Reprinted in Computer Structures, ed. Bell and Newell, McGraw-Hill, 1971, pp 291-300

A User Machine in a Time-Sharing System

B. W. LAMPSON, W. W. LICHTENBERGER, MEMBER, IEEE, AND M. W. PIRTLE

Abstract—This paper describes the design of the computer seen by a machine-language programmer in a time-sharing system developed at the University of California at Berkeley. Some of the instructions in this machine are executed by the hardware, and some are implemented by software. The user, however, thinks of them all as part of his machine, a machine having extensive and unusual capabilities, many of which might be part of the hardware of a (considerably more expensive) computer.

Among the important features of the machine are the arithmetic and string manipulation instructions, the very general memory allocation and configuration mechanism, and the multiple processes which can be created by the program. Facilities are provided for communication among these processes and for the control of exceptional conditions.

The input-output system is capable of handling all of the peripheral equipment in a uniform and convenient manner through files having symbolic names. Programs can access files belonging to a number of people, but each person can protect his own files from unauthorized access by others.

Some mention is made at various points of the techniques of implementation, but the main emphasis is on the appearance of the user's machine.

INTRODUCTION

A CHARACTERISTIC of a time-sharing system is that the computer seen by the user programming in machine language differs from that on which the system is implemented [1], [2], [6], [10], [11]. In fact, the *user machine* is defined by the combination of the time-sharing hardware running in user mode and the software which controls input-output, deals with illegal actions which may be taken by a user's program, and provides various other services. If the hardware is arranged in such a way that calls on the system have the same form as the hardware instructions of the machine [7], then the distinction becomes irrelevant to the user; he simply programs a machine with an unusual and powerful instruction set which relieves him of many of the problems of conventional machine-language programming [8], [9].

In a time-sharing system which has been developed by and for the use of members of Project Genie at the University of California at Berkeley [7], the user machine has a number of interesting characteristics. The computer in this system is an SDS 930, a 24 bit, fixed-point machine with one index register, multi-level indirect addressing, a 14 bit address field, and 32 thousand words of 1.75 μs memory in two independent modules. Figure 1 shows the basic configuration of equipment. The memory is interleaved between the two modules so that processing and drum transfers may occur simultaneously. A detailed description of the various hardware modifications of the computer and their implications for the performance of the overall system has been given in a previous paper [7].

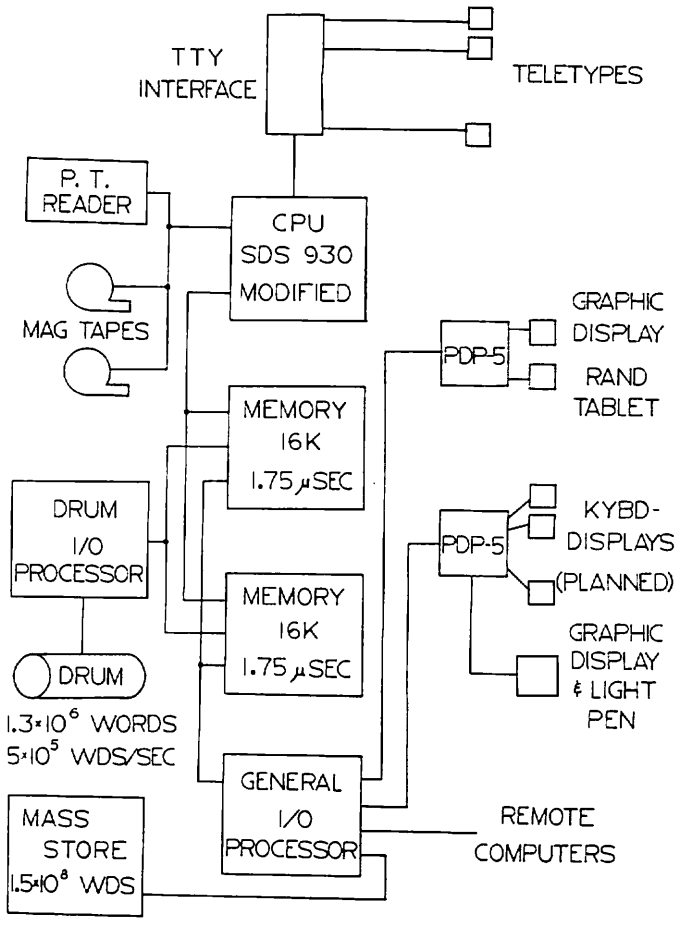


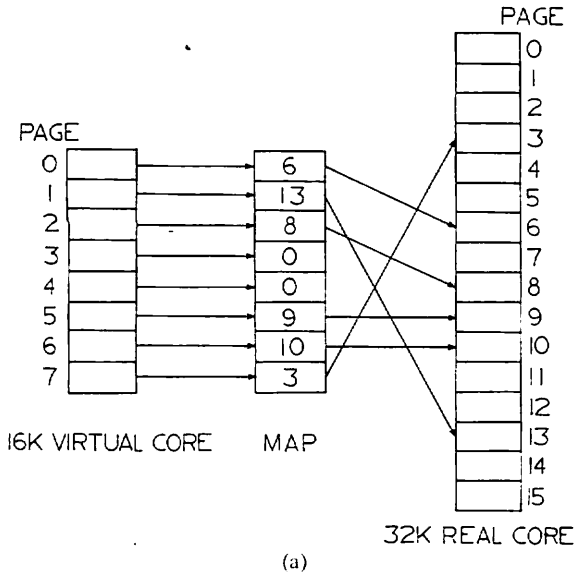
Fig. 1. Configuration of equipment.

Briefly, these modifications include the addition of monitor and user modes in which, for user mode, the execution of a class of instructions is prevented and replaced by a trap to a system routine. The protection from unauthorized access to memory has been subsumed in an address mapping scheme: both the 16 384 words addressable by a user program (logical addresses) and the 32 768 words of actual core memory (physical addresses) have been divided into 2048-word *pages*. A set of eight six-bit hardware registers defines a *map* from the logical address space to the real memory by specifying the real page which is to correspond to each of the user's logical pages. Implicit in this scheme is the capability of marking each of the user's pages as unassigned or read-only, so that any attempt to access such a page improperly will result in a trap.

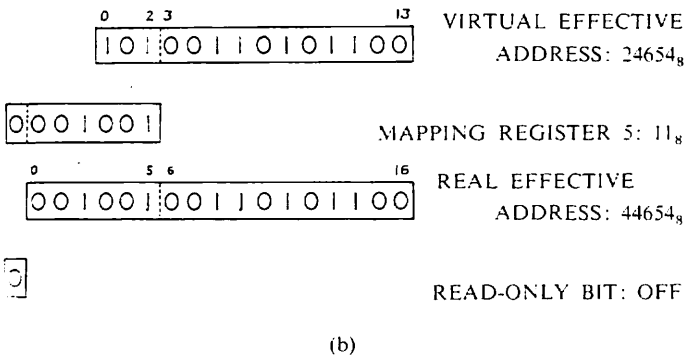
All memory references in user mode are mapped. In monitor mode, all memory references are normally ab-

Manuscript received July 12, 1966; revised August 29, 1966. The work for this paper was supported in part by the Advanced Research Projects Agency, Department of Defense, Contract SD-185.

The authors are with the University of California, Berkeley, Calif.



(a)



(b)

Fig. 2. The hardware memory map. (a) Relation between virtual and real memory for a typical map. (b) Construction of a real memory address.

solute. It is possible, however, with any instruction in monitor mode, or even within a chain of indirect addressing, to specify use of the user map. Furthermore, in monitor mode the top 4096 words are mapped through two additional registers called the monitor map. The mapping process is illustrated in Fig. 2.

Another significant hardware modification is the mechanism for going between modes. Once the machine is in user mode, it can get to monitor mode under three circumstances:

1) if a hardware interrupt occurs,
2) if a trap is generated by the user program as outlined, and,
3) if an instruction with a particular configuration of two bits is executed. Such an instruction is called a system programmed operator (SYSPOP).

In case 3), the six-bit operation field is used to select one of 64 locations in absolute core. The current address of the instruction is put into absolute location zero as a subroutine link, the indirect address bit of this link word is set, and another bit is set, marking the memory location in the link word as having come from user-mapped memory. The system routine thus invoked may take a parameter from the word addressed by the SYSPOP, since its address field is not interpreted by the hardware. The routine will address the parameter indirectly through location zero and, because of the bit marking the contents of location zero as having come from user mode, the user map will be applied to the remainder of the address indirection. All calls on the system which are not inadvertent are made in this way.

A monitor mode program gets into user mode by transferring to an address with mapping specified. This means, among other things, that a SYSPOP can return to the user program simply by branching indirect through location zero.

As the above discussion has perhaps indicated, the mode-changing arrangements are very clean and permit rapid and natural transfers of control between user and system programs. Advantage has been taken of this fact to create a rather grandiose machine for the user. Its features are the subject of this paper.

Basic Features of the Machine

A user in the Berkeley time-sharing system, working at what he thinks of as the hardware language level, has at his disposal a machine with a configuration and capability which can be conveniently controlled by the execution of machine instruction sequences. Its simplest configuration is very similar to that of a standard medium-sized computer. In this configuration, the machine possesses the standard 930 complement of arithmetic and logic instructions and, in addition, a set of software interpreted monitor and executive instructions. The latter instructions, which will be discussed more fully in the following, do rather complex input-output of many different kinds, perform many frequently used table lookup and string processing functions, implement floating point operations, and provide for the creation of more complex machine configurations. Some examples of the instructions available are:

1) Load A, B, or X (index) registers from memory or store any of the registers. Indexing and indirect addressing are available on these and almost all other instructions. Double word load and store are also available.
2) The normal complement of fixed-point arithmetic and logic operations.
3) Skips on various arithmetic and logic conditions.
4) Floating point arithmetic and input-output. The latter is in free format or in the equivalent of Fortran E or F format.
5) Input a character from a teletype or write a block of arbitrary length on a drum file.
6) Look up a string in a hash-coded table and obtain its position in the table.
7) Create a new process and start it running concurrently with the present one at a specified point.
8) Redefine the memory of the machine to include a portion of that which is also being used by another program.

It should be emphasized that, although many of these instructions are software interpreted, their format is identical to the standard machine instruction format, with the exception of the one bit which specifies a system interpreted instruction. Since the system interpretation of these instructions is completely invisible to the machine user, and since these instructions do have the standard machine instruction format, the user and his program make no distinction between hardware and software interpreted instructions.

Some of the possible 192 operation codes are not legal in the user machine. Included in this category are those hardware instructions which would halt the machine or interfere with the input-output if allowed to execute, and those software interpreted instructions which attempt to do things which are forbidden to the program. Attempted execution of one of these instructions will result in an *illegal instruction* violation. The effect of an illegal instruction violation is described later.

Memory Configuration

The memory size and organization of the machine is specified by an appropriate sequence of instructions. For example, the user may specify a machine which has 6K of memory with addresses from 0 to 13777_8; alternatively, he may specify that the 6K should include addresses 0 to 3777_8, 14000_8 to 17777_8, and 34000_8 to 37777_8. The user may also specify the size and configuration of the machine's secondary storage and, to a considerable extent, the structure of its input-output system. A full discussion of this capability will be deferred to a later section.

The next few paragraphs discuss the mechanism by which the user's program may specify its memory size and organization. This mechanism, known as the *process map* to distinguish it from the hardware memory address mapping, uses a (software) mapping register consisting of eight 6-bit bytes, one byte for each of the eight 2K blocks addressable by the 14 bit address field of an instruction. Each of these bytes either is 0 or addresses one of the 63 words in a table called the private memory table (PMT). Each user has his own private memory table. An entry in this table provides information about a particular 2K block of memory. The block may be either *local* to the user or it may be *shared*. If the bock is local, the entry gives information about whether it is currently in core or on the drum. This information is important to the system but need not concern the user. If the block is shared, its PMT entry points to an entry in another table called the shared memory table (SMT). Entries in this table describe blocks of memory which are shared by several users. Such blocks may contain invariant programs and constants, in which case they will be marked as *read-only*, or they may contain arbitrary data which is being processed by programs belonging to two different users.

A possible arrangement of logical or virtual memory for a process is shown in Fig. 3. The nature of each page has been noted in the picture of the virtual memory; this information can also be obtained by taking the corresponding byte of the map and looking at the PMT entry specified by that byte. The figure shows a large amount of shared memory, which

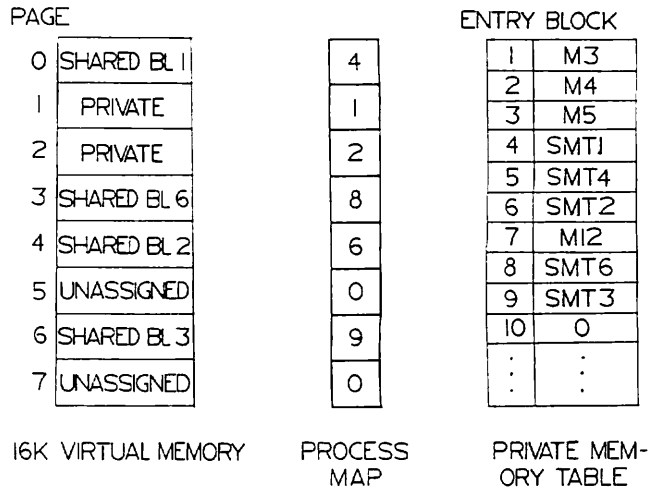


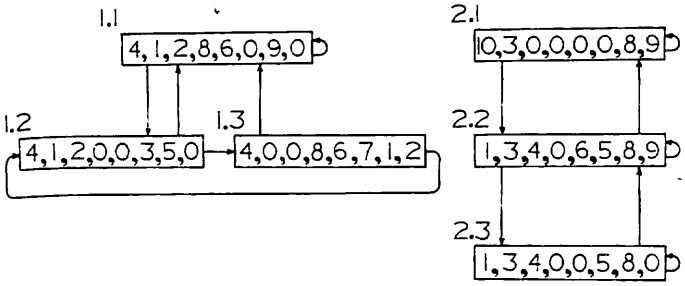
Fig. 3. Layout of virtual memory for a typical process.

suggests that the process might be a compilation, sharing the code for the compiler with other processes translating programs written in the same source language. Virtual pages one and two might hold tables and temporary storage which are unique to each separate compilation. Note that, although the flexibility of the map allows any block of code or data to appear anywhere in the virtual memory, it is certainly not true that a program can run regardless of which pages it is in. In particular, if it contains references to itself, such as branch instructions, then it must run in the same virtual pages into which it was loaded.

Two instructions are provided which permit the user to read and modify his process map. The ability to read the process mapping registers permits the user to obtain the current memory assignment, and the ability to write the registers permits him to reassign memory in any way which suits his fancy. The system naturally checks each new map as it is established to ensure that the process is not attempting to obtain unauthorized access to memory which does not belong to it.

When the user's process is initiated, it is assigned only enough memory to contain the program data as initially loaded. For instance, if the program and constants occupy 3000_8 words, two blocks, say blocks 0 and 1, will be assigned. At this point, the first two bytes of the process mapping register will be nonzero; the others will be zero. When the program runs, it may address memory outside of the first 4K. If it does, and if the user has specified a machine size larger than 4K, a new block of memory will be assigned to him which makes the formerly illegal reference legal. In this way, the user's process may obtain more memory. In fact, it may easily obtain more than 16K of memory simply by addressing 16K, reading and preserving the process mapping register, setting it with some of the bytes cleared to zero, and grabbing some more memory. Of course, only 16K can be addressed at one time; this is a limitation imposed by the address field of the machine.

There is an instruction which allows a process to specify the maximum amount of memory which it is allowed to have. If it attempts to obtain more than this amount, a *memory violation* will occur. A memory violation can also



PMT 1	PMT 2	SMT
1 M3	1 SMT1	1 M1
2 M4	2 SMT5	2 M16
3 M5	3 M7	3 M2
4 SMT1	4 M8	4 M10
5 SMT4	5 M9	5 M11
6 SMT2	6 SMT2	6 M6
7 M12	7 M13	.
8 SMT6	8 SMT3	.
9 SMT3	9 M14	.
10 0	10 M15	
.	.	
.	.	
.	.	

Fig. 4. Process and memory configuration for two users. (The processes are numbered for each user and are represented by their process mapping registers. Memory blocks are identified by drum addresses, which are written M1, M2 . . .)

be caused by attempts to transfer into or indirect through unassigned memory, or to store into read-only memory. The effect of this violation is similar to the effect of an illegal instruction violation and will be discussed.

The facilities just described are entirely sufficient for programs which need to reorganize the machine's memory solely for internal purposes. In many cases, however, the program wishes to obtain access to memory blocks which have been created by the system or by other programs. For example, there may be a package of mathematical and utility routines in the system which the program would like to use. To accommodate this requirement, there is an instruction which establishes a relationship between a name and a certain process mapping function. This instruction moves the PMT entries for the blocks addressed by the specified process mapping function into the shared memory table so that they are generally accessible to all users. Once this correspondence has been established, there is another instruction which allows a different user to deliver the name and obtain in return the associated process map. This instruction will, if necessary, make new entries in the second user's PMT. Various subsystems and programs of general interest have names permanently asigned to them by the system.

The user machine thus makes it possible for a number of processes belonging to independent users to run with memory which is an arbitrary combination of blocks local to each individual process, blocks shared between several processes, and blocks permanently available in the system. A complex configuration is sketched in Fig. 4. Process 1.1 was shown in more detail in Fig. 3. Each box represents a process, and the numbers within represent the eight map bytes. The arrows between processes show the process hierarchy, which is discussed in the next section. Note that the PMT's belong to the users, not to the processes.

From the above discussion, it is apparent that the user can manipulate the machine memory configuration to perform simple memory overlays, to change data bases, or to perform other more complex tasks requiring memory reconfiguration. For example, the use of common routines is greatly facilitated, since it is necessary only to adjust the process map so that 1) memory references internal and external to the common routine are correct, and 2) the memory area in which the routine resides is read-only. In the simplest case, in which the common routine and the data base fit into 16K of memory, the map is initially established and remains static throughout the execution of the routine. In other cases where the routine and data base do not fit into 16K, or where several common routines are concurrently employed, it may be necessary to make frequent adjustment to the map during execution.

Multiple Processes

An important feature of the user machine allows the user program, which in the current context will be referred to as the controlling process, to establish one or more subsidiary processes. With a few minor exceptions, to be discussed, each subsidiary process has the same status as the controlling process. Thus, it may in turn establish a subsidiary process. It is therefore apparent that the user machine is in fact a multi-processing machine. The original suggestion which gave rise to this capability was made by Conway [3]; more recently the Multics system has included a multi-process capability [4], [5], [13].

A *process* is the logical environment for the execution of a program, as contrasted to the physical environment, which is a hardware *processor*. It is defined by the information which is required for the program to run; this information is called the *state vector*. To create a new process, a given process executes an instruction which has arguments specifying the state vector of the new process. This state vector includes the program counter, the central registers, and the process map. The new process may have a memory configuration which is the same as, or completely different from, that of the originating process. The only constraint placed on this memory specification is that the total memory available to the multi-process system is limited to 128K by the process mapping mechanism, which is common to all processes. Each user, of course, has his own 128K.

This facility was put into the system so that the system could control the user processes. It is also of direct value, however, for many user processes. The most obvious examples are input-output buffering routines, which can operate independently of the user's main program, communicating with it through memory and with interrupts (see the following). Whether the operation being buffered is large volume output to a disc or teletype requests for information about the progress of a running program, the degree of flexibility afforded by multiple processes far exceeds anything which could have been built into the input-output system. Furthermore, the overhead is very low: an additional process requires about 15 words of core, and

process switching takes about 1 ms under favorable conditions. There are numerous other examples of the value of multiple processes; most, unfortunately, are too complex to be briefly explained.

A process may create a number of subsidiary processes, each of which is independent of the others and equivalent to them from the point of view of the originating process. Figure 4 shows two simple multi-process structures, one for each of two users. Note that each process has associated with it pointers to its controlling process and to one of its subsidiary processes. When a process has two immediate descendants, as in the case of processes 1.2 and 1.3, they are chained together on a ring. Thus, three pointers, up, down, and ring, suffice to define the process structure completely. The up pointers are, of course, redundant, but are convenient for the implementation. The process is identified by a *process number* which is returned by the system when it is created.

A complex structure such as that in Fig. 5 may result from the creation of a number of subsidiary processes. The processes in Fig. 5 have been numbered arbitrarily to allow a clear description of the way in which the pointers are arranged. Note that the user need not be aware of these pointers; they are shown here to clarify the manner in which the multiple process mechanism is implemented.

A process may destroy one of its subsidiary processes by executing the appropriate instruction. For obvious reasons this operation is not legal if the process being destroyed itself has subsidiary processes. It is possible to find out what processes are subsidiary to any given one; this permits a process to destroy an entire tree of sub-processes by reading the tree from the top down and destroying it from the bottom up.

The operations of creating and destroying processes are entirely separate from those of starting and stopping their execution, for which two more operations are provided. A process whose execution has been stopped is said to be *suspended.*

To assure that these various processes can effectively work together on a common task, several means of interprocess communication exist. The first allows the controlling process to obtain the current status of each of its subsidiary processes. This status information, which is read into a table by the execution of the appropriate system instruction, includes the current state vector and operating status. The operating status of any process may be

1) running,
2) dismissed for input-output,
3) terminated for memory violation,
4) terminated for illegal instruction violation, or
5) terminated by the process itself.

A second instruction allows the controlling process to become dormant until one of its subsidiary processes terminates. Termination can occur in the following four ways:

1) because of a memory violation,
2) because of an illegal instruction violation,
3) because of self-termination.

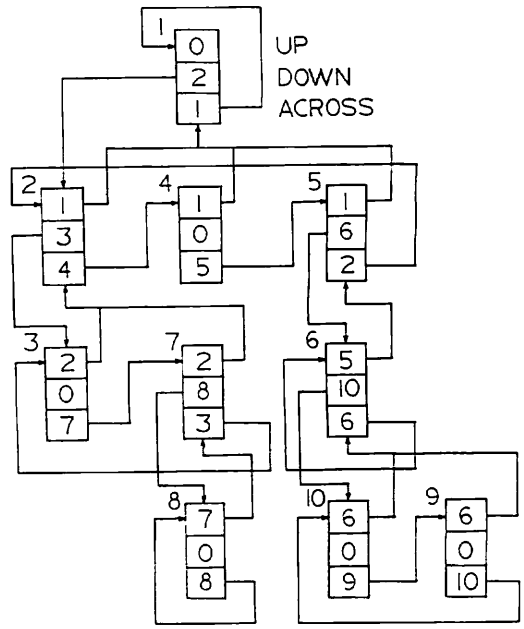


Fig. 5. Hierarchy of processes.

Interactions described previously provide no method by which a process can attract the attention of another process which is pursuing an independent course. This can be done with a program interrupt. Associated with each process is a 20-bit interrupt mask. If a mask bit is set, the process may, under certain conditions (to be described in the following), be interrupted; i.e., a transfer to a fixed address will be simulated. The program will presumably have at this fixed address the location of a subroutine capable of dealing with the interrupt and returning to the interrupted computation afterwards. The mechanism is functionally almost identical to many hardware interrupt systems.

A process may cause an interrupt by delivering the number of the interrupt to the appropriate instruction. The process causing the interrupt continues undisturbed, but the nearest process which is either on the same level as the one causing the interrupt or above it in the hierarchy of processes, and which has the appropriate interrupt armed, will be interrupted. This mechanism provides a very flexible way for processes to interact with each other without wasting any time in the testing of flags or similar frivolous activities.

Interrupts may be caused not only by the explicit action of processes, but also by the occurrence of several special conditions. The occurrence of a memory violation, attempted execution of an illegal instruction, an unusual input-output condition, the termination of a subsidiary process, or the intervention of a user at a console (by pushing a reserved button) all may cause unique interrupts (if they have been previously armed). In this way, a process may be notified conveniently of any unusual conditions associated with other processes, the process itself, or a console user.

The memory assignment algorithm discussed previously is slightly modified in the presence of multiple processes.

When a process is activated, one of three options may be specified:

1) Assign new memory to the process entirely independently of the controlling process.
2) Assign no new memory to the process. Any attempt to obtain new memory will cause a memory violation.
3) If the process attempts to obtain new memory, scan upward through the process hierarchy until the topmost process is reached. If at any time during this scan a process is found for which the address causing the trap is legal, propagate the memory assigned to it down through the hierarchy to the process causing the trap.

Option 3) permits a process to be started with a subset of memory and later to reacquire some of the memory which was not given to it initially. This feature is important because the amount of memory assigned to a process influences the operating efficiency of the system and thus the speed with which it will be able to respond to teletypes or other real-time devices.

The Input-Output System

The user machine has a straightforward but unconventional set of input-output instructions. The primary emphasis in the design of these instructions has been to make all input-output devices interface identically with a program and to provide as much flexibility in this common interface as possible. Two advantages result from this uniformity: it becomes natural to write programs which are essentially independent of the environment in which they operate, and the implementation of the system is greatly simplified. To the user the former point is, of course, the important one.

It has been common, for example, for programs written to be controlled from a teletype to be driven instead from a file on, let us say, the drum. A command exists which permits the recognizer for the system command language and all of the subsystems to be driven in this way. This device is particularly useful for repetitive sequences of program assemblies and for background jobs which are run in the absence of the user. Output which normally goes to the teletype is similarly diverted to user files. Another application of the uniformity of the file system is demonstrated in some of the subsystems, notably the assembler and the various compilers. The subsystem may request the user to specify where he wishes the program listing to be placed. The user may choose anything from paper tape to drum to his own teletype. In the absence of file uniformity each subsystem would require a separate block of code for each possibility. In fact, however, the same input-output instructions are used for all cases.

The input-output instructions communicate with *files*. The system in turn associates files with the various physical devices. Programs, for the most part, do not have to account for the peculiarities of the various actual devices. Since devices differ widely in characteristics and behavior, the flexibility of the operations available on files is clearly critical. They must range from single-character input to the output of thousands of words.

A file is *opened* by giving its name as an argument to the appropriate instruction. Programs thus refer to all files symbolically, leaving the details of physical location and organization to the system. If authorized, a program may refer to files belonging to other users by supplying the name of the other user as well as the file name. The owner of a file determines who is authorized to access it. The reader may compare this file naming mechanism with a more sophisticated one [12], bearing in mind the fact that file names can be of any length and can be manipulated (as strings of characters) by the program.

Access to files is, in general, either sequential or random in nature. Some devices (like a keyboard-display or a card reader) are purely sequential, while others (like a disk) may be either sequentially or randomly accessed. There are accordingly two major I/O interfaces to deal with these different qualities. The interface used in conjunction with a given file depends on whether the file was declared to be a *random* or a *sequential* file. The two major interfaces are each broken down into other interfaces, primarily for reasons of implementation. Although the distinction between sequential and random files is great, the *subinterfaces* are not especially visible to the user.

Sequential Files

The three instructions CIO (character input-output), WIO (word input-output), and BIO (block input-output) are used to communicate with a sequential file. Each instruction takes as an operand a *file number*. This number is given to the program when it opens a file. At the time of opening a file it must be specified whether the file is to be read from or written onto. Whether any given device associated with the file is character-oriented or word-oriented is unimportant: the system takes care of all necessary character-to-word assembly or word-to-character disassembly.

There are actually three separate, full-duplex physical interfaces to devices in the sequential file mechanism. Generally, these interfaces are invisible to programs. They exist, of course, for reasons of system efficiency and also, because of the way in which some devices are used. The interfaces are:

1) character-by-character (basically for low-speed, character-oriented devices used for man-machine interaction),
2) buffered block I/O (for medium-speed I/O applications),
3) block I/O directly from user core (for high-speed situations).

It should be pointed out that there is no particular relation between these interfaces and the three instructions CIO, WIO, and BIO. The interface used in a given situation is a function of the device involved and, sometimes, of the volume of data to be transmitted, not of the instruction.

Any interface may be driven by any instruction.

Of the three subinterfaces under discussion, the last two are straightforward. The character-by-character interface is, however, somewhat different and deserves some elaboration. Devices associated with this interface are generally (but not necessarily) used for man-machine interaction. Consider the case of a person communicating with a program by means of a keyboard-display (or a teletype). He types on the keyboard and the information is transmitted to the computer. The program may wish to make an immediate response on the display screen. It many cases this response will consist of an echo of the same character, so that the user has the feeling of typing directly onto the screen (or onto the teleprinter).

So that input-output can be carried out when the program is not actually in main memory, the character-by-character input interface permits programs a choice of a number of *echo tables;* it further permits programs a choice of grade of service by permitting them to specify whether a given character is an attention (or *break*) character. Thus, for example, the program may specify that each character typed is to be echoed immediately and that all control characters are to result in activation of the program regardless of the number of characters in the input buffer. Alternatively, the program may specify that no characters are echoed and every character is a break character. By changing the specification the program can obtain an appropriate (and varying) grade of service without putting undue load on the system. Figure 6 shows the components of the character-by-character interface; responsibility for its operation is split between the interrupt routine called when the device signals for attention and the routine which processes the user's I/O request.

The advantage of the full-duplex, character-by-character mode of operation is considerable. The character-by-character capability means that the user can interact with his program in the smallest possible unit—the character. Furthermore, the full-duplex capability permits, among other things 1) the program to substitute characters on strings of characters as echoes for those received, 2) the keyboard and display to be used simultaneously (as, for example, permitting a character typed on a keyboard to pre-empt the operation of a process. In the case of typing information in during the output of information, a simple algorithm prevents the random admixture of characters which might otherwise result), and 3) the ready detection of transmission errors.

Instructions are included to enable the state of both input and output buffers to be sensed and perhaps cleared (discarding unwanted output or input). Of course, it is possible for a program to use any number of authorized physical devices; in particular, this includes those devices used as remote consoles. A mechanism is provided to permit output which is directed to a given device to be copied on all other devices which are *output linked* to it (and similarly for input). This is useful when communication among users is desired and in numerous other situations.

The sequential file has a structure somewhat similar to that of an ordinary magtape file. It consists of a sequence of *logical records* of arbitrary length and number. On some

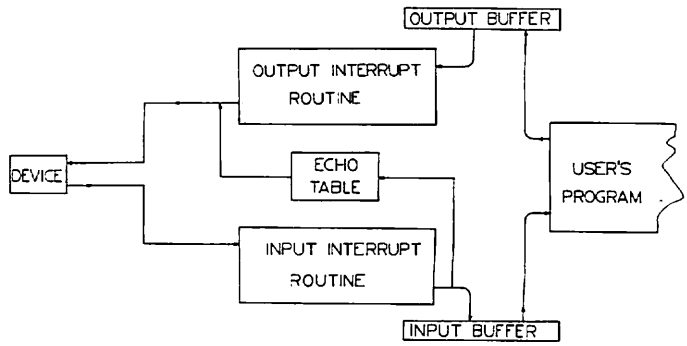


Fig. 6. The character-oriented interface.

devices, such as a card reader or the teletype, a file may have only one logical record. The full generality is available for drum files, which are the ones most commonly used. The logical record is to be contrasted with the variable length physical record of magtape or the fixed length record of a card. Instructions are provided to insert or delete logical records and increase or decrease them in length. Other instructions permit the file to be "positioned" almost instantaneously to a specified logical record. This gives the sequential file greater flexibility than one which is completely unaddressable. This flexibility is only possible, of course, because the file is on a random-access device and the sequential structure is maintained by pointers. The implementation is discussed in the following.

When reading a sequential file, CIO and WIO return certain unusual data configurations when they encounter an end of record or end of file, and BIO terminates transmission on either of the conditions and returns the address of the last word transmitted. In addition, certain flag bits are set by the unusual conditions, and an interrupt may be caused if it has been armed.

The implementation of the sequential file scheme for auxiliary storage is illustrated in Fig. 7. Information is written on the drum in 256-word physical records. The locations of these records are kept track of in 64-word index blocks containing pointers to the data blocks. For the file shown, the first logical record is more than 256 words long but ends in the second 256-word block. The second logical record fits in the third 256-word block and the third logical record—in the 4th data block—is followed by an end of file. If a file requires more than 64 index words, additional index blocks are chained together, both forward and backward. Thus, in order to access information in the file it is necessary only to know the location of the first index block. It may be worthwhile to point out that all users share the same drum. Since the system has complete control over the allocation of space on the drum, there is no possibility of undesired interaction among users.

Available space for new data blocks or index blocks is kept track of by a bit table, illustrated in Fig. 8. In the figure, each column represents one of the 72 physical bands on the drum allocated for the storage of file information. Each row represents one of the 64 256-word sectors around a band. Each bit in the table thus represents one of the 4608 data blocks available. The bits are set when a block is

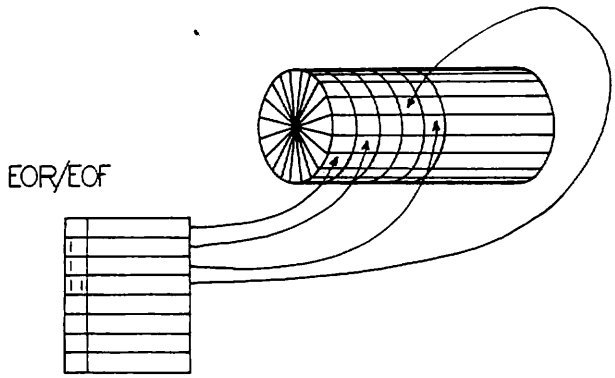


Fig. 7. Index blocks and pointers to data blocks.

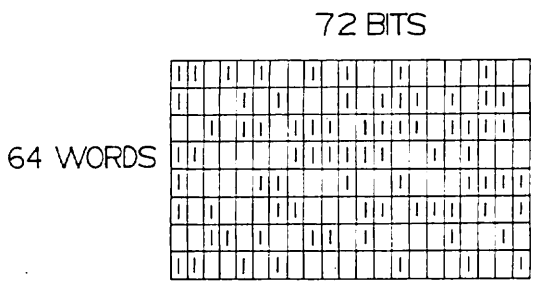


Fig. 8. Bit table for allocation of space on the drum.

Main Memory	Secondary Memory
LDA* ADDR	LAS ADDR
STA* ADDR	SAS ADDR

(a)

Address	Instruction
600	LAS 1450
1450	16345
16345	1234567

Effect: 1234567 → A

(b)

Fig. 9. Load and store form main and secondary memory. (a) Instructions. (b) Addressing.

in use and cleared when the block becomes available. Thus, if a new data block is required, the system has only to read the physical position of the drum, use this position to index in the table, and search a row for the appearance of a 0. The column in which a 0 is found indicates the physical track on which a block is available. Because of the way the row was chosen, this block is immediately accessible. This scheme has two advantages over its alternative, which is to chain unused blocks together:

1) It is easy to find a block in an optimum position, using the algorithm just described.
2) No drum operations are required when a new block is needed or an old one is to be released.

It may be preferable to assign the new block so that it becomes accessible immediately after the block last assigned for the file. This scheme will speed up subsequent reading of the file.

Random Files

Auxiliary storage files can also be treated as extensions of core memory rather than as sequential devices. Such files are called *random files*. A random file differs from a sequential file in that there is no logical record structure to the file and that information is extracted from or written into the random file by addressing a specific word or block of words. It may be opened like a sequential file; the only difference is that it need not be specified as an output or an input file.

Four instructions are used to input and output words and blocks of words on a random file. To permit the random file to look even more like core memory, an instruction enables one of the currently open random files to be specified as the *secondary memory* file. Two instructions, LAS (load A from secondary memory) and SAS (store A in secondary memory), act like ordinary load and store instructions with one level of indirect addressing (see Fig. 9) except, of course, that the data are in a random file instead of in core memory.

Random files are implemented like sequential files except that end of record indicators are not meaningful. Although as many index blocks are used up as required by the size of a random file, only those data blocks which actually contain information will be attached to a random file. As new locations are accessed, new data blocks are attached.

Subroutine Files

Whereas it makes little sense to associate, say, a card reader with a random file, a sequential file can be associated with any physical device in the system. In addition, a sequential file may be associated with a subroutine. Such a file is called a *subroutine file*, and the subroutine may thus be thought of as a "nonphysical" device. The subroutine file is defined by the address of a subroutine together with information indicating whether it is an input or an output file and whether it is word or character oriented. An input operation from a subroutine file causes the subroutine to be called. When it returns, the contents of the A register is taken to be the input requested. Correspondingly, an output operation causes the subroutine to be called with the word or character being output in A. The subroutine is completely unrestricted in the kinds of processing it can do. It may do further input or output and any amount of computation. It may even call itself if it preserves the old return address.

Recall that for sequential files the system transforms all information supplied by the user to the format required by the particular file; hence, the requirement that the user, in opening a subroutine file, must specify whether the file is to be character or word oriented. The system will thereafter do all the necessary packing and unpacking.

Subroutine files are the logical end-product of a desire to decouple a program from its environment. Since they can do arbitrary computations, they can provide buffers of any

desired complexity between the assumptions a program has made about its environment and the true state of things. In fact, they make it logically unnecessary to provide an identical interface for all the input-output devices attached to the system; if uniformity did not exist, it could be simulated with the appropriate subroutine files. Considerations of convenience and efficiency, of course, militate against such an arrangement, but it suggests the power inherent in the subroutine file machinery.

Summary

The user machine described was designed to be a flexible foundation for development and experimentation in man-machine systems. The user has been given the capability to establish configurations of multiple processes; and the processes have the ability to communicate conveniently with each other, with central files, and with peripheral devices. A given user may, of course, wish only to use a subsystem of the general system (e.g., a compiler or a debugging routine) for his particular job. In the course of using the subsystem, however, he may become dissatisfied with it and wish to revise or even rewrite the subsystem. The features of the user machine not only permit this activity but make it easier.

Acknowledgment

The software portion of the system was designed and written in part by L. P. Deutsch, who is entitled to equal credit with the authors for the ideas in this paper. L. Barnes also contributed significantly to the final result.

Bibliography

[1] H. S. Bright, "A Philco multiprocessing system," *1964 Proc. AFIPS Conf.*, vol. 26, Pt. II, pp. 97–141.
[2] W. T. Comfort, "A computing system design for user service," *1965 Proc. AFIPS Conf.*, vol. 27, Pt. I, pp. 619–626.
[3] M. Conway, "A multiprocessor system design," *1963 Proc. AFIPS Conf.*, vol. 24, pp. 139–146.
[4] F. J. Corbato and V. Vyssotsky, "Introduction and overview of the MULTICS system," *1965 Proc. AFIPS Conf.*, vol. 27, Pt. I, pp. 185–196.
[5] J. Dennis and E. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, pp. 143–155, March 1966.
[6] J. Forgie, "A time- and memory-sharing executive program for quick-response on-line applications," *1965 Proc. AFIPS Conf.*, vol. 27, Pt. I, pp. 599–609.
[7] W. Lichtenberger and M. W. Pirtle, "A facility for experimentation in man-machine interaction," *1965 Proc. AFIPS Conf.*, vol. 27, Pt. I, pp. 589–598.
[8] B. W. Lampson, "Interactive machine-language programming," *1965 Proc. AFIPS Conf.*, vol. 27, Pt. I, pp. 473–481.
[9] J. McCarthy, S. Boilen, E. Fredkin, and J. Licklider, "A time-sharing debugging system for a small computer," *1962 Proc. AFIPS Conf.*, vol. 23, pp. 51–57.
[10] J. D. McCullogh, K. Speierman, and F. Zurcher, "Design for a multiple user multiprocessing system," *1965 Proc. AFIPS Conf.*, vol. 27, Pt. I, pp. 611–617.
[11] J. I. Schwartz, "A general-purpose time-sharing system," *1964 Proc. AFIPS Conf.*, vol. 25, pp. 397–411.
[12] R. C. Daley and P. G. Neumann, "A general-purpose file system for secondary storage," *1965 Proc. AFIPS Conf.*, vol. 27, Pt. I, pp. 213–229.
[13] J. H. Saltzer, "Traffic control in a multiplexed computer system," Mass. Inst. Tech., Cambridge, Mass., Tech. Rept. MAC-TR-30, July 1966.

A Time-Sharing System Using an Associative Memory

A. B. LINDQUIST, MEMBER, IEEE, R. R. SEEBER, AND L. W. COMEAU

Abstract—**The hardware scheme designed to implement an experimental IBM System/360, Model 40 time-sharing system will be discussed. The concept of a virtual system will be introduced which allows up to fifteen simultaneous users on the system, each user assuming he has a complete hardware-software of his own. The problem of building and interfacing a 64-word associative memory mapping device into an existing hardware system will be reviewed.**

Manuscript received August 1, 1966.
The authors are with Systems Development Division, IBM Corporation, Poughkeepsie, N. Y.

Introduction

THE advanced technology group at the IBM Poughkeepsie Laboratory has designed and built a hardware configuration for an experimental time-sharing system to be used as a research tool at the IBM Cambridge Scientific Center. Many multiprogrammed or multiple-accessed systems [1]–[10] have been built or proposed; however, there is much controversy on the organization and efficiency of such systems. The goal for this time-sharing system is to gather data on the operational characteristics of