An algorithm for scheduling storage on a non-paged computer

By D. C. Knight*

A method is described of allocating core storage on a computer without paging facilities, working in a time-shared environment. The algorithm includes techniques for the dynamic repositioning of program segments, and the dumping of segments to a secondary store, in order to enlarge and consolidate unused portions of the core store. The paper also gives the results obtained from the random simulation of a system requiring the scheduling algorithm, to check out the logic, and to provide patterns of behaviour.

(First received July 1967, and in revised form November 1967)

1. Introduction

This paper presents a method for allocating core storage between multiple users of a computer that does not have paging facilities. That is, the computer is used in the conventional time-shared (or store-shared) mode, with several users or jobs requiring core storage at any instant. The users may be accessing the computer from many remote terminals, or be running jobs via a standard time-shared batch processing system, or a combination of the two. The particular configuration used to implement the algorithm was an English Electric KDF9 computer with 32K of core store, backed by a 4 million word disc in addition to the standard peripherals. The storage scheduling was required as part of an experimental system (called DEMOCRAT), being developed at the National Physical Laboratory, Teddington, to provide multi-access and time-shared facilities on the KDF9.

2. The DEMOCRAT system

In order to explain the scheduling problem a brief description of the DEMOCRAT system is required. A basic self-contained segment of program is called a *module*, and is normally held as a file on the disc. It has associated with it various parameters such as: number of core words required, number of 40-word disc sectors occupied, a unique identifier and security password, and several priorities. The priorities include a software priority giving the importance of the job relative to central processor time, and a space priority giving the importance relative to core store allocation. The module also contains an area for dumping the machine registers such as the nesting stores, the link address, the subroutine jump nesting stores, and the modification registers. This dump gives the values of the registers when the module was last receiving service from the central processor, i.e. was last 'current'.

The KDF9 operates under control of a master program which is resident in core, and runs in a privileged mode called Director mode. This routine is activated by, and recognises, all interrupts and requests for special services such as input/output. Before handing control back to a normal-mode program the Director may set hardware storage lock-outs, using the base address and number of locations registers. In this way several segments of program can occupy core store, with only one receiving service at any time, and each protected from the other by storage lock-outs.

Under DEMOCRAT the modules in core store are linked together on a chain of latched modules. When the current module causes an interrupt or requests some service, a jump is made to the Director mode routine, called INTERFACE. Its first job is to dump the contents of all registers into the module area. Then it recognises and processes the interrupt or provides the service (e.g. by initiating a call to a service module).

INTERFACE then selects the next module on the chain to be made current by choosing one such that:

(*a*) it requires further processing, or has been called by another module, i.e. it is 'active',
(*b*) it is not inhibited waiting for the completion of an interrupt or a service,
(*c*) it has the highest software priority of modules satisfying (*a*) and (*b*).

Before returning control to the selected module, the registers are reset from its dump area, and the storage lock-outs are set. By suitably defining and dynamically adjusting the priority system, DEMOCRAT avoids long delays on low priority jobs, while ensuring quick response for high priority ones.

3. Storage allocation

A particular service handled by INTERFACE is the loading and latching to the chain of a new module. This may be required as a sub-module or new segment of an existing module on the chain, or may be a new job initiated by a remote user or the background system. This service is provided by a special module called WANTED, which contains the scheduling algorithm. WANTED checks the validity of the new module, attempts to find room for it in the core store, and if successful reads it down from the disc before latching it to the chain.

* *National Physical Laboratory, Teddington, Middlesex, England* (while on leave of absence from the *Computing Research Section, C.S.I.R.O., Australia*); now at *King's College, Strand, London, W.C.2.*

The KDF9 storage lock-out facility requires that a currently available piece of core be a contiguous set of 32-word segments. Thus modules must be located in continuous blocks of store to make use of this protection. At any instant when a new module is required to be loaded, the store consists of alternating used and unused blocks, with the lower part allocated as system resident area, and not available to the loading procedure. Each latched module may be thought of as being preceded by a gap, which may be of zero length measured in 32 word units (see **Fig. 1**).

Hopefully there will exist a gap which is large enough to contain the new module, in which case loading can proceed immediately. In practice this is unlikely, so that a strategy has to be adopted aimed at enlarging the gaps until one becomes large enough.

Each latched module can be thought of as having one of five statuses, which change dynamically:

(*a*) Inactive and erasable, i.e. no more servicing is required, and the space occupied is re-usable,
(*b*) inactive, to be dumped on the disc before the space can be re-used,
(*c*) active and movable in the store,
(*d*) active and both movable, and able to be dumped (if its space priority is lower than the new module),
(*e*) active but immovable, either to disc or in store.

The status is determined by consideration of the type of module, and whether it is waiting for some service or interrupt to be completed. That is, a module can be dumped to the disc and retrieved later, or moved in the store, if its exact location is of no importance to any other module, and it is not in the middle of an I/O transfer.

The first improvement to the gaps is made by unlatching from the chain and erasing all modules of status (*a*). This will free all areas no longer required.

The next thing, if this fails, is to locate movable regions in the store—that is sequences of gaps and movable modules (status (*b*), (*c*), or (*d*)) bounded by immovable ones (or the boundaries of core store). To each region is associated a region gap, which is the sum of all gaps in the region. If one of these region gaps is large enough, then the region is consolidated by moving each module into the gap on its left (down the store) in turn, accumulating the gap on the right until it is large enough for the new module.

If no region gap is large enough, then each region is inspected to see if the modules in it will fit into other gaps outside the region or other region gaps, so that the region gap can be made large enough. The modules are checked in decreasing order of size, so as to minimise any movement that may result.

If this proves successful for a region, then the modules for which gaps have been found are moved out, and the remaining ones moved down the store to consolidate the region gap until it becomes large enough for the new module to be loaded. When moving modules into another region, it may also be necessary to move those region modules down the store to consolidate the gap.

The final stage of the loading strategy is to see if the region gaps can be made even larger by dumping onto the disc some of the modules that cannot be moved out into other gaps. First, modules of status (*b*) are inspected in descending order of size in each region, to see if dumping them would enlarge the respective region gap to the required size (the gap being already augmented by the modules which can be moved elsewhere). If still no region is potentially large enough, then the same procedure is tried for modules of status (*d*). If either of these is successful for some region, then the necessary modules are dumped onto the disc, all possible modules are moved out of the region, and the remainder moved down the store until the consolidated gap becomes large enough.

A failure of all these techniques means that the new module cannot be loaded at this time, and the WANTED module gives up the attempt. In this case a further attempt will be made to load when the module requiring it is next being serviced.

Neither moving nor dumping of modules is performed until the strategy has been shown successful in theory. That is, no adjustments to the modules in core are attempted until it is known that the final gap will be large enough. However, while determining this, the original status of the latched modules may have been changed, due to an interrupt. To avoid trouble, each module to be interfered with is first unlatched from the chain, then its status is reassessed. If this indicates that the module is no longer movable or dumpable as the case may be, it is relatched unaltered, and the attempt to load the new module is abandoned. This is justifiable on the grounds that any interruption of WANTED will be on a higher priority level than the module requiring the service.

4. Dumped modules

Modules dumped onto the disc are of two types. First, inactive ones are dumped rather than erased because the next user will require the latest updated version (this could apply to some system modules). A flag is set in the original copy of the module held on the

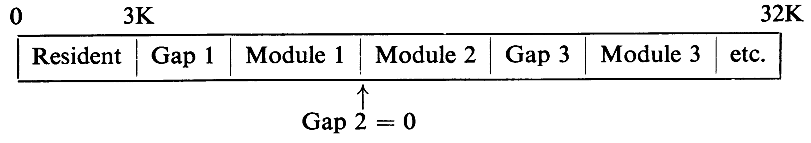

Fig. 1. Diagram of core store allocation

disc, so that the next call will in fact cause the dumped version to be loaded, and not the original. Secondly, active modules are placed temporarily onto the disc to make room for modules with higher space priorities. However, it is imperative that these dumped modules find their way back onto the latched chain. To achieve this a queue of dumped active modules is kept and inspected at regular intervals. This queue contains both the identifiers and the software priorities of the modules. At each inspection of the queue the priorities are increased, and compared with the priorities of the latched chain. If a dumped module is at a higher level than anything on the chain, it is loaded back into the store by WANTED, and will receive instant servicing. In this way no dumped module will stay out of the chain for too long, although a very low space priority module could be dumped several times before it has completed its activities. The queue is also limited in size, so that when full, no further dumping is allowed until there is room again.

5. Discussion

Fig. 2 gives a simplified flow diagram of the scheduling algorithm.

The problem is very much different from that experienced on a computer with paging facilities. In the latter case the core store is allocated in pages or sections, each having protection facilities. The scheduling becomes a matter of finding enough free pages to satisfy the demand, with some pages being dumped to a secondary storage device, but with no necessity to shuffle sections of store to improve gaps.

It is obvious that with this type of problem, the techniques to search for an optimum gap are in theory very many. For instance when searching for a gap into which to move a module from one region, a secondary level of moving could be performed in a second region, to move modules out into other gaps again, and so on. Similarly dumping could be performed at many levels. Essentially the problem could be tackled by the recursive use of one level of moving and dumping, to allow deeper levels to be reached.

It would also be desirable to simulate these multi-level searches in order to find the optimal strategy in terms of the number of words moved and dumped, before actually performing any moving or dumping. Unfortunately this optimisation could be very lengthy in terms of machine time, and could result in a great deal of movement both of words in store, and between store and disc. In fact it is necessary to compromise at some point and restrict the technique to a reasonable level of moving and dumping.

Only one level of dumping was allowed in the implementation to limit the use made of the disc. It was felt that if too many transfers were made, the machine would be tied up waiting for them to be initiated, and the disc itself could soon become overloaded. Two levels of moving were allowed to avoid excess word shuffling and to prevent the algorithm becoming too complicated. With a basic module such as WANTED, which by its nature must remain resident in store, an effort must be made to limit its size. Dumping was given a lower priority than moving, even though a disc transfer once initiated can proceed in parallel with other computations. However, this was again considered necessary as another means of limiting disc usage.

As an extension to this work, it is possible to divide the scheduling algorithm into two logical parts each contained in a separate module. Only the first part would have to be resident in core, thus freeing more space for general use, and also eliminating the need to keep the algorithm to a minimum size and complexity. The first part would attempt to load the new module into an existing gap in core, without any further manipulation. Only if this failed, would the second part be needed. This would then attempt to find space by moving and dumping the modules on the chain.

Part one of the scheduling must be capable of loading part two into a suitable gap. If such a gap cannot be found, then the loading attempt fails. This implies a recursive use of part one. The recursive facility could be extended to allow the loading of other modules sometimes required by the WANTED module, and normally resident in core. This would return even more core store to general use.

6. Simulation and testing of the algorithm

In order to test out the logic of the algorithm it was first coded in ALGOL, omitting the actual movement of words within the store. A random number generation subroutine was included to produce two sets of random variates. The first sequence was used to simulate module status by selecting a value between 0 and 5 from a rectangular distribution, thus giving an equal likelihood for all five statuses. The second sequence was used to simulate the new module size, by sampling a truncated approximation to a normal distribution to give a value between 1 and 60. $\left(x = \left|\sum_{i=1}^{k} u_i\right| + 1\right.$, where $k \geqslant 10$, and $a \leqslant u_i \leqslant b$ is taken from a suitable rectangular distribution$\left.\right)$. The available storage was set arbitrarily at 60 units as this gives 30K words in units of 500 words. Because new modules, or modules unlatched and moved, are latched to the end of the chain, the order of the links in the chain is not the order of the modules in core. The WANTED module keeps tables giving for each module in store such particulars as size, starting address, starting address of previous link in chain, and size of gap preceding the module. These tables are kept up to date and used to formulate the loading strategy for each request.

By varying the parameters k, a, b of the random number generator for module size, various sample sets have been loaded and statistics compiled, to determine the effi-

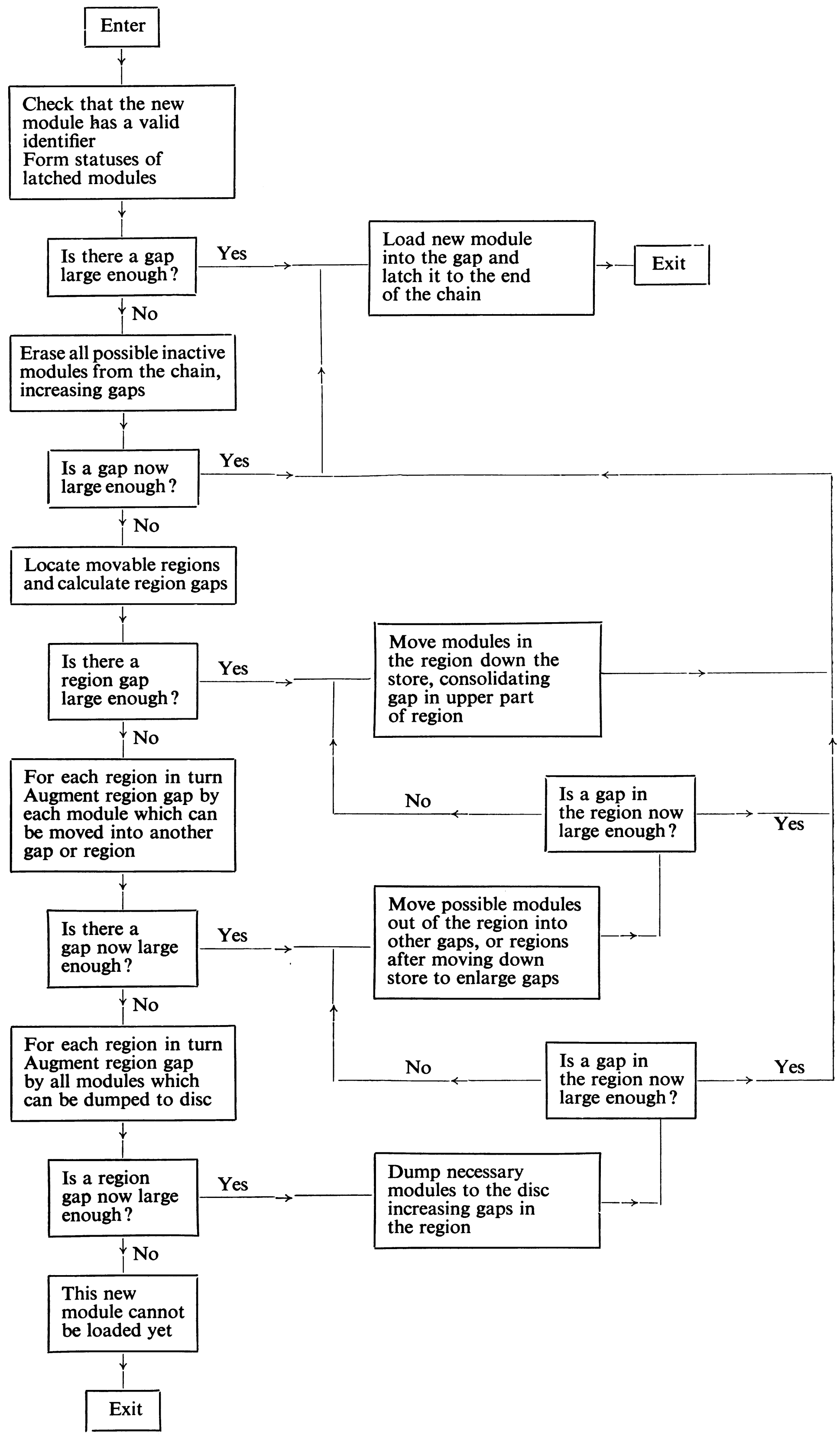
Enter
Check that the new module has a valid identifier Form statuses of latched modules
Is there a gap large enough?
Yes
No
Load new module into the gap and latch it to the end of the chain
Exit
Erase all possible inactive modules from the chain, increasing gaps
Is a gap now large enough?
Yes
No
Locate movable regions and calculate region gaps
Is there a region gap large enough?
Yes
No
Move modules in the region down the store, consolidating gap in upper part of region
For each region in turn Augment region gap by each module which can be moved into another gap or region
No
Is a gap in the region now large enough?
Yes
Is there a gap now large enough?
Yes
No
Move possible modules out of the region into other gaps, or regions after moving down store to enlarge gaps
For each region in turn Augment region gap by all modules which can be dumped to disc
No
Is a gap in the region now large enough?
Yes
Is a region gap now large enough?
Yes
No
Dump necessary modules to the disc increasing gaps in the region
This new module cannot be loaded yet
Exit

Fig. 2. Simplified flow diagram of scheduling algorithm

Table 1 Results from simulated runs (total store = 60)

	SAMPLE 1	SAMPLE 2	SAMPLE 3	SAMPLE 4	SAMPLE 5
Mean module size	6	11	9	21	12
Standard deviation	4·5	8·5	7·5	16	4
Total number loaded	128	128	128	128	128
Number loaded directly	108	98	101	88	95
Number loaded after moving	15	12	16	10	12
Number loaded after dumping	4	12	10	27	18
Number loaded after moving and dumping	1	6	1	3	3
Number delayed at least once	22	34	26	51	43

Table 2 Simulated sample 1 (mean 6, s.d. 4·5)

SIZE OF MODULE	1-5	6-10	11-15	16-20
Total number loaded	65	36	24	3
Number loaded directly	63	30	13	2
Number of delays	2	10	12	5
Number moved	18	4	2	0
Number dumped	0	2	3	0

Table 3 Simulated sample 2 (mean 11, s.d. 8·5)

SIZE OF MODULE	1-5	6-10	11-15	16-20	21-25	26-30	31-35
Total number loaded	40	31	20	15	15	4	3
Number loaded directly	40	27	15	8	5	3	0
Number of delays	0	4	8	10	20	4	9
Number moved	12	7	4	3	1	1	0
Number dumped	1	6	1	3	4	2	2

Table 4 Simulated sample 5 (mean 12, s.d. 4)

SIZE OF MODULE	1-5	6-10	11-15	16-20	21-25
Total number	10	31	52	28	7
Number loaded directly	10	27	37	17	4
Number of delays	0	12	22	19	8
Number moved	3	10	7	1	0
Number dumped	0	3	12	7	0

ciency of the algorithm, and its behaviour under differing conditions.

Table 1 gives a summary of the results for five such sets, each loading 128 modules. **Tables 2, 3** and **4** give a further breakdown of the results for the first, second, and last samples. They show the frequency, for different sizes, that modules are (*a*) loaded into available gaps without having to move or dump anything, (*b*) delayed loading because of lack of space, and (*c*) moved or dumped to make room for incoming modules.

The frequency of delays included the fact that some modules are delayed more than once. Approximately 50% of delays have been shown to be the result of status changing during a loading attempt. This would not be so frequent in a real situation, since in the simulated case each status is equally likely at any moment of sampling.

As might be expected in a sample with a low mean size, fewer modules need to be dumped or delayed than in a sample with a large mean because there are more gaps which can be manipulated. The amount of movement around store is relatively constant for all samples, as the gap size tends to be a reflection of the mean size of modules as they leave store for one reason or another.

With a very large mean such as sample 4, the number of delays rises considerably, as does the number of modules dumped. This is the result of a fairly rigid store structure, with few modules and therefore little chance of finding a region which can be manipulated.

Sample 5 is very close to the distribution of jobs run under the present time-shared batch processor on the KDF9. The fewer number of small modules due to a small spread, results in more dumping than with the the other samples of similar mean size.

Acknowledgement

The DEMOCRAT system for the KDF9 computer, and the associated file structure, were designed and implemented at the National Physical Laboratory by a team including Dr. J. L. Martin (now at King's College, London), Mr. G. G. Alway, Mr. B. A. Wichmann, and Mr. M. Woodger.