R. L. ASHENHURST, Editor

Time Sharing on a Computer with a Small Memory

R. O. Fisher* and C. D. Shepard
University of Illinois,† *Urbana, Illinois*

Techniques to make time sharing attractive on a computer with a small central memory are presented. "Small" is taken to mean that only one user program plus a monitor will fit into the memory at any time. The techniques depend on having two levels of secondary storage: level 1, several times larger than the main memory and quite fast; and level 2, many times larger and slower than level 1.

Introduction

It has been suggested by John McCarthy that at least one million words of directly addressable core memory are necessary for effective time-shared computer use [1]. In fact, the major effort in the time-share world today is directed toward systems using extremely large hardware configurations. However, many installations are interested in providing time-sharing facilities, but have relatively small hardware configurations. We feel that a useful, time-shared system can be implemented in these cases. For example, one might set up a small time-share system with a slow response time which would still be an immense improvement over a batch job turn-around time of 2–4 hours. Also, by changing the emphasis from time sharing as a utility to specialized time sharing, one can have a feasible time-shared system on a small core computer [2]. In addition, interactive processors can be provided such that each processor handles many consoles each time it comes into core, thereby minimizing the number of swaps required to service these consoles. With several users in these specialized, easily serviced modes of interaction, one can make available a few general purpose slots without causing a serious degradation of the over-all system response. This paper presents some of the difficulties inherent in such a system together with techniques which can be used to overcome these difficulties.

* Present address: Texas Instruments, Dallas, Texas

† Department of Computer Science. This work was supported in part by Atomic Energy Commission Contract No. AT(11-1)-1469.

Preliminary Discussion

This paper was written on the basis of experience gained in developing a time-share system on the University of Illinois' ILLIAC II. Some of the details peculiar to this implementation are found in the Appendix. In order to make this paper of general interest, we make the hardware configuration shown in Figure 1 the basis of our discussion. It will, however, be clear during much of the discussion that we have a particular system in mind.

It is essential that the hardware for a time-shared system have the following minimal set of properties:

(1) A secondary storage large enough to hold a reasonable amount of information permanently for each potential user of the system.

(2) Interrupts, particularly timer, memory violation, protected order, and illegal order interrupts.

(3) A multiplexer for consoles.

The following are additional assumptions about our configuration:

(1) The core is not large enough in general to hold more than the monitor and one user program.

(2) There are two levels of secondary storage: level 1 which is relatively small but fast (drum); and level 2 which is large but slow (disk).

(3) Transfers between core, drum, and disk are by block (i.e., a fixed number of words).

Some of the difficulties inherent in a small core time-

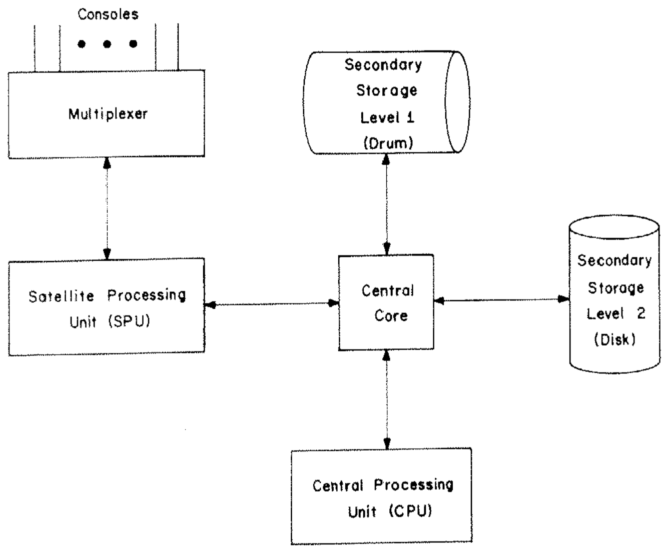
Consoles
Multiplexer
Secondary Storage Level 1 (Drum)
Satellite Processing Unit (SPU)
Central Core
Secondary Storage Level 2 (Disk)
Central Processing Unit (CPU)

Fig. 1. Hardware configuration

shared system are as follows:

(1) Since only one user program can be resident in core, swap times are unusually important.

(2) The monitor must be kept as small as possible. Thus the options available to the user through the monitor will have to be a carefully chosen subset of what might be provided.

(3) When a user program generates an I/O request, there is no other user program which might perform useful computation while the request is being serviced.

(4) There is a lack of core for buffers in which to queue I/O requests.

(5) Effective utilization of the drum is difficult.

(6) Only a few lines from consoles can be kept in the monitor.

The techniques used to overcome these difficulties are described in the remainder of the paper. Each of the following aspects of the software design will be related to the difficulties for which it provides partial solutions:

(A) Method of file organization.

(B) Secondary storage access optimization and minimization.

(C) Console communication.

(D) Processors for interactive languages.

Method of File Organization

File organization for time-shared systems has been described at length elsewhere [3, 4]. Our problem is to define a feasible file organization for a small core computer.

A limited amount of the disk is set aside for user and system scratch and for saving core loads. The rest of the disk is dedicated to a three-level file-by-name system (Figure 2) consisting of the Master ID Table (level 1), User File Dictionaries (level 2), and Named Files (level 3). (These levels are not to be confused with the two levels of secondary storage.) The Master ID Table contains all legal ID's (IDentification Numbers) and a pointer to a User File Dictionary for each ID. The User File Dictionary contains a dictionary entry for each named file belonging to or referenced by that user. A named file belongs to exactly one User File Dictionary, but may be referenced by many User File Dictionaries. Each named file consists of several ordinary files (see below).

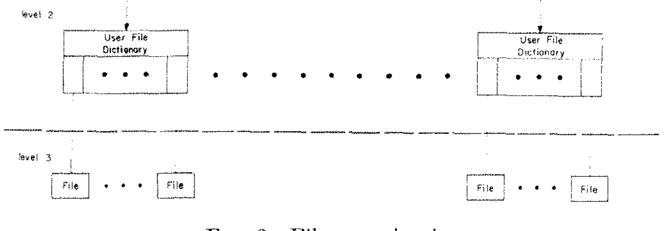

FIG. 2. File organization

Each ordinary file is a collection of linked disk tracks. Each track contains the following system information: the addresses of the previous track, this track and the next track. The first and last tracks of a file contain an end-of-file flag and a pointer to the User File Dictionary to which this file belongs. A file can contain images of BCD cards, binary cards or offline printer lines. For convenience, all three are referred to as lines. The format of lines is fixed because of a lack of core in which to handle arbitrary formats. Each line is prefixed by the following system information: line number, line type, length of this line, and length of previous line. The first and last lines in a track contain an end-of-record flag. With this symmetric linking, files appear to be bidirectional tapes without the disadvantages of tapes. For example, if the first and last addresses of active files are kept in core, "rewind" time becomes negligible. Also, writing into input files is very easily handled. Since one cannot depend on users to maintain the physical and logical linking, and since virtually every program requires I/O, it seems reasonable to place the file-by-name program in the monitor. By having a User File Dictionary (and associated ID in the Master ID Table) with a Dictionary Entry for each User File Dictionary, the system can manipulate the User File Dictionaries as ordinary files. Thus all references to the file-by-name area, either by system or user, can be required to go through the above monitor program.

The files making up a named file are:

Source—The source file contains the main body of the named file, i.e., program or data. If the source file is data then *binary* and *listing* (below) are irrelevant. Each line in a source file is numbered.

Table of Contents—For source files longer than four tracks, a subfile is kept containing the following information for each track: track address, largest line number in this track, and number of free words in this track. This information is used to facilitate merges into a file and selective listing from a file.

Changes—When changes to a named file are typed in, they are entered into this file. Certain commands (e.g., RUN) cause the changes to be sorted and then merged into the source file. This together with the table of contents allows an extremely efficient algorithm for updating a file (e.g., only those tracks of the source file which will be changed need to be read in).

Assembly Listing—The assembler or compiler listing for each program is automatically maintained for user convenience.

Binary Image—If a program has not been changed since the last assembly or compilation the binary image is available.

The dictionary entry for a named file contains the first track address, the last track address and the number of tracks for each of the files above. It also tells whether binary, listing, and/or changes files exist for this named file.

This organization of named files allows the system to use previously generated binary images of subroutines, thereby eliminating all unnecessary translations. It also facilitates running programs which consist of many subroutines. When a run is given at a console, the system checks to see which (if any) of the files requested have been changed. Those files that have been changed are then updated. The system next translates these subroutines and obtains a relocatable binary file, which is then available for all future runs.

Finally, the names of all files requested are passed to the loader, a library search is performed, and a core image formed. The library search (which includes a search of the user's file dictionary) is necessary because the programs requested in the run command may contain references to other programs not explicitly mentioned in the run command. This implicit referencing saves the user from having to list all subroutine names required for the run. We feel that this organization allows as much flexibility and convenience as might be expected on a small core computer.

Secondary Storage Access Optimization and Minimization

Initially it was thought that the frequently needed systems programs and tables could be allocated semipermanent drum space. The rest of the drum would then be used for buffering and holding core loads. However, the sheer bulk of such systems programs along with the fluctuating load of a time-shared system made this infeasible (e.g., when console usage is heavy different systems programs are needed than when background is being processed). Also, with very few drum buffers available, I/O would have to go directly to disk, causing costly I/O waits. The above problems are essentially solved by the following proposal made by S. J. Nuspl.

All data movement to and from disk and drum is handled by a program in the monitor. Specific references to the drum are disallowed, and thus this program can buffer I/O to and from the disk through the drum. Calls are available to it which allow system programs to make efficient use of the buffering capabilities. There are two general areas: minimization of disk use and optimization of disk accesses.

Minimization

Whenever any program reads or writes a block on the disk, it can specify that the block will be needed again, and the monitor will attempt to keep that block on the drum. More generally, a program can issue an informative call saying that it will soon be needing a particular block from the disk. If the block is not already on the drum, the monitor attempts to find room on the drum. Each block from disk which finds its way to the drum has a status parameter associated with it. This status has one of four values, whose meanings are given in Table I. In addition, there is another bit for each drum block called the WRITE bit. It is turned on when a block of core has been buffered onto the drum in preparation for writing on the disk. Any drum block which has the WRITE bit turned on is protected until the DISK WRITE has been given or until a CANCEL WRITE is given. CANCEL WRITE may be given when a program finishes using temporary disk storage which has found its way to the drum. The temporary storage may be read (with a WILL NEED) one or more times before the CANCEL WRITE is given. If the temporary storage is still on the drum when the WRITE is cancelled, it is then not necessary to move the blocks to the disk.

TABLE I. Drum Status Parameter

Status	*Meaning*
0	Drum block is free, and may be used by any block needing a space on the drum. However, if the block presently occupying the space is requested before the space is reassigned, no disk access is necessary.
1	Drum block holds information which will be needed again. The more recent the "will need" command, the higher the priority.
2	Drum block being put to special high priority use—not available for any other use. This status is available only to selected system programs.
3	Drum block locked out and unused in normal running. Recovery and engineering test information stored here.

A stack-down list of all blocks on the drum is kept, and references to data already on the drum (with a WILL NEED) cause the entry to be moved to the top of the stack so that it will be the last to leave the drum. That is, data that is most frequently referenced will tend to remain on the drum. Thus when a disk call is made, the drum table is first searched for the disk address; if it is not found then a direct READ FROM DISK is given.

After a compilation, for example, all other jobs for that compiler should be given a high priority since it is now on the drum.

Blocks with status equal to 2 do not go into the stack-down list. These are blocks which may not be referenced often enough to stay on the drum with status 1, but which must be available immediately when needed (real-time display, etc.).

Optimization

Also incorporated in the monitor is a tight algorithm which provides for minimal head movement while moving data to and from the disk. Specifically, most output from core is first written on the drum and an entry containing the disk destination of the output is made in a table. The position of the disk arms is then taken into consideration in making the choice of which drum block should next be moved to disk. More specifically, the priority of the waiting transfers is dynamically re-assigned so that the transfer requiring the least time for moving the heads will be chosen next. The monitor maintains a queue of pending transfers, consisting of:

(1) READS: disk → drum
(2) WRITES: drum → disk

When the drum is not heavily used, READS are given priority in order to bring the drum usage up. In normal use, READS and WRITES have equal priority. In heavy use, when the drum is full, or nearly so, WRITES are given priority, in order to try to make room on the drum.

The scheme outlined above for secondary storage control is one of the key factors in making time sharing feasible on a small core computer.

With these specifications available, the scheduler is able to take advantage of the knowledge that a certain job is pending execution and use the WILL NEED option to overlap the core load to the drum during execution of the current core load. Similarly, when a swap is undertaken the active core load is moved to the drum whence it is gradually moved back to the disk (if necessary) while the next core load is in execution. To fully appreciate the savings over sending core loads straight to the disk one must realize that a full core swap to disk takes about 10 times more time than to drum. Thus we have gone a long way toward alleviating the first difficulty listed in the Introduction.

The scheme also allows the possibility of foreseeing input requests by the user program and bringing the required block from disk to drum ahead of time. Further, output requests are handled very rapidly by being buffered onto the drum for later transfer to the disk.

Using the drum as a buffer helps alleviate the lack of core available for buffering.

Finally, the scheme provides automatic dynamic allocation of the drum, and so provides very effective utilization of the drum.

Console Communication

Since communication with the Satellite Processing Unit (SPU) is line by line there is the problem of what to do with lines in a block-oriented computer.

One could bring in the command processor each time a console line is received; however, this would raise the system overhead to an intolerable level if many consoles were active. Further, this is unnecessary if the line is data (as opposed to a command) since it can simply be packed away and looked at later (e.g., typing lines into a file). One is immediately led to the conclusion that the SPU should flag lines as being data or command, the latter informing the monitor that the command processor will be needed. One could then pack the lines into a buffer and bring in the command processor when the buffer becomes full or when a command is received. However, this does not solve the problem of sending lines to the SPU. Bringing in the command processor each time a line (or even a few lines) is sent raises the system overhead again. Further, such tasks as listing programs do not require a high level processor. A solution to these problems seems to be achieved by having one block (which will remain on the drum in status 2) assigned to each active console. When a line is received, the corresponding drum block is read in and sent back containing the line. If the line received was a command or if the drum block is nearly full, a flag is set indicating that the command processor is needed. Likewise, when outputting lines, the appropriate drum block is read in and a line removed each time the SPU sends a request for a line.

Other advantages become apparent in situations such as the following: An interactive core load associated with the console goes into execution and generates output for that console. These output lines are buffered onto the drum until the drum block becomes full or until the user's time slot is finished and a swap takes place. In either case the monitor starts sending lines from the drum block as soon as they are available. In the ideal situation the user has filled up the drum block before being swapped and the console can be kept busy printing for three or four minutes before that core load is needed again and before disk accesses associated with this console are required, once again reducing the number of swaps required. Similarly, when a console user wishes a program to be listed an entire block is moved from disk to drum and the monitor takes over to provide automatic listing of the entire drum block. For paper tape input, lines are buffered onto the drum until the block becomes full and then the entire block is moved to the disk. (One disk access for each 30–50 lines of input.)

Processors for Interactive Languages

Swap time is probably the most important factor in the building of a time-shared system with a reasonable response time. In fact, most of the techniques so far described help to reduce swap time. Our design of processors for interactive languages is intended to reduce it further.

Each interactive language processor is an integral part of the software system. Each one has facilities available to it which are not available to the user in general. (1) Each processor can communicate with many consoles. In fact, the processor itself determines the consoles with which it interacts. (2) Each processor is allowed to request additional time slots so that each console with a pending request can be serviced once.

Each processor is non-self-modifying, hence only the data pertaining to the consoles need be saved. The interactive processors can be designed with a time limit for each console so that all consoles are serviced within the time allotted to the interactive processor. Thus, users requiring a small amount of compute time can expect fast response time. It has been our experience that many users have found one such processor [7] very useful.

Conclusion

While several aspects of time-shared computer use have been discussed, many others have not. Nonetheless, the authors believe the suggestions made in this paper will prove useful to other groups who might be thinking of implementing a time-share system on a small memory computer. In particular, we believe that we have made several promising steps in the direction of overcoming the problems associated with a small memory in a time-shared environment.

Acknowledgments. The authors acknowledge the benefit of criticisms of early versions of the paper by Professors C. W. Gear, M. Paul, and B. Squires. They would also like to thank Mr. J. D. Madden for encouraging them. Finally, they would like to thank the members of the staff of the Department of Computer Science at the University of Illinois, including L. Greninger, S. J. Nuspl, F. K. Richardson, and A. Otis, all of whom made many useful suggestions.

RECEIVED AUGUST, 1966

REFERENCES

1. McCARTHY, JOHN. Time-sharing computer systems. *In* Martin Greenberger (ED.), *Computers and the World of the Future,* M.I.T. Press, Cambridge, Mass., 1962, p. 232.
2. LICHTENBERGER, R. WAYNE, AND PIRTLE, MELVIN W. A facility for experimentation in man-machine interaction. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1, pp. 589–598.
3. CRISMAN, P. A. (ED.) *The Compatible Time-Sharing System: A Programmer's Guide.* M.I.T. Press, Cambridge, Mass., 1965.
4. DALEY, R. C., AND NEUMANN, P. G. A general-purpose file system for secondary storage. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1, pp. 213–230.
5. BREARLEY, H. C. ILLIAC II, a short description and annotated bibliography. *IEEE Trans. EC-14*, 3 (June, 1965), 399–403.
6. GEAR, C. W. Optimization of the address field computation in the ILLIAC II assembler. *Comput. J. 6* (1964), 332–335.
7. GEAR, C. W. Tipsy, a fast response interactive processor. Programming Memo. No. 50, Dept. of Computer Science, U. of Illinois, Urbana, Ill.

APPENDIX

We first give a few statistics. The ILLIAC II [5] has an 8K core of 52-bit words, the drum holds 8 core loads, and the disk holds about 150 drum loads. The core cycle time is 1.75μsec, the core is four times faster than the drum, and the drum is about 15 times faster than the disk. Maximum access time for the drum is 16msec and for the disk 214 msec. A block is defined to be 256 words. A hardware restriction is that only block transfers to drum are possible.

The SPU used is the PDP-7 with the 630 multiplexer. It has a 8K core of 18-bit words, its core cycle time is 1.75 μsec, and it can service up to 64 consoles.

Planning on the system began in the spring of 1965. A pilot system, which has been running 13 hours a day since February 1966, required between 1 and 2 man years to implement. It services four consoles, which are connected directly to ILLIAC II instead of coming through the PDP-7. Other than that, the hardware configuration for the pilot system is the same as that shown in Figure 1. Because console lines come directly into ILLIAC II core in the pilot system, a system processor is swapped in each time a line arrives. Handling the line may mean still another swap to get the core image for the console from which the line came. This core image is likely on the disk, so that two or three seconds may have gone by before the line is processed by the correct core image and control returned to the core image which was present when the line came in. This procedure, while relatively straightforward, results in lengthy waits at the consoles because of excessive swapping. While this is bearable when only four consoles are attached to the system, it would not be feasible if many more were attached directly to ILLIAC II.

The pilot system includes processors for three languages: Tipsy, FORTRAN II, and NICAP (New Illinois Compiler and Assembler Program) [6]. The FORTRAN and NICAP processors are ordinary implementations of a compiler and a machine language assembler, respectively. The Tipsy processor is interactive, procedure-oriented and designed to handle several consoles at once. It is an interactive processor as discussed in the main body of the paper.

The pilot system includes as background, the batch processor which previously occupied most of the time of ILLIAC II. In fact, runs from consoles in the pilot system are made via this batch processor.

Finally, the command language for the pilot system includes commands for the following functions: 1. Creation and modification of named files; 2. simple file searching; and 3. running of files through one or another of the system processors.

The design and implementation of the pilot system is due primarily to the efforts of F. K. Richardson and C. W. Gear.