McKEAG
and
WILSON

Studies in Operating Systems

APIC
Studies in Data Processing No. 13

Studies in Operating Systems

R. M. McKEAG and R. WILSON
edited by D. H. R. HUXTABLE

Academic Press
London New York San Francisco
A Subsidiary of Harcourt Brace Jovanovich, Publishers

STUDIES
IN
OPERATING SYSTEMS

A.P.I.C. Studies in Data Processing
General Editor: C. A. R. Hoare

1. Some Commercial Autocodes. A Comparative Study
   E. L. Willey, A. d'Agapeyeff, Marion Tribe, B. J. Gibbens and Michelle Clarke
2. A Primer of ALGOL 60 Programming
   E. W. Dijkstra
3. Input Language for Automatic Programming
   A. P. Yershov, G. I. Kozhukhin and U. Voloshin
4. Introduction to System Programming
   Edited by Peter Wegner
5. ALGOL 60 Implementation. The translation and use of ALGOL 60 Programs on a Computer
   B. Randell and L. J. Russell
6. Dictionary for Computer Languages
   Hans Breuer
7. The Alpha Automatic Programming System
   Edited by A. P. Yershov
8. Structured Programming
   O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare
9. Operating Systems Techniques
   Edited by C. A. R. Hoare and R. H. Perrott
10. ALGOL 60 Compilation and Assessment
    B. A. Wichmann
11. Definition of Programming Languages by Interpreting Automata
    Alexander Ollongren
12. Principles of Program Design
    M. A. Jackson
13. Studies in Operating Systems
    R. M. McKeag and R. Wilson

A.P.I.C. Studies in Data Processing
No. 13

STUDIES
IN
OPERATING SYSTEMS

R. M. McKEAG
and
R. WILSON
edited by
D. H. R. HUXTABLE

1976

ACADEMIC PRESS
LONDON NEW YORK SAN FRANCISCO
A Subsidiary of Harcourt Brace Jovanovich, Publishers

ACADEMIC PRESS INC. (LONDON) LTD.
24/28 Oval Road
London NW1

United States Edition published by
ACADEMIC PRESS INC.
111 Fifth Avenue
New York, New York 10003

Library of Congress Catalog Card Number: 75-45751
ISBN: 0-12-484350-6

Typeset by Typesetting Services Ltd, Glasgow and Edinburgh
Printed by Whitstable Litho, Straker Brothers Ltd.

Preface

This book is the second in a series publishing the results of a research project into Operating Systems Techniques. This project was undertaken by the Computing Science Department of the Queens University, Belfast, under the direction of Professor C. A. R. Hoare. It was sponsored by International Computers Limited and the Advance Computer Technology Project (Contract No. K78B/308) by whose kind permission this book is published.

The purpose of the research was to investigate, clarify and evaluate the practical techniques which have been used in the implementation of successful Operating Systems and to produce a reliable guide upon which an operating system designer can base his decisions.

The programme of work split into two distinct phases. Firstly, the collection and presentation of the structure and technology of selected operating systems. Secondly, the analysis of the techniques used in each system to achieve a particular function and to relate the technique with its objective and operational context.

This book presents the results of the first phase of the study and contains the descriptions of four key operating systems chosen because each has made a significant contribution to the development of operating systems.

Despite great differences in system structures and terminology it has been possible to create a series of essentially uniform descriptions. At the lowest levels of detail this has not been entirely possible, and each study is therefore supplemented by a short glossary of terms particular to the system, and to correlate the standardized terminology with that originally chosen by the system designer.

The information contained in this book is not normally available even to the users of the systems. It therefore fills a significant gap in the information available to users, teachers of computing science, students and designers of operating systems.

It will enable the user to gain a closer insight into the system he is using and to understand the extent to which existing techniques can solve his problems. To the teacher and student it will provide for the first time the basis for comparative studies. To the designer it will represent a source book of information on techniques and their practical application.

Finally, the authors wish to express their gratitude for the encouraging assistance received from the designers of the systems and their organizations for the way in which detailed design documentation has been made available, and for permission to publish the resultant descriptions of their system.

International Computers Limited
Bracknell, Berkshire.

D. H. R. HUXTABLE
R. H. McKEAG
R. WILSON

Contents

III. T.H.E. Multiprogramming System

R. M. McKeag

I. Introduction

The T.H.E. Multiprogramming System was designed and implemented by E. W. Dijkstra and five other members of the Department of Mathematics at the Technological University of Eindhoven: C. Bron, A. N. Habermann, F. J. A. Hendriks, C. Ligtmans and P. A. Voorhoeve.

Dijkstra (1968b) defines the purpose of the system to be the smooth processing of a continuous flow of user programs as a service to the University, and the objectives to be: firstly a reduction of turnaround time for short programs, secondly the economic use of peripheral devices, thirdly automatic control of backing store combined with economic use of the central processor and fourthly the economic feasibility to use the machine for those applications for which only the flexibility of a general purpose computer is needed, but (as a rule) not the capacity nor processing power. Bron (1972) adds a further objective: ease of use, by both operator and programmer.

The user sees the system as a batch processing one with no file store. To submit a job he supplies a paper tape containing his program in ALGOL 60 (the only language used in the system); the rudimentary job description is at the beginning of the tape and consists of the word "**algol**" followed by the maximum numbers of non-pre-emptible resources (viz., punches, plotter and magnetic tape decks) that the program will require, for these are the only resource requirements of which he is certain—he could do no more than estimate the others. Alternative forms of job are "**correction**" followed by a list of editing instructions and "**runst**" to obtain a print-out of the run statistics or log maintained by the system. Once started, a job has available to it two reader streams, two punch streams, one plotter stream and one printer stream. There are no restrictions on run time nor, subject to physical limitations, on store; the operator, however,

has ample opportunities to dispose of a program that is attempting to use excessive amounts of these resources.

The operating system does no job scheduling and this is done by the operator. In general he attempts to have one long job and up to three shorter jobs running simultaneously. The only constraint imposed by the system is that the number of magnetic tape decks required by a program must be available before that program is run. In addition, there is one extra "job stream" permanently reserved for the running of correction jobs. One unusual task that the operator has to perform is to collate the sheets of printer output: since the printer works on a sheet-at-a-time basis and not a file-at-a-time basis, it occasionally happens that the output of one user's file is temporarily interrupted to enable another user's output to be printed.

The computer chosen for this system was an Electrologica X8 with initially 32K and subsequently 48K 27-bit words of core store with a cycle time of 2·5 μs. In addition to a 512K drum with 1K words per track and a revolution time of 40 ms, the configuration includes three 1000 characters per second paper tape readers, three 150 characters per second paper tape punches, a plotter, a line printer and a console teletypewriter. Some time after the original system came into operation three magnetic tape decks were added.

The overall performance of the multiprogramming system can be measured against the manufacturer's single programming operating system. At Eindhoven both systems are used with the former taking an increasing share of the workload despite being 20% less efficient than the latter. The reasons why the multiprogramming system is preferred are firstly the short turn-around time for the majority of jobs and secondly the capability of running very large programs using the virtual store; it is this virtual store, implemented by software paging, that leads to the inefficiency figure of 20% mentioned above.

In this description of the T.H.E. Multiprogramming System certain liberties have been taken with the nomenclature. This has been done because this description is one of a series in which a number of operating systems are examined and treated in a reasonably uniform manner. A glossary of some of the terms used, together with their Eindhoven equivalents, appears towards the end of this chapter to assist those who are familiar with the T.H.E. terminology. The programming notation used is based upon ALGOL 60 with the addition of the data structuring notations of Pascal and a few other extensions whose meanings should be readily apparent.

Finally I should like to express my appreciation of the considerable help I have received from E. W. Dijkstra and C. Bron of the University of Eindhoven, and of the guidance I have received from C. A. R. Hoare of the Queen's University of Belfast.

II. System Structure and Synchronization

A. OBJECTIVES

The system is organized as a society of sequential processes which progress in parallel with undefined relative speeds and which communicate with one another using well defined synchronization rules. In this way the problems of dealing with irreproducible time-dependent phenomena, arising from real-time interrupts or from the arbitrary scheduling of resources, are avoided.

The control of the resources of the system is organized hierarchically: at a particular level the resources of a given type are treated in such a way that at all higher levels the physical resources have lost their identities and have been replaced by virtual resources. Thus a process at one level operates entirely in terms of virtual resources at lower levels and need not be aware of what processes and resources exist at higher levels. The simplicity of this structure enables the logical soundness of the design of the system, and the correctness of the implementation, to be readily verified.

B. A SOCIETY OF PROCESSES

1. *Introduction*

The foundations of the T.H.E. system have been clearly described by Dijkstra (1968a, b). He has explained that the structure of the system was considerably influenced by his fears about the possibility of debugging software that depends upon irreproducible real-time interrupts. Consequently, since only the time succession of logical states of a sequential process is meaningful and not the speed with which the process progresses, the system was designed as a society of sequential processes that progress in parallel with undefined speed ratios.

Much of each process is entirely independent of all other processes in the system and so its correctness is entirely independent of the speed of the process. In the other parts, in which the process wishes to inspect and update common data or to synchronize itself with respect to external events such as interrupts, the synchronization is explicit: no reliance is placed on the fact that a particular event will occur at a particular time. Thus the speed with which a process progresses can never affect the interior logic of that process.

In the T.H.E. system there are 15 processes: one corresponding to each independent activity in the system. In particular there is one process associated with each peripheral device: the drum, the operator's console, the printer, the plotter, each of the three readers and each of the three punches; each of these processes synchronizes its activity with respect to interrupts

from its peripheral device. There are also five processes corresponding to the five user programs that may be executed concurrently.

2. *Mutual Exclusion—The Binary Semaphore*

Part of each process consists of operations upon data local to that process. The rest consists of operations upon common variables. It is important that the latter operations are protected so that while one process accesses a set of shared data all other processes are excluded from referencing that data. This is known as the "mutual exclusion" problem and a section of program in which exclusive access is required to some common data is called a "critical section". It behoves any process wishing to enter a critical section to test whether it is excluded from doing so; if so the process must wait; and when it may enter it must signal to other processes that they are excluded. On leaving a critical section the process must signal that the shared data is again accessible, and it must accept responsibility for waking one and only one waiting process, if there is one. It is also necessary that these entry and exit operations are not interrupted by some other process.

For this reason Dijkstra (1968a) has introduced a mechanism called a "semaphore" which is a non-negative integer variable that is subject to two operations, P and V. A V-operation increments by one the value of the semaphore in one indivisible operation. A P-operation performs, in one indivisible operation, a test to determine whether the value of the semaphore is zero and, if not, decrements the value by one; if however it is zero the process suspends itself pending the occurrence of a corresponding V-operation and thus the satisfactory completion of the interrupted P-operation. In practice the effect of the V-operation is extended to test whether some other process may now complete its interrupted P-operation and, if so, to permit it to continue. The V-operation derives its name from the Dutch word "verhogen" meaning "to raise", while the P-operation takes its name from the non-existent portmanteau word "prolagen" which is compounded from the Dutch verbs "proberen verlagen" meaning "to attempt to lower".

A semaphore that takes only the values 0 and 1 is called a "binary semaphore" or "Boolean semaphore". Such a semaphore may be used to control access to critical sections: a binary semaphore *mutex*, with initial value 1, may be declared in association with a set of shared variables; prior to entering a critical section associated with this data, a process would perform a P(*mutex*) operation and, on exit, V(*mutex*). A mutual exclusion semaphore is characterized by the fact that each process that uses it will always perform an ordered pair of operations (P then V) upon it. At any moment there may be as many as $(n-1)$ processes waiting on *mutex*, where n is the number of processes interested in the data.

3. *The Private Semaphore*
Associated with each process is a "private semaphore". This is a binary semaphore, initially 0 in value and characterized by the fact that only the process to which it belongs ever performs P-operations upon it. It is used when its process is unable to continue, typically because some resource is unavailable. At some time another process will release the resource and, on noting that the suspended process can now continue, will perform a V-operation on the semaphore.

4. *Producers and Consumers—The General Semaphore*
It is common for two (or more) processes to work in such a manner that one produces items which the other consumes. The consumer must have some way of suspending itself if production cannot keep pace with consumption. For this reason the "general semaphore" is introduced: it can take any non-negative integer value and is 0 initially. Whenever an item has been produced the producing process performs a V-operation upon the semaphore associated with the queue of items; and whenever a consumer is ready to consume another item it performs a P-operation on the semaphore and is suspended if the queue of items is empty. The value of the semaphore therefore represents the length of the queue of items.

5. *The Implementation of Semaphores*
In the EL X8 computer of the T.H.E. system the indivisibility of the V- and P-operations can be ensured by the use of an instruction to "add contents of store location to register, leaving the result in both". However advantage is taken of the fact that the computer has only a single processor and so mutual exclusion can be achieved by disabling interrupts for short periods.

Semaphores in this system are classified as either "software semaphores" or "hardware semaphores". The latter are those whose V-operations are performed not by software processes but by peripheral devices. Dijkstra (1968b) calls this "an interrupt system to fall in love with"; it is described more fully later. The main characteristic of a hardware semaphore is that its value is stored in a fixed main store location known to the hardware of the peripheral device. Every semaphore contains, as well as its value, a count of the number of processes waiting on it.

C. A HIERARCHY OF RESOURCES

1. *The Structure*
The purpose of the operating system is to share the hardware amongst the various processes that run concurrently. These hardware resources are organized in a hierarchical manner, and at each level of the hierarchy some

resource type is treated in such a way that at all higher levels the physical resources of that type have lost their identities and have been replaced by virtual resources.

At the lowest level the dispatching of processor time takes place. Above this level the actual number of processors in the computer is irrelevant and each process is given the impression that it has its own dedicated processor; this virtual processor is of course slower than the actual processor and less regular but that does not affect the interior logic of the process. This level consists of some shared routines (and data) which are used to implement the P- and V-operations and which also respond to clock interrupts by switching the processor from one process to another. By disabling interrupts a process may ensure that it has exclusive access to this *dispatcher*.

The mapping of the virtual store onto the physical (core and drum) stores is to be found at the next level. At higher levels the two-level physical store ceases to exist and the identification of information takes place entirely in terms of the virtual store. This level contains several shared routines (and data structures) by which processes map their virtual addresses onto actual addresses, acquire and release areas of core and drum store, and communicate with the *drum process* whose task is to transfer information between core and drum in synchronism with the latter.

The next level is concerned with the sharing of the operator's console. At higher levels each process is under the impression that it has its own console. The level comprises several shared routines (and their data) used by processes to acquire and release the console and to communicate with the *console process*. This process interprets messages typed by the operator and routes each to its destination process.

The remaining resources—the readers, punches, printer and plotter—are treated at the next level. Corresponding to each of these eight input/output devices is a process which, like the *drum process* or *console process*, may be considered to be a software extension of the hardware with which it is associated. Higher level processes perform their input and output by operating upon a number of virtual devices which, at this level, are mapped onto the actual devices. Thus at this level there are a number of streams of buffered input and output (these represent the virtual devices), together with the shared routines for operating upon the streams and for communicating with the device processes.

Above the hierarchy of resources and its 10 supervisor processes are to be found the five user processes that constitute the uppermost level. Four of these are general purpose and run the majority of the user jobs, while the fifth is reserved by the data preparation staff for editing paper tape and is special purpose in that it always uses the same reader and punch.

It can be seen that a program at a particular level operates entirely in

terms of virtual resources provided at lower levels and is unaware of what processes and resources exist at higher levels. If a program needs to use a resource that is not available at that level in virtual form, then it must work in terms of the physical resource. Thus in this system the *drum process* and the shared routines and data dealing with processor dispatching and store management must occupy a permanent allocation of core store. For reasons of efficiency other routines and data of the operating system also reside permanently in main store.

Processes that make use of the virtual store are obliged to recognize that the size of the underlying physical store is limited, and so they must cooperate in ensuring that sufficient storage is always available to enable the system to continue working satisfactorily without being slowed down or stopped by a bottleneck. This means that from time to time a process must wait until an adequate amount of store is available whereupon it receives permission from one of its fellows to continue.

Except that mutual exclusion is occasionally ensured by disabling interrupts, all synchronization is effected using semaphores. Processes that invoke the shared routines described above use mutual exclusion semaphores to protect the data accessed by those routines, they use private semaphores when they are obliged to wait for permission to continue, and they use general semaphores to coordinate their producer or consumer relationships with other processes.

2. *Reasons and Consequences*

The reasons for assigning particular resources to particular levels were pragmatic. Console control is at a higher level than store management so that the former can make use of the virtual store provided by the latter and thereby avoid the unnecessary expense of permanently occupying a partition of main store. The input/output processes must be able to converse with the operator and so must be at a higher level than the *console process.*

In retrospect, however, this allocation of resources to levels turns out to be correct for a more fundamental reason. If a program invokes an operation upon a virtual resource (and this operation may involve a delay if no corresponding real resource is immediately available) then it is important that the time taken by the operation is smaller (by one or more orders of magnitude) than the "grain of time" in which the invoking program works. So if an operation upon a virtual resource is to be considered as atomic at higher levels, the resources of the system must be assigned to levels in the hierarchy in such a way as to reflect the grains of time in which they work. For example, while it is reasonable for the *console process* to invoke the *drum process* and suffer no significant delay the converse is not true and so store management must be at a lower hierarchical level than console control.

Since a call on a lower level of the hierarchy is to be considered as atomic it is important that any delay suffered during the operation be limited; in other words it is important that no scheduler in the operating system should delay any of its callers for too long. Not only would an excessive delay upset the grains of time of programs at higher levels and thus make response times and turnaround times unpredictable for the user, but, and this is more important, serious instabilities may develop in the system. For example, if a process is starved of processor time for too long it may find, when it is at last dispatched, that all of its pages have been removed from main store; such instabilities tend to spread like wildfire.

The grains of time argument apart, there are several good reasons for adopting a hierarchical structure. One is that it is a simple method for controlling the complexity of a large programming task: the system designer can construct each level with little or no regard to the rest of the system: he works in terms of virtual resources supplied by lower levels and he is unaware of what exists at higher levels (except in a very broad sense). Consequently the designer of a scheduler for a particular level has little data available to him and so is obliged to construct an algorithm that is not only simple (and hence cheap and readily amenable to analysis) but also independent of, rather than attuned to, the workload it is to support.

An impressive consequence of using this structuring method is that Dijkstra (1968b) was able to claim that "the resulting system is guaranteed to be flawless" because he was able to verify the logical soundness of the design and to test the correctness of the implementation. The verification of the design was established by regarding each process as being cyclic with a neutral "homing position" which it leaves on accepting a task and to which it returns when and only when that task has been completed; the system is at rest when all processes are in their homing positions. The harmonious cooperation of the processes was then proved in three stages. Firstly, as each process activated to perform a single task can be shown to generate a finite number of tasks, and as circularity of task generation is excluded because a process can only generate tasks for processes at lower levels of the hierarchy, no single initial task can give rise to an infinite number of new tasks. Secondly, it was demonstrated that it is impossible for all processes to return to their homing positions leaving a generated but unaccepted task. Finally, it was proved, using induction, that after the acceptance of an initial task all processes eventually return to their homing positions.

Verifying the correctness of the implementation consisted of testing the lowest level by itself at first, and then adding and testing the other levels one by one. The testing of each level required that a set of test processes had to be written to force the program being tested into all its "relevant states" and to check that the system continued to work as expected. That the team

were able to decide what the relevant states were, that they were able to convince themselves that they had not overlooked any such state, and that they were able to generate them all, were due to the simplicity of the scheduling algorithms and the hierarchical structure which ensured that there were few states to be tested at each stage.

D. PROTECTION

An important feature of the T.H.E. system is that every process is well behaved and so the operating system is absolved from the policing function normally associated with supervisors. This is achieved by limiting users to high level languages (ALGOL 60 in this case) so that much checking can be performed at compilation and the rest is carried out during execution by routines that are invoked by calls compiled into the user program. The efficiency of this approach is seen when one compares it with the more usual strategy in which the operating system checks all supervisor calls for validity. For example, in the latter case, a call upon one level of the supervisor from a higher level of the supervisor would result in an unnecessary check, and every attempt to use any resource would be accompanied by a check that the resource has been allocated to the prospective user; in most operating systems such checks are frequently expensive.

E. PERFORMANCE

Perhaps the best measure of the success of the system is the extent to which it has influenced other operating system designers. The use of semaphores for synchronization is widespread as is admiration for the hierarchical structure. What is surprising is that the structure has rarely been copied. B. J. Moore has remarked that "there seems to be a cosmic force that makes operating systems monolithic"—perhaps the self-discipline of system designers needs to be reinforced by suitable design language notations in order to oppose this force successfully!

III. Process Dispatching

A. OBJECTIVES

A principal objective in dispatching is to execute short jobs quickly by letting new jobs receive priority over old jobs for a time. Another important

objective is to keep the peripheral devices busy by letting peripheral bound jobs take precedence over processor bound jobs. Both these aims are achieved by favouring jobs that have used little processor time in the recent past.

In dispatching it is important to minimize the chances of bottlenecks developing, and to this end the *dispatcher* services system processes, especially those at a low level in the hierarchy, before user processes; it can also give preferential treatment to processes while they are obeying critical sections.

B. MAIN FEATURES

The *dispatcher* maintains a priority-ordered *dispatch queue* of all the processes; each time the *dispatcher* is entered it selects for processing the highest priority process ready to run. Every two seconds the *dispatch queue* is reordered: the *drum process* remains at the head followed by the *console process* and then the other peripheral processes; at the tail are the user processes: for each user process an exponentially damped count of the processor time it has used is maintained and the processes are ordered by increasing time count. In addition, there is a facility whereby a process that wishes to receive preferential treatment for a short time, typically while obeying a critical section, may be marked as "urgent"; the action of the dispatcher is thus modified to search for the highest priority urgent process before attempting to run any of the non-urgent ones. If there is no process ready to run then the *dispatcher* goes into an idling loop.

The *nucleus* may be entered as the result of an interrupt: either a clock interrupt in which case the *dispatch queue* is reordered, or a peripheral interrupt in which case the suspended process awaiting this interrupt is released. In either case the *dispatcher* is then invoked to select a process to run.

The *nucleus* may also be entered as a result of a process wishing to perform a V-operation on a semaphore. If there is a process waiting on this semaphore then the highest priority urgent process or, failing that, the highest priority non-urgent process that is waiting is released. The process performing the V-operation may take this opportunity to dispose of its "urgent" status. Finally, control returns to the original process unless there is now the possibility that another process has higher urgency or priority in which case the *dispatcher* is invoked.

The *nucleus* may also be entered by a process wishing to perform a P-operation on a semaphore. If the P-operation is unsuccessful the process marks itself as "suspended" and invokes the *dispatcher*. Before doing so, however, it may take the opportunity to mark itself as "urgent".

C. DATA STRUCTURES

Each of the 15 entries in the *dispatch queue* is a pointer to the base of the stack of the corresponding process. The base of each stack contains certain control information in fixed positions: this includes an indication of the urgency of the process and whether it is active or suspended and, if the latter, a pointer to the semaphore on which it is waiting. When a process is suspended the values of its registers are preserved in the base of the stack. The other item of control information stored in the base of the stack is an exponentially damped count of the processor time used by the process; it is this value that is used when the *dispatch queue* is re-ordered every two seconds.

The time count for process i at time t_0 represents the following value:

$$\text{time}_i(t_0) = \alpha \int_{-\infty}^{t_0} f_i(t)\, e^{-\alpha(t_0 - t)}\, dt$$

where α is the damping factor and $f_i(t)$ takes the value 1 while process i is running and 0 while any other user process is running. It is assumed that time t increases when a user process is being executed and remains stationary otherwise. The values of "time" are only relevant for user processes since they are the only processes that are rearranged in the *dispatch queue*.

After a time interval Δt during which process j is running:

$$\begin{aligned}
\text{time}_i(t_0 + \Delta t) &= \alpha \int_{-\infty}^{t_0 + \Delta t} f_i(t)\, e^{-\alpha(t_0 + \Delta t - t)}\, dt \\
&= \begin{cases} e^{-\alpha \Delta t}\, \text{time}_i(t_0) & i \neq j \\ e^{-\alpha \Delta t}\, \text{time}_i(t_0) + \alpha\, e^{-\alpha \Delta t}\, e^{-\alpha t_0} \displaystyle\int_{t_0}^{t_0 + \Delta t} e^{\alpha t}\, dt & i = j \end{cases} \\
&= \begin{cases} e^{-\alpha \Delta t}\, \text{time}_i(t_0) & i \neq j \\ e^{-\alpha \Delta t}\, \text{time}_i(t_0) + e^{-\alpha \Delta t}\, (e^{\alpha \Delta t} - 1) & i = j \end{cases}
\end{aligned}$$

Rather than update all values of time_i we only update time_j, so:

$$\text{time}_i := \begin{cases} \text{time}_i & i \neq j \\ \text{time}_i + (e^{\alpha \Delta t} - 1) & i = j \end{cases}$$

Now we may make the approximation:

$$e^{\alpha \Delta t} - 1 \approx \alpha\, \Delta t$$

and so the only computation necessary is to add $\alpha\, \Delta t$ to time_j after process j has used Δt units of processor time. As α is chosen to be 2^{-10} and $\Delta t \leqq 200$ (units of 10 ms), the maximum value of the main error term introduced

by the approximation is given by $\frac{1}{2}(\alpha\,\Delta t)^2 \approx 0{\cdot}02$ units where $\alpha\,\Delta t \approx 0{\cdot}2$ units.

On every clock interrupt each time_i is normalized by division by $\sum_{2s} \Delta t$ to prevent overflow. The effect of this implementation is to record for each process between eight and 10 seconds of history. One result of this is that each new job starts life with a high priority since it takes several seconds for the tape containing a new job to be read in and during this period the value of the job's time count remains almost stationary while those of the other processes increase. When the system is initially started, $\text{time}_i = 0{\cdot}2$ for each of the five user processes.

D. PERFORMANCE

The dispatching algorithm is simple enough for its effects to be readily apparent. Firstly, urgency can reduce the chance of bottlenecks developing in critical sections. Secondly, the grain of time rule is respected by keeping supervisor processes at the head of the *dispatch queue*. Thirdly, peripherals are kept busy, users are not indefinitely ignored, and short jobs receive fast service because of the periodic re-ordering of the user processes on the *dispatch queue*. Fourthly, the overheads can be seen to be small but this is because there are only five user processes to be sorted; if there were considerably more then it would be cheaper, but probably just as effective, to perform a partial sort rather than a complete sort.

IV. Store Management

A. OBJECTIVES

The purpose of the store management system is to present to its users (i.e., all processes at higher levels of the hierarchy) a large uniform virtual store that is both larger and easier to use than the two-level physical store.

The ways in which the virtual store is used have been very carefully designed to respect the physical limitations on the amount of backing store and main store available.

B. IMPLEMENTATION OF VIRTUAL STORE

1. *Main Features*

The store management system provides a large virtual store of 512-word pages which are mapped onto the smaller real stores (consisting of core and drum) which are divided into 512-word "page frames". A *main frame table* and a *drum frame table* are maintained to record the states of the main store

and drum store frames. In addition there is, for each virtual page that has been declared, a descriptor indicating the location of the frame, if any, in which it is held. The movement of pages between main store and drum is carried out by a *drum process*. The page replacement strategy is "least recently used".

The paging system is implemented entirely by software: each program ensures that the pages it wishes to use are in main store at the right times, and this entails explicit invocation of the *drum process* when pages have to be read from or written to drum; the code to cater for such page accesses is generated by the compiler.

2. *Data Structures*

(*a*) *Drum Store.* The drum consists of 512 tracks, each of which contains two page frames. The positions of the frames are staggered around the drum so that the frames pass the heads in order of (cyclically) increasing frame number. The address of the first location of frame f is given by word f of track (f **mod** 512).

The *drum frame table*, which is resident in main store, may be represented as follows:

drum frame table: **array** *drum frame number* **of** *drum frame state*

where:

type *drum frame number* = 0 ... 1023
type *drum frame state* = (*free, in-use*).

The *drum frame table* is arranged in such a way that a simple algorithm using normalizing orders can quickly discover which free frame will next come under the writing heads; in this way, by taking advantage of the fact that logically contiguous pages need not occupy contiguous drum frames, it is possible to minimize the delay in writing a page to drum.

(*b*) *Main Store.* The main store originally consisted of 32K words but was later increased to 48K. The *main frame table* contains, in general, for each main frame a four-word entry describing the state of that frame. The exceptions are the 23 frames at the lower end of main store as they contain resident supervisor pages and so their states never change. The *main frame table*, which is resident, may be represented as follows:

```
main frame table: array main frame number of
                  case main frame state: (free, in-use, incoming, victim)
                  of
                    free: (next: (main frame number, null));
                    in-use: (copy: (drum frame number, null);
                             lock: 0 ... 15;
                             time: 0 ... ∞);
```

incoming: (*copy*: *drum frame number*;
lock: 0...15;
time: 0...∞);
victim: ( )
end

where:

type *main frame number* = 23...95 (say).

It may be seen that each main frame has one of four states:

(i) Free—All the free frames are chained together to form a stack, the top element of which is referenced by a *free frame pointer*:

free frame pointer: (*main frame number, null*).

(ii) In-use—The page occupying an in-use frame may be unique, or it may have a copy on the drum, thus by-passing the need to dump the page if it is to be discarded. If the page is altered in any way the corresponding drum frame is freed and the "copy" field is set to null. Certain pages must be locked down in main store: for example those associated with dispatching and store management, stack pages and those pages containing code currently being executed; as the same page may be used simultaneously by several processes a count is kept of the number of times each page has been locked. Locking may also be used when a process wishes to make, say, one access to an array element and wishes to ensure that the page is not discarded between checking its presence and performing the access: if the page is in main store this may be accomplished by temporarily disabling interrupts but otherwise the process requests that the page be read and promptly locked, and then, after the array element has been accessed, the page may be unlocked (as far as this operation is concerned) by decrementing the lock count. Another use of locking is by the ALGOL system which is allowed to lock a second page of code in addition to the one currently being executed: this is to speed the use of simple parameter subroutines. Whenever a main frame is accessed, a "timer" is incremented and its value is recorded in the *main frame table* entry for that frame: this is used by the "least recently used" algorithm when selecting a page to be discarded.

(iii) Incoming—Such a frame is awaiting the transfer of a page from drum and so is in transition from "free" to "in-use".

(iv) Victim—At any one moment there is one and only one "victim" frame; its use is described below.

(*c*) *Descriptors.* As each page is declared, a descriptor is set up for it in main store; this descriptor usually resides in the same place throughout the

lifetime of the page and so each page is identified by the address of its descriptor. The descriptors of the pages belonging to a process reside in the stack of that process, but the descriptors of pages of input/output streams (which, for the time being, belong to neither the process that produced them nor the process that will consume them) are stored in a fixed area of main store. Consequently, when a process unchains a page from an input stream or chains a page onto an output stream it moves the appropriate descriptor to or from its stack. Descriptors may be represented as follows:

```
type descriptor = case descriptor state: (none, main, coming, drum) of
                      none: ( );
                      main: (location: main frame number);
                      coming: (location: main frame number);
                      drum: (location: drum frame number)
                  end.
```

Initially the contents of a page will be empty and so no frame will be associated with it. The first time that an assignment is made to the page a main frame is selected as follows. The selection algorithm produces a "candidate" which will be a free frame, if there is one, or otherwise the least recently used frame that is not locked. If this candidate is free or has a copy on drum then it is immediately chosen, otherwise the current "victim" is chosen and the candidate becomes the new victim and is written away to drum. Should reference be made to a page that has been discarded, that page is read to the victim and a new victim is selected. There may of course be a delay in using a victim for a new page because its old page may not have been completely dumped. There is no danger of a program trying to read the old page back from drum before it has been written there from the victim because drum transfers are dealt with on a "first come, first served" basis.

(*d*) *Drum Transfers.* A process wishing to read a page from the drum or copy a victim to the drum communicates with the *drum process* by placing a message in a cyclic *message buffer* and performing a V-operation on the associated semaphore. All this takes place within a critical section. The requesting process then waits upon its private semaphore until its request has been dealt with. With N concurrently running programs a buffer with a capacity of $N+1$ messages is always sufficient. It was found that the extra complexity involved in using a smaller buffer was nearly prohibitive. Associated with the cyclic message buffer are two pointers: an *insertion pointer* which is updated in the critical section in which messages are added to the buffer, and an *extraction pointer* which is updated by the *drum process*:

```
message buffer: array message number of (message, null);
insertion pointer, extraction pointer: message number
```

```
where:
    type message number = 0...N;
    type message = record
                       semaphore of calling process;
                       read page: case transfer1: (yes, no) of
                                      yes: (source: drum frame number;
                                            destination: main frame number);
                                      no: ( )
                                  end;
                       write page: case transfer2: (yes, no) of
                                      yes: (source: main frame number;
                                            destination: drum frame number);
                                      no: ( )
                                  end
                   end.
```

Each message consists of two parts: the first is concerned with the acquisition of the required page (or empty frame) and the second is concerned with the disposal (if relevant) of the page residing in the newly selected victim frame. The requesting process supplies the identity of its own semaphore so it can be woken by the drum process between dealing with the two parts of the message.

C. DISTRIBUTION OF VIRTUAL STORE

1. *Main Features*

Although the number of pages of virtual store that may be declared is unlimited, the number that may be in use at any moment is limited by the size of the drum. Of these latter pages a certain number are permanently reserved for system and library routines and so on. Of the rest, some are free for allocation, and each of the others belongs to one of three categories according to whether it has been allocated to an input stream, an output stream or a user program. The numbers of pages belonging to these categories, or combinations of these categories, are restricted by policy decisions that reflect the physical limitations; consequently the programs that use the virtual store must cooperate with one another to observe these restrictions.

Thus, a process wishing to increase the number of pages belonging to one of these three categories will only do so provided that the limits will not be exceeded, if this is not possible the process suspends itself. Conversely, removal of a page from some category does, in general, impose upon the removing process an obligation to arouse a waiting process, if there is one. In order to ensure that no process is held up permanently in this manner, the policy adopted is to ensure that output continues as fast as possible,

and to impose upon the output processes the obligation to keep the other processes as active as possible. For a detailed description of this strategy for balancing the use of the virtual store, see Bron (1972).

2. *The "Sugar Lump"*

As described above, about a hundred pages are used by permanent material such as library routines and so forth. Of the rest, *total* pages say, some will be free and each of the others will belong to one of three categories:

i = number of pages belonging to input files,
o = number of pages belonging to output files,
p = number of pages belonging to user programs.

These three quantities are constrained to satisfy four relations:

$$\begin{aligned} i+o+p &\leqq total \\ i+o &\leqq maxio \\ i+p &\leqq total-reso \\ i &\leqq maxio-reso \end{aligned}$$

The first restriction arises from the physical size of the drum. The second is due to the fact that, for reasons of simplicity and efficiency, the descriptors to pages belonging to input and output files are kept in main store and are limited to a certain number, *maxio*, which is chosen to be 256: this represents a quarter of the drum and is sufficient to buffer a complete reel of paper tape. The third restriction is due to a wish to reserve a certain number of pages, *reso*, for output use: the reason for this is that it is important to ensure that when storage is in short supply it is still possible for the output peripherals to continue working and thus reduce the danger of store exhaustion; the value of *reso* is chosen to be 64, in other words a quarter of the input/output area, and the part that these pages play will be described later. The fourth restriction reinforces the third by ensuring that in addition to the *reso* pages that are reserved for output buffering there are also *reso* descriptors in the input/output area.

The state space delimiting the range of values of (i, o, p) may be illustrated by Fig. 1, termed the "sugar lump", in the accompanying diagram. The faces are the planes:

$$\begin{aligned} i &= 0 \\ o &= 0 \\ p &= 0 \\ i+o+p &= total \\ i+o &= maxio \\ i+p &= total-reso \\ i &= maxio-reso \end{aligned}$$

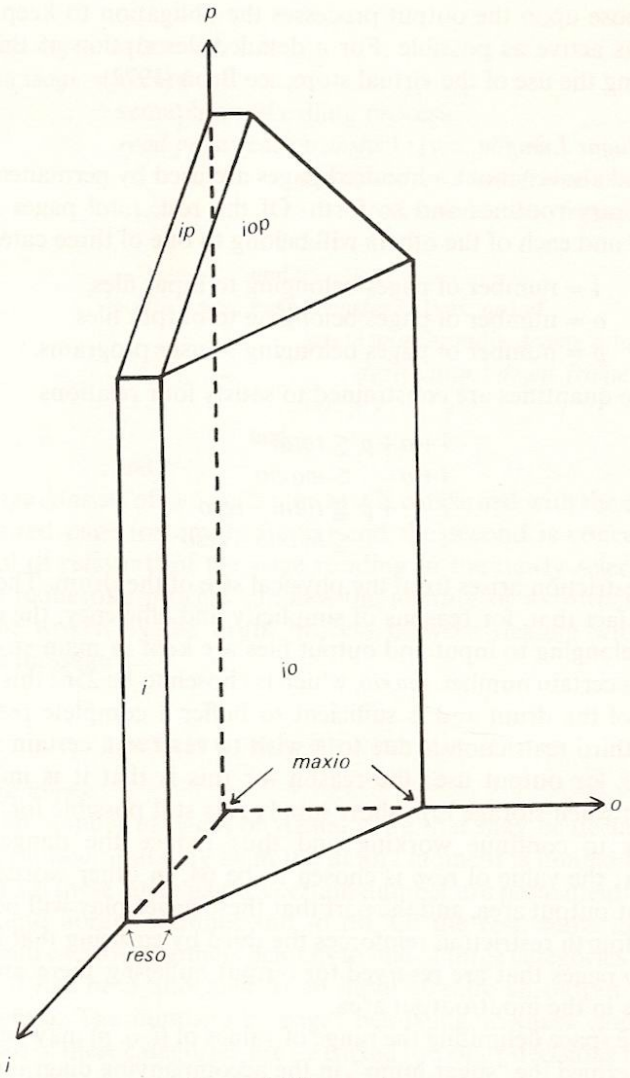

FIGURE 1. The sugar lump.

The sugar lump is a state space, and any process that causes a change of state, and thus a movement within the sugar lump, must ensure that the bounds of the space are not violated. For this purpose the following four quantities are introduced to denote the amount of "elbowroom" left in the four inequalities, that is they relate to the numbers of pages currently free for specific purposes:

fiop	"free for input, output and programs"	(initially = *total*),
fio	"free for input and output"	(initially = *maxio*),
fip	"free for input and programs"	(initially = *total* − *reso*),
fi	"free for input"	(initially = *maxio* − *reso*).

Sequencing of regions of program that increase or decrease these quantities is done in such a way that the quantities never become negative. The six possible changes of state that may occur, together with the associated changes of the four quantities are as follows:

(i) *p*-increase in a user process—decrease *fip* and *fiop*.
This occurs when an array page is first used. Since stack pages are locked down in main store, only the array pages and pages containing program code contribute to *p*; when the compiler generates code it stores it in an array.

(ii) *p*-decrease in a user process—increase *fip* and *fiop*.
This occurs when arrays are discarded at block exit, when code pages are discarded at the end of a program or when pages of input file are discarded after being processed.

(iii) $i \rightarrow p$ transition in a user process—increase *fio* and *fi*.
This occurs when a program unchains a page from an input stream, preparatory to processing it; the new page is treated as part of an array.

(iv) $p \rightarrow o$ transition in a user process—decrease *fio* and increase *fip*.
This occurs when a program chains a page onto an output stream from an array.

(v) *i*-increase in an input process—decrease *fiop*, *fio*, *fip* and *fi*.
This occurs when an input process adds a newly filled page to an input stream.

(vi) *o*-decrease in an output process—increase *fiop* and *fio*.
This occurs when an output process removes one or more pages from an output stream preparatory to emptying them.

3. *The Strategy*
As mentioned above, any process contemplating causing a change of state must respect the bounds of the sugar lump. Two cases arise, according to whether the face encountered is, on the one hand, the *i* or *ip* face or, on the other, the *io* or *iop* face.

(i) The *i* and *ip* faces—When one of these faces is encountered during an *i*-increase or *p*-increase there is little to be done apart from signalling "store exhaustion". The reason is that one cannot rely upon some other user process

to perform an $i \rightarrow p$ transition or a p-decrease: the other processes may themselves need to wait for a new page to be input or for a new program page to be made available. The strategy must therefore be to keep away from these two faces where possible. This is done in two ways. Firstly, each user process will warn the operator if it is getting near to the *ip* face and he may act accordingly. Secondly, each input process normally reads at full speed but will pause if it finds that the input stream it is producing has reached half the size of the space still available for input.

(ii) The *io* and *iop* faces—When one of these faces is encountered during a p-increase, a $p \rightarrow o$ transition or an i-increase, the process may safely suspend itself if it can be sure that there will be an o-decrease to relieve the shortage of pages. To ensure this it is necessary to guarantee that at least one output process is active while $o > reso$ and then we may leave to that process the task of waking suspended processes. In practice, although it is not strictly necessary, this task is also performed by each user process whenever it engages in an $i \rightarrow p$ transition: this can enable a suspended process to continue sooner than it might otherwise have done. A user process performing a p-decrease is not so burdened, however, as this change of state occurs comparatively frequently.

In order to guarantee that output will continue when $o > reso$ it is necessary to draw a distinction between "active output" and "passive output". The former consists of all those pages that the system knows it can output if necessary, while the latter consists of those whose output cannot be guaranteed because there is doubt as to whether the appropriate device will be available. Thus, in order that the system can guarantee output if $o > reso$, it must ensure that the total amount of passive output is limited to *maxpas*, where $reso - maxpas$ is sufficiently large for active output to continue. Now the basic unit of each output stream depends upon the associated peripheral device: for punches and the plotter it is a single page but for the printer it is up to four pages (corresponding to a sheet of printer output), and so *maxpas* is chosen to be $reso - 4$. The system thus imposes an upper bound, *maxpas*, on the number of passive output pages and forces output to take place if $o > maxpas$. That this is always possible will be shown in the section on Input/Output Control in which the classification of output as active or passive is treated in detail.

D. STORAGE OF ARRAYS

Many paged operating systems succumb to thrashing when faced with large matrices or other two-dimensional arrays. If the elements of such an array are stored by row but accessed by column (or vice versa) then the entire

array may have to be kept in main store. In the T.H.E. system this serious problem is avoided by storing the elements by sub-array as in the example in Fig. 2. The example shows a matrix of 128×128 real numbers, each of which occupies two words. By storing a 16×16 sub-array on each page, only eight pages are required in main store at a time if the array is being accessed by row or by column; if the array were stored by row then two rows would be stored on each page and the numbers of pages that would have to be in main store would be one if access were by row or 64 if access were by column. Storage by sub-array is also cheaper when relaxation techniques are applied to the array. There is a further benefit: it is generally cheaper on virtual store if the array is a locally dense sparse matrix: for example a lower or upper triangular matrix would only use 36 pages instead of 64.

In principle this technique could be used with arrays of any number of dimensions but in this system it is only used for two-dimensional arrays.

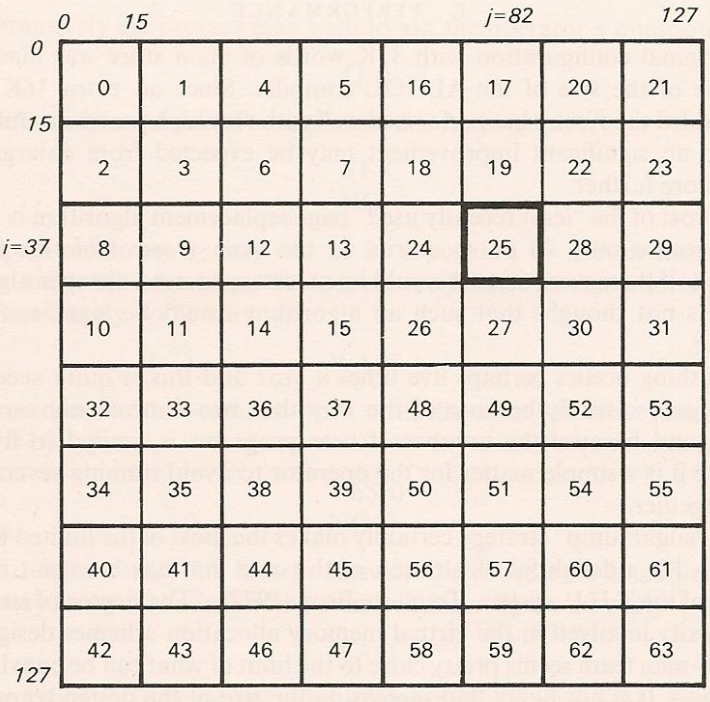

FIGURE 2. Array storage.

Address calculation is a simple matter using Dijkstra's "zip fastener" technique which has been described by Hoare (1972). The numbering of the pages is shown in the diagram and the numbering of the array elements within each page is similar. The address of any array element may be obtained by interleaving the bit patterns of the binary representations of the subscript values of that element. For example the address of the element with subscripts $i = 37$ and $j = 82$ may be obtained as follows:

$i = 37 =$	0		1		0		0		1		0		1	
$j = 82 =$		1		0		1		0		0		1		0
	0	1	1	0	0	1	0	0	1	0	0	1	1	0
	page 25						element 38							

E. PERFORMANCE

The original configuration with 32K words of main store was inadequate because of the size of the ALGOL compiler. Since an extra 16K words were added the system has performed well with very high processor utilization and so no significant improvement may be expected from enlarging the main store further.

The cost of the "least recently used" page replacement algorithm is acceptable because only 70 or so entries in the *main frame table* have to be scanned: if there were more it would be necessary to use a cheaper algorithm but it is not thought that such an algorithm would be significantly less effective.

Thrashing occurs perhaps five times a year and this is quite acceptable. It happens so rarely because of the way that two-dimensional arrays are stored, and because the number of user programs is limited to five, and because it is a simple matter for the operator to avoid running several large jobs together.

The "sugar lump" strategy certainly makes the most of the limited backing store and is a delightful illustration of the care that has been put into the design of the T.H.E. system. To quote Bron (1972): "The degree of structural complexity involved in the virtual memory allocation schemes designed by our five-man team seems pretty close to the limit of what can be convincingly conceived. It is not likely that increasing the size of the design team would allow for the design of significantly more complex structures."

V. Console Control

A. OBJECTIVES

The purpose here is to share the operator's console amongst the processes that wish to converse with the operator, and thus to give each of these processes the impression that it has its own console. For reasons of simplicity the forms that conversations may take are restricted.

B. MAIN FEATURES

The essential features of the console control system have been presented in a simplified form by Dijkstra (1968a). He postulates a number of processes, each of which can converse with the operator. The forms that these conversations may take are as follows. The simplest conversation consists of a single message sent to the operator by process i and requiring no answer, thus:

M(i)

Alternatively the process may wish to ask the operator a question:

Q(i)

to which is expected one of several answers, for example:

YES
or: NO

The operator, however, may not be in a position to answer immediately but he may wish to release the console for further conversations pending his delayed reply, so he may type:

WAIT

The conversation thus remains open until the operator subsequently completes it by typing:

YES(i)
or: NO(i)

where it is necessary for him to identify the process he is addressing as there may be several outstanding conversations. Again the reply

WAIT

is acceptable to indicate that the operator has changed his mind since indicating that he wished to type.

Two further rules are imposed. Firstly, when the console becomes free, the operator, if he wishes to complete an outstanding conversation, should have priority over processes wishing to start new conversations. Secondly, when the operator types something that is rejected as inapplicable, he is given the opportunity to try again.

These conventions dictate the main features of the console control system. The first point is that the operator must have some way of announcing his intention to type and he must be able to make such an announcement even if a message or question is being output to the console; he must also be able to detect when the machine is ready to accept his intervention. The second point is that when the operator resumes an outstanding conversation by typing a "delayed" answer, there may be several processes awaiting such replies, or there may be none; it is therefore necessary to introduce a *console process* whose sole task is to accept the operator's announcement that he wishes to type, to read his message when the console is free, to check the message for applicability and to route it to the appropriate process.

C. NORMAL CONVERSATIONS

Since the console is shared by a number of processes it behoves a process wishing to output a message to reserve the console first and afterwards to release it again, thus:

...; reserve; output message; release;...

where the procedures *reserve* and *release,* operating under the protection of a mutual exclusion semaphore, inspect and update the common state variables. There are two main state variables:

state: (*free, output, immediate reply, delayed reply*);
operator priority: *Boolean.*

The former indicates whether the console is reserved for use and, if so, for what purpose, while the latter is set to true if the operator indicates, while output is in progress, that he wishes to type; when the console is next free, *operator priority* is set to false and *state* is set to *delayed reply.* While trying to *reserve* the console, a process may find that *state* $\neq$ *free* in each case it must await permission to start its conversation, releasing the mutual exclusion semaphore for the duration of the wait.

Similarly a process wishing to ask a question must first reserve the console and, after typing the question, must wait for a response from the operator:

reserve;
output question (*i*);
case *response* **of**

```
        YES: positive reaction;
        NO: negative reaction
    end.
```

The body of the function *response* involves waiting for an answer which may be delayed by the operator typing "WAIT" and replying later; in either case the console must be released after the answer has been supplied. The procedures *reserve* and *release* and the function *response* may thus be programmed as follows:

```
    procedure reserve;
        {if state ≠ free then wait until conversation may start;
        state := output};
    procedure release;
        if operator priority then {operator priority := false;
                                   state := delayed reply}
                             else {state := free;
                                   signal that conversation may start};
    function response (me: process identifier): (YES, NO);
        begin state := immediate reply;
              who := me;
              wait until conversation is over and who = me;
              response := reply;
              release
        end
```

where we have introduced two new state variables, *who* and *reply*, which are inspected by processes awaiting replies and are set up by the *console process* when it accepts an answer typed by the operator.

The *console process* is cyclic. It spends most of its time waiting for the operator to announce that he wishes to type. When this happens it inspects the state variables, using the function *state of console*, to discover whether the console is available and, if so, whether an immediate reply to a question is expected or whether the input is to be treated as a delayed reply to an earlier question. If the console is not being used for output then the operator's answer is read and examined: if it is acceptable then it is transmitted, via the state variables, to the waiting process; the procedure *respond* carries out this latter task. We may program *state of console* and *respond* as follows:

```
    function state of console: (output, immediate reply, delayed reply);
        begin if state = output then operator priority := true;
              if state = free then state := delayed reply;
              state of console := state
        end;
    procedure respond (answer: (YES, NO); whom: (process identifier,
    immediate));
```

```
        begin if whom ≠ immediate then who := whom;
              reply := answer;
              signal that conversation is over
        end.
```

The outline of the *console process* is:

```
    console process: process
    cycle
        begin wait for incoming message;
            case state of console of
            immediate reply: begin input answer A;
                                  case A of
                                  "WAIT": release;
                                  "YES": respond (YES, immediate);
                                  "NO": respond (NO, immediate);
                                  other:
                                  end
                             end;
            delayed reply:   begin j: process identifier;
                                  input answer A;
                                  case A of
                                  "WAIT": release;
                                  "YES(j)": respond (YES, j);
                                  "NO(j)": respond (NO, j);
                                  other:
                                  end
                             end;
            output:
            end
        end console process.
```

A semaphore is used to preserve the integrity of the state variables and so each of the procedures and functions that operate upon them is preceded by a P-operation on that semaphore and followed by a V-operation. The semaphore is also released while a process is waiting for a conversation to start or finish. An array of semaphores represents "conversation may start" and "conversation is over and *who* = *me*" with an array of integer variables denoting which, if either, of these events each process is awaiting.

D. ABNORMAL CONVERSATIONS

Normally conversations are under the control of the processes, not the operator; but occasionally the operator may wish to stop a user process either temporarily or permanently and this is not a message that the process

is expecting. What happens is that the *console process* recognizes a "STOP" message and, by inserting a P-operation in front of the next instruction to be executed by that process, obliges it to wait on a semaphore.

Either a "GO" message or an "OUT" message is now applicable. On receiving the former the *console process* performs a V-operation. On receiving the latter the *console process* changes the value of the instruction counter of the suspended process to point to a location where the finishing rites are initiated; it then performs a V-operation to enable the user process to terminate itself.

This, however, is not the complete picture since a process may have obligations to fulfil and therefore cannot be thrown out immediately. The concept of mortality is therefore introduced: if a user process is mortal it may be stopped, if immortal then the *console process* deals with the "STOP" message by setting to true a variable, *stop request,* of the user process: the P-operation is now delegated to the user process itself, which, when it returns to mortality and finds *stop request,* suspends itself. Since sections of immortality may occur in nested fashion, each user process has an *immortality counter*; when this counter goes to zero the process becomes mortal again.

E. PERFORMANCE

The design of console conversation systems brings with it the dangers of synchronization errors and large overheads. These have been avoided here by restricting the forms that conversations may take. Despite this restriction (or, more probably, because of it) the system is easy for the operator to use and it permits him to recover readily from his mistakes.

VI. Input/Output Control

A. OBJECTIVES

The system presents to each user a number of "streams" for input and output, thereby abstracting away from the actual peripheral devices. The extensive buffering of data in these streams enables files to be produced and consumed smoothly despite considerable differences in speed between these two operations. The buffering of complete or substantially complete files imposes a great load on the limited store and so the system endeavours to maintain a reasonable supply of free page frames for various purposes, partly by slowing the buffering of input when space is short and partly by keeping the output devices busy. Generally, peripheral devices are not

assigned to the output of a file if there is a danger that the output might last an indefinite length of time but, on the occasions when this is necessary, care is taken to reduce the likelihood of deadlock.

B. LOGICAL INPUT/OUTPUT

Each user process has available to it two reader streams, two punch streams, one plotter stream and one printer stream. A user program may send several independent files in succession to each of its output streams; so such a stream may contain several files of which some may have been produced by the current user program with the others having been produced by previous programs run by the same process. The newest file in an output stream may perhaps be still incomplete, while the oldest may be in the process of being printed or punched or plotted. Similarly, a succession of files are read from each input stream but, as the buffering of an input file never starts until the user program is ready to use it, such a stream will never contain more than one file at a time and this will be incomplete, being consumed from one end while new portions are being read and added to the other end.

The user process transfers data between its own work area and its streams a "portion" at a time: generally a portion consists of a page of data but in the case of the printer it corresponds to a sheet of printed output and consists of up to four pages. Since the printer switches its attention from one printer stream to another on a sheet-at-a-time basis an "end of sheet" marker must be stored in the last page of each portion; the other output devices work on a file-at-a-time basis and so the user must insert an "end of file" indication in the last page of each file. During its production and consumption each portion is treated as an array local to the relevant (user or device) process.

Chaining portions onto streams or unchaining them from streams involves no manipulation of the portions themselves but only of their descriptors which are transferred between the process' stack and the area reserved for the descriptors that point to the pages of input/output streams. Each of the words in this latter area belongs to a chain, there being one chain for each stream and another of unused locations. Associated with each stream is a general semaphore and chaining a portion onto a stream (or unchaining a portion from it) involves a V- (or P-) operation on the semaphore whose value therefore represents the length of the stream.

The magnetic tapes are treated a little differently from the other devices as pages are input or output directly by the user processes, thus obviating the need for device processes and streams. The only difference that the user notices is that he has to open and close tape files explicitly.

C. PHYSICAL INPUT/OUTPUT

1. *General*

The interface between a peripheral device and the process that controls it consists of two queues made up of a chain of input/output requests linked together: one queue is of pending requests while the other is of those that have been completed by the device; this arrangement is illustrated in Fig. 3.

Associated with the pending queue P_i is a general semaphore, p_i, representing the length of the queue and, likewise, the semaphore c_i denotes the length of the completed queue, C_i.

The four operations on the chain are as follows:

(i) attach request to queue P_i; V(p_i)

(ii) P(p_i); detach request from queue P_i

(iii) attach request to queue C_i; V(c_i)

(iv) P(c_i); detach request from queue C_i.

In step (iv) it may happen that queue C_i is found to be empty and so the device process is suspended pending a V(c_i) operation; this state is indicated

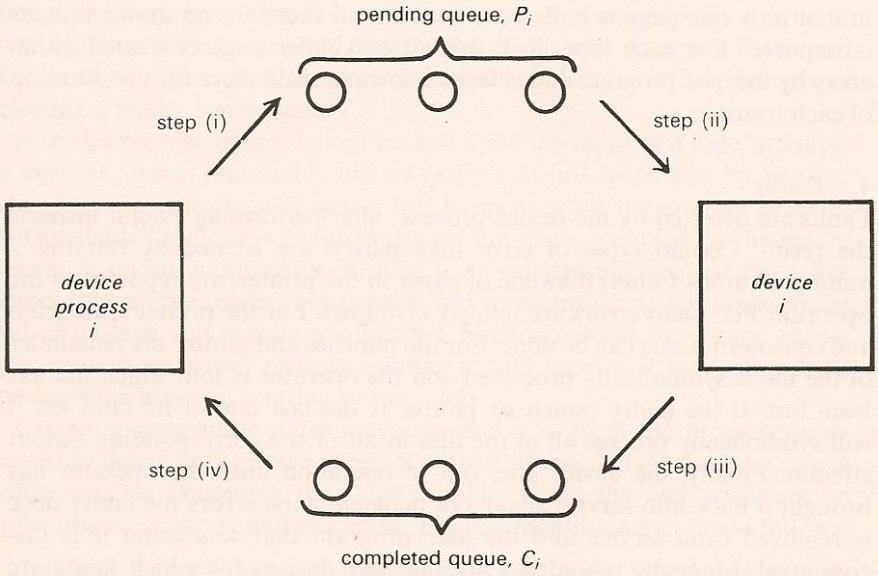

FIGURE 3. Buffering of input/output requests.

by a Boolean variable s_i whose value is "true" in such a case. Steps (ii) and (iii) are performed by the channel hardware which also causes an interrupt if s_i is true at the time of a V(c_i) operation. The channel hardware uses a busy form of waiting if unable to perform a P(p_i) operation in step (ii). In step (i) the addition of a request to an empty pending queue means that the $link_i$ pointer to the request at the head of queue P_i must be reset by the process for subsequent use by the device; this $link_i$ pointer is normally updated by the device on the completion of a request. For convenience there are two Boolean variables associated with each device: they take the values $p_i > 0$ and $c_i > 0$ and are implemented as flip-flops.

2. *Slow Devices*

For each paper tape reader, two buffers of 24 characters are used in the manner illustrated above. A reader process packs three characters to a word and writes these words one by one to its buffer page which, when full, is attached to the appropriate reader stream. For the plotter and punches the scheme is the inverse. For the printer the buffering is more complicated because lines may be double printed—the units of buffering are "completely specified lines".

3. *Magnetic Tapes*

The magnetic tape interface differs from that of the other types of device in that only one page is buffered at a time and therefore no device process is required. For each tape deck the current buffer page is treated as an array by the user program and is locked down in main store for the duration of each transfer.

4. *Faults*

Faults are detected by the device process: after performing P(c_i) it inspects the result. Certain types of error (like parity) are treated by retrying a number of times. Others (like end of paper in the printer) are reported to the operator. Persistent errors are treated as follows. For the printer (and drum and console) nothing can be done. For the punches and plotter the remainder of the file is symbolically processed and the operator is told which file has been lost. If the faulty punch or plotter is the last one of its kind left it will symbolically process all of the files in all of the corresponding output streams. Finally, the device goes out of operation until the operator has brought it back into service again. On magnetic tape errors the faulty deck is removed from service and the user program that was using it is discontinued. Magnetic tape decks are the only devices for which faults are reported back to the user program as the programmer may wish to re-use

part of the tape. In particular, whenever a user finds an untimely end, the state in which each open tape file is left is reported back to him.

C. DEVICE SCHEDULING

1. *Magnetic Tapes*

Magnetic tape decks are allocated statically and released dynamically. That is to say, a program will not be accepted for execution unless all the decks it requires are available; and the program may reduce its claim when it no longer needs a deck and any remaining claim will automatically be reduced to zero when the program terminates.

2. *Readers*

The normal state for an idle reader is "ready and willing to read". On being supplied with a tape (assumed to be a job tape) a reader process searches for some user process with no program to run and supplies it with the tape. If there is no free user process then the reader process waits until one becomes free. Thus all the operator has to do to run a job is to place the job tape in a free reader.

The other common requirement is that a running user program may want a data tape. It waits until some reader is free, reserves it and asks the operator to put the required tape in the specified reader. Generally the operator will do as requested and will say so to the user process. Occasionally, though, he may reply that the required tape is not immediately available, in which case the user process releases the reader and suspends itself until the operator signals that the tape is available, whereupon the business of selecting a reader starts again.

A reader process, once it has embarked upon the input of a tape, attempts to keep its stream reasonably full by reading at full speed and by pausing when there is a shortage of space: this happens when the amount of space still available for input is less than double the space occupied by the reader stream in question. However, if there is an outstanding request for a free reader then all the readers continue at full speed until one has come to the end of its tape and can be re-allocated. Such a request for a tape reader may come from a user process wanting a data tape or it may come from the operator urgently wanting to run a job.

Certain incompatible states may arise and it is up to the processes concerned to detect and resolve these incompatibilities. This is done by, say, a reader process examining the state variables of all the user processes on appropriate occasions, and vice versa. The incompatible states are as follows:

(i) a reader may have been supplied with a job tape, and a user process may be idle waiting for another job;

(ii) a user process may be waiting for a reader, and a reader may now be free.

(iii) there may be an outstanding request from the operator to disconnect a reader, and the reader may now be idle or may have been given a job tape but cannot find a user process ready for it.

3. *Printer*

The printer process examines all the printer streams in its attempt to select a completely generated file, favouring the stream with the maximum value of

(*stream length, in portions* − *time, in portions, that last portion was attached*).

The selected file is then output as fast as possible.

If there is no complete file the process waits for one unless there is a shortage of space in which case an unfinished file is selected using the same criterion as before. For the purposes of this strategy, space is deemed to be in short supply if the total number of pages containing printer output exceeds the number of pages, *fpas*, still available for passive output. (Recall that in the section on Store Management a distinction was drawn between active and passive output and that an upper limit, *maxpas*, was imposed on the latter. The difference, *fpas*, (initially = *maxpas*) is introduced to denote the number of pages "free for passive output".)

If, during the printing of an incomplete file, the user process finishes producing the file then the complete file is output without interruption. Otherwise, the printer process attempts to output 20 consecutive sheets from the file, pausing when necessary for production to keep up with consumption. At the end of the 20 sheets—or earlier, if a shortage of space for passive output is detected during one of the pauses—the printer process selects another file or possibly even the same file, using the same criteria as before.

As a consequence of this strategy, if the printer process suspends itself it must be reawoken as soon as there is a shortage of space for passive output, as soon as a printer file has been completed, or as soon as the portion for which the printer is waiting has been produced. The onus is on the process that causes one of these changes of state to detect that the printer is waiting and to restart it.

4. *Plotter and Punches*

(*a*) *Scheduling Policy.* A plotter or punch process generally attempts to select a finished file for output and, if there are several streams that have not been selected for output and yet contain finished files, the process chooses the one with the largest value of

(*stream length, in portions* − *time, in portions, that last portion was attached*).

Should there be none, the process suspends itself unless some other user process is being held up by a shortage of space for passive output: in such a case the device process attempts to relieve the shortage by selecting an unfinished file, provided that no danger of deadlock arises (this is checked by the "banker's algorithm" which is described later). The stream selected is the one with the largest unfinished file or, failing that, a stream that would contain such a file were its user process not suspended due to the shortage of space for passive output. Having selected a finished or unfinished file the process plots or punches it as quickly as possible.

If a device process suspends its activities because of a lack of suitable candidates for output it must eventually be reawoken when a user process finishes producing a file which is then eligible for selection or when there is a shortage of space for passive output or when some unfinished file that is currently being output is finished and the banker's approval for the selection of another unfinished file is consequently forthcoming.

(*b*) *Passive Output.* The distinction between active and passive output has already been drawn: the former consists of all those pages that the system knows it can output if necessary, while the latter consists of those whose output cannot be guaranteed because there is doubt as to whether the appropriate device will be available. More specifically, only pages belonging to plotter or punch files contribute to the amount of passive output as the printer can, when necessary, switch from one file to another and back again and therefore cannot be held up indefinitely. All pages belonging to punch and plotter files are active if there is at least one appropriate device that can, within a finite time, be ready to output them: included in this category are all files that have already been selected for output and also all unselected but finished files for which there is at least one device of the right type that has not tied itself up for an indefinite period by embarking upon the output of an unfinished file. It follows that if all devices of a particular type (punch or plotter) are outputting unfinished files, all other files of that type are passive. Furthermore all unfinished unselected files also contribute to the amount of passive output since their selection for output is subject to approval by the banker's algorithm and such approval cannot be guaranteed in advance.

As already described, an upper limit, *maxpas,* has been imposed on the amount of passive output, and the quantity *fpas* has been introduced to denote the number of pages "free for passive output". The following changes of state alter the value of *fpas* and, as printer files do not contribute to passive output, refer only to punch and plotter files. Note that no active page ever becomes passive; this is ensured by selecting unfinished files for output only when there are no finished files.

(i) File selection by an output process: the selection of a finished file in preference to an unfinished one avoids the possibility that an unselected finished file would have to change from active to passive if all the appropriate devices started to output unfinished files; in the absence of a finished file the selection of an unfinished file gives rise to:

increase *fpas* by the size of the selected file.

(ii) File completion by a user process: when a selected file is finished all finished files for devices of the type in question can change from passive to active:

increase *fpas* by the sizes of those finished files.

If, on the other hand, the newly finished file has not yet been selected for output then its pages become active:

increase *fpas* by the size of the newly finished file

or stays passive, depending on whether or not there is a device of the right type that has not been assigned to the output of an unfinished file.

(iii) $p \rightarrow o$ transition in a user process when a page is chained onto a passive (i.e., unselected) file:

decrease *fpas*.

During a $p \rightarrow o$ transition it may prove impossible to decrease *fpas* because the limit for passive output may have been reached. Eventually, however, either an unselected file will be finished and its passive pages will become active, or else all active output will be disposed of and an unfinished file will be selected and its pages will cease to be passive.

(*c*) *Banker's Algorithm.* There is a danger with dynamically allocated non-pre-emptible resources (in this case the plotters, the punches and the readers), that a deadlock may occur. If, for example, each of two processes needs both of two resources, a deadlock or "deadly embrace" results if each process has one resource (which it has no intention of releasing) and requires the other (which is not available). The banker's algorithm is designed to avoid such a situation—it is due to Dijkstra (1968a) and has been extended by Habermann (1969).

The basic problem may be expressed as follows. A banker has a finite *capital* expressed in florins. He accepts any number of borrowers, of whom each has a *loan* which may be decreased or increased in units of a florin up to a maximum pre-stated *need,* which must not of course exceed the banker's *capital.* Each borrower guarantees that he will eventually return his complete *loan.* When asked for a florin the banker must decide whether the resulting situation would be safe from the possibility of deadlock; if not the borrower must wait to have his request granted.

To decide whether a situation would be safe the banker must check whether

all transactions are able to finish. The algorithm starts by inspecting whether or not at least one borrower has a *claim* (*claim* = *need* − *loan*) not exceeding the banker's *cash* (*cash* = *capital* − Σ*loan*). If so this borrower can complete his transactions and so the algorithm investigates the remaining borrowers as if the first one had finished and returned his complete *loan*. The situation is safe if all transactions can be completed.

In practice, as L. Zwanenburg has pointed out, the only situations to be investigated are those where, starting from a safe state, borrower k has asked for a florin. If this borrower is able to complete his transactions the situation is known to be safe. An algorithm for this case is given below; there are assumed to be a maximum of n borrowers.

```
function safe (k: 0...n−1; cash: integer): Boolean;
    begin able to finish: array 0...n−1 of Boolean; i: 0...n−1;
        for i := 0...n−1 do able to finish [i] := false;
        repeat i := k;
            repeat if able to finish [i] or claim [i] > cash
                    then i := (i+1) mod n
                    else {able to finish [i] := true;
                        cash := cash+loan [i]}
            until i = k or able to finish [i];
        until i = k;
        safe := able to finish [k]
    end
```

This algorithm is readily extended to deal with several "currencies" or resource types. This is necessary in the T.H.E. system where the plotter and the punches need the approval of the banker before embarking on the output of unfinished files.

Readers are not subject to the banker and so the danger of deadlock exists. Bron (1972) has explained that it was decided not to burden the user programmer with the need to drop his reader claim explicitly and that the alternative of maintaining the claim over the lifetime of the program would be unduly restrictive.

It should be noted that the banker's algorithm takes no scheduling policy decisions—it merely checks whether a particular request is safe; there may be several safe requests.

E. PERFORMANCE

At first glance the use of the multi-currency banker's algorithm appears to be expensive, but in this system it is designed to be very rarely invoked and, when it is, few resources and processes are involved. Although readers

lie outside the banker's algorithm they have never been the cause of a deadly embrace.

Important though the avoidance of deadlocks may be, the avoidance of near-deadlocks is at least as valuable. This is the justification for the carefully designed algorithms that schedule the use of the peripheral devices and the virtual store. They are designed to ensure that the system will continue to function well when demand for devices or storage is high or is unusually biased towards a certain type of device or category of storage.

The unconventional but effective decision to treat the printer as a pre-emptible device, by scheduling it a sheet-at-a-time rather than a file-at-a-time, places only a negligible burden on the operators since very few files are ever split.

An advantage of the scheme whereby input and output streams are buffered in the virtual store is that frequently the pages being input or output never need to be dumped onto backing store.

At a lower level, the semaphore interface between the peripheral devices and the computer leads to a worthwhile reduction of overheads when compared with more conventional systems.

VII. System Maintenance and Monitoring

A. RESTARTS

The operating system, including library routines, is initially and rapidly loaded into the computer from magnetic tape. There is no other form of restart since there is no file store to maintain and jobs are not stored in the computer prior to being scheduled. Should a system failure occur, the jobs currently running are lost and will have to be re-run; also the outputs of previous jobs may not have appeared and are thus lost. The accounting information in the system log is also lost.

System failures are rare and are generally due to hardware faults. However if the operator deliberately chooses to let one or more programs acquire too much store the system may fail owing to a shortage of store. Furthermore, as has been explained, a deadlock may be caused by requests for tape readers but this has never happened in practice.

B. LOGGING

For accounting purposes the system maintains a log of jobs run, including a record of the resources used by each. These run statistics are accumulated

in seven array pages that are permanently present and considered as library pages; access to them is made exclusive by means of a semaphore.

The system also maintains a record of the amount of time spent on useful computation and the amount of time spent idling. Apart from this no other statistics are gathered. However it appears from observation that the configuration is well balanced with satisfactorily high processor utilization—between 87% and 92%, according to Bron. Until the main store was increased from 32K to 48K the shortage of core was the limiting factor.

C. OPERATOR CONTROL

All job scheduling is performed by the operator who attempts to run one long job and up to three short ones at a time. He may (and should) terminate any job that runs for too long since it is not automatically aborted.

Each job starts life with an allowance of a certain number of pages. Should it require more it asks the operator who may grant it as many pages as he thinks fit or may abort it. The operator is thus given plenty of opportunity to prevent the system from failing because of a shortage of store. As Bron (1972) has pointed out, such a shortage usually arises from outrageous array growth or uncontrolled recursion, but it only occurs as a result of concurrently running several genuinely large jobs perhaps five times a year.

D. MAINTENANCE

The documentation of the operating system is not complete but the designers wrote down very clear accounts of the most important features of the system (in particular Dijkstra, 1964, 1965a, 1965b). In those reports they explained and justified their design decisions, defined their principal data structures and developed their scheduling algorithms in ALGOL 60. Only a fraction of the software was developed in ALGOL though and because everything was implemented in assembly language the software is difficult to modify.

This has not, however, proved to be much of a drawback, partly because the structure of the operating system lent itself fairly readily to testing, and partly because sufficient care went into the design and implementation to ensure that very few mistakes occurred. A certain amount of difficulty was encountered in writing the ALGOL compiler since, unlike most paging systems, the paging was visible to the compiler writers who had to incorporate code to deal specifically with inter-page accesses.

The only errors known to escape the testing stage were either in the ALGOL compiler (and some of these were concerned with inter-page access) or in the library of input/output procedures.

VIII. Glossary of Terms Local to T.H.E.

The T.H.E. terms, where they differ from those used in this chapter, are given in brackets.

BANKER'S ALGORITHM TERMS:

capital—maximum number of resources available to be borrowed.
cash—number of resources currently available to be borrowed.
need$_i$—maximum number of resources that may be borrowed by process *i*.
loan$_i$—number of resources currently borrowed by process *i*.
claim$_i$—number of resources that may still be borrowed by process *i*.

console process—[message interpreter] a process that interprets messages typed by the operator and routes each to its destination process.

dispatcher—a routine that switches the processor from one process to another.

dispatch queue—[priority list] a priority-ordered queue of all the processes in the system.

drum frame table—[drum page table] a table recording the state of every drum store page frame.

drum process—[segment controller] a process whose task is to transfer pages between main store and drum store.

free frame pointer—a pointer to the next free entry in the *main frame table*.

immortality counter—a counter local to each user process and used to note whether the process is "mortal" (and therefore immediately able to terminate the user program it is executing) or "immortal" (and therefore has obligations to fulfil before it can terminate its program).

INPUT/OUTPUT TERMS:

P_i—a first-in-first-out queue of requests pending for device *i*.
C_i—a first-in-first-out queue of requests completed by device *i*.
p_i—[AFT_i] a general semaphore denoting the length of queue P_i.
c_i—[IFT_i] a general semaphore denoting the length of queue C_i.
link$_i$—a pointer to the head of queue P_i.
s_i—[LV_i] a flip-flop used to denote whether a process is suspended awaiting the completion of a request by device *i*; on completing a request the device causes an interrupt if s_i is set true.

main frame table—[core table] a table recording the state of every main store page frame.

message buffer—a buffer used to pass messages to the *drum process*; an *insertion pointer* to the buffer is used when adding messages and an *extraction pointer* is used when messages are removed by the *drum process*.

mutex—a mutual exclusion semaphore used to control access to critical sections.

nucleus—a routine that fields interrupts, implements semaphore operations and includes (and invokes) the *dispatcher*; as such, the *nucleus* underlies all the processes in the system.

stop request—a Boolean variable local to each user process and used to inform that process that it should terminate the user program it is executing.

SUGAR LUMP TERMS:

total—[tot] maximum number of pages available for holding temporary material (viz., input files, output files and user programs).

maxio—[transp] maximum number of pages available for holding input and output.

maxpas—maximum number of pages available for holding passive output.

reso—minimum number of pages available for holding output.

i—number of pages currently holding input.

o—number of pages currently holding output.

p—number of pages currently holding programs.

fiop—number of pages currently available for holding input, output and programs.

fio—number of pages currently available for holding input and output.

fip—number of pages currently available for holding input and programs.

fi—number of pages currently available for holding input.

fpas—number of pages currently available for holding passive output.

In addition, the following general terms, with their T.H.E. equivalents given in brackets, are used in this chapter: file [document], page [segment], page frame [page], process [AM (abstract machine)], supervisor process [CM (constant machine)], user process [PM (programmable machine)].

IX. References

Bron, C. (1971). Tapereaders in the multiprogramming system. *Univ. Eindhoven, Dept. Maths. Report CB44.*

Bron, C. (1972). Allocation of virtual store in the T.H.E. multiprogramming system. *In* "Operating Systems Techniques" (C. A. R. Hoare and R. H. Perrott, eds), 168–184. Academic Press, London.

Dijkstra, E. W. (1964). Segment control. *Univ. Eindhoven, Dept. Maths. Report EWD 113.*

Dijkstra, E. W. (1965a). The multiprogramming system for the EL X8 THE. *Univ. Eindhoven, Dept. Maths. Report EWD 126.*