The Design of the Venus Operating System

Barbara H. Liskov
The MITRE Corporation*

The Venus Operating System is an experimental multiprogramming system which supports five or six concurrent users on a small computer. The system was produced to test the effect of machine architecture on complexity of software. The system is defined by a combination of microprograms and software. The microprogram defines a machine with some unusual architectural features; the software exploits these features to define the operating system as simply as possible. In this paper the development of the system is described, with particular emphasis on the principles which guided the design.

Key Words and Phrases: operating systems, system design, levels of abstraction, machine architecture, microprogramming, segments, semaphores, multiprogramming, virtual machines, processes, process communication, virtual devices, data sharing, resource management, deadlock

CR Categories: 4.30, 4.32, 4.41, 6.21

* Bedford, MA 01730. This work was supported by Air Force Contract No. F19(628)-71-C-0002.

Presented at the Third ACM Symposium on Operating Systems Principles, Palo Alto, California, October 18–20, 1971.

Introduction

The Venus Operating System is an experimental multiprogramming system for a small computer. It supports five or six concurrent users, who operate on-line and interactively through teletypes. It may be distinguished from other multiuser systems in that it primarily caters to users who are cooperating with each other, for example, a group of users sharing a data base or building a system composed of cooperating processes.

The operating system was produced to test the following hypothesis: the difficulties encountered in building a system can be greatly reduced if the system is built on a machine with the "correct" architecture. We had in mind a complex system whose data and processing requirements vary dynamically, for example, an operating system or an on-line data management system. We felt that machines available today do not support the programming of such systems very well and that considerable software complexity is introduced to cope with the inadequacies of the hardware.

First, it was necessary to build the machine with the correct architecture. This was done through microprogramming on an Interdata 3 computer; the result is called the Venus machine. The microprogram contained solutions to some of the time-consuming and complex tasks performed by such systems as well as useful mechanisms for building these systems.

Next, it was necessary to use the Venus machine for a software application. An operating system was selected as the initial effort because it was the type of system the machine was intended to support and because it would provide a facility which could support later applications.

System Design Principles

Two main principles were followed in the design of the operating system:

1. The system was built as a hierarchy of levels of abstraction, defined by Dijkstra [1]. We expected this would lead to a better design with greater clarity and fewer errors. A level is defined not only by the abstraction which it supports (for example, virtual memories) but also by the resources which it uses to realize that abstraction. Lower levels (those closer to the machine) are not aware of the resources of higher levels; higher levels may apply the resources of lower levels only by appealing to the functions of the lower level. This reduces the number of interactions among parts of a system and makes them more explicit. Examples of levels of abstraction, which occur in both the microprogram and software, are given throughout the paper.
2. The features of the Venus machine were allowed to influence the operating system design in order to evaluate the effect of the architecture on the development of the software.

Several other principles also guided the design of the system:

3. Efficiency of performance was always considered when making design choices but was not the major criterion.
4. Independent users were protected, insofar as possible, from each other's mistakes by limiting the effects of errors to the user or group of cooperating users involved.
5. Users were given as much access as possible to the features of the machine and the software mechanisms developed for the operating system.

This paper is primarily concerned with the development of the operating system according to the design principles with special emphasis on the two main principles. In the next section the Venus machine features of most influence on the operating system design are described. The following section explains how these features were extended to support the design of the operating system. Then the design of the resource management portion of the system is described. Finally, the design experience is evaluated. A user's view of the system is given in the Appendix; it is intended to clarify the abstract view of the system contained in the body of the paper by showing how the system supports the user in performing his job.

The Venus Machine

Hardware

The Venus machine was built by microprogramming an Interdata 3, a small, slow, and inexpensive computer. The microprogram is stored in a read-only memory; it is limited to 2,000 microinstructions, which imposes fairly severe restrictions on its content.

The Interdata is connected to several teletypes, a card reader, a printer, and two magnetic tapes. In addition, there is a small paging disk with a capacity of half a million bytes of storage. The disk and tapes have direct memory access through hardware selector channels; other devices transfer data a byte at a time.

The Microprogram

In addition to an ordinary instruction set, the Venus microprogram supports a number of nonstandard architectural features [2]. Those features most important to the design of the operating system are briefly described in this paper; they are: segments, multiprogramming of 16 concurrent processes, a microprogrammed multiplexed I/O channel, and procedures.

Segments. Segments are named virtual memories, as defined for Multics [3]. Each segment contains a maximum of 64 thousand bytes of data and has a 15-bit name; segments are the primary storage structure on the Venus machine. Segments and core memory are both divided into 256-byte pages. Information about the contents of each core page is kept in a single, centralized core-resident table, the core page table, which is used by the microprogram to map virtual addresses into real addresses. All references to a given virtual address will be mapped by the microprogram into the same real address, which implies that segments are physically shared among processes. In addition, there is no way for one process to protect a segment from access by other processes. This restriction makes sense only because the Venus machine was designed to support a system composed of *cooperating* processes.

Paging on the Venus machine is performed on demand. If the microprogram cannot locate the desired segment page in the core page table, it starts a software routine, the page fault routine, to fetch the page from the disk. The page fault routine acts like a subroutine of the microprogram, called by the microprogram when needed and returning, via a special instruction, to the microprogram at the point where the page fault was detected.

Multiprogramming. A process is defined to be a program in execution on a virtual machine.[1] The Venus microprogram supports 16 virtual machines, each consisting of an address space and a work area. The address space encompasses all segments and is the same for all processes, although only a few segments are of interest to a particular process. The work area is permanently located in core and contains about 150 bytes of process-related information including the general registers, program counter, and other information about the state of the process.

[1] The reader is referred to Saltzer [4] for a more precise definition of "process." The Venus virtual machine is an instance of Saltzer's "pseudo processor."

Scheduling of the CPU to the processes is performed by the microprogram, which enabled us to define a single uniform mechanism for passing control of the processor among the processes. This mechanism, used even for indicating the end of I/O, is provided by semaphores, first defined by Dijkstra [1]. Semaphores are used to control the sharing of resources and to synchronize processes. Two operations may be performed on semaphores: P and V. P is performed when a process wishes to wait for an event to occur or a resource to become available. A process performs a V to free a resource; either a process or the I/O channel performs a V to signal the occurrence of an event.

Dijkstra defined a semaphore to be an integer variable and an associated waiting list. In Venus the waiting list is represented by a queue, and the association is made explicit by defining a semaphore to be an ordered pair (*count*, *link*). If *count* is negative, its absolute value equals the number of processes on the queue. In this case, *link* points to the work area of the process which most recently performed a P on the semaphore; in the work area of that process is a pointer to the work area of the next newest process on the queue, and so on. Also in the work area of each process is a priority which can be set by software. When the time comes to remove a process from the queue (because a V was performed on the semaphore), the process with the highest priority is removed; if the highest priority is associated with more than one process, the one which has been on the queue the longest is selected.

Process swapping in Venus only occurs as the result of P's and V's being performed by processes or by the I/O channel. All processes in Venus are in one of three states: a single *running* process, processes which are *ready* to run, or processes which are *blocked*. Ready processes are listed on a special queue, the J queue. Blocked processes are on queues associated with semaphores. If the running process performs a P, requesting a resource which is not available or waiting for an event which has not occurred, it becomes blocked; at this point, the highest priority, oldest process on the J queue is selected to be the new running process. If a V is performed, some blocked process may become ready. If this process does not have higher priority than the running process, it is added to the J queue; otherwise, it becomes the running process, and the old running process is added to the J queue. The running process always has a priority at least as high as all ready processes.

Input/Output Channel. The microprogrammed I/O channel relieves the software from all real-time constraints associated with devices by permitting the specification of I/O transfers in increments which suit the requirement of the devices. The microprogram runs the channel between the execution of instructions; from the software point of view, the channel is running simultaneously with the execution of instructions. The channel signals the completion of I/O by performing a V on a special semaphore located in the work area of the process which started the transfer. When the process wishes to synchronize with the I/O, it performs a P on this semaphore.

Fig. 1. Levels of abstraction supported by the Venus microprogram.

ABSTRACTION	RESOURCES	METHOD OF APPEAL
SEGMENTS	CORE PAGE TABLE, DISK	SEGMENT REFERENCE, EL1 INSTRUCTION
VIRTUAL DEVICES	DEVICES, DEVICE STATUS WORD TABLE	SIO INSTRUCTION, CHANNEL COMMANDS
VIRTUAL MACHINES	J QUEUE, PROCESSOR	P AND V INSTRUCTIONS

Procedures. Each procedure is stored in a unique segment. Call and return instructions switch a process from the instructions in one procedure to those in another; they also save and restore part of the work area at the time of the call. Arguments and values can be passed in separate segments which are used as pushdown stacks, referenced by push and pop instructions.

Sharing procedures is desirable on the Venus machine. Of course, only reentrant procedures can be successfully shared. The primary support for reentrant procedures comes from the separate virtual machines. The call, return, push, and pop instructions provide a reentrant procedure interface. In addition, no easy way of storing into procedures is available, and segments provide private and temporary work space whenever this is needed.

Levels of Abstraction

The design principle of levels of abstraction was applied to the microprogram as well as to software. Levels supported by the microprogram and the page fault handler include the virtual memory abstraction, the virtual machine abstraction, and the I/O channel which really provides virtual devices—for example, a card reader which reads an entire card at once. Figure 1 illustrates the resources and methods of appeal associated with these levels.

Extensions to the Venus Machine

The second main design principle was to let the features of the Venus machine influence the design of the operating system. Thus we expected the system to use segments for data and to be composed of a combination of reentrant procedures and asynchronous processes. A feeling for the usefulness of such structures can be gained by considering how they correspond to the way in which the system most conveniently performs its work.

We were interested in designing a system to support multiple users. Each user is assigned a separate virtual machine and is thus represented by an independent process. To support the users, the system must perform certain tasks. Some will most naturally be performed on the user's virtual machine by reentrant system procedures, which use segments to hold user-related data. Other system tasks are logically asynchronous with the user (for example, running I/O devices for the system as a whole). These tasks can be made physically asynchronous by assigning them to separate processes. Thus the work done by the system can be distributed among the processes so that logically concurrent and asynchronous tasks can be performed by physically concurrent and asynchronous processes. This leads to clarity in the design.

To make the features of the Venus machine more convenient to use, several levels of abstraction were defined in software. Two levels are described below: the first, dictionaries, supports the convenient use of segments; the second, queues, supports communication between processes.

Dictionaries

Building a system out of segments containing procedures and data requires cross-referencing between segments. The Venus machine supports references to segments by internal segment name, which is not a very convenient item for a programmer to use or remember. In addition, internal segment names are dynamically assigned (by the page fault handler); the programmer needs a static name. Therefore, external segment names were introduced.

To support external segment names, a mapping between external and internal names is required. This is supplied by *dictionaries*, which are stored in segments. An external segment name is actually a pair of symbolic names: the symbolic name attached to the segment and the symbolic name of the dictionary which should be used to perform the mapping. One special dictionary contains the mappings between the symbolic and internal names of all dictionaries. The two-level external name allows related segments to be grouped together (referenced through the same dictionary) and makes it easier for a user to obtain unique names.

Although dictionaries are primarily intended to support external segment names, they are actually a general mapping facility and are occasionally used as such by the system.

Queues

Processes which synchronize with each other may also need to send and receive information. P's and V's permit processes to wait for or signal the occurrence of events but do not contain any information about what the event is. A process may be waiting for several different events; it must know which event occurred in order to take appropriate action. A mechanism was required which allows one process to send information and the same or another process to receive it. *Queues* were selected for this purpose because they supply a chronological ordering for their elements. All queues are held in a single "queue segment"; the location within this segment of the head of a particular queue is obtained from an associated "queue dictionary."

Resource Management

An operating system must manage the resources available on a machine in such a manner that deadlock is avoided and access to resources is distributed fairly among the users. Resource management in Venus is described here in some detail, partly because this is an interesting part of the operating system and partly as an example of the use of levels of abstraction and distribution of work among processes. Resource management is primarily provided by independent processes synchronizing and communicating with each other.

If several processes compete freely for resources, deadlock may result. The simplest example of deadlock is:

Process 1 owns Resource A and is waiting for Resource B.
Process 2 owns Resource B and is waiting for Resource A.

Neither Process 1 nor Process 2 can continue. We wanted to avoid deadlock in the Venus Operating System, which required careful design of resource management. Resources to be managed by software include segments and input/output devices; core, the paging disk, and the CPU are managed by the microprogram. Virtual machines must also be managed, but only in the sense that no more can be assigned to users than are available.

Management of Shared Data Segments

The management of shared data segments is difficult because general solutions which insure that deadlock will not occur tend to prevent users from running even when the situation is perfectly safe [5, 6, 7]. In our system, users must control the sharing of user-defined data segments. The availability of semaphores provides users with a tool for controlling sharing by means of an algorithm of their own choosing. If the algorithm does not work correctly, only the group of users would be affected.

There still remains the problem of system data structures which are also available to users, for example, the dictionaries. Dictionaries are intended to be shared and may be referenced simultaneously. While a dictionary is being changed, it does not contain consistent data; thus there is a need for mutual exclusion here. Associated with each dictionary is a semaphore which is used to control sharing. Dictionaries may be accessed only by calls to special reentrant dictionary procedures, which perform P's and V's on the associated semaphore where appropriate. Restricting dictionary access to these procedures guarantees that the semaphores will be used correctly (for example, every P will eventually be followed by a V) and that no access errors will occur.

Access to any shared system data structure (queues are another example) is always limited to a group of procedures. This group comprises a level of abstraction which owns the accessed segments as a resource.

Management of I/O Devices

All I/O devices are managed similarly. The system retains ownership of the devices; users are given access for well-defined and limited transactions, which insures that deadlock cannot occur. This section discusses how devices are managed; the next section explains how levels of abstraction were used to accomplish the management.

Teletypes. Our system is on-line and interactive; therefore every user needs a teletype. One solution is to assign each user a teletype which he alone can use. This requires one teletype to be designated the "operator's console" and to be handled differently since the system occasionally needs a teletype to announce special conditions and errors. Furthermore it would not be natural for one user to send a message to another user's teletype or for the system to do so.

The solution chosen defines the teletype on which a user starts running as his "preferred" teletype. He can reference this device symbolically and will ordinarily use it, but he is not constrained to do so and is unable to prevent others from using it. A teletype may be accessed without interruption long enough to write and then read one line. This insures that the standard use of teletypes, which is the user responding to the program (with a command) when told to do so, can occur without interference from other users.

Other Devices. The card reader, the printer, and the two magnetic tapes are all in high demand. Because the disk is small, symbolic data can be maintained in core and disk storage for only a limited time. The card reader and tapes provide the only reasonable method for entering large amounts of symbolic data; only on tape can edited symbolic data be saved and then retrieved later. The printer provides the only reasonable method for listing symbolic data. Users will probably require lengthy access to several of these devices each time they run.

All users benefit when the system retains control, thus insuring that the devices will be run efficiently and kept busy as long as there is work to do. For user convenience, the devices may be accessed without interruption for a fairly long time, for example, long enough to list the assembly of a user's procedure or read the cards in a user's card deck.

Levels of Abstraction

The system performs device management by providing the user with virtual devices which are quite different from real devices. This is accomplished through several levels of abstraction, which are spread over several processes (see Figure 2).

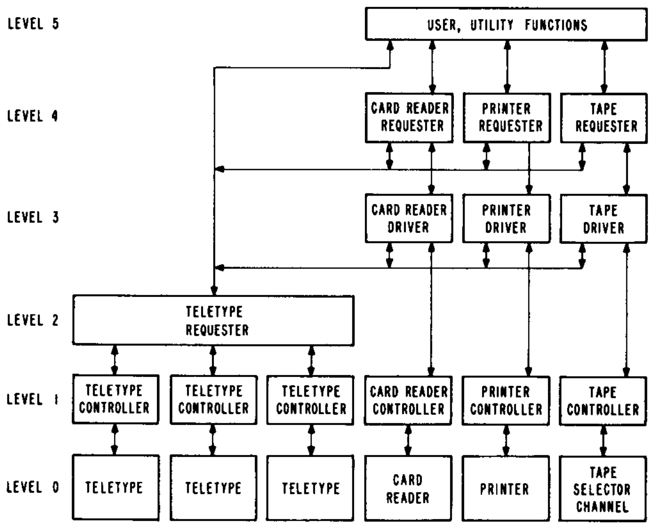
Level 0, which the microprogram supplies, provides devices which have no real-time constraints but which require core buffers. Level 1 is made up of software Controllers, one for each device. Each Controller is an independent process (although one reentrant procedure may serve as several Controllers). The virtual devices supported by the Controllers have the following characteristics:

1. Only a few types of transfers are possible, for example, exactly one card may be read.
2. The buffers for the transfer are located in segments.
3. A transfer on the device may be requested at any time, regardless of whether the device is currently in use. It will be performed when the device is available.
4. The completion of the transfer is signalled by the performance of a V on a semaphore specified by the transfer request.

Above the Controller level, teletypes are handled quite differently from the other devices.

Teletypes. A single reentrant procedure, the Teletype Requester (level 2), handles teletypes. Primarily it provides an interface between the user and the Controllers; it runs on the user's virtual machine.

Fig. 2. Levels of abstraction used in resource management.



Other Devices. The card reader, printer, and tapes are accessed for much larger amounts of data than are accepted by the Controllers. The Drivers (level 3) support this abstraction. Each Driver handles one device and is an independent process. The characteristics supported are:

1. The device is accessed a segment at a time. The Driver breaks the segment into buffers acceptable to the Controller.
2. The Driver defines the type of synchronization required. The Card Reader Driver builds segments containing the images of card decks whenever there are cards in the hopper; user synchronization is required only to obtain the completed segments. The Printer Driver requires notification that a segment should be printed; no synchronization with the completion of printing is required. The Tape Driver requires notification to read or write a segment on a tape; the user must wait for completion.

Above each Driver is a Requester (level 4) which primarily provides an interface between the user and the Driver and runs on the user's virtual machine. These Requesters are more elaborate than the Teletype Requester. For example, the Printer Requester helps the user build printable segments, while the Tape Requester is an interactive procedure which reads and writes segments on tape on user command.

Conclusions

The Venus Operating System has been supporting multiple users for several months. Two to three man-years were spent building the machine; then an estimated six man-years were required to design and implement the system (including utility functions, such as the Assembler, Editor, and debugging aids, as well as operating system functions). We feel this surprisingly brief development time indicates that the architecture of the Venus machine is suitable for supporting the programming of complex software and, in fact, reduces the difficulties encountered in building such software.

In addition, the use of levels of abstraction proved a valuable tool because it provided a way of thinking about the design with clarity and precision. Errors were uncovered while putting the system together, but the rule about ownership of resources was for the most part faithfully followed, minimizing errors resulting from interactions of parts of the system. Such errors as were discovered resulted from errors internal to some piece of the system which was not fully checked out and errors traced to breaking the rule about resources. An added benefit of levels of abstraction is the ease with which the system can be modified; its design is stratified so that the effects of proposed modifications are plainly visible.

Acknowledgments. The author wishes to thank several colleagues at The MITRE Corporation, and especially J.A. Clapp, for many helpful discussions of both the Venus Operating System and this paper, and the referee for his valid and useful criticisms.

Appendix—A User's View of the System

The user views the system in two ways. First, he must write his programs; for this he is interested in available data structures and system procedures. Then he is interested in running and debugging his programs.

Programs running under Venus use segments for all storage—procedures, data, and pushdown stacks. Because segments are available by external name through dictionaries, users can readily share data and procedures. Users may also synchronize with other users through queues and semaphores.

A user runs on-line and interactively. He starts up by typing a command on a teletype and is assigned a virtual machine under the control of an interactive system procedure called the Loader. The Loader recognizes many commands; the most important is the "E" command, which passes control to a specified system, utility, or user procedure. Important system and utility procedures are the Assembler, Editor, Tape Requester, and Debugger.

The user submits card decks to be read prior to his run; he may access all decks through a command to the Loader. The names of the segments containing the card deck images are entered in a dictionary associated with the user. The user may read segments from tape; the names of these segments are also entered into his dictionary where they may be assembled or edited on command. Binding of intersegment references occurs on completion of assembly.

The user executes procedures by giving the "E" command. Execution can occur with or without debugging; no change to the procedures being run is required. Debugging is handled by an interactive system procedure which runs before the execution of every instruction and can be used to stop execution at a specified "breakpoint." A dialogue with the user then commences in which the contents of a segment or his work area may be displayed and modified, a new breakpoint specified, and control returned to the interrupted procedure or the Loader.

The system also provides interactive interrupt procedures which run as the result of exceptional conditions, for example, a stack underflow or overflow. They make use of a subset of the debugging commands, permitting the user to discover the reason for the error, restore his data, and return to the Loader.

When the user has finished running he saves his symbolic data on tape and then informs the Loader, which releases his virtual machine and destroys his symbolic data. His checked-out programs may be entered in the system and become accessible to others through dictionaries.

References

1. Dijkstra, E.W. The structure of the 'THE'—multiprogramming system. *Comm. ACM 11*, 5 (May 1968), 341–346.
2. Huberman, B.J. Principles of operation of the Venus microprogram. MTR 1843, F19(628)-71-C-0002, The MITRE Corporation, Bedford, Mass., May 1970.
3. Corbató, F.J., and Vyssotsky, V.A. Introduction and overview of the Multics system. Proc. AFIPS 1965 FJCC, Vol. 27, Pt 1, Spartan Books, New York, pp. 185–196.
4. Saltzer, J.H. Traffic control in a multiplexed computer system. Tech. Rep. TR-30, Proj. MAC, MIT, Cambridge, Mass., June, 1966.
5. Habermann, A.N. Prevention of system deadlocks. *Comm. ACM 12*, 7 (July 1969), 373–377, 385.
6. Holt, R.C. Comments on prevention of system deadlocks. *Comm. ACM 14*, 1 (Jan. 1971), 36–38.
7. Coffman, E.G. Jr., Elphick, M.J., and Shoshani, A. System deadlocks. *Computing Surveys 3*, 2 (June 1971), 67–78.