous partition sizes is

<= M:

$$\sum_{} p(h\_min,t) <= M.$$

Of course, if the load is sufficiently heavy, the inequality becomes an equality, and thrashing is held to the level determined by the mhbpf, h_min. On the other hand, if there is not a sufficient load, the mhbpf for each process will be greater than h_min. Of course, fixing h to h_min does not guarantee that we will prevent thrashing; it depends also upon how steady p(h,t) is with respect to small changes in t.

The problem of implementing this strategy is that p(h_min,t) cannot be computed without foreknowledge of the process's behavior. Consequently, a practical algorithm must use a function which is capable of estimating p(h_min,t). The working set estimate above can be seen to be an attempt at this, however, it is not a very effective estimate for the reasons noted above. In the following analysis I will attempt to create an estimator for p(h_min, t) which is better at tracking the actual value of p(h_min,t) than the working set estimate currently used.

Saltzer's linear model (1) states that p(h,t) is a linear function of h:

---

(1) I am assuming that Saltzer's model can be applied to single processes, an assumption which is plausible but not yet experimentally verified. The linear model does not have to apply exactly in order for this scheme to work, but I develop the argument from this statement of the linear model for the purpose of conceptual clarity.

$$p(h,t) = k(t)*h,$$

where k is only a function of the program behavior at t. As a consequence, if we know the value of p for some specified value of h, $h0$, we can determine p for all values of h:

$$p(h,t) = \frac{h}{h0} * p(h0,t)$$

In particular, assuming that $k(t)$ has a bounded first derivative, and that we have the value of $p(h0,t)$, we can get a good estimate for p a short time later by assuming that $k(t)$ is not varying very much over that short period:

$$p(h,t+dt) = \frac{h}{h0} * p(h0,t)$$

It can easily be seen that this last equation can be used as the basis for a rather simple multiprogramming control. If we take $p(h0,t)$ to be the partition size which we last allowed the process to run with, and $h0$ to be the observed mean headway between page faults for that period, than we can easily obtain an estimate for the immediate future of that process of $p(h\_min,t)$ by that equation, setting $h = h\_min$. We thus have the following iterative algorithm which tracks the partition size of a process:

$$pe = \frac{h\_min}{h\_obs} * pa$$

where pe is the new estimate of the required partition size of the process for an mhbpf of h_min, h_obs is the observed recent mhbpf, and pa is the amount of main memory which was assigned to

the process during the recent past (over which we measured h_obs).

This algorithm is actually more powerful than the assumptions indicate. Even if Saltzer's linear model does not hold for a single process, this iterative algorithm will track p(h_min,t) as long as the rate of change of p with time is small.

The only problem with implementing this algorithm is that it is somewhat difficult to determine a value for pa, the amount of memory actually assigned to the process, since several processes are competing for main memory. Under a sufficient load, of course, the constraint that the scheduler tries to make as many processes eligible as possible will force pa to be approximately equal to the previous estimate of pe. However, under lighter or more transient loads, this will not be the case. Since we would wish pe to be relatively independent of load fluctuations, we have to handle this case.

Under a light load, the scheduler will not have enough processes to wholly utilize main memory resources. Consequently, each process's pa > pe. When there are processes whose partition sizes, pe, do not fit in primary memory, it is possible to have situations where a process's assigned memory, pa, is less than pe. This latter condition can also arise from constraints, such as the current max_eligible and min_eligible, placing bounds on the number of processes the scheduler can make eligible. In order to keep the partition size estimates, pe, stable through

transient loads and independent of time, it is thus important to get an accurate estimate of the memory assigned to the process to compute new values of pe.

Making the assumption that the effective partitioning of memory among the eligible processes is proportional to their minimum partition size, we can say then that

$$pa = \frac{pe}{u}$$

where u is the ratio of the sum of all eligible processes' pe values and the main memory size M:

$$u = \frac{1}{M} * \sum_{i \in \{\text{eligible processes}\}} pe(i)$$

This is basically a primary memory usage factor. Combining the last two equations, we get the iterative algorithm for computing the next pe value, pe':

$$pe' = \frac{h\_min * pe}{h\_obs * u}$$

The remaining task is to determine how to compute the value u, and how to parameterize the algorithm (i.e., set h_min) so that once the parameters are set, all configurations will obtain optimum performance. Another problem is to determine how frequently to iterate the computation, and a final problem is how to choose an initial estimate, pe_init.

Since u is just an average, over the last iteration of the algorithm, of the memory usage fraction, it can be computed in any number of ways. An approximation to the average will suffice, since the value is not particularly critical. Consequently, for an initial attempt, the system can sample the memory usage fraction at each page fault, then average.

I think that for the purposes of an initial experiment, the computation of pe can be iterated each time the process loses eligibility, since the value is needed no more frequently than that.

Parameterization of h_min and pe_init is a more difficult problem. I think that one can say that h_min must be linear with the size of main memory, and independent of the number of CPU's and the speeds of memory and CPU. The characteristics of the paging devices do affect it heavily, though, if one wants to get optimum performance -- the bulk store should affect things, especially, since processes will probably not be switched upon page faults to the bulk store. I think a good choice for a parameterization is given by following two equations though:

$$h\_min = ch * M$$

$$pe\_init = cp * M$$

where cp and ch are tuning parameters, and M is of course the size of primary memory available for paging.

VI. Plans for Testing the Algorithm

For my thesis, I intend to investigate the characteristics of this algorithm experimentally, using Doug Hunt's standard Multics load programs, and compare performance with the current multiprogramming algorithm. If possible, I hope to get a chance to test this algorithm as a replacement for the current algorithm. Several minor problems, of course, suggest themselves.

First, the pre-paging algorithm interferes significantly with the operation of this multiprogramming algorithm, since it introduces undesirable feedback into the mhbpf of a process. Consequently, I intend to perform my initial tests with the pre-paging algorithm removed from the system. In any case, it is not clear what performance improvement it gives the system, and with the bulk store it will not be used, owing to the transfer time of the bulk store.

Second, I may have to provide constraints on eligibility such as the current algorithm requires, so that response time is held down and so that undesirable oscillations or other problems in my algorithm don't cause bad responses to pathological loads. It should be noted that too high a level of multiprogramming does tend to increase interactive response time (1) undesirably, so

---

(1) By response time here I mean the real time taken for a process to go from a blocked state to a running state then back to a blocked state, executing for a particular amount of CPU time. There is no good measure of this in the system now,

the maximum bound is important. The minimum bound will prevent excessive idle time caused by a program with a partition size greater than or equal to the amount of core available. Parameterization of such bounds to adapt to various configurations and policy requirements is somewhat difficult, and probably will be developed from experimental results.

Third, the algorithm does not explicitly take into account the burst of page faults which occurs when a process is made eligible after not running for a period of time. This could cause the algorithm to determine a partition size which is larger than necessary for such processes. I think the best way to avoid this problem is to make the sampling times at which pe is recomputed sufficiently far apart so that the short period of high page fault rate has little effect. It is possible that this is not an adequate solution, in which case some sort of smoothing must be employed to damp out oscillations in the calculated partition size.

A fourth problem arises in the attempt to allow the user a call of the order of hcs_$reset_working_set, by which the user can inform the system of changes in his memory reference behavior. The possibility always exists that the user will lie, in which case he may be able to prejudice the scheduling algorithm in his favor. I admit that I have no good ideas here at present. The solution would be to have some sort of penalty for lying; perhaps

---

unfortunately.

charging for excessive memory competition would be an effective deterrent to lying.

VII. Expectations

The results of implementing the scheme above, with modifications as necessary to achieve this goal, should basically concern only the extreme classes of users. I don't expect a spectacular improvement in system performance under the current load, nor do I expect response time to get significantly quicker. Rather, I expect a stabilization effect, in which the system as a whole becomes more stable in response to load, and the more average user programs should become more consistent in their memory usage behavior. The extreme classes of users will be treated more fairly than they are at present, since they will be scheduled with the correct amount of competition to ensure reasonable performance.