University of Edinburgh

Department of Computer Science

The Edinburgh Multi-Access System Scheduling and Allocation Procedures in the Resident Supervisor

by

N. A. Shelness, P. D. Stephens
and
H. Whitfield

EMAS Report 4 Reprinted 1977

James Clerk Maxwell Building
The King's Buildings
Edinburgh EH9 3JZ
031-667 1081

This paper was presented at The International
Symposium on Operating Systems, Theory and Practice,
April 23rd-25th, I.R.I.A., Paris.

THE EDINBURGH MULTI-ACCESS SYSTEM

SCHEDULING AND ALLOCATION PROCEDURES

IN THE RESIDENT SUPERVISOR

N. H. Shelness

Department of Computer Science

Edinburgh University

P. D. Stephens

Edinburgh Regional Computing Centre

H. Whitfield

Mathematisch Instituut

Rijskuniversiteit te Groninigen

INTRODUCTION

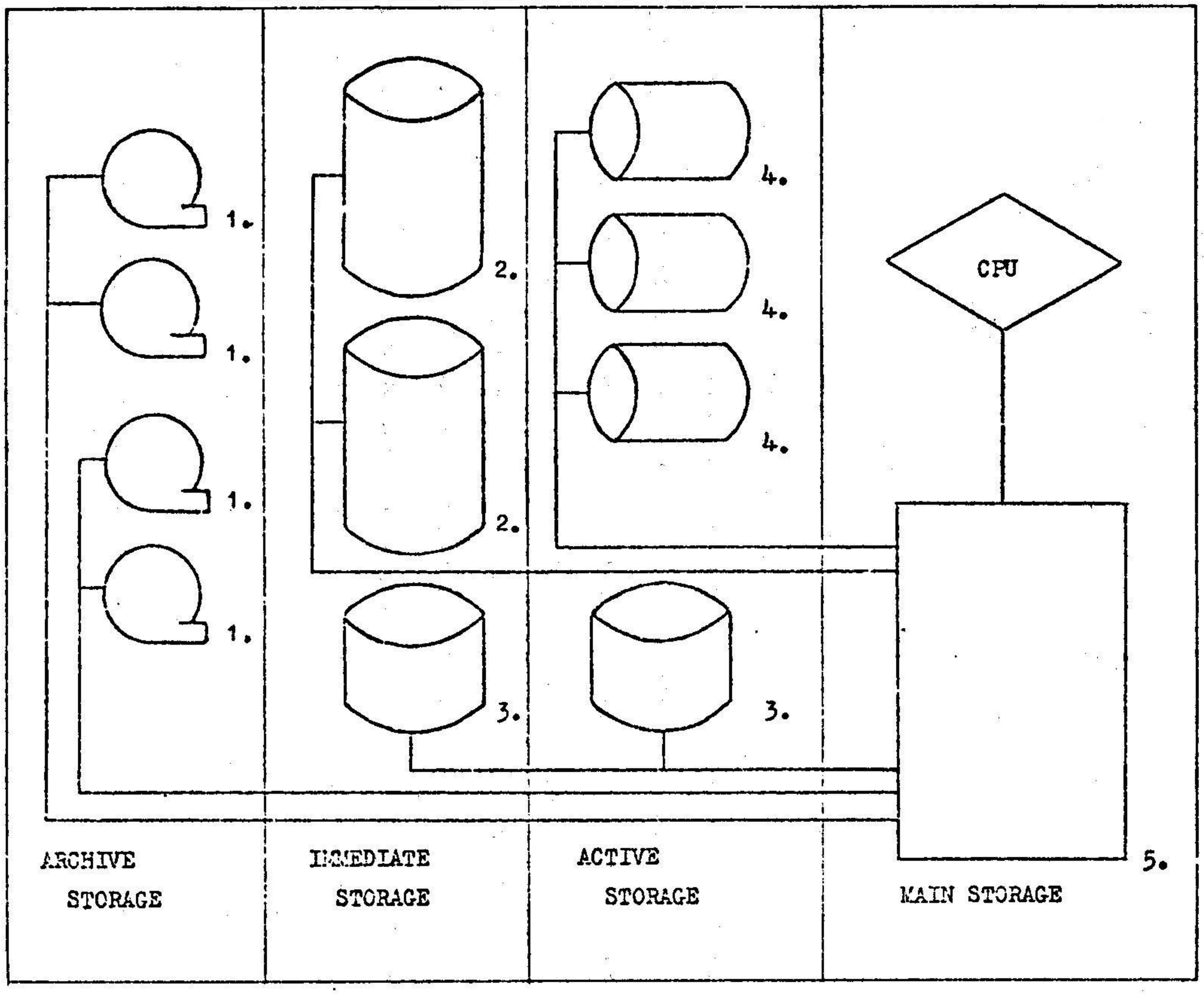
The Edinburgh Multi-Access System is a large, multi-user, interactive, paged, virtual memory operating system developed in Edinburgh over the last nine years. An initial design was formulated between 1964 and 1966 [w1]. Following this an attempt at implementation was undertaken by a large University/manufacturer team: The Edinburgh Multi-Access Project. This effort commenced in 1967 and came to an end in the summer of 1970, having failed to implement a satisfactory system. The reasons for this will not be discussed here.

The system that existed at the end of the initial project was not robust enough for even such simple system development tasks as editing and compilation. There was, though, a nucleus of code on which a small team, varying in size from 3 to 7, was able, over the next year, to build a system robust enough for general release. The results of this effort, along with the performance characteristics of the system in the summer of 1972 are presented in a paper written at that time [w2]. Since then a new team including some who were associated with this earlier effort, have rewritten a large part of the resident supervisor, with a view to generality, and portability. A number of scheduling and allocation procedures have also been changed as a result of performance monitoring and new design insights. These have resulted in a considerable performance improvement over that reported in [w2]. It is the purpose of this paper to discuss the current scheduling and allocation policies, and the motivation for them.

PHYSICAL SYSTEM STRUCTURE

The hardware (ICL 4/75) corresponds to that of many large European third generation machines, being a copy of an (RCA SPECTRA 70) which is itself a copy of an (IBM S/360), to which an address translation facility (paging) was added as an afterthought. It possesses four types of memory the first three of which (tape, diskfile and drum) are connected to the last (core) by conventional selector channels. On the initial 1 megabyte configuration in Edinburgh (figure 1), there are 4 nine

FIRST EMAS CONFIGURATION IN EDINBURGH



1. 1600 BPI. Tape
2. 350 Megabyte Disk
3. 6 Megabyte Disk
4. 2 Megabyte Drum
5. 1 Megabyte Core

Figure 1

track tape units on two channels, 2 three hundred and fifty megabyte diskfiles on one channel, 3 six megabyte disk packs on one channel, and 3 two megabyte drums, also on a single channel. The diskfiles have 32 tracks per cylinder, with 2 1/2 pages per track, and 1023 cylinders each. The mean seek time is 100 milliseconds and the maximum transfer rate is 50 pages a second. The drums have 4 pages per track, a rotational time of 20 milliseconds, and a maximum transfer rate accordingly of 200 pages per second. None of the rotating memory devices posses a rotational sensing mechanism. They are not synchronized, nor may more than one device on a channel be transferring at a time. The strategies used by the device handlers to surmount these difficulties [f1] will be discussed later; however, it is true to say that the system possesses less utilizable transfer capacity than one has been led to believe necessary in systems of this type [11].

LOGICAL SYSTEM STRUCTURE

The system implements a four level storage hierarchy consisting of archive storage (tape), immediate storage (diskfile), active storage (drum) and main storage (core). The unit of allocation in archive store is the file (1-4096 pages), in immediate storage the segment (a set of 1-16 contiguous pages), and in active and main store the page (4096 8 bit bytes). The sole unit of information transfer between levels is the page.

The three fastest levels of the storage hierarchy are managed as a whole by a *resident supervisor* on the behalf of up to 63 dynamically created *virtual processors*. Each virtual processor has a linear *virtual address space* of 16 megabytes. Segments of each virtual address space are associated with segments of immediate memory in one of a number of *access modes*: *private read*, *private write*, *shared read* and *shared write*. This is accomplished by placing an entry in the appropriate slot of the virtual processor's *master page*. Hence, as in MULTICS [c1], there is no file I/O in the conventional sense, all access to files being performed through the virtual memory mechanism.

Each virtual processor contains two virtual processes: a *director* process and a *user* process. The director process runs a paged supervisor [r1], the code and global tables of which are *shared* between all director processes. The segments of virtual memory in which this shared material resides, as well as segments private to each paged supervisor are *shaded*, and hence inaccessible to the user process. The director process maintains the master page of the virtual processor, both for itself and the user process. Director sub-processes perform console interaction and other external functions through communicating with the system processes that perform those tasks. In addition a *critical section* of the paged supervisor maintains a *file system*, for the entire system, on immediate storage. The director process of each virtual processor creates an environment in which the associated user process is aware only of *named* sequences of bytes called *files* which are *connected* into its unshaded virtual memory at specified segments, and of virtual addresses.

The user processes run one of a number of sub-systems [m1]. There are two types of user processes: executive processes and normal processes. The sub-systems of executive processes perform specific system functions that are not as time dependent as those performed in the resident supervisor. These functions include: the handling of unit record device I/O, the demons executive; the transfer of files to and from archive storage, the volumes executive; the testing of online peripherals, the engineers executive; and systems maintenance functions, the manager executive. The first two executives run in background mode without an interactive console, while the last two run in foreground mode, and are initiated from an interactive console. Director and executive processes have the same level of software privilege as processes in the resident supervisor, that is the ability to communicate with any process, either resident or virtual. Normal user processes, having a lower level of software privilege, may communicate only with their own director processes. The functions provided by a normal process' sub-system [b3] are those often thought to be part of a system: loading, command interpretation, compiling, editing etc. In EMAS the programmes that perform all of these functions have an identical status with programmes provided by the user, and in fact, a user may easily add to the standard sub-system or provide his own, should he so desire.

The structure of processes in both the virtual processors and the resident supervisor is provided implicitly by the structure of the IMP high level programming language [s2] [b4], in which the entire system is written. In order to allow fuctions to be

serviced by processes in either the resident supervisor or a virtual processor, all inter-process communication and synchronization is performaed through a single message switching mechanism, provided by the most basic software level - the kernel. The message switching mechanism is described in detail in [w2].

There are three levels of hierarchy in the resident supervisor, in addition to those already described in the virtual processors. A chart of the system hierarchy is presented in figure 2.

MAIN MEMORY MANAGEMENT

In all of the large and fully implemented paged systems known to the authors [b1] [c1] [g1] [13], there are at least three distinct supervisory processes which control the processing of user tasks, or as we chose to call them virtual computations. These are a process scheduler, a global paging manager and a CPU scheduler. We will not concern ourselves in this paper with access control functions that are performed by a segment manager or its equivalent.

The process scheduler selects a virtual processor from among those desiring to perform a virtual computation, and inserts it into the multi programming set MPS. The process scheduler is initiated by the page manager when space is available in the MPS.

It will usually take into account, in making its choice, the cpu and main storage requirements of previous computations performed in each virtual processor. In so doing it determines the system's response to various classes of computation. It will assign to the virtual processor it selects an amount of CPU time that the virtual computation may use before being removed from the MPS and _rescheduled_ - a _cpu allocation_. The process scheduler will not assign any limit to the number of pages the processor may aquire in main store - a _main store allocation_. The decision as to which pages will be resident in main store at any instant, the _resident page set_ RPS, will be made by the global paging manager over the entire MPS. In EMAS this is not the case

The EMAS process scheduler assigns to the selected virtual processor both a cpu and a main storage allocation. Having done this, there is no longer any need for a global paging manager. It can be replaced by a number of _local paging managers_ provided on a one to one basis for each virtual processor in the MPS. By replacing the global paging manager, which requires to operate over the _domain_ [s1] of the entire paging system, by local paging managers, each operating over the domain of a single virtual processor, we immediately reduce _complexity_ and increase _systems reliability_. A _critical failure_ that occurs, be it hardware or software induced, while in a local paging manager, need only affect a single virtual processor, not the entire system. A second benefit of the EMAS approach is that we eliminate two of the major problems, and greatest sources of programming complexity, encountered by a global paging manager: preventing

thrashing, and preventing throughput degradation across the entire system as a result of having a virtual computaion in the MPS which displays unstable paging behaviour. In order to see why this is the case, it is neccessary to examine how a global paging manager manages the RPS.

Control of the RPS. is maintained through a mechanism of page replacement. When a page fault occurs, a choice is made, by the paging manager, of a page currently in the RPS to be replaced by the newly required page. In practice there is usually a buffer pool of pages not included in the RPS, so that the newly required page may be fetched immediately, rather than having to wait for the replaced page to be written back to secondary storage if neccessary.

There are basicaly two algorithms used by a global paging manager for making its replacement choice: least recently used LRU and working set WS [d1]. In the first algorithm the least recently used page in the RPS is replaced. In the second algorithm a free page is replaced. A page is free if it is not in the union of the working sets of any virtual processor in the MPS.

The elimination of thrashing in an LRU driven page replacement scheme is difficult, but not impossible. The majority of systems being considered by the authors use an LRU algorithm and don't thrash. This is achieved, at the expense of greater complexity in the CPU scheduler, through altering the size of the active MPS by varying the CPU priority of virtual processors in the MPS.

```
record format CAT LAY (byte integer CATEGORY, PRIORITY, STORE, &
                       ASPERD, ASMAX, ASMIN, &
                       NCY1, NCY2, NCY3, NCY4, &
                       integer CPU TIME, STROBE TIME)
!
! NCY1 is next category if process runs out of core.
! NCY2 is next category if process exceeds time limit.
! NCY3 as ncy2 but core used less than next smallest core limit.
! NCY4 is category if process goes to sleep.
! ASMIN is the unconditional allocation of active store.
! ASMAX is the largest amount of active store that can be held.
! ASPERD is the number of MPS residency periods before recomputing WS.
!
!           hexadecimal constants are bracketed by X' and '.
!
! CPU TIME is cpu allocation in 8 microsecond units. X'00020000'=1 sec.
! STROBE TIME is the period over which the main storage WS is computed.
!

const record array CAT TAB(1:20) (CAT LAY) = &
 1, 1, 42, 20, 80, 64, 17, 15, 11, 14, X'00020000', X'00004000',
 2, 1, 16, 20, 80, 64,  3,  2,  0,  2, X'00010000', X'00010000',
 3, 1, 24, 20, 80, 64,  4,  3,  2,  3, X'00020000', X'00020000',
 4, 1, 42, 20, 80, 64,  4,  4,  3,  4, X'00040000', X'00010000',
 5, 1, 16, 40, 80, 64,  8,  6,  0,  5, X'00010000', X'00010000',
 6, 4, 16, 20, 80, 64, 10,  7,  0,  5, X'00080000', X'00020000',
 7, 5, 16, 20, 80, 64, 10,  7,  0,  5, X'00140000', X'00020000',
 8, 1, 24, 40, 80, 64, 11,  9,  5,  8, X'00020000', X'00010000',
 9, 4, 24, 10, 80, 64, 13, 10,  6,  8, X'00140000', X'00020000',
10, 4, 24, 10, 80, 64, 13, 10,  7,  8, X'000C0000', X'00020000',
11, 2, 32, 40, 80, 64, 14, 12,  8, 11, X'00020000', X'00020000',
12, 4, 32, 10, 80, 64, 16, 13,  9, 11, X'00140000', X'00020000',
13, 5, 32, 10, 80, 64, 16, 13, 10, 11, X'00180000', X'00020000',
14, 2, 42, 20, 80, 64, 17, 15, 11, 14, X'00020000', X'00020000',
15, 4, 42, 10, 80, 64, 19, 16, 12, 14, X'00140000', X'00020000',
16, 5, 42, 10, 80, 64, 19, 16, 13, 14, X'00140000', X'00020000',
17, 3, 52, 20,128, 80, 20, 18, 14, 17, X'00040000', X'00010000',
18, 4, 52,  5,128, 80, 20, 19, 15, 17, X'000E0000', X'00010000',
19, 5, 52,  5,128, 80, 20, 19, 16, 17, X'000A0000', X'00020000',
20, 3, 52, 20,128, 80, 20, 18, 15, 17, X'00040000', X'00008000'

const byte integer array CHOICE(0:63) = &
          5,1,2,1,2,1,2,1,1,2,1,1,2,1,3,1,
          2,1,1,4,1,1,2,1,2,1,1,2,1,1,3,1,
          1,2,1,1,2,1,2,1,4,1,1,2,1,1,3,1,
          1,2,1,1,2,1,2,1,1,3,1,2,1,2,1,1
```

1 - 39
2 - 18
3 - 4
4 - 2
5 - 1

figure 3

processor will remain in the MPS until one of two events occur: it attempts to exceed one of its local constraints, or it goes to sleep. At this point it is removed from the MPS, and its next category determined. There are four possible category transitions. The four cases that determine which of the four transitions is to be made are:

1). The virtual processor's working set attempts to grow larger than its main store allocation.

2). The virtual computation overruns its CPU allocation with a working set that would fit in a smaller category.

3). The virtual computation overruns its CPU allocation with a working set that fits into the current category.

4). The virtual processor goes to sleep.

If the virtual processor is still awake, it is immediately placed on the priority queue associated with its new priority. In this way a virtual processor follows a path through the category table towards an entry that matches its current behaviour. If that behaviour is for the most part stable, then we can expect many of the transitions to be back into the same category, and this is in fact the case (figure 4).

The means by which a virtual processor is selected from a priority queue, the choice algorithm, is exceedingly simple. A circular table is cycled through one by one. Each entry contains

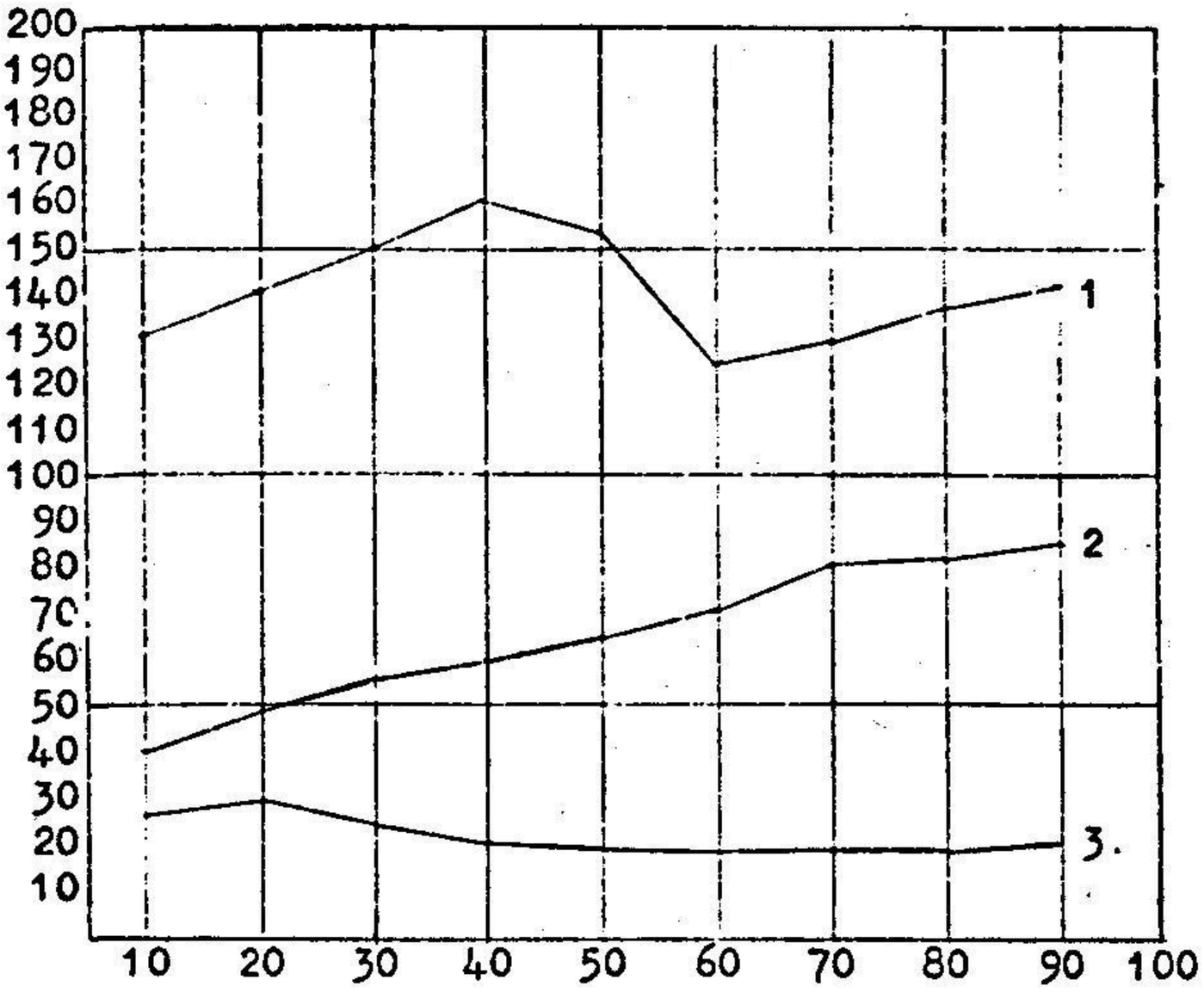
EMAS VERSN 781A DATE: 05/03/74 TIME: 14.31.01

CATEGORY TABLE MOVEMENT

FROM \ TO	1	2	3	4	5	6	7	8	9	10
1	396	0	0	0	0	0	0	5	0	0
2	0	2323	342	0	0	0	0	0	0	0
3	0	894	1911	479	0	0	0	0	0	0
4	0	0	472	600	0	0	0	0	0	0
5	0	0	0	0	14694	115	0	2666	0	0
6	0	0	0	0	8	0	5	0	0	151
7	0	0	0	0	0	0	171	0	0	80
8	0	0	0	0	2683	50	0	9164	58	0
9	0	0	0	0	0	0	2	9	0	4
10	0	0	0	0	122	0	75	39	0	167
11	0	0	0	0	0	0	0	3874	57	0
12	0	0	0	0	0	0	0	24	0	15
13	0	0	0	0	0	0	0	37	0	35
14	0	0	0	0	0	0	0	13	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	5	0	0
18	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	5	0	0

FROM \ TO	11	12	13	14	15	16	17	18	19	20
1	2	0	0	0	0	0	588	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	5842	0	0	0	0	0	0	0	0	0
9	1	0	59	0	0	0	0	0	0	0
10	2	0	49	0	0	0	0	0	0	0
11	4124	111	0	4678	0	0	0	0	0	0
12	54	20	14	0	0	101	0	0	0	0
13	25	0	90	0	0	139	0	0	0	0
14	4779	77	0	3105	120	0	1850	0	0	0
15	68	0	57	21	544	26	0	0	500	0
16	122	0	57	22	0	955	0	0	320	0
17	11	0	0	1525	145	0	951	146	0	766
18	2	0	0	5	0	18	1	297	55	250
19	15	0	0	145	0	237	28	0	2417	250
20	16	0	0	725	207	0	130	185	0	2115

Figure 4

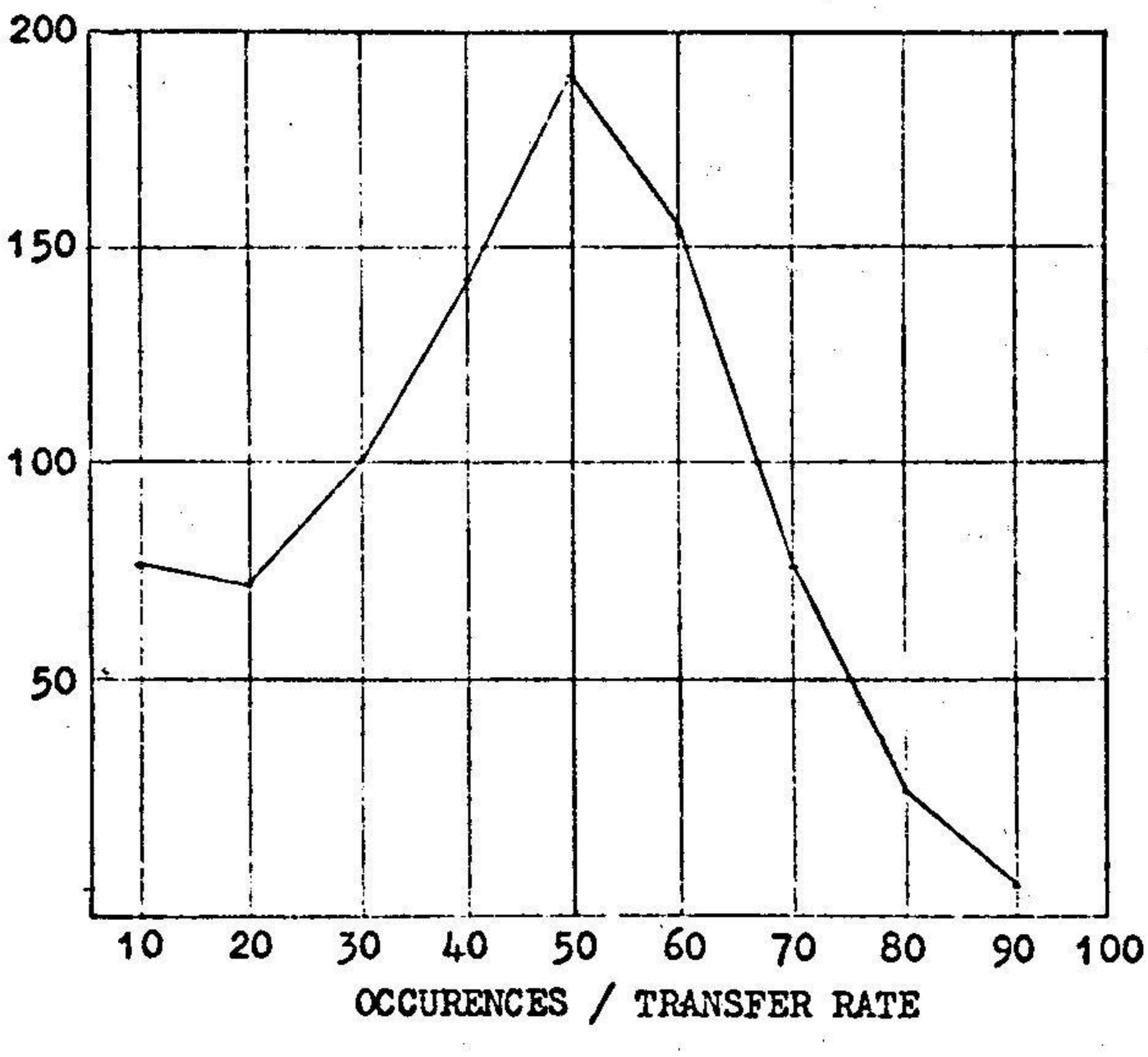


Transfer rate in pages/sec.
$\frac{3}{4}$ megabytes of core
2 drums, 1 pseudo drum
32 users

16 minute sample

KEY

1. Diskfile demand reads.
2. Drum and pseudo drum demand reads.
3. Drum and pseudo drum prepage reads.



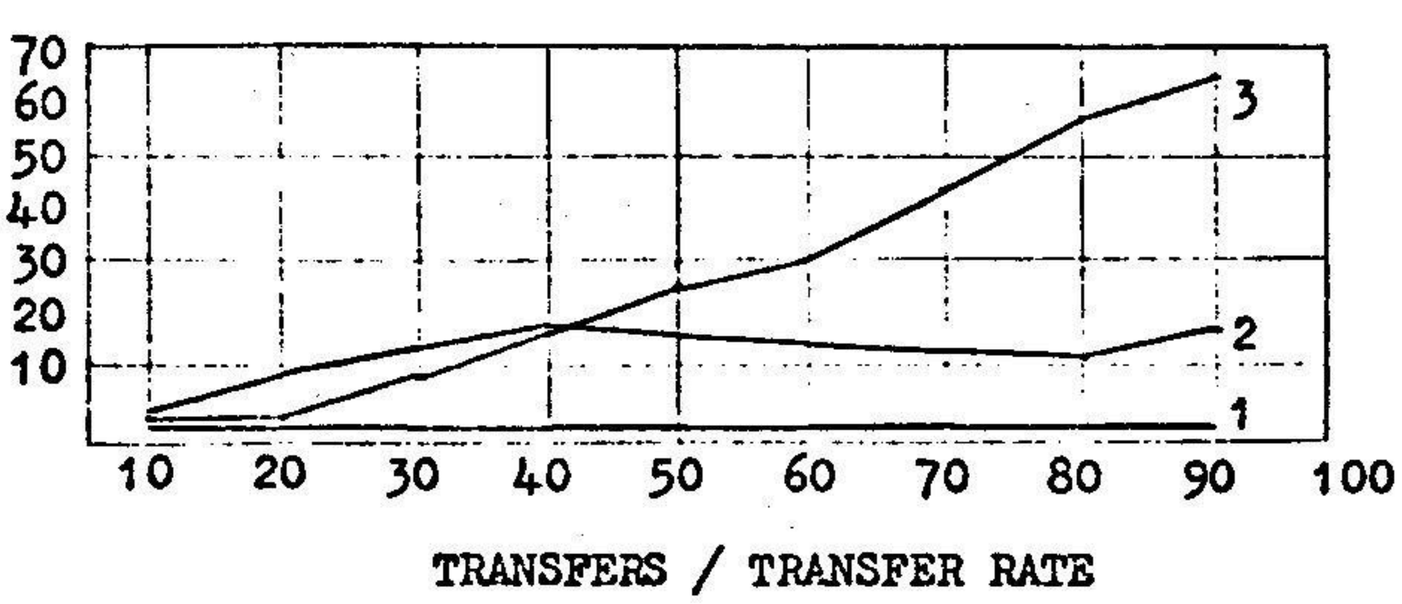


figure 5

the identity of the next priority queue to be chosen. In the initial choice algorithm described in [w2], once a priority queue was selected, and if there was a virtual processor on it, it was allowed to enter the MPS only when the amount of unallocated main store was greater than the selected virtual processor's main store allocation. If the priority queue was empty, or after the selected virtual processor had entered the MPS, the choice algorithm was re-enabled to select another priority queue.

The decision to use the number of unallocated main store pages as the global constraint on a virtual processor's entering the MPS was our first choice, on which we intended to improve if monitoring justified it. We have now done this monitoring and it has allowed us to ascertain that with the original global constraint, the main store was not used as effectively as it might have been. There are three identifiable reasons why:

The first is that the measure of unallocated main store excludes pages that are free due to sharing. This occurs because a page that is shared among two or more virtual processors is included in the allocations of each of the virtual processors using it, yet there is only one copy in main store.

The second is that a virtual processor's working set is by definition always less than or equal to its allocation. There is thus the likelihood that a number of allocated main store pages remain free.

The third relates directly to the performance characteristics of the drums and diskfiles. It is a result of the difference in wait time of a page that is fetched on demand against the wait time of those that are prepaged (figure 5). Prepaged pages arrive three times faster than demand pages. The reasons for this will be discussed later. This difference means that allocated pages that are free while a virtual processor with a large main store allocation demand pages up to its working set, could have been usefully utilized by a prepaging virtual processor that had a small main storage and cpu allocation, as such a virtual processor could have come and gone while the other was still demand paging.

To overcome these deficiencies in the initial global constraint, two changes were made. Monitoring indicated that sharing within a single mix of resident virtual processors was relatively stable, and tended to change only when a new virtual processor entered, or an old virtual processor left the MPS. In light of this evidence, the choice algorithm was trivially modified to take account of sharing. A virtual processor is now allowed to enter the MPS if its allocation is less than the unallocated store plus the ammount of shared store. A virtual processor may also enter the MPS and prepage up to the unallocated plus shared limit, even if this is less than its full allocation. It is then allowed to run, subject to the constraint that a minimal number of free pages remain, and it is a short computation. Otherwise the computation is suspended until more main store can be allocated to it. We refer to this process as partial prepaging.

Recently the installation of a second machine with switchable peripherals, especially drums, has allowed us to experiment further, with the effects upon main store utilization of various sizes of active and main storage. One result we have arrived at is that the performance of the system in an interactive environment seemed to be limited by the amount, rather than the transfer capacity, of active storage, as we had previously believed. In fact a 3/4 megabyte machine with four drums, seems to result in smaller queues, and hence faster response than a one megabyte machine with three drums. This is an extreemly recent result, and we are not yet completely sure of its validity.

INTERACTIVE RESPONSE

In the discussion so far we have ignored the choice algorithm itself, concentrating instead on paging behaviour. here again problems arose that had not been originally foreseen. These occurred if a priority queue was empty, especially if it was a high priority queue on which small interactive computations are held. For this allowed the store to fill up with virtual processors chosen from lower priority queues - virtual processors with large store and cpu allocations.

The problem that arises is, that once these virtual processors enter the MPS, they block the entry into the MPS of virtual processors that arrive on higher priority queues in the interim. If three or four large virtual processors, with a cpu allocation of ten seconds, are resident together, it could be as long as

thirty to forty seconds, in the worst case, before another processor can enter the MPS. This problem has been overcome by limiting the multi-programming level among virtual processors chosen from low priority queues. This guarantees that a certain amount of main store will always be free for allocation to high priority virtual processors. Doing this does not radically affect the cpu utilization of the system, as a single low priority virtual processor is capable of saturating the cpu when its working set is fully resident.

The second change to the choice algorithm was motivated by a political decision; EMAS was to be first and foremost an interactive, rather than a remote batch, system. Thus another simple amendment was made.

If a virtual processor remains active for more than a certain period of elapsed time, currently 6 minutes, or if it was initiated by the batch scheduler, it is considered to be penalized with respect to more interactive virtual processors - those that go to sleep from time to time. If a virtual processor is penalized its paging behaviour remains the same as if it were not. Its store and cpu allocations, category and priority are determined normally. The difference is that three out of every four times it is selected, it is returned to the back of the priority queue from which it was selected, rather than being allowed to enter the MPS. Thus penalized virtual processors enter the MPS less often, unless there are no unpenalized virtual processors on the same queue, in which case they enter the MPS normally.

THE CPU SCHEDULER

Another area where some improvement has been achieved is that of cpu scheduling. This area has been of little interest to us, and we believe, that while it may still be possible to effect major improvements in system performance, through improved cpu scheduling, it is unlikely. The goal of cpu scheduling in EMAS is one of satisfying a number of simple constraints. Context changes of a virtual processor are expensive and should be minimised. Demand paging virtual processors should get the cpu as soon after satisfying a page fault as possible, and small computations should be processed in as short an elapsed time as possible, so as to free main store for the next small computation.

In the system described in [w2] a simple queue of virtual processors able to take the cpu (the run queue) was maintained, and we restricted ourselves to experiments with various durations of CPU time slice. What we discovered is that the throughput of small interactive computations increased as we reduced the time slice, though of course at an increased cost in context switching overhead. At that time we settled on a time slice of thirty milliseconds, and worked on improving other areas of the system where we felt our efforts would yield greater results. We have now though managed to satisfy our design constraints more fully. To do this we introduced a system of three run queues of differing absolute priority. On the first we placed virtual

processors that had not completed their previous time slice before page faulting, on the second we placed those virtual processors that had completed a time slice, and were not on the third and lowest priority queue on which we put penalized virtual processors. By doing this we were able to increase the time slice to 100 milliseconds and still improve the throughput of small virtual computations.

ACTIVE MEMORY MANAGEMENT

It is in the area of drum handling that we have made our most important improvements, which have in turn impacted back into the scheduling of other resources.

The most important change came as the direct result of the discovery that the drums were only performing one fifth as fast as we thought they were. That the system had been capable of the performance reported in [w2] with drums transferring pages at a rate slower than some disk subsystems has a message in it somewhere. The cause of this performance degradation will be obvious to anyone familiar with the early history of drums. The electronic switching between tracks was taking longer than the inter-record gap. For once this was not a case of the software being unable to keep up, but of a failure of the drum control unit itself to keep up. Luckily there was enough room left over on a track to interpose three dummy records between each page frame. The page frames then occupying records 0,2,4, and 6 rather than 0,1,2, and 3 as before. It is a testimony to the

flexibility of the system software as it currently exists, that the change required modification of one table in the rotating memory handler and anotner in the drum formatting programme. A changeover was achieved successfully in the first systems development slot after the discovery.

The other changes to the rotating memory handler came as a result of experimentation. We have concluded that systems efficiency is improved by prepaging, and by ordering transfer requests in each of the sector queues by type. We place demand page reads first, then prepage reads, and finally pageout writes. This prevents the blocking of demand pages by other forms of transfer, and of reads by writes. It is our understanding, that others have reached a similar conclusion [12]. It is a little frightening to think that drum controllers have been built that do not impose this sort of discipline, especially as they have been constructed after a great deal of modelling and simulation.

With respect to prepaging, it is obvious that the amount one does is tied to the characteristics of one's backing store. If an immediate access medium such as bulk store is used, then of course it makes no sense to do any prepaging. In our configuration, the following seems to occur. The average number of virtual processors in the MPS is six, of which one is on the cpu, one is in a run queue, and four are waiting for the arrival of one or more pages. If we operated a pure demand page strategy, even with shortest latency time first SLTF ordering of transfers on each drum, we would not achieve much better

throughput than that produced by a first in first out FIFO ordering. For there would be little more than a single transfer queueing on each drum. By prepaging we increase the number of transfer requests pending on each drum, and hence in each sector queue. We thus gain the benefits of a SLTF ordering on transfers. Approximately 80% of all pages entering the APS are prepaged. Of the prepaged pages 20% are never used. While this figura seems high; it should be remembered that a high proportion of prepaged pages (well over half) are transferred in sector windows that would not otherwise be used, and hence to a certain extent are transfered at no cost.

Because of the performance mis-match between the diskfiles and drums, a mis-match that grows considerably worse as the queue of transfers pending on each device grows, it is imperative that the majority of transfer requests be for pages, copies of which are held on the drum. Back of envelope calculations indicated that we should aim for a transfer ratio of 20 drum transfers to 1 diskfile transfer, as this would create a balanced load on each device. Because of the shortage of page frames in active memory it is neccessary to optimize the set of pages held in active memory, with respect to their expected future use. It seemed reasonable given our success with a page allocation strategy for main storage, that we should employ a similar strategy for active store. This is especially true if one takes into account the consistency rule for information in immediate storage. This rule states: that every time a page belonging to a virtual processor is updated in immediate storage every page belonging to that virtual processor in immediate storage should also be updated.

This guarentees that a virtual processor's *image* in immediate storage is always self consistent, and that the *immediate storage image* always represents an *instantaneous image* of the process, though rarely the *real time image*. This rule yields a form of *automatic checkpointing*.

The observant reader may have noticed a flaw in this argument. What does one do with write shared pages? When these are updated in immediate storage, parts of the images of other virtual processors are also updated. There is in EMAS no attempt to propogate the consistency rule from virtual processor to virtual processor, through the write shared material. As file indexes are connected into multiple virtual memories in write shared mode, it would require the updating of all immediate storage images any time a single virtual processor's immediate storage image was updated. Hence automatic checkpointing in the sense it was initially intended does not exist, though the continued application of the rule does minimize the chance of inconsistency in a user's files as the result of a system failure. One effect of adhering to the consistency rule is that it is impossible to operate a global allocation policy in active storage.

The method of allocation employed is briefly as follows. A virtual processor is allowed to build up pages in active storage, until one of four events occur. It disassociates a file from its virtual address space (*disconnection*), it overflows its active storage allocation, it remains asleep for two minutes, or a certain number of residency periods have passed without its *active storage working set* having been recomputed.

The pages that a processor uses during each period in the WFS are noted, and a cyclic record of page use during the last four periods is kept on its master page. It is from this record that the active storage working set of the virtual processor is calculated. There are three progressively stiffer algorithms that are applied depending upon global active storage saturation.

WS = The union of the four periods.

WS = The union of the intersection of the three oldest periods and the most recent period.

WS = The null set.

All updated pages belonging to that virtual processor are then updated in immediate storage, and those pages in active storage no longer in the virtual processor's working set are deleted. Here as in main store, a virtual processor cannot acquire more than its fair share of a system resource.

PSEUDO MEMORY

There is one other change to the system that needs to be mentioned, and this is the introduction of a *pseudo drum* as a *memory level extension*. The concept of allocatable memory extension is exceedingly simple and general.

It is a well known allocation problem, that one needs to keep a certain amount of allocatable resource in hand to avoid *deadlocks*, or in our case the need to remove a virtual processor

from a memory level prematurely - a form of thrashing. It is thus impossible to utilize all of a memory level, unless one somehow extends the allocatable memory at that level, so as to use all of the real memory. This extension is effected through the use of pseudo memory.

Pseudo memory can be used at any memory level that has an address continuity with the next. Examples are: main memory - mass memory, mass memory - drum memory and drum memory - disk memory. The main memory - drum memory boundary does not display this characteristic of address continuity, as one is addressed in bytes, and the other in pages. We are thus able to use a pseudo drum, which is part of a small disk pack, the rest of which is used as a collection pool for event monitoring records, but not pseudo main memory. With this extension the active storage allocation algorithm can endeavour to allocate all of the real drum pages, secure in the knowledge that if it overflows the real drum, there are pseudo drum pages in which to put the information. We are thus able to achieve full drum utilization.

Needless to say pages are moved from the pseudo drum onto real drum when space becomes available on the latter.

It should be obvious from this discussion about utilizing drum storage that we do not fully utilize main storage. Figure 6 shows main store utilization during a 24 hour period. It is for others to decide how we fare in relation to other techniques.

MACHINE A

201 allocatable pages
3 drums and 1 pseudo drum

from: 23:28 on: 27/02/74
  to: 23:18 on: 28/02/74

Queues sampled every ten seconds.
Each point represents 1000 samples.

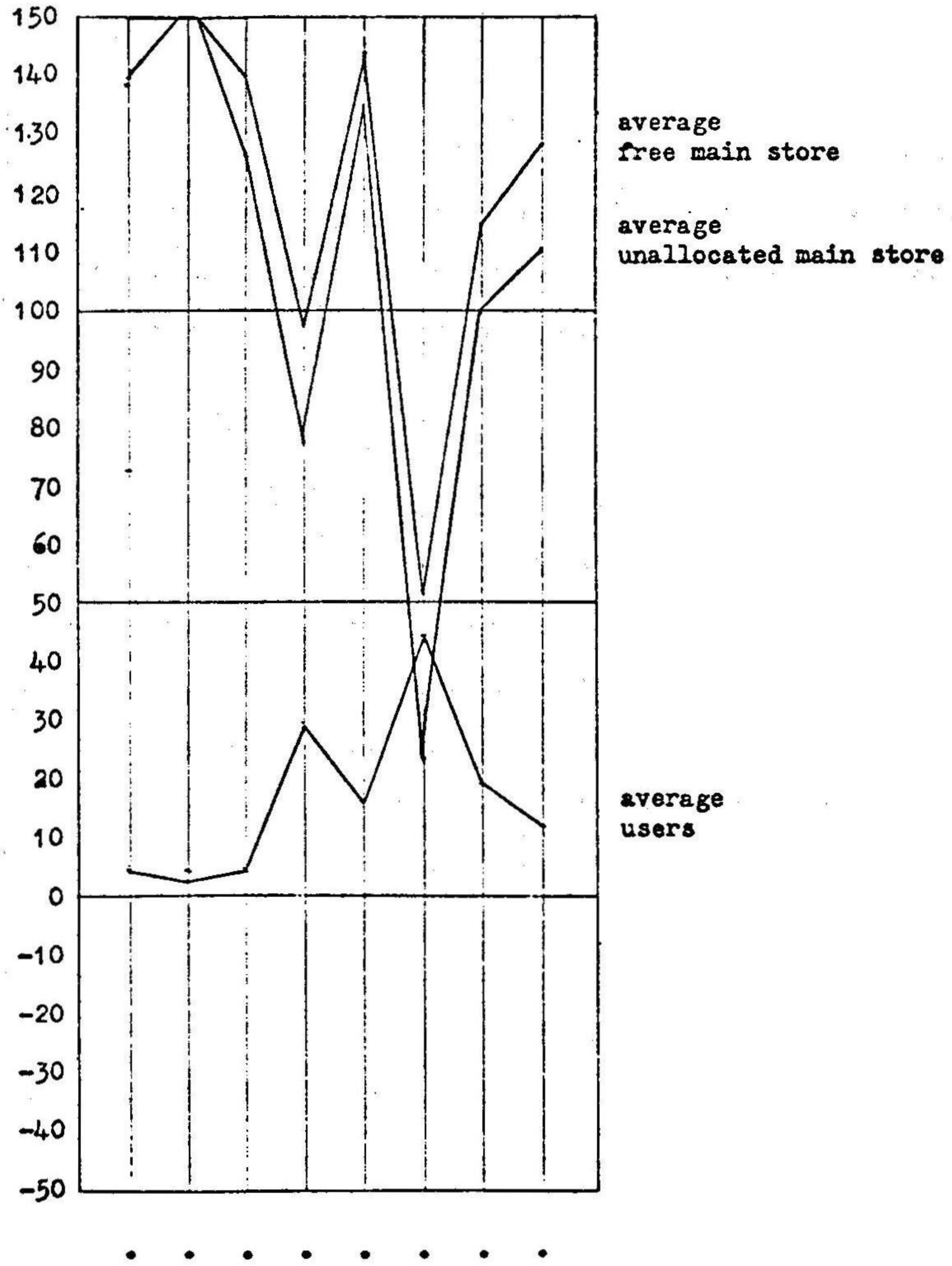


figure 6

SYSTEM PERFORMANCE

The system currently supports a maximum of 55 simultaneous users, with the *elbow* (the point at which the first critical resource becomes overloaded) in the response curve appearing at somewhere between 45 and 50 users, depending upon the *mix*. Under this type of loading, the drum channel transfers about 65 pages a second, and the diskfile channel about 6, of which less than 1 - 2 are demand reads. Approximately 90% of the cpu is utilized with this number of users, the remaining 10% being free, due to an instantaneous lack of compute bound virtual processors. In general 58% of the processor is given to the virtual processors, while the remaining 40% is spent in the resident supervisor. Over 75% of this time is spent in only two resident processes: The drum handler: and the working set calculator, This time would be radically reduced, with the addition of appropriate *hardware mechanisms* in each case: hardware drum scheduling, and access and usage information on the the segment and page tables, rather than the store. There are in fact 8 keys which have to be read out and reset on each page, at every *strobe period*. Given the current hardware configuration we see no way of improving these figures while running interactively with many users, and fast response.

While running *batch work* overnight with 6 batch streams, the system achieves effectively 100% cpu utiliazation, 85% of the time being spent in virtual processors.

The meantime between crashes due to hardware malfunction is currently 25 hours, and we encounter approximately two failures a month, due to software malfunction. This while the supervisor is still under development, and being changed about twice a week. When running proven supervisors, we have found that it is possible to achieve an essentially zero software error rate. Both the hardware and software error rates are lower than they might be, due to extensive checking. graceful degradation features, and the vetting of all incoming messages by system processes.

ACKNOWLEDGEMENT

There have been too many people involved in the development of EMAS to mention them all individually. Specific thanks though are due to Professor S. Michaelson for creating the environment in which this work could be undertaken, and to Dr J. G. Burns, who provided moral support, when few external to the project believed it would be successful. Special thanks are due also to Professor B. Galler of the University of Michigan, without whose encouragement this paper would not have been written.

BIBLIOGRAPHY

b1). Bobrow, G. D., et. al., 'TENEX, A Paged Time Sharing System for the PDP 10' , CACM, March 1972, pp. 135-143

b2). Bobrow, G. D., personal communication

b3). Burns, J. G. et al, The EMAS User Manual, Edinburgh Regional Computing Centre, 1972

b4). Barritt, M. M., et. al., The IMP language manual Edinburgh Regional Computing Centre, 1970

c1). Corbato, F. J. and Vyssotsky, V. A., 'Introduction and Overview of the MULTICS System' , Proc. AFIPS 1965 FJCC Vol. 27, Part 1, pp. 185-196

d1). Denning, P. J., 'The Working Set Model for Program Behaviour' , CACM, May 1968, pp. 323-333

d2) Denning, P. J., 'Virtual Memory' , Computing Surveys, Vol 2 No. 3, September 1970

f1). Fuller, S., 'Performance Characteristics of an I/O Channel With Multiple Paging Drums' , technical report # 27, Stanford Electronics Laboratory, August, 1972

g1). Galler, B., personal communication

l1). Lauer, H. C., 'Bulk Core in a 360/67 Time-Sharing System' , Proc. AFIPS 1967 FJCC, Vol 31, pp. 601-609

l2). Lynch, W. C., personal communication

l3). Lett, A. S., and Konigsford, W. L., 'TSS 360: A Time Shared Operating System' , Proc. AFIPS 1968 FJCC, Vol 33, Prt 1, pp. 15-28

m1). Millard, G., 'The Standard EMAS Sub-System' , Accepted for publication in The Computer Journal

o1). Organick, E. I., The MULTICS System, An Examination of its Structure, MIT Press, Cambridge Mass., 1972

r1). Rees, D. J., 'The EMAS Director' , Accepted for publication in The Computer Journal

s1). Spier, M. J., 'A Model Implementation for Protective Domains' , International Journal of Computer and Information Science, Vol 2 No 3, September 1973, pp. 201-228

s2). Stephens, P. D., 'The IMP Programming Language' , Accepted for publication in The Computer Journal

w1). Whitfield H., 'The Organization of the University of Edinburgh Time Sharing System', International Seminar on Advanced Programming Systems, Vol. ii, No. v, Jerusalem, 1968

w2). Whitfield, H. and Wight, A. S., 'EMAS, The Edinburgh Multi-Access System' , The Computer Journal, Vol. 16 No. 4, November 1973

w3). Wilkes, M. V., 'The Dynamics of Paging' , The Computer Journal, Vol 16 No. 1, February 1973, pp. 4-9