The design of multiple-access computer systems: part 2

By M. V. Wilkes and R. M. Needham*

In a previous paper, one of the authors discussed some of the hardware and software problems facing the designer of a multiple-access computer system. The present paper carries the discussion further and surveys certain areas of system design that are at present far from being well understood. In particular the problem of communication between processes in a multi-computer configuration is discussed. The authors offer no final solutions, but endeavour to set in perspective some of the problems that must be solved before highly efficient multiple-access systems can be designed.

In a recent paper entitled "The design of multiple-access computer systems" (Wilkes, 1967, referred to here as *Part 1*), one of us discussed, largely from the point of view of the software programmer, some of the problems involved in the design of large-scale multiple-access systems. In this paper, we propose to consider possible computer systems in greater detail and to discuss the related software problems. It would be much too early to offer any final solutions, but we will attempt to put in some sort of relationship to one another the various approaches that may be made and to illuminate some of the fundamental problems that have to be solved.

By far the most straightforward system is that of simple swapping in which only one object program is in core at a given time. It seems likely that, if drums are to be used for swapping, then swapping time will make this method uneconomic in the future. If, however, mass core memories become cheap enough, they could be used instead of a drum, and this might change the situation. The aim would be to make the transfer rate sufficiently high to consume all the storage cycles of the high-speed store; since mass core is, by its nature, slower than main core, this implies that the mass core should be designed with sufficient parallelism to permit rapid transfer of blocks of information, and that the circuits should also be designed with this in view. At the present time, the cost of doing this is very high. One type of mass core now being developed, which has a word length of several hundred digits in which can be accommodated a number of computer words, would lend itself well to this application. On the whole, the economics of simple swapping with mass core memory do not look very attractive at the present time. On the other hand, it is difficult to overemphasize the advantage of being able to use very simple software that is cheap both to construct and to maintain and that imposes very small system overheads.

Some time ago, when it appeared that mass core would have an access time perhaps ten times as long as main core, and when it also appeared that there would be no time advantage in transferring blocks of consecutive words compared with the transfer of the same number of words randomly placed in memory, one of us put forward an organization based on the slave memory principle (Wilkes, 1965). This suggestion does no appear to have been taken up, and it is possible that the premises are now false. However, it is also possible that the proposal has not been properly understood, and we believe that it is worthy of further study.

What one might term the conventional approach—in the sense that many people seem to adopt it unquestioningly—is to have a large main core memory backed by a drum, together, of course, with the large disc file that appears in all systems. The large main core memory consists of a number of modules which are connected through a criss-cross switch to several processors. The design of the switch must allow for simultaneous paths to be set up, so that all the processors can be connected to a memory module at the same time. The whole system is multi-programmed, there being a number of object programs in core at once. Undoubtedly, we shall see such systems in operation and undoubtedly they will work. In the present state of knowledge, however, the construction of a supervisor for such a system is an immense task, and when constructed it has severe run-time overheads. No doubt substantial improvements will be possible with further study, and it is probable that the greater part of the development effort available in the near future will be devoted to such systems.

An alternative to the multi-processor configuration is the *multi-computer* configuration. Here we have a number of processors, each with its own core memory and constituting, in effect, a complete computer; the peripheral capability of some of these computers may, however, be rudimentary compared with that of conventional computers. Some discussion of the special communication problems that occur in multi-computer situations will be given later. It is possible for a multi-computer configuration to be operated either on the principle that there is one program resident in each computer at a time, or with multi-programming in each computer.

The hardware configurations that can be devised are, of course, endless, and it is not to be expected that all actual configurations will fit neatly into the classification just attempted. For an example of a different approach to the design of a multi-computer situation, see Aschenbrenner *et al.* (1967).

* *University Mathematical Laboratory, Corn Exchange St., Cambridge.*

In order to illustrate the somewhat different significance that attaches to swapping in a computer with automatic paging and a virtual memory, than attaches to it in one of more conventional design, we will consider two systems. Both have a disc and a high-speed drum on which system files and user files are stored; which files go on the disc and which on the drum is at the management's discretion. In the first system, programs are read into core from the files containing them when the programs are first loaded; data files remain on the disc or drum and are accessed in the ordinary way. Some space on the drum is set aside for swapping and, when a user's quantum of time is exhausted, his program is transferred from core to the drum. Later, when that user's turn comes up again, the program will be reloaded. Thus, at any time, a given user's program is either in core or exists on the drum as a core image. When the program eventually becomes dead or dormant, the core image still exists on the drum; eventually it will be overwritten by the next program activated by that user. Until this occurs, however, the program is available for postmortem examination or reactivation and, in addition, the user can, if he wishes, by making use of a SAVE command, file it away for future use.

The second system also has files on a disc and on a drum, and it has, in addition, the full paraphernalia of segmentation, paging, and virtual memory. When the scheduling algorithm adds a process to the list of those active, such pages belonging to that process as are required are brought into core by the paging algorithm. Any page which has not been accessed for some time is liable to be over-written if space is required for another page belonging to the same or another process; if it has been changed since being loaded, but not otherwise, the page will be written back to the filing system. When a process is rendered inactive by the scheduling algorithm, there is strictly no need for any special swapping action to be taken, since inactive pages will, in the ordinary course of events, soon be overwritten by the paging algorithm. However, an improvement in efficiency can be achieved if the paging algorithm is apprised by the supervisor of the fact that certain pages have become inactive and are available for immediate overwriting.

It is to be noted that when a virtual memory is used in this way, there is no concept of a core image and no SAVE command. The user thinks of his program, together with any data segments attached to it, as existing in the virtual memory during the whole time that it is being run; when it becomes dead or dormant, it is still there with any changes that have taken place during running. If the user wishes to preserve the original program, he must make a copy of it (or of the parts which are liable to be changed by running) before he activates it.

The system just outlined has the advantage that the extra mechanism—over and above the paging mechanism—that must be provided for swapping is very simple, or even non-existent. As far as files held on the drum are concerned, we reap the full benefits that paging gives by way of reduction in traffic to and from the core memory· However, for files held on the disc, there is the longer transfer time to be taken into account whenever a page is brought down. Swapping is, in fact, taking place to and from the disc, and the drum is not being used at all.

The extra swapping time does not matter if it can be overlapped by multi-programming. As pointed out in the next section, however, multi-programming is not without its costs. Various methods for making better use of the drum can be suggested. It is possible, for example, to copy programs from the disc on to the drum before they are offered to the supervisor for running, and to copy them back again, if necessary, afterwards. The objection to this procedure is that it is likely to defeat its own object by creating a good deal of extra traffic between disc and drum. Another method which, we understand, is being followed in one implementation, is to swap pages to and from the drum, even though their real home may be on the disc; eventually, the versions held on the disc of pages that have been modified during the running of the program will have to be updated. Quite complicated book-keeping is necessary, and some additional traffic from the drum via the core to the disc results.

The efficiency of a multiple-access system based on a virtual memory must depend very much on the solutions obtained to the problems that have just been discussed. So far, very little experience in this area has been made public.

The consequences and costs of multi-programming

With multi-programming, several object programs are resident in core at the same time, in addition to the permanently resident part of the supervisor. The object of multi-programming is to reduce the processor idle time that can otherwise occur when the single object program in core is waiting for a response (from the drum, a disc file, a magnetic-tape drive, or some other peripheral), and the supervisor is unable to make use of the time. Clearly this benefit is obtained only if there is enough core to accommodate the two or more object programs. If, for example, there is 32K of core available for object programs, and 32K-jobs are permitted, then it is doubtful, given an average mix of jobs, whether there will be enough short ones for the extra throughput achieved by multi-programming to justify the overheads incurred in providing it. These overheads manifest themselves in a larger supervisor and in more supervisor time. Thus, if jobs of maximum size are common, in order to make multi-programming pay off, one must buy a larger core memory—probably twice as large—as is necessary to hold the largest program permitted. This extra memory, together with the extra overheads of the supervisor, represents the price paid for the greater throughput.

In the above, we have assumed that there is no paging. Paging enables better use to be made of core memory, so that less memory is required for a particular job. When this has been allowed for, however, the argument given above remains very much the same. More core is needed if several object programs are to be active at a given time than if only one is to be active.

One consequence of multi-programming that is unpopular with programmers is that the management is driven to put pressure on them to keep their programs short. In the computer systems with which we have been familiar until recently, a given program might just as well occupy the whole available memory, since any memory left over could not be used for any other purpose. Apart from multi-programming, however, the importance in multiple-access systems of reducing loading and swapping time would, in any case, have led to a demand on programmers to keep programs short.

Multi-programming enables a greater throughput to be obtained by reducing processor idle time. As we have just seen, however, this advantage is not obtained without some cost in equipment. The fall in the cost of processors enables us to consider an alternative way of securing a high throughput, namely to provide a number of processors that can run simultaneously, and to allow any given processor to remain idle while the program on which it is working is waiting for a response from a peripheral device. This avoids the complications of multi-programming. A multi-processor system, in which the processors share common memory, can be used in this way. However, if multi-programming were abandoned, one would perhaps be drawn towards the multi-computer arrangement in which each processor has permanently associated with it enough memory to hold the largest program for which that processor is intended to cater. The total memory required is probably not much more than that needed for a multi-processor system (without paging) and no criss-cross switch is necessary. This latter item is quite expensive and its presence tends to degrade the performance of high-speed core memory by the transmission delays that it inevitably introduces. In fact, it is hard to avoid the conclusion that the impossibility of designing a criss-cross switch that does not introduce some delay, means that, if core memories and processors of the highest possible speed are to be used, then a multi-computer, rather than a multi-processor configuration, must be chosen. For systems of normal speed, there is little doubt that the multi-processor solution is more economic at the present time; the situation could easily become changed, however, as the cost of processors falls.

An important type of multi-computer system is one in which a small peripheral computer is used for communicating with consoles and other peripheral devices, and in which a large system, which can itself be multi-computer or multi-processor, is used for processing. The connection between the small computer and the large system is made via the disc. The small computer contains the file master, together with the file directory, and an editing program. Files containing programs and data are created by users at consoles, or read in through input devices. These files have names by which they are known to the file directory. The large system draws its programs and data from named files on the disc and similarly returns its results to named files. In addition to programs for file management, the small computer could very well contain other programs designed to give rapid interaction with the user in typical circumstances; an example is a JOSS type system. Thus a great part of the rapid conversational response needed by the average user would be provided in a very efficient way, that is, by a separate small computer (no larger than is necessary) dedicated to the purpose.

The organization just outlined would suffice if no full conversational mode working (other than that provided by the peripheral computer) were required. A program would, however, only be able to communicate with its console via the disc. Some limited conversational mode working might be provided in this way, but to go further would require the provision of a direct transfer channel between the memory of the peripheral computer and the main core memory. Additional supervisory overheads would, of course, be incurred in administering this channel.

Organization of disc storage

It was mentioned in Part 1 that the first step in the design of a filing system is to establish a satisfactory method of controlling the allocation of space on the disc. This space is divided into records, the length of which depends on the particular disc file used; in a typical case, a record consists of 512 computer words. If files were of fixed length, then an easy and efficient procedure would to be allocate a number of consecutive records to each file. Since, however, files vary in length dynamically, it is necessary to use a chaining technique of the type familiar in list processing. In the simplest form of this technique, one or two words in each record are used to contain the link, the rest containing data. Initially, all the tracks on the disc are chained together to form a free list.

A preferable system is to maintain a table, or *storage map*, containing an entry for each record on the disc. Chaining of records is then done by means of appropriate entries in the storage map and not by recording links in the records themselves. This has the advantage that administration of the records can be performed without accessing them. Administration in this sense includes setting up the free list in the first place, returning records to it when they are no longer needed, locating particular records in a multi-record file, and checking the consistency of the filing system after a failure has occurred. With some hardware systems there is a further advantage in not having to use any of the words in a record to contain links. This is the case in the Atlas 2 system, where the record is 512 words, both on the disc and on magnetic tape; it would be rather inconvenient if disc records contained only 510 words.

A disc map normally occupies several thousand computer words and is rather large to keep permanently in the memory. It can, however, be divided into sections, and if there is some sort of paging, software-simulated or otherwise, then there is a good chance of the part of the map required being available in core.

The problem of organizing space on a drum is very similar to that of organizing space on a disc, and much of what has been said about discs applies equally well to drums. Indeed, if a system is equipped with both a disc and a drum, there is everything to be said for the principle that all space on the drum, other than that used for swapping, or for holding parts of the supervisor that are not permanently resident in core, should be treated as far as possible in the same way as space on the disc, and handled by a common system. This means that the system can put a file equally well on the disc, or on the drum; it is, in fact, a user or management decision where a particular file goes, the decision being taken on considerations of response time and economics. It is to be understood that, in what follows, references to a disc can, in general, be interpreted as though they were references to a drum, and *vice versa*.

It is important to keep clearly in mind the relative roles of character files and binary files. Character files are associated with the functions that, in earlier days, would have been dealt with by primary input and output; they are like character strings on punched paper tape. Binary files are much more like magnetic tape on to which blocks of binary information can be transferred from core. The analogy is very close if a magnetic tape system with addressable records is used. As pointed out in Part 1, however, the distinction between character and binary files lies not in the way they are stored on the disc, or treated by the filing system, but rather in the way they are accessed.

When a character file is used for input, access is usually required to it line by line or character by character. It is one of the tasks of the supervisor to provide this facility. Whenever a line or a character is required, the user program makes a call on the supervisor to deliver it. Since transfers to and from the disc can only take place in units of complete records, this implies that space in core memory must be used for buffering. The management of such buffering space calls for careful consideration.

If programs are normally run to completion once they have been loaded into core, that is, if swapping is the exception rather than the rule, then input buffering can be done wholly in space available to the supervisor. If a program is swapped out of memory, any information being buffered on its behalf is abandoned, and when the program returns to memory, the buffer is refilled. If a good deal of swapping takes place, it is better that the buffer space should be regarded as part of the user's working space, although it is used not by his program but by the supervisor, and the long arm of the supervisor extends to prevent him from accidentally writing into it. The advantage of treating the buffer as a part of the user's space is that, when the user's program is swapped out of core, anything in the buffer goes with it, and automatically returns when the program returns.

A similar buffering problem arises when a character file receives output from a program. In this case, the necessary buffer space is probably better regarded as belonging to the supervisor. Once information has reached the buffer, it is beyond the reach of the program, and the supervisor can write it into the file at any time that the space occupied by the buffer is needed for some other purpose.

In a virtual memory environment, the supervisor has exactly similar responsibilities for the buffering of character files. A character file used for input can be attached as a segment either to the user program or to the supervisor; a character file used for output is probably better attached to the supervisor.

It might be remarked here that, unless pages of a small size can be used, considerations of efficiency may render it inexpedient to use an automatic paging system for buffering small quantities of information. This remark applies even more to the buffering of information from teletypes than to the buffering required in accessing a character file.

File names and the avoidance of clashes

When access is first required to a file, an appeal must be made by the user program to the file master. The file is referred to by its full alphanumeric name, including the name of the directory in which it is to be found. The file master checks that the file exists and that access is permitted. It then passes information about the file and the user concerned to the supervisor. The file is then said to be *open* and the user program may make future references to it by calling on the supervisor. The supervisor maintains a list of the files that are open in this way, together with a direct reference to the number of the record on the disc where the file is to be found. If the file extends over more than one record on the disc, this reference is updated as the file is worked through, use being made of the storage map for this purpose. Once a file is open, it is not necessary for a user program to refer to it by its full alphanumeric name; instead, it can be referred to by a number giving its position in the list.

When the Cambridge multiple-access system was developed, it was possible to make use of a system of stream numbers already provided by the supervisor for dealing with input and output and for transfers to and from magnetic tape. Under this system, the programmer writes his program so as to take inputs from *input streams*, and to send outputs to *output streams*. The connection of these streams to physical input and output devices, or to magnetic tape, is effected not by the program, but by a preliminary job description. This system was easily extended to make it possible to connect streams to files.

The advantage of proceeding in this way is that the program can be written entirely in terms of streams, leaving until later the specification of where information comes from or where information goes to. Thus, by writing appropriate job descriptions, a program can be fed with information from a document read in from a tape reader, typed on a console, or taken from a file; and similarly for output. The request to the file master to open a file is made when the job description is processed, and this necessary procedure is thus completely divorced from the accessing of the file by the program.

It will be appreciated that, since an ordinary user writes many programs over a period of time and creates many files, some system such as that outlined is absolutely necessary in order to avoid name clashes. If programs had written into them the actual names of files that they require, and if different programs used the same names with different meanings, then much confusion could result. At the best, much tedious renaming of files would be necessary.

When the Cambridge system is used for processing jobs off line, the job description is one of the documents submitted along with the program and other input documents. All of these documents can, if the programmer wishes, be taken from named files. When working from a console, the programmer can type what is equivalent to a job description in the form of parameters following the command that initiates his job. Before making any attempt to run the job, the system checks that files referred to in the job description exist and are accessible to the user concerned. If all is not in order, the online user is informed immediately. The program may still fail, however, if it makes reference to a stream which has not been set up by the job description.

The CTSS proceeds in a similar way. Programs are written so as to refer to named files. If, when the program is activated, these files are found to exist, the program runs normally. If a file is missing, the user receives a message "NEED ALPHA BETA" or whatever the name of the missing file may be. He can then type "USE GAMMA DELTA" and the program will proceed using the file GAMMA DELTA instead of the file ALPHA BETA.

Communication areas

It is frequently necessary that concurrently running processes should be able to communicate with one another. They may do this via *communication areas* in storage. It is obviously important that once one process has begun to make changes to the information held in a communication area no other process should be allowed to access the information until the changes have been completed. When the communication area is in core, this may be achieved by using one of the registers in it to contain a *flag*. The flag is regarded as set or unset according as the register contains one or the other of two arbitrarily selected quantities, e.g. zero or one. When a process is about to make a change to the information in the communication area, it first sets the flag. Any other process finding the flag set will itself refrain from initiating any similar action. A difficulty, however, arises if there is an interval between the testing of a flag, finding that it is unset, and then setting it, since, if the process is interrupted before these operations are all complete, confusion can result. This situation can be avoided if the processor has in its order code an order which enables the flag to be tested and set in one storage cycle. The absence of such an order can lead to serious inconvenience and much unnecessary calling on the supervisor. A suitable order is one which tests the content of a storage register and, if the content is found to be zero, sets it to some non-zero value; if the content is non-zero, the order brings about a jump. Any order which changes the content of the storage location without destroying all evidence of what was there before can, however, be pressed into service; an example is an order which adds a number to the number in a storage register in one storage cycle.

It is sometimes necessary to make use of communication areas on a drum or disc. An example of such a communication area is a file directory. Here one runs up against the difficulty that drums are not usually provided with a writing operation that preserves evidence of the former content of a record. It would, of course, be possible to design the hardware so that the entire drum—or the relevant section of it—could be busied long enough for the record to be read into core and written back in a modified form. The problem is complicated by the fact that requests are not necessarily serviced in the order in which they are made on account of the need to minimize latency time. A change of ordering can also occur when a transfer has to be repeated because it has not been made correctly.

A better plan is to associate a flag with a communication area on drum or disc, and to keep the flag in core storage. Since it is in core, this flag can be tested and set in one storage cycle by means of an order of the type just described.

The above solution is not, however, possible in a multi-computer configuration in which the separate computers all have access to a common drum or disc, but have no core in common. In fact, if there is no direct communication between the computers at all, very serious problems are presented to the software designer. These problems can be avoided completely by the simple expedient of providing a small amount of immediately accessible storage available to all the computers. Even the provision of a single flip-flop is enough to turn the software designer from a worried man into a contented one. He would, however, be able to make good use of a whole register (composed of flip-flops) whose contents could be tested and selectively changed by the execution of a single order. A less satisfactory alternative is the provision of interrupt connections between the computers.

The serious problems of communication that arise if there is no communication between the computers will be illustrated by considering the case of the important configuration outlined above, in which a small computer services user's requests, and passes problems for solution to a larger system which then sends the results back. It is assumed that the only communication between the two computers is via the disc, and that, although both computers have access to the disc, they cannot both access the same record at the same time. A possible solution to the communication problem will be described

Two communication areas on the disc are provided; one, the *forward communication area* is used to send messages from the small computer to the large system, and another, the *backward communication area*, is used to send messages from the large system to the small computer.

The small computer decodes the job description and places in the forward communication area a specification of the job to be done associated with a serial number which will be referred to as the *cipher*. Ciphers will eventually repeat, but it is assumed that there are a large number of them. It is also assumed that either the communication area consists of a single record, or that the information in each record is complete in itself, so that the information in one record can be safely used while that in another record is being modified.

The large system periodically examines the forward communication area, and copies particulars of the next job to be done. The large computer makes a note of the cipher in a list and proceeds to do the job; in due course, it puts the results, or rather references to files containing the results, in the backward communication area, together with the cipher.

Periodically the small computer examines the backward communication area and passes information back to the user. When it has done this, it deletes the request from the forward communication area. The next time this area is examined by the large system, it notes that there is no longer any reference to one of the ciphers in its list; it deletes this cipher from the list and deletes the corresponding results from the backward communication area. The fact that each communication area is changed by one of the computers only, eliminates any possibility of the information becoming inconsistent.

It will be seen that the system overheads consequent on not having some direct communication between two computers are very serious. We have, however, not enumerated them all. It is necessary for both computers to have access to the same filing system, which implies, among other things, that they must make use of a common pool of space on the disc. Communications relating to filing and the administration of this space must be passed from one computer to another via cumbersome machinery similar to that which has just been described in detail. The result is unnecessary complication and duplication of software.

Restarting after system failure

Making it possible for the system to be restarted after a failure with as little loss as possible should be the constant preoccupation of the software designer. Even if all software bugs are eventually eliminated, hardware faults will always occur. Clearly, no software system can be entirely protected against hardware failure, but some are better than others. Often, what appear to be simple and straightforward software solutions, which involve a minimum of disc transfers, have to be rejected since the protection they give is not as good as it could be. Restart procedures should be designed into the system from the beginning, and the necessity for the system to spend time in copying vital information from one place to another should be cheerfully accepted.

It must be recognized that there will be occasions on which the situation after a failure is such that a complete reinitializing of the system is necessary. This will involve making use of the archive tapes to restore the filing system to what it was at some previous epoch.

It is good practice in systems programming to minimize the time during which the contents of a communication area—or indeed of any area containing systems information—are being modified and hence are temporarily inconsistent, since the consequence of a system failure occurring during this time can be serious. Often, when a number of things have to be done, it will be found that one particular order of doing them gives minimum vulnerability.

It is important that, as part of the procedure used to restart the system after a failure, the information in communication areas should be made consistent, even if it is not possible to make sure that it is correct. Otherwise, the system may operate in a crippled form and may, perhaps, eventually be brought to a halt by reason of a software tangle from which recovery is impossible. This remark is particularly important in relation to the file directory and disc storage map. An inconsistency may result in some records on the disc being neither recorded as free nor as belonging to one of the files in the directory. If they are left as they are they will be lost to the system. In the absence of further information as to their identity, the best thing that can be done with such records is to put them on the free list. Many other examples can be given of the way in which inconsistent information can lead to eventual trouble. For example, inconsistent scheduling information can easily cause the supervisor to go into a loop.

Redundant information can be included in supervisor communication or data areas in order to enable errors caused by system failure to be corrected. Even a partial application of this idea could lead to important improvements in restart capability. A system will be judged as much by the efficiencies of its restart procedures as by the facilities that it provides.

In preparing this paper, we have drawn heavily on experience obtained in designing the Cambridge multiple-access system, and our debt to our colleagues, in particular to D. F. Hartley and B. Landy, is a heavy one. We are equally indebted to experience obtained when visiting Project MAC at M.I.T. and to the many discussions that we have had with members of the staff of that project.

References

Wilkes, M. V. (1967). The design of multiple-access computer systems, *Computer Journal*, Vol. 10, p. 1.

Wilkes, M. V. (1965). Slave memories and dynamic storage allocation, *IEEE Trans. on Electronic Computers*, Vol. EC-14, p. 270.

Aschenbrenner, R. A., Flynn, M. J., Robinson, G. A. (1967). Intrinsic multiprocessing, *AFIPS Conference Proc.*, Vol. 30, p. 81.