The dynamics of paging

M. V. Wilkes

Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge CB2 3QG

A general model of a paging system is given that can be used to describe a wide variety of particular systems referred to in the literature. This model is discussed from the point of view of control engineering, with particular reference to the avoidance of instability (thrashing). A distinction is drawn between different levels of control and between systems that are *tuned* and those that are *under control*. The relationship of the approach made in the paper and that *via* queuing theory is discussed.

(Received September 1972)

The *kinetics* of operating systems has received much attention in the literature. Kinetics is the study of how a system can move without violating its constraints, and it is relevant to the problem of designing a system that is adequately, but not over, constrained. The proper interlocking of processes, so that deadly embraces cannot occur, comes under this heading; the constraints can be applied through the use of nested process calls, flags, semaphores, and other devices. There is no discussion, however, in kinematics of how the system will actually move in response to applied forces. That discussion belongs to *dynamics*.

The last 20 years have seen the systematic development of *control engineering* which aims, in the various fields in which it is practiced, at understanding the dynamics of systems and how they can be controlled. The control engineer is concerned with such things as maximising throughput, minimising cost, and safeguarding the quality of the product; in general terms, with the achieving of maximum efficiency, however that efficiency may be defined. Typically, if one attempts to push efficiency too far, one runs the risk of instability or of sudden loss of performance. To the extent that a computer system handling a stream of jobs can be regarded as analogous to a mechanical system or process plant, then the approach of the control engineer is relevant.

We are not yet in a position, and perhaps never will be, to write down equations of motion for a computer operating system. However, this does not exclude the design of a control system. Indeed, it is just in circumstances where the dynamical equations are not fully understood, or when the system must operate in an environment that can vary over a wide range, that control engineering comes into its own.

Some general remarks are made below about the principles on which control systems are designed. These lead to a discussion of how the same principles may be applied to the control problem associated with computer operating systems intended to process a stream of users' jobs.

Control fundamentals

The degree of complexity, or otherwise, that is necessary in a control system will depend on the degree of stability implicit in the basic design of the system to be controlled. If this is such that there is inherent negative feedback, then smooth operation may be possible without any super-imposed control system. An example is an electric motor which, if properly designed, will maintain a speed sufficiently constant for many applications without any form of control, drawing from the supply the amount of power required to meet the varying demands of a changing load. The simplest form of super-imposed control is that summed up by the words *steam engine governor*. This type of control acts retrospectively on the occurrence of an error; it is, in fact, error driven, and even in the steady state the error is never completely annihilated. A control system as simple as the steam engine governor will only work satisfactorily in an extremely small number of cases, another of which is the control of temperature by means of a thermostat. Watt, who is credited with the invention of the steam engine governor, was evidently lucky in that he had no problem over stability, and it was perhaps because this luck was not repeated that there was such a long interval before the modern subject of control engineering began to develop. As soon as control is applied to anything but the first derivative, or any delay is introduced, the problem of instability arises; one has only to imagine a steam engine governor that controlled the rate of flow of fuel to the boiler, or one in which a deliberate delay in the actuating of the steam valve were introduced, to appreciate this point. The theory of error-driven control systems in which the variables are continuous is now well understood.

In the case of systems in which a delay necessarily occurs between an action becoming necessary and the decision to take it becoming effective, a control system that can look ahead is required. If the delay is small the look ahead can be based on the current rate of change; the widely used p.i.d. (proportional, integral, derivative) controllers work on this principle. If, however, the delay is more serious, such methods are not sufficient to achieve stability and some knowledge of plant dynamics must be built into the control system. This is where a model of plant operation or of product behaviour comes in. The control system consults the model before deciding on the action to be taken in the plant. An essential feature is that the model should operate more quickly than the plant. In some cases, hill-climbing techniques can be used, that is, alternative courses of action can be tried on the model and the best one selected. Models typically contain parameters whose values must be continuously estimated with the aid of data obtained from the plant; a case in point is a parameter describing the efficiency of a catalyst in a reaction vessel. If, however, one does not wish to be concerned with the details of a model, one can think of it as a black box into which a number of inputs are fed and from which come one or more outputs that are used by the control system to decide on action. The fact that the model has to determine its own parameters means that the outputs are functions of past values of the inputs as well as of current values and may, therefore, be biased towards the past.

In some cases, *open loop control*, based on a model, may be entirely satisfactory in itself. An example is a simple autopilot that automatically determines from a model the degree of banking necessary to execute a prescribed turn. On the other hand, it may be necessary to supplement open loop control by an additional super-imposed control loop of the steam engine governor type.

Often it is necessary to add *safety devices* that come into action when the control system fails. These may be divided into two

classes: those which enable operations to continue and those which shut the plant down altogether. An example of the first class is the safety valve on a boiler and of the second the emergency control rods in a nuclear reactor. In many cases, the former class are not correctly to be regarded as safety devices at all, but as an additional level to the control system. Their presence may enable the next lowest level of control to be simplified and designed in a less conservative manner, with a resulting improvement in average efficiency. Although blowing off steam is inefficient, the presence of the safety valve enables a boiler to be run nearer to its maximum safe working pressure than would otherwise be the case.

Application to computer operating systems

An example of the direct application of control engineering principles to the control of a computer operating system will be found in Wilkes (1971). This concerns the automatic control of the number of users allowed to log-in to a time-sharing system. The object is to allow this number to be as great as possible without the degradation of response that comes from allowing it to be too large. The controlled variable is the total number of tasks awaiting attention within the system. If the average activity of the users were constant then, of course, it would simply be necessary to choose the right number of users to produce tasks at the required rate. In practice, the average activity varies over wide limits and it is the function of the control system to compensate for this. A system, known as the *load leveller*, that performed these functions, was designed by R. Mills for the CTSS at MIT. The load leveller was a good example of a super-imposed control system. It ran as though it were a user's program—although one having special privileges—and it could be used or not used according to requirements.

The aspect of operating system design that will be discussed in this paper is paging. It is notorious that the use of apparently innocuous scheduling and paging algorithms can give rise to the type of unstable behaviour known as thrashing. The achievement of optimum use of core memory and processor time, while avoiding the occurrence of thrashing, may be regarded as posing a problem in control engineering. The literature contains descriptions of a number of systems based on differing policies for scheduling and paging. These systems are usually described by their authors in isolation and it is not easy to see how they are related to one another. An attempt will be made here to provide a general description, or model, of a paging system that is applicable to a wide range of particular systems. This model will be discussed from the control engineering point of view. Later it will be shown how some of the paging systems referred to in the literature can be described in terms of the model.

Queuing theory

The approach made in this paper is complementary to the more usual statistical approach by way of queuing theory. Statistics, being concerned with an average over all possible situations, can throw no light on the way a system will behave in particular circumstances; that is, it can throw no light on dynamics. On the other hand, statistical considerations are highly relevant to the discussion of certain matters not treated in the present paper; for example, how much channel capacity, buffer space, etc. must be provided in order to secure efficient operation without congestion.

Process administration

An operating system must maintain lists of processes that are in various stages of passage through the system. Three such lists may be identified as being essential; in practice these lists may be subdivided, but we need not here be concerned with the subdivisions. The *waiting process list* contains processes that have been presented to the system, but have not yet been accepted by the supervisor for the allocation of a share of processor time. The *accepted process list* contains processes that have been so accepted, while the *active process list*, or *loaded process list*, contains processes that are actually loaded, which in a paging system is taken to mean that they are entitled to have pages in core. The number of processes on the active list is called the *level of multiprogramming* and it is denoted by L. L may have a constant value or it may vary dynamically.

The distribution of the available memory space between processes may be regarded as a problem in resource allocation. Since, however, core space is limited and must be re-used, there is a corresponding problem in resource de-allocation. These two aspects, those of allocation and de-allocation, may be divorced from one another by maintaining a reserve of free page-frames which are not allocated to any process and are, therefore, available for allocation when required. The reserve of free page-frames is replenished by a routine—*the de-allocation routine*—that withdraws page-frames from processes, while the *allocation routine* is responsible for issuing page-frames to processes that require them. The special case in which the reserve is maintained at zero level is mentioned below.

Given the above framework of definitions, a system may then be defined by specifying the following policies:

Scheduling policy

This is the policy according to which processes are transferred from the accepted list to the active list and *vice-versa*. It also covers the transfer of processes from the waiting list to the accepted list to make up for processes that have run to completion. The scheduling policy can be implemented by three separate routines, one for each of the three types of transfer just mentioned.

Processor allocation policy

This governs the allocation of processor time to the processes on the active list, and is implemented by one routine.

Page allocation policy

This is implemented by the two routines that have already been mentioned, namely the allocation routine and the de-allocation routine. To complete the specification of the paging policy, it is necessary to specify the circumstances in which the two routines are called in.

Choice of policies

It is now possible to list the various factors which may be taken into account in specifying the three policies, and to make some general remarks about the design of the policies themselves and the influence that they have on the characteristics of the system. Clearly, it is not possible to make an exhaustive listing of all factors that might be taken into account in the definition of a policy.

From the present point of view, the important decisions to be taken under the heading of scheduling policy concern the transfer of processes between the accepted list and the active list. In a time-sharing system, it is to be expected that a process that has failed to run to completion, after being on the active list long enough to receive a certain quantum of processor time, will be transferred back to the accepted list to wait its turn for another period on the active list later. In a non-time-sharing system a process, once transferred to the active list, would stay there until it had run to completion.

Some of the factors that may be taken into account in deciding whether to transfer a process from the active list back to the accepted list are as follows:

1. The amount of processor time that it has received while on the active list. Usually, on being transferred to the active list, a process is given an allocation of time that may depend,

for example, on the total amount of processor time that it has received and on the number of times that it has had a spell on the active list.
2. Whether the process has reached an input/output, as distinct from a page, wait, or is waiting for a semaphore.
3. The current observed requirement of the process for page-frames (considered in relation to any estimate made when the process was transferred to the active list or to the aggregate requirements of other processes).

A process is removed from the system altogether when it runs to completion.

The decision to transfer a process from the waiting list to the accepted list may depend on the following factors:

1. Whether the process is free to run; for example, whether any magnetic tapes needed have been mounted.
2. The number of processes on the accepted list.
3. The number of processes on the active list.
4. The number of processes that have run to completion in the recent past.
5. The priority (externally determined) attaching to the process.

A processor may become free as a result of the process on which it is working running to completion, being arbitrarily removed from the active list, or reaching a page wait. Depending on the policy adopted, it may also become free if the running process has exhausted a slice of time allocated to it, without necessarily having consumed all its allocation of time for its current period of residence on the active list. When a processor becomes free, the following factors may be taken into account when deciding which process is to be given the use of the processor next:

1. An arbitrary ordering of processes, for example their positions on a round-robin.
2. The relative seniority of the processes as determined by the order in which they entered the active list.
3. The relative priorities of the processes.

Alternatively, the choice may be made on a random basis.

In the case of the allocation and de-allocation routines we have to consider when they are triggered, what action they take when triggered, and the criteria on which that action is based. The allocation routine can be triggered:

1. When a new process enters the active list.
2. When a process demands a page.

The action taken can be:

1. To allocate a single page.
2. To allocate a specified number of pages.

Factors taken into account in deciding whether the allocation can be met may be:

1. Whether the process already has in core as many pages as it is allowed.
2. The seniority of the process.
3. The priority of the process.

The de-allocation routine can be triggered:

1. When a process leaves the active list.
2. When the reserve of free page-frames becomes zero or falls below a specified limit.
3. At regular or quasi-regular intervals.

The action taken can be:

1. To add a single page-frame to the reserve of free page-frames.
2. To add a specified number of page-frames.
3. To add to the reserve all page-frames satisfying certain criteria.

Factors taken into account in deciding whether a given page may be removed from core, and in which order pages shall be so removed, may be as follows:

1. Time, either real time or process time, that has elapsed since the page was last accessed.
2. Whether the owning process is on the active list or not.
3. Total number of pages already owned by the process.
4. Seniority of owning process (absolute or in relation to those of other processes waiting for pages).
5. Whether the page is owned by a single process or shared by several processes.

Nothing has been said about the policy in relation to the reserve of free page-frames, since this follows naturally from the policies that have been discussed above. If, for example, the de-allocation routine is triggered only when the reserve is exhausted, and if that routine delivers a single page, then in effect the reserve is maintained at zero level. Naturally, certain short cuts in implementation would be adopted if this were the case. One would probably have a single *placement routine* instead of separate allocation and de-allocation routines. The placement routine would have the responsibility both of deciding what page-frame to declare free and of issuing it to the demanding process.

A distinction may be drawn between policies in which a limit is set on the number of pages a process may own (not necessarily the same for all processes) and those in which no such limit is placed. This corresponds to the distinction made by Denning (1970) between local and global placement policies.

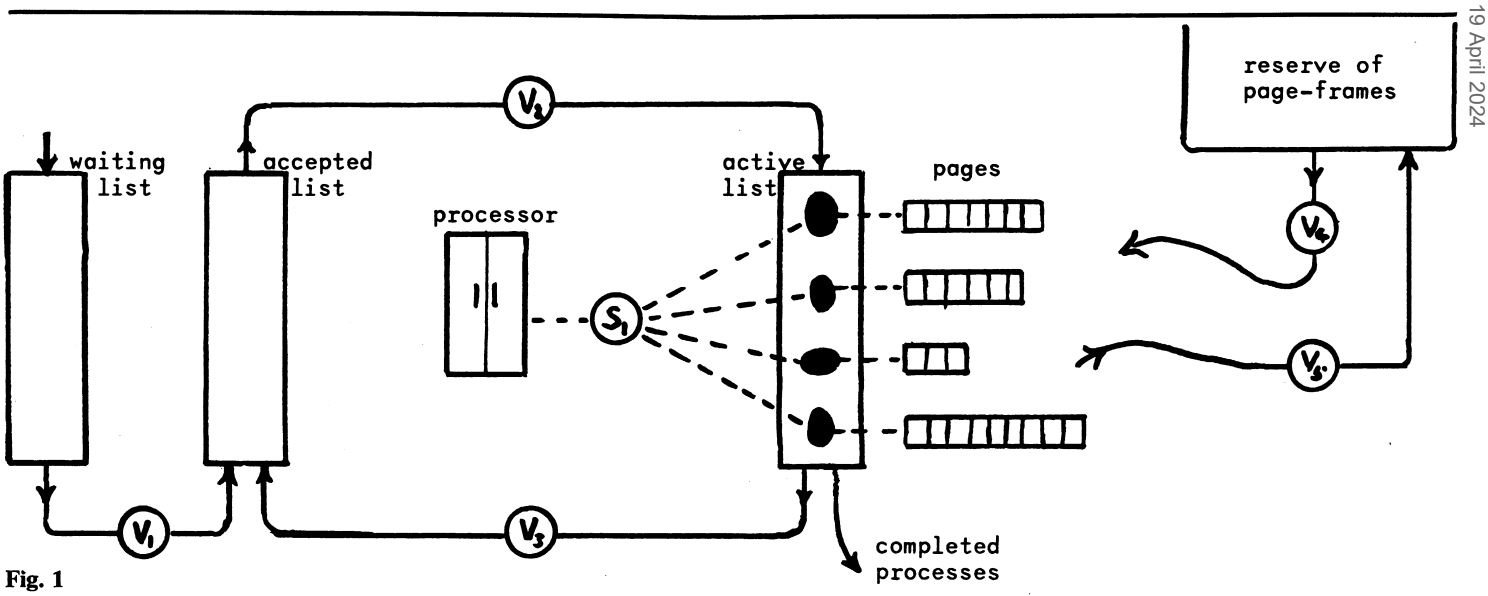

Fig. 1

The above listings of factors to be taken into account are not intended to be by any means exhaustive. For example, in addition to the externally determined priority of a process, account may be taken of whether it belongs to a foreground or a background job and of its need for peripherals, such as magnetic tape drives.

Flow of processes

Fig. 1 shows in diagrammatic form the way in which processes flow through the system, or rather through the part of it with which I am concerned in this paper. The flow of processes between the accepted list and the active list is controlled by two 'valves', V_2, V_3, and the flow of processes from the waiting list to the accepted list is controlled by the 'valve' V_1. A number of processes of varying shapes and sizes are shown as being currently on the accepted list with pages attached to them. Pages are withdrawn to the reserve through the 'valve' V_5 and supplied to the processes as required through the 'valve' V_4. The processor—only one is shown in the diagram for simplicity—can be associated with any process on the active list through the 'switch' S_1. It is to be emphasised that the figure is entirely diagrammatic and that the analogy with a process plant is not to be taken too seriously. V_1, V_2, . . ., V_5 and S_1, may be referred to as *control points.*

Control of V_3 by one of the algorithms implementing the scheduling policy, of S_1 by the processor time allocation routine, and of V_4 and V_5 by the allocation and de-allocation routines is straightforward, since the necessary decisions can be taken on the basis of information available at the time without any need to look into the future. The information taken into account can vary greatly from one system to another and the routines are not necessarily simple. The three policies on which the routines are based should be chosen with two objects in view. One is that together they should give the system the maximum degree of intrinsic stability in the sense that an electric motor has intrinsic stability. Some policies would obviously do just the contrary, for example one that caused the most recently used page-frame, rather than the least recently used one, to be declared free. It is not, however, always easy to see what the effect of a particular policy will be as regards intrinsic stability, and the matter is referred to again when examples are given. The second object in choosing policies is that they should maximise the proportion of time the processor spends in doing useful work and hence maximise the throughput of the system.*

In a process plant, a control engineer would perhaps install at points such as V_3, V_4, V_5, S_1, simple controllers taking account of present values and possibly also of present rates of change. Control of V_2, however, is not such a simple matter. If too few processes are allowed to enter the active list, there will be processor idle time and consequent inefficiency. On the other hand, if the number of processes on the active list is allowed to increase beyond a certain point, there will be insufficient pages to satisfy their needs and thrashing will occur. If processes were uniform in their paging requirements, the routine for controlling V_2 could be designed to keep the number of processes on the active list to a predetermined number. Indeed, some systems do work in this way. The number is chosen conservatively, in the hope that, with the work load actually encountered, the efficiency will be acceptable and thrashing will occur only rarely.

On the other hand, it is possible to devise methods of control of V_2 which make use of recorded knowledge of the past behaviour of a process to predict its future behaviour. This, in effect, is to use a model, and all such methods stem from the original concept of the *working set* introduced by Denning (1970). Although the working set is the main ingredient in such models of process behaviour, hypotheses about the circumstances in which a process changes course and starts building up a fresh working set may be incorporated, as may also means of estimating the working set of a process which has not yet run. It would be nice to have a model that was capable of more accurate prediction and relied less heavily on observations from the past. It is difficult, however, to discern any principles on which such a model could be constructed.

Control of V_1 involves a different set of considerations. If too many processes are allowed to pass on to the accepted list, the penalty will be, not instability, but a poor response. Control of V_1 is thus bound up with the problem of keeping the total load of the system within its capabilities, rather than with the problems of paging with which this paper is primarily concerned. The control of V_1 will not, therefore, be discussed further.

It has been remarked that there may be, in any control system, a number of levels of control. In the present context, an additional level of control may be superimposed on a system designed along the lines that have been described. One could, for example, arrange to detect the onset of thrashing—by an increase in processor idle time, or in the rate of paging—and to take suitable action to restore stability. For example, the number of processes on the active list could be temporarily reduced. This action could be taken either automatically or through operator intervention.

Attempts have been made to design systems which always tend to move towards a condition of thrashing and, as this condition becomes detected, to back off. Such a system of control, if effective, would be error-driven along the lines of the stem engine governor. There are two reasons why I would not expect such an approach to be very successful. In the first place, one normally thinks of error-driven control systems as applied to continuous variables and there are difficulties in adapting them to the discontinuous case. This is particularly so in the present instance, since the number of jobs on the active list is quite small. The other reason is that a paging system should, for efficiency, operate so that it is well away from the thrashing point, and not at the point at which thrashing is incipient or about to become serious. A system that was continually going into thrashing and backing off would tend to spend far too much of its time in a state of mild, or worse than mild, thrashing, or in recovering from the disruptive effects of the drastic action necessary to bring the thrashing under control.

Overall view

The overall view that emerges from the above discussion is of two levels of control. On the main level there are a number of independent but co-operating routines, using as input the current state of the system; however, one control point, namely V_2, is singled out as needing special attention. Since time must elapse between the taking of a decision to transfer an additional process to the active list and the becoming evident of the consequences, in terms of system behaviour, of that decision, it would appear to me that the total system can be described as being *under control* only if a policy based on prediction is adopted for the control of V_2. I do not deny that systems based on other policies may give satisfactory results in practice, but I would regard such systems as being essentially *tuned* to their work load and would expect retuning—to the extent even of drastic modification of the algorithms—to be necessary to

*From the point of view of the overall design of the system there are many considerations other than the avoidance of processor idle time. Processor efficiency must not, for example, be secured at the cost of excessive investment in core memory or channel capacity. However, once the other parameters are determined—that is, once the configuration has been fixed—it remains to choose the policies under discussion so that the maximum amount of work is done in a given time.

make them work equally well in differing circumstances. I would also expect occasional trouble when the work load happened to contain a mix of jobs having unusual space requirements. Even with predictive control, there will necessarily be an element of tuning on account of the basic limitations of the working set model on which the prediction must be based.

The superior level of control is of the safety valve type, and must include some provision for human decision, if only to purge from the system jobs whose working sets are too large for them to be effectively run.

Particular systems

An attempt will now be made to show, with the aid of the table, how a number of the paging policies that have been implemented or suggested may be described in the foregoing terms. Naturally, in such a table, only an outline can be given. Each row of the table refers to a particular combination of policies and the last column indicates, where appropriate, a system that has been implemented or discussed in the literature that comes near to the one of which an outline is given in the preceding columns.

A few comments are made on each system, but it is not part of the plan of this paper to survey, or discuss in detail, the various systems that are possible.

The system outlined in the first line of the table uses simple demand paging with a global placement algorithm according to which the page longest in core is always over-written irrespective of the process to which it belongs. There is no time-sharing and a process reaching an I/O wait is not removed from the active list, although it may in the course of time lose most or all of its pages. The second line shows a similar system with

Table Examples of particular paging policies

	V_2 LOADING OF A NEW PROCESS	S_1 ALLOCATION OF PROCESSOR	V_3 UNLOADING OF A PROCESS	V_4 PAGE FRAME ALLOCATION	V_5 PAGE FRAME DE-ALLOCATION	NOTES[1]
1	keep L constant	round-robin; a process loses processor on reaching an I/O wait	never	on demand	zero reserve; take LRU[2]	*c.f.* Atlas[3]
2	ditto	round-robin; a process loses processor on reaching a page wait	on reaching an I/O wait; on exhausting time allocation	ditto	ditto	time sharing with demand paging
3	keep sum of working sets less than core capacity	Senior process free to run pre-empts processor	on reaching an I/O wait; on exhausting time allocation	by pre-paging: on demand	on completion; on unloading; if reserve empty take LRU[2]	*c.f.* MULTICS[4]
4	keep L constant	highest priority process free to run pre-empts processor	never	on demand	on completion; if reserve empty take LRU[2] from process of lowest priority that owns pages	*c.f.* Wharton[5]
5	ditto	ditto	ditto	ditto	ditto, but also: at regular intervals take pages from processes not waiting for pages	*c.f.* Lynch[5]
6	load if space allocation (based on working set) is less than reserve	round-robin; a process loses processor on reaching a page wait	on reaching an I/O wait; on exhausting time allocation; on exceeding space allocation	by pre-paging; on demand	on completion; on unloading; at intervals take pages not in current working set of any process.	*c.f.* EMAS (Whitfield[6])

1. Systems mentioned in this column resemble in some, if not all, of their features the one outlined in the earlier columns.
2. Least recently used page irrespective of owning process.
3. See Kilburn *et al.* (1962).
4. See Organick (1972).
5. See Alderson *et al.* (1971).
6. Private communication.

time-sharing. In both these systems, the processes on the active list are arranged in a round-robin and whenever a process reaches a page wait the processor is offered to the process next in cyclic order. The disadvantage of this strategy is that control tends to pass to the process that has been waiting longest and hence is most likely to have lost some of its pages. It would be better to offer the processor, whenever the current process reaches a page wait, to one of the other processes chosen at random. This is, in fact, what the system degenerates into under conditions of heavy paging when any process to which the processor is offered will most likely not be free to run. A disadvantage of these types of strategy is that the processes compete with each other for pages on equal terms and the consequence can easily be a disorderly scramble.

This defect is corrected in the system used in MULTICS which is essentially that outlined in line 3 of the table. The processes on the active list are arranged in an order of seniority which is, in fact, their order of loading. When a process leaves the active list, then the process below it moves up in seniority and a new process may be loaded to become the junior process. The processor is always allocated to the senior process that is free to run. Any process emerging from a page wait preempts the processor from the running process if that process is lower in seniority. In this way, some order is introduced into the competition for pages. The junior process perhaps obtains the use of the processor only for brief intervals of time; these, however, may be sufficient to enable it to load the pages that it will require for continuous running when it has risen in seniority. A form of prediction based on the working set model is used to control V_2. The actual algorithms used in MULTICS are complex and information about them will be found in Organick (1972).

Line 4 shows a non-time-sharing system of interest in that it provides implicit control of the level of multiprogramming. Processes are again arranged in an order, although the order is determined by an externally assigned priority rather than by seniority of loading. In Wharton's system (Alderson, *et al.*, 1971), on which this line of the table is based, the priority applies, not only to the allocation of the processor, but also to the delivery of pages from the drum. If, when a processor demands a page, the reserve is empty, a page-frame is commandeered from a process junior to the demanding process, the process chosen for raiding being the most junior one that has pages in core. If the demanding process is itself the most junior process, or if all the processes junior to it have no pages in core, then the demanding process is halted. There may thus be a number of processes on the active list that have no pages in core. The effective level of multiprogramming is given by the number of processes that have pages in core, rather than by the number on the active list, and varies dynamically. The system has the property that the highest priority process runs in effect as though it were alone in the computer. There is no reason why it should not go on demanding pages until it occupies all the page-frames in core and no other process is able to run. In some situations this may be just what is desired; in others, it is to be regarded as a disadvantage that the senior process never loses pages, even those that it has not touched for some time.

The system shown in line 5 resembles that in line 4, modified, however, by the introduction of a 'drain' designed to bring pages back into circulation (Alderson, *et al.*, 1971). Refinements in strategy of this type can be most valuable in increasing the efficiency of the system. Similar refinements are often made with a view to increasing its intrinsic stability. However, they do not always have the effect desired, since it is extremely difficult to visualise what the consequences of any particular change will be in practice and even more difficult to make a theoretical analysis. In another modification to line 4 that was tried by Alderson *et al.*, arrangements were made to drain pages from processes at a rate proportional to the number of pages that they had in core. Since the higher priority processes were given priority in access to the drum, it was thought that, under conditions of heavy paging, the algorithm would then establish a bias in their favour as regards the acquisition of pages. Experience showed, however, and further reasoning confirmed, that this is not always sufficient to prevent the onset of thrashing, and that the algorithm is helpless in suppressing thrashing once it has started.

Line 6 refers to a paging system based on a thorough-going use of the working set model that has been implemented for EMAS, a system developed on an ICL (English Electric) 4-75 computer by a team led by H. Whitfield at Edinburgh University. It is of interest in that it uses a local, as distinct from a global, paging policy. A process, on being transferred to the active list, is given a *core allocation* and a *time allocation*. In the case of a process that has not yet run, these are determined by arbitrary rules. Periodically the working set of each process on the active list is re-computed and page-frames containing pages no longer in the working set are added to the reserve. If, in spite of this, the process tries to exceed its core allocation, it is immediately returned to the accepted list. It is similarly returned if it exhausts its time allocation or if it reaches a console wait. When a process is so removed, its working set is re-computed and both its core allocation and its time allocation are re-assessed. The re-assessment is based on the original assessment and on the reason for removal. A process that has run out of time, but whose working set is within its core allocation, will perhaps next time be allowed more time but less core. When it is returned to the active list, its working set will be pre-paged. Similarly, a process that has run out of space will have its core allocation increased. However, the fact that it has run out of core is taken as an indication that quite possibly it has moved to another part of its virtual memory, so that its former working set is no guide to the number of pages that it will require in future. No pre-paging is done, therefore, when such a process is re-activated, and it must load its pages one by one on a demand basis. The core allocation, rather than the size of the working set itself, is used to control V_2; no process is transferred to the active list unless there is enough space in the reserve to meet its maximum authorised requirement as specified by its core space allocation. The level of multiprogramming is thus determined dynamically. A consequence of the use of a local paging policy is that each process is able to operate within the number of page-frames allocated to it, and the misbehaving of one process cannot affect other processes.

References

ALDERSON, A., LYNCH, W. C., and RANDELL, B. (1971). Thrashing in a multiprogrammed paging system, *International Seminar on Operating System Techniques*, Belfast. Academic Press, 1973.

DENNING, P. J. (1970). Virtual memory, *Computing Surveys*, Vol. 2, No. 3, p. 153.

KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., and SUMNER, F. H. (1962). One-level storage system, *IRE Transactions*, Vol. EC-11, p. 223.

ORGANICK, E. I. (1972). MULTICS system, MIT Press.

WILKES, M. V. (1971). Automatic load adjustment in time-sharing systems, *ACM Sigops Workshop on System Performance Evaluation*, Harvard, p. 308.

WILKES, M. V. (1972). *Time-sharing computer systems*, 2nd edition, MacDonald, London; American Elsevier, New York.