The structure of a multiprogramming supervisor

By D. F. Hartley, B. Landy and R. M. Needham*

The paper discusses the basic logic of the supervisor which has been developed for the Titan computer at the Mathematical Laboratory, Cambridge University. The emphasis is on the underlying principles rather than the operating system as seen by the user.

(Received June 1968)

The supervisor in operation on the Titan (prototype Atlas 2) computer at Cambridge University provides a combined jobshop and multiple-access service. Work may be presented to the system either in the conventional manner or from a console. A user may make use of different modes of access at different stages of his work, and at all times he has available a disc-based filing system with magnetic-tape backup.

In this paper we are concerned with the underlying mechanism of the system, in particular with the means for activating and de-activating processes, the ways in which processes interact, the handing to and fro of work space, and the general multi-programming organisation. No attempt will be made to explain in detail the function of the various processes that this mechanism controls, or to describe how the system looks to the user. The same underlying mechanism could be used to support a variety of operating systems with user characteristics differing markedly from those of the Cambridge system. One such system is in operation at the present time, and will be referred to later in the paper.

A few facts about the Titan computer are required for the purposes of the present paper. The computer has three sequence control counters (program counters) called *main control*, *extracode control*, and *interrupt control*, respectively. Main control is used for user level programs, and interrupt control for the immediate processing of interrupts; when the computer is on interrupt control, further interrupts are held off automatically by hardware. When the computer is on extracode control, certain storage access privileges, which are debarred when on main control, are available; interrupts are allowed. Extracode control is used for the bulk of the supervisor, and for certain functions available to user programs. Such programs may call for one of these functions by executing an *extracode instruction*, which causes control to go to one of 512 entry points which are fixed and protected, at the same time switching the computer from main control to extracode control. The computer is provided with storage protection and relocation hardware; these features are briefly described where needed in the exposition. Peripheral control is almost entirely performed by interrupts, control bits and data being associated with specially addressed registers.

Routines and processes

In describing any system that makes use of multiprogramming, it is necessary to distinguish between *routines*, which are blocks of code, and *processes*. At any time, there may be a number of processes that are free to run. There may be other processes that are halted waiting for response from a peripheral device or for some other reason. The same routine may form part of a number of processes, some of which are halted in that routine and some of which are free to run in it. One sometimes speaks of a routine as being halted, although, in fact, it is the process using the routine that is halted.

Types of routine

There are three types of routine with which we are concerned. *Object programs* are entirely unprivileged and are of no urgency as far as the system is concerned. They run on main control, except when they are using extracode functions when they run on extracode control. *Interrupt* routines arise from hardware interrupts and are dealt with immediately; since they are themselves non-interruptable, they must be short. Interrupt routines run on interrupt control. *Supervisor routines* run on extracode control, except when they need to protect themselves from interruption; in these circumstances, they can run for brief periods on interrupt control. Supervisor routines make use of a number of *central routines* in the supervisor for such purposes as the acquisition and relinquishing of storage space.

Coordinator

As its name implies, the coordinator directs and controls the whole functioning of the supervisor, including the idling that takes place when there are no jobs to be run. The coordinator is entered whenever an object program has occasion to request supervisor activity, or when such activity must be temporarily suspended in order that another activity may take place. It can also be entered from an interrupt routine. If the call to the coordinator is from an object program or from an interrupt routine that has interrupted an object program,

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

the coordinator initiates the requested activity immediately. Otherwise, the coordinator places a request for the activity to take place on a queue in a manner described below; that is, entry to the supervisor is queued rather than nested.

Bases

When an activity is said to be queued, the object that is actually put on the queue is a *base*. This is a consecutive sequence of 2 to 32 memory registers, located in a fixed region of core set aside for the purpose. It is referred to by the address of its first element relative to the beginning of the region.

A base contains:

(*a*) One half-word used for chaining the base to a queue.

(*b*) An entry address for the routine that will perform the activity with which the base is associated. Provision is made for this address to be set when the base is queued. If the routine is halted, the entry address is set to the point at which execution is ultimately to be resumed.

(*c*) Queuing data. This includes the priority of the process, whether the routine associated with it is resident in core and, if not, whereabouts it is to be found.

(*d*) Some registers, varying in number from base to base, for dumping index registers when the activity is halted.

Some bases also contain:

(*e*) A small amount of working space for the process.

(*f*) One or more words used for associating data held in core with the base. The data are stored in chains with backward and forward pointers. This method of storage enables the data to be moved freely about in core when necessary.

(*g*) An extra half-word for chaining the base on to a second queue (see below under OP bases).

It will be seen that a base is linked both to a routine and to the data that will be used by the routine when it is activated. Together, the routine and the data define a process. A number of bases may be linked to the same routine, but each will be linked to different sets of data. For example, there is one base associated with each paper-tape reader; these bases are linked to the same routine but to different data areas. Processes associated with bases are referred to as SERs. The historical derivation of this name will not be given since it would confuse the reader.

Bases are of two kinds:

(1) Those permanently associated with activities that are able to run as independent processes. Examples are the KILL JOB SER and the 1 SECOND SER.

(2) Those assigned to object programs. These are referred to as *OP bases* and there is one for each object program in execution. OP bases are associated as required with supervisor routines that perform activities (for example, changing phase or opening a stream) that may be requested by object programs. In addition to data space needed by the associated routine, OP bases also have some space chained to them for holding information relating to the object program.

When an object program is not running its processor status (including the main control counter, which gives the re-starting address) may be stored here.

An SER is either halted, free to run, or dormant. It may be halted at its own request, or it may be halted in a central routine that it is currently executing. A common case of an SER being halted in a central routine is when an SER has called on a space routine to provide storage space for an object program and the storage space is not available. When halting is necessary, the coordinator is entered with a statement of the halt reason and (usually) with the address at which execution of the SER is ultimately to be resumed. The coordinator places the base on the appropriate halt queue and writes the resumption address into the base.

SERs that are halted are chained together on a halt queue; there are many halt queues corresponding to the different halt reasons. Whenever a halt condition clears, the coordinator is asked either (1) to transfer all the bases from the corresponding halt queue to the free queue, or (2) to put the first base from the halt queue on to the free queue, or (3) to put a specified base from the halt queue on to the free queue. These requests to the coordinator are always made by an SER. Often the SER that makes the call is the one that has cleared the halt condition. When the halt condition is cleared by an interrupt, the request is made by an SER called by the corresponding interrupt routine.

SERs that are free to run are chained to the free queue. For efficiency reasons, this queue is implemented as three separate queues connected head to tail. When an SER is queued, it is placed at the bottom of the appropriate sub-queue according to its priority. An exception is the RETRIEVER (referred to below) that is always treated as having lower priority than any other SER. An SER which is *dormant* is on no queue at all.

It would have been very convenient if the design of the supervisor had been such that halted SERs could be released for any one of several reasons instead of for one reason only. An example would be to release an SER either when a console user types a carriage return or when a certain time is expired. In the present implementation, requirements of this kind must be met by indirect means.

Central subroutines typically have two entry points. In one case, the SER calling them is halted if it makes a request that cannot be met (for example, in the case of a space routine, if no space is available). If a subroutine is entered at the other entry point, the process is not halted if the request cannot be met, but the subroutine returns control and reports the reason.

In addition to the chains corresponding to the free and halt queues, there is an additional chain running through all object program bases. An object program is either:

(1) Halted in an SER, in which case the SER will be on the corresponding halt queue.
(2) Running.
(3) Free to run but interrupted. In this case (unless it was the last object program to run) its processor status will be recorded in registers chained to the base.

When the coordinator searches the OP chain to find an object program free to run, it examines the marker in each base that indicates whether the object program corresponding to that base is halted or not. It will pass control to the first object program it finds which is not halted, after restoring its processor status if need be.

Space allocation

Atlas 2 is provided with base and limit registers for storage lockout; the base registers act as relocation registers. However, the content of a base register is not added to the address in the instruction, but instead the logical OR of the content of the base register and the address is formed. This imposes an awkward restriction on where program or data segments may start when they are loaded into memory. A segment of n words can only start at an address of the form 2^m, where m is such that $2^m \geqslant n$. This together with the fact that a section of core within the limits defined by the base and limit registers can be separately locked out, led to the adoption of an unusual method of utilising storage space.

Since object programs can only start at certain addresses, there is inevitably space left over when the object programs selected for loading have been fitted into memory. This space is utilised on a temporary basis for buffering and for holding sections of the supervisor—known as *chapters*—that are not permanently resident in core (see below). Similar use is made of space that has been allocated to a program but not yet used. In each case the space occupied is held on sufferance and must be yielded up if it is required by an object program. The responsibility for freeing space that has been temporarily occupied in this way, and is now required by an object program, lies with the space routines of the supervisor. They free space either by moving information from it to some other part of the memory that may be free, or by authorising its overwriting.

Object programs once loaded are never shifted to another part of core, a procedure that might be adopted in a computer provided with a genuine additive base register and efficient hardware instructions for moving material. Again, the supervisor in its present form has no provision for swapping (there being no drum) and it is for this reason that fully conversational working is expensive on the Atlas 2.

The core memory is divided into two parts. The *fixed area* of about 15K holds permanently resident code and provides some working space. The *space routine domain* is available for object programs, unused space between them being used in the manner described above for bufferage and for chapters of the supervisor that are not permanently resident. It is found convenient in a 64K configuration to allocate $8\frac{1}{2}$K of this space specially for pure procedures and to leave $\frac{1}{2}$K permanently available to provide some elbow room for the supervisor in times of storage congestion. These figures are different in a 128K configuration. (All store figures refer to words of 48 bits.)

There are about 32K words, including code associated with SERs and system routines, and tables, in the supervisor altogether, all but about 15K being non-resident. The system initialises itself with all the supervisor in core; as the core fills up with object programs, more and more of the supervisor becomes non-resident. It is found that operation becomes very inefficient if there is only room in core for two or three of the normally non-resident chapters of the supervisor, since furious and almost cyclic retrieving then tends to occur.

A distinction was made above between storage space being allocated to a program and actually being used by that program. This distinction is built into the implementation. The *master space scheduler* allocates space to an object program when that program asks for it. This, however, is a pure book-keeping transaction, and no corresponding adjustment is made to the storage lockout settings. In consequence, when the object program tries to access the space that it has been given, a memory fault occurs. It is as a result of this fault that the space routines, which are responsible for disposing of information temporarily in occupation and giving the space to the object program, are called into action. Experience has shown that this method of scheduling enables extremely economical use to be made of the available core space.

Space routines

The largest unit in which space other than that actually containing user programs is handled is the *block* of 512 words. It is in terms of (consecutive) blocks that space is allocated to object programs. A supervisor chapter also occupies one block. *Blocklets* consist of 64 words; they are typically chained together to provide buffering space, and are made by splitting up blocks.

At any time, the space routine domain is made up of:

(1) Space physically containing object programs (that is, not merely allocated to object programs by the master space scheduler).
(2) Space containing supervisor chapters.
(3) Space containing blocklets. In addition:
(4) There may be space that is unoccupied. There are two chains of unoccupied space, the *gap chain*, containing free blocks, and the *free blocklet chain*, containing free blocklets.

A request made by an SER to the space routines may be for the delivery of any one of the following:

(1) A particular block required by an object program (for example, for growing within the allocation made by the master space scheduler).
(2) Any block to receive a chapter of the supervisor.
(3) A blocklet.

Similarly, a request can be made for space to be taken back. The drasticness, or otherwise, of the action taken to free space with which to satisfy the demand is determined by a priority system, as described below. Such action can be:

For (1) above:

If the block is on the gap chain, deliver it.
If the block contains a chapter, shift or overwrite the chapter.
If the block contains blocklets, not on the free chain, shift them, or authorise their overwriting provided a copy exists on the disc.

For (2) above:

If there is a block on the gap chain, deliver it.
Overwrite a chapter.
Try shifting blocklets to free a block.
If necessary, overwrite some blocklets.

For (3) above:

If there is a blocklet on the free blocklet chain, deliver it.
Authorise the overwriting of a blocklet of which a copy exists on the disc.
If necessary, cut up a block to make more blocklets.

If none of these things are possible, or desirable, the space routine either halts the requesting SER or returns with a FAIL indication; which of these two things happens depends on the manner in which the space routine has been entered.

Associated with the space routines is a garbage collection routine that combines blocklets into complete blocks when necessary.

Requesting and keeping priorities

Three distinct priority numbers are associated with chapters of the supervisor that are not permanently resident. These are:
Requesting priority, that applies when space is being requested for a chapter; *live keeping priority*, which is associated with a chapter currently resident in core and actively in use by one or more SERs; *dead keeping priority*, associated with a currently resident chapter that is not in use, i.e. all the SERs using its code are dormant. It is by comparing the requesting priority of a chapter whose loading has been requested by an SER with the appropriate keeping priority of chapters already in core that the space routines determine—when the gap chain is empty—whether a given chapter is to be overwritten or not. The dead keeping priority is fairly high for the chapter containing the line printer translation table, for example, and low for the chapter containing the 1 MINUTE SER that is, as its name implies, activated only once a minute. The priority numbers adopted for use were arrived at by accumulating statistics of actual operation. Loading of requested chapters is performed by an SER known as the RETRIEVER, which as explained earlier, has lower priority than any other SER on the free queue.

Blocklets also have requesting and keeping priorities, but not dead keeping priorities. The keeping priority can be made effectively infinite, in which case the blocklet remains in core until the priority is altered or the blocklet is relinquished; such a blocklet is said to be *tied*. This is done, for example, with blocklets containing data from a paper tape reader that are waiting to go to the disc.

PQ bases

PQ bases (the origin of this strange name is not worth explaining) are short sequences of consecutive words used for recording information needed by the supervisor. They are constructed out of tied blocklets; an individual PQ base lies wholly within a blocklet, but the same blocklet, can contain several PQ bases. Thus while a PQ base is typically 6 to 8 words long, it can go up to 60 words. The issuing and return of PQ bases is handled by a sub-system that makes use of the space routines for the purpose of obtaining blocklets. The addressing system used to implement PQ bases limits the amount of space that may be used for this purpose to 64 blocklets; this has proved an unfortunate limitation and shortage of PQ bases sometimes leads to system delay.

It is worth-while giving one or two examples of the use of PQ bases. Each input stream has a PQ base associated with it; this contains a pointer to the current blocklet being used as a buffer by that input stream and to the current place in that blocklet. It also contains the value of the current partially unpacked word and whether the stream is exhausted or a not. Similarly, a PQ base associated with a document in the well (see below) contains the name of the document, a statement of how long it is, the absolute address on the disc at which it starts, the current record on the disc being accessed, and whether the record is complete or not. PQ bases are associated with output streams, and with magnetic tapes; they are also used to contain job titles and operator output or input messages in course of being processed.

The idling loop

It will be remembered that, in addition to the regular chains (halt and free) through the bases, the coordinator maintains a separate chain through the object program bases. Whenever the coordinator can find no SER free to run, it goes down this chain and starts the first OP that is not halted (or, more strictly, is not trying to run an SER that is halted). If no such OP can be found, the coordinator queues and immediately enters the IDLING SER.

Actions performed by the IDLING SER are as follows:

(1) The running of a simple arithmetic test routine for the accumulator.
(2) Testing of a marker that is set when an SER is queued in consequence of an interrupt. If this happens, exit from the IDLING SER takes place and, as is the case after exit from all SERs, control goes to the coordinator. The chain of free SERs will then be scanned, and the SER that has just been queued will be entered.
(3) Keeping an account of the number of idling cycles. Since normally the idling loop is interrupted at least once a second by the timed interrupt calling for the 1 SECOND SER, if this count exceeds a certain quantity it is an indication that the clock has stopped.

Object program selection

An object program once started will run (1) to completion, (2) until it is halted (in an SER), or (3) until an interrupt takes place. The interrupt may be a simple one involving no call to an SER and soon over. It may, however, call an SER and, since this call takes place during the running of an object program, the SER is executed immediately; it may, during its running, free or queue other SERs. Like all SERs when it terminates it sends control to the coordinator which, after running all SERs that are now free, will again turn its attention to the chain of OPs that are ready to run. Since the activity that has taken place could very well have the effect of freeing an object program higher on the queue than the one that was interrupted, a change of object program may, therefore, occur at this point; if so, the processor status for the displaced object program is dumped into the set of blocklets provided for this purpose and chained to the OP base.

As an efficiency aid a flag is set when an object program is freed as the result of an SER running, and the search of the object program chain takes place only if this flag has been set.

1 SECOND SER

At rather more than one second intervals(1·28 seconds to be precise), a clock interrupt takes place and the associated interrupt routine queues the 1 SECOND SER. The principal functions of this SER, which has been mentioned several times, will be summarised; they are:

(1) To scan the peripherals to see if one has recently become engaged, and if so to queue the appropriate SER.
(2) To give an alarm if an unreasonable time has gone by without an expected interrupt having come from a peripheral.
(3) To scan the chain of object programs and find the one which has had the most processor time since the last scan (this figure is recorded in the OP base). The object program in question is removed to the end of the chain.
(4) To look for messages prepared for typing to the operators, and queue an SER if there are any.
(5) At proper intervals, to queue the 1 MINUTE SER.

Among the functions of the 1 MINUTE SER are:

(1) To subtract a constant from the priority rating of all jobs on the abridged job list (see below).
(2) To remind operators of missing documents and to abandon jobs if necessary.
(3) To put out the time on the operator's typewriter.
(4) To initiate the logging of information.

Some longer interval timing is handled (conveniently if not always logically) by the file organisation program, which for reasons of its own is equipped with a mechanism for initiating standard operations that are required to be performed periodically such as incremental dumping.

The well

The well is, in effect, a short-term filing system that enables material to be stored away on the disc in the form of a sequence of blocklets chained together. The well routines also administer transfer of this material to and from core. The well was originally designed to hold documents in batches, and the terminology reflects this intention; however, it was in practice found more convenient to work with single documents and the words 'batch' and 'document' are to be taken as synonymous.

A batch (or document) is described by a *batch record* which is accommodated in six words in a PQ base. In the case of a stream associated with an object program, the batch record consists of (1) the batch number, which is a combination of the cipher of the job (see below) and stream number, together with a digit indicating whether the stream referred to is an input or an output stream, (2) type and size of stream, and (3) the absolute address on the disc of the record in which the first block of the batch is stored.

The unit of allocation of space on the disc is the block, but the well routine is capable of writing in smaller units down to one blocklet, although to do so is relatively inefficient. The well routines attempt to accumulate material in core until there is a block full, but, if buffer space gets short, they write on the disc the blocklets that have been accumulated. Blocklets used for buffering are obtained from the space routines as required and returned when they are finished with. Object programs can create output at a great rate, faster in fact than the disc can absorb it; it is therefore arranged that an object program is halted if more than 12 blocklets of a particular stream have accumulated in the well buffer awaiting disc writing.

The well reading routines are designed so that they will work under conditions of extreme shortage of core space. When a document from the well is to be read into core, the well routines try to procure up to four groups of 8 blocklets each from the space routines. The exact number requested in a particular case depends on the rate of consumption if this can be estimated. They ask for the groups of blocklets successively with decreasing priority, entering the space routines in such a way that they are not halted if space is not available. They then read enough material to fill the blocklets that have been obtained. When all but a few of the blocklets have been used by the requesting program, the reading procedure is repeated. It may be, however, that material is read from the well and not immediately consumed by the user; it may indeed never be consumed. Blocklets in use for well bufferage are, therefore, regarded as potentially available to the space routines if space becomes short, and are allotted decreasing keeping priorities corresponding to the decreasing requesting priorities with which they were obtained. In order to keep the core space situation as fluid as possible, the well routine gives transfers out of core priority over transfers into core.

The well routines can be used for dealing with magnetic tape, this, however, being a facility provided for users and not for the system. They will also perform such functions as turning an output stream into an input stream, which they do by making the appropriate change in the batch record. Note that the stream numbers are in the batch records, and that there is no central list of stream numbers.

It may be useful to describe the sequence of events that take place when an input stream is created and connected to a file. The object program executes an extracode in which the file title and stream number are parameters. The SER called by this extracode requests access to the file from the file master; if all is in order, the file master delivers the disc address and marks the file as open for reading. The SER then makes up the batch record and puts it in the pool of batch records. When the stream is no longer required, the file master is requested to close the file. In order that this should be possible, the batch record contains a marker to show that it refers to a stream connected to a file. Otherwise, the batch record is identical to one for a stream that came from a peripheral.

Scheduling and administration of object programs

The handling of objects programs has two sides, which are here distinguished as administration and scheduling. The former consists of operations which have to be done by some means, but which, apart from the orders obeyed in carrying them out, do not influence the overall efficiency, or reflect a managerial policy. Under this head come such things as checking that all peripheral input is complete, ensuring that magnetic tapes are mounted, arranging that all peripheral output is satisfactorily completed, ensuring that the correct code translations are performed and that input from permanent files is available. Scheduling on the other hand involves managerial decisions, such as a decision to favour short small jobs compared with big long ones.

We now follow the course of a job from the point at which the paper tapes are inserted in the reader. It should be clear which operations are scheduling and which are administrative .

When a tape reader is engaged, this fact is detected by the 1 SECOND SER. This in turn queues an SER that starts the tape reader, gets the right translation table from the disc, and demands a blocklet from the space routines. Successive rows of tape are read until a new line is encountered. The SER then sends control to the NEWLINE subroutine that detects whether the document in the tape reader is a job description, a titled document, or an erroneously punched tape.

If the document is a job description, the NEWLINE subroutine allocates a *job cipher;* if one is not available, reading is halted. A cipher is a number from a pool, re-used when available. It is used to identify the job throughout its life in the machine. At the present time, 100 ciphers are used in a rotating sequence; 12 are reserved for jobs created by the file master, and the rest are used for regular jobs subject to some detailed restrictions on job type. The job description is checked for error and packed into a convenient form. Two lists are prepared; one, in the fixed area, contains the titles of all documents needed by the job and the other, in a PQ base, contains the names of magnetic tapes. When all documents (as checked against the former list) have been read the job is said to have *arrived.*

Abridged job list

This important list, held in a region of consecutive memory, contains single word entries indexed by the cipher. Each word contains information about the current status of the job to which it refers. Immediately after a job description has been read, the word indexed by a particular cipher contains a count of the input documents needed by the corresponding job; when all the documents have come in, the word contains a marker to the effect.

The entries in the abridged job list are chained in four queues as follows:

queue 0	regular off-line jobs
queue 1	jobs created from a console by a RUNJOB command or created by off-line jobs
queue 2	magnetic tape jobs (necessarily off-line at present)
queue 3	console jobs, and jobs created by the file master.

The operator can, by typewriter message, assign a particular incoming job to any queue.

The allocation of a base

The first stage of scheduling is the allocation of a base to the object program. Each job waiting for a base is given a *rating number* which is arrived at as follows. First a number is computed from the time estimate, store size, and other parameters of the job. This number is multiplied by 10 for jobs on queue 0, by 6 for jobs on queue 1, and by 3 for jobs on queue 2; it is left unaltered for jobs on queue 3. Every minute, the rating for each job not yet started is decreased by a constant, this being one of the functions of the 1 MINUTE SER. Under all ordinary circumstances, the priority ratings obtained in the above way have the effect of giving high priority to jobs on queue 3. Individual jobs may be given high priority by operator message.

A job which requires magnetic tapes to be mounted is not given a base until its tapes are ready. The issuing of directions to the operators for mounting tapes is a function of the *magnetic tape scheduler*. This routine has little scope for optimising, since jobs using the same tapes must be run in the order in which they are put into the machine in case the programmer is using the tapes for communication between one job and following jobs. The tape scheduler does, however, avoid issuing futile requests such as asking for a tape to be removed from one deck and immediately mounted on another.

A job, which is complete as far as peripheral input and (where applicable) magnetic tapes are concerned, is said to be *ready*. The first requirement for a job to be able to proceed is that it should possess an OP base. Whenever a base becomes available, it is assigned by the job scheduler to the job with the lowest rating number, subject to rules which are designed to balance the loads of various types of job. A selection is first made of the type of job to be started, and then the job of that type with the lowest rating number is chosen. The rules at present in use are as follows:

1. Reject from consideration off-line non-magnetic-tape jobs if more than 3 such jobs are running.
2. Reject from consideration off-line, magnetic-tape jobs if more than 6 off-line jobs of any sort are running.

Then, subject to the job class not being rejected:

3. If there are less than 5 console jobs running, start one if there is one.
4. If there are less than 3 off-line non-tape jobs running, start one if there is one available.
5. If there are less than 4 tape or off-line jobs running, start one if there is one available.
6. Otherwise, start the job with the least rating number, if there is one whose class is not ruled out by (1) and (2).

These rules are not claimed to be optimal, but they seem to have had some success in load-balancing. They were designed to take account of the following observations:

1. Unless there are enough off-line jobs multi-programming with each other and with console jobs, there will be idle time because of them all getting held up at the same time on account of tape and disc delays.
2. If there are too many off-line jobs multi-programming with each other, overheads increase, and on-line response is degraded.
3. Tape units are a scarce resource and once tapes for a particular job have been mounted in is sensible to favour that job.
4. It is very desirable to start console jobs without delay, but one must keep enough off-line work going to avoid idle time.

At the time of writing there are commonly 20 jobs of all sorts multi-programming at a time.

The base assigned to an object program stays with it throughout its existence until the job finally goes through the *end program* sequence. Assigning a base is a distinct matter from giving the job core space to use, since the amount of core space required may vary widely during the process of the work. From the point of view of space scheduling, a job goes through a series of *phases*. The characteristic of a phase change is that the master space scheduler makes a new allocation of core space. For example, a job might successively go through an editing, a compiling, and a loading phase; the core allocation for each of these phases would be quite independent of the others. Note that there is, in consequence, no compulsion on the system to run one phase of a job immediately the previous one has finished; the job in its new phase can go into the queue for space. Thus, if one job is running for a long time in a phase that needs a large amount of core space, so that there is, for example, only 8K left, a large number of jobs may go through a compiling phase using the Autocode compiler which needs only $6\frac{1}{2}$K, although none of the compiled programs can be loaded until the large job has run to completion.

When a job has acquired a base, and its entry in the abridged job list has been marked accordingly, the job scheduler queues the (free standing) START OP SER on its behalf. In due course, this SER will pick up information from the job description, set up the input and output streams (opening such files as are necessary) and queue the PHASE CHANGE routine on the OP base belonging to the job in question.

The PHASE CHANGE routine, as its name implies, is activated, not only when a job is initiated, but whenever it changes from one phase to another. The phase change routine first returns to the space routines any core space that the job may possess. It then requests space for the new phase from the master space scheduler; the object program is placed on a *space wait* chain, and halted until the space is allocated. When space becomes available, the master space scheduler runs down the space wait chain, and selects the first job that is waiting for space and will fit into the space available. Space is allocated to object programs in multiples of 4K words. Each object program on the space wait chain has associated with it a number called the *skip count*, which is reduced by 1 each time the job is passed over by the master space scheduler when it is seeking a job to go in the available space. The initial value of the skip count depends on the priority of the job, and on the amount of space needed. In order that a large job shall not be indefinitely delayed, a job whose skip count has become zero is not passed over; instead, the master space scheduler is halted. In due course, enough space will necessarily become free to fit in the waiting job, and normal operation will be resumed.

It is now possible to summarise the basis on which the allocation of central processor time between the various object programs in core is done. It has been explained that the coordinator maintains a chain through the OP bases, and is so constructed as always to run the highest non-halted OP on the chain. It was also explained that one of the functions of the ONE SECOND SER is to demote to the end of the chain the object program which has received most central processor time during the second just expired. The joint application of these two rules constitutes nearly all the multiprogramming scheduling that is found to be needed. It ensures that console jobs, which are frequently in input or output wait, are near the head of the queue, and that tape-limited or disc-limited jobs are favoured in a similar way. If all the jobs in hand are processor limited, then the result is a round robin, and no special priority is given to console jobs; this does not happen often enough to be worth worrying about.

There is one departure—in favour of the file organisation program, or *file master*—from the strict application of the above rules that is worth mentioning. The file master, which it is outside the scope of this paper to describe, runs as an object program in its own right, with its own base and cipher. Since the file master can, by its nature, never run for a very long period, and since many other programs may be waiting for access to files, it is found desirable to keep the file master always at the head of the queue regardless of anything that may happen in exceptional cases.

Re-starting after system failure

One blocklet per cipher is reserved on the disc for the purpose of holding restart information. When the status of a job changes (from the point of view of restart) this blocklet must be updated. The information held in the restart blocklet is sufficient to enable all jobs that had not been run, or had not finished running, and all output from jobs that had run to completion to be recovered.

In order to make a restart possible, certain information is preserved during running, when it would otherwise be abandoned; for example, an input stream that originally came from a peripheral, and has been thrown away by a program, is, in fact, kept intact on the disc until the job is completed.

General comments

The reader may wonder why it is necessary to treat SERs and object programs differently in the matter of scheduling, and why they should run on a different control. The reasons, of which there are a number, all derive from the fact that much more knowledge is available about SERs than about object programs. It is known that SERs will do their job quickly, or at any rate will come to a halt quickly. Accordingly, there is no danger of making nonsense of object program scheduling if they are allowed to take precedence. Again, it can be assumed that SERs are fully debugged (!), and there is, therefore, no need to protect the system by setting the memory protection registers before they are called in. Whenever an object program is changed, the status of the processor, including contents of all index registers (of which there are many in the Atlas 2), must be saved; in the case of an SER, these operations can be curtailed or omitted, since the actions to be performed by the SER are known in advance; on the Atlas 2, the saving in supervisor overheads made in these ways is substantial.

It will be seen that the reasons just quoted for treating SERs and object programs differently are partly fundamental and partly arise from constraints imposed by the hardware. It is likely that, when computers designed specially for time sharing are available, the latter will be far less serious. The fundamental reasons will, however, still carry sufficient weight for a distinction between SERs and object programs to be highly desirable in the interests of efficiency. We may remark in this context that the Cambridge machine spends some 13% of its time in the supervisor, less than 1% in the idling loop and thus over 85% obeying user-level programs. This measurement relates to a period when 16 remote terminals were being supported at a time, together with an off-line throughput of 800–1000 jobs per day.

A supervisor based on an organisation very similar to that described in this paper is used on the I.C.T. Atlas 2 at AWRE Aldermaston, the original development work having been done by a joint I.C.T. and Cambridge University group. The basic organisation of SERs, chapters, blocklets, etc., and the coordinator, are only trivially different if at all. From the point of view of the user, however, the two operating systems are totally different. The AWRE machine runs without multiprogramming of object programs, with a well on tapes instead of on a disc, and has no provision for multiple access. This is the software configuration desirable for the kind of work customary at that installation.

Acknowledgements

The work reported here is the work of a team, which in its first period included a number of people from I.C.T. Ltd., principally C. R. Spooner, T. P. Wansbrough, P. M. Brenan, and R. A. Jennings. From the Cambridge side in addition to the authors, D. W. Barron was involved in the early planning and M. J. T. Guy, P. R. Radford and J. C. Viner in the implementation. The authors are most grateful to M. V. Wilkes for his encouragement in preparing the paper and his help in drafting it. Residual incomprehensibility is likely to be due to not taking his advice.

Appendix

The way in which the structure is used is illustrated by the following list of SER bases, with brief notes on the purpose of the associated SERs.

SER base	*Notes*
Object program	There are 39 of these which are described in the text.
Input peripheral	There are 2 per peripheral, one for starting and one for running. 12 bases in all.
Output peripheral	There are 3 per peripheral, for starting, running, and exception conditions. 24 in all.
Magnetic channels	These administer the autonomous channels. There are 2.
Consoles	These form a pool, and administer transfers to and from the on-line consoles. 29 bases.
Timed routines	The 1-second and 1-minute routines described in the text.
Well	Four bases serve the SERs working the Well.
Idling	This SER is described in the text.

The remaining bases serve SERs with more particular purposes:

Blocklet reserve	Maintains a reserve of blocklets to enable certain other SERs to avoid halting.
Operator communication	Two bases: the SERs handle operator communication via the console typewriter or ordinary peripherals.
Tape scheduler	Allocates magnetic tapes to drives, checks their identity, etc.
Date	Handles change of date and time and messages about them.
Job decoder	Decodes Job Descriptions when a new job is put in at the peripherals.
Job arrived	Does certain administrative tasks when peripheral input of a job is complete.
Job scheduler	Decides which job to start next.
Start OP	Sets up the object program base for a job and starts it.
OP dump	Removes object programs from core to disc in case of dire space congestion.
Kill job	Removes a job from the system, whatever its status.
PDP7	Works the link to the PDP7. Logically it is not unlike a channel SER.
Map dump	Copies the disc allocation maps from core disc when necessary.
Status messages	Responds to operator queries about system status.
Output peripheral messages	Handles messages about output peripherals, e.g. to reload printer.