STRUCTURED OPERATING SYSTEM ORGANIZATION *

Martin V. ZELKOWITZ
Department of Computer Science, University of Maryland
College Park, Maryland 20742, USA

Received 2 July 1974
Revised version received 10 September 1974

operating systems program modularity resource allocation software reliability

1. Introduction

Operating system design was once a complex art that few understood (including many of the designers), but it is slowly becoming a science where many of the fundamental ideas are crystallizing into a set of basic axioms. The purpose of this paper is to present one set of ideas and show how they can be developed into a reliable system. The system will be hierarchically structured and has a powerful protection mechanism that allows for reliable system operation. Due to increased use of communications between computers, it is felt that operating system design should reflect this development and allow for networks of computers.

2. Operating system structure

An operating system consists of a set of independently executing programs called *processes*. Each of the processes execute on one or more central processors – usually in a multiprogrammed manner. Thus at the most primitive level, two system operations must be defined – process communication and process creation.

2.1. Communication

Process communication is usually implemented by one or more of the following techniques: shared memory, hardware trap instructions and message communication. It will be shown later that shared memory has certain drawbacks in order to keep processes isolated, thus it will be assumed that all communication is via messages. The function of the trap instruction will be to invoke operations in the primitive operating system (called the *kernel*) which provides the basic functions of process creation and process communication.

A *message* is simply a stream of characters. In order to implement them an I/O mechanism for processes must be established. It will be assumed that all messages are handled via *ports* [8]. A port can be viewed as an entrance into a process. A process is provided with the primitive operations (via trap instructions) of allocating a port, sending a message to a port owned by some process and receiving a message from a port that it itself owns. When sending a message, a process either may request that execution be suspended until the process receiving the messages replies, or may continue processing and wait for a reply at a later time. It will be assumed that each port has a unique name, and processes have the ability of passing the name to selected other processes.

This ability to pass port names selectively leads to two important features in operating system design – the creation of capabilities and the establishment of a protection scheme. The process that is to create a new function first establishes a port for that function. Any message then received on that port is interpreted

* This research was sponsored in part by contract N00014-67-A-0239-0032 from the Office of Naval Research to the University of Maryland.

as a request for that function. With this interpretation, the ability to send a message to this port is equivalent to being able to perform this function, or the port itself can be assumed to be a *capability* (the capability of performing the function).

Since processes have the ability of creating ports and passing these names to other processes, a form of protection can be implemented. Only processes that need a certain capability will be passed the appropriate port name. If the passing of a port name is also a primitive operation and handled by the kernel of the system, then this kernel can determine with which ports a given process may communicate, and thus no process will be able to forge a message to an unauthorized port.

2.2. Process hierarchy

Systems are usually designed using an abstract virtual machine approach [1,2]. This is sometimes called an "onionskin" design. At the lowest level is the basic hardware of the machine. Using this hardware primitive new operations are implemented for the next level virtual machine. Process creation and message communication are such operations. Using this level a higher level virtual machine is implemented which adds new primitive operations, until the final user level is implemented which contains such primitive operations as core allocation, accessing file systems and the like.

An extension to the port concept explained previously can allow this hierarchy to be readily implemented. It will be assumed that each process has an associated capability *vector* passed to it by the process that created it. This vector contains a subset of the capabilities (ports) available to its creator. A process may either add or delete entries from its capability vector and a process (if it has the capability) may pass a port name to another process. A process may only communicate with ports that are in its capability vector, and thus a structured communication scheme can be organized among all of the processes in the system – a strict hierarchy if the capability to pass a portname does not exist, and a more general hierarchy if that capability is passed.

2.3. Errors

An important aspect in any system design is the processing of errors. Hardware errors generally either halt a machine of nothing is specified about an error, or will activate an error routine if something is specified (an interrupt is "enabled"). This analogy can be implemented in the virtual machine design. At any level if a process has not anticipated an error, then the process will be terminated; if it has anticipated the error, then the appropriate error routine will be executed. This organization can be implemented as an extension to the capability vector, called the *interrupt vector.*

At the lowest level the interrupt vector is essentially the hardware interrupt mechanisms of the hardware. For each type of hardware interrupt (including the hardware trap instructors) the kernel of the system will enable the appropriate interrupt routine via a message to the port of the routine that processes the interrupt. (Of course the system must be sure that for time-critical interrupts, such as those that effect moving peripheral devices, the messages to these interrupt ports be given high priority and be processed immediately.)

At each successive level, for each capability that is passed to a process via the capability vector, a port name is passed as an interrupt vector entry. If a process sends a message to a function that generates an error condition, the called process will generate an error reply. This reply is interpreted by the kernel as a message to the port in the interrupt vector entry that corresponds to the capability just invoked. Usually there will be a suspended process waiting for a message on this interrupt port. This interrupt process can either halt the process in error or send a "normal" reply.

If a process wishes to process its own errors, it has the capability (if passed it) to alter its interrupt vector with a port name it owns. In this manner error conditions either stop a process by being reflected as a message to an ancestor process, or are error replies to the process (on a port possibly different from the normal reply port).

This organization should not incur significant

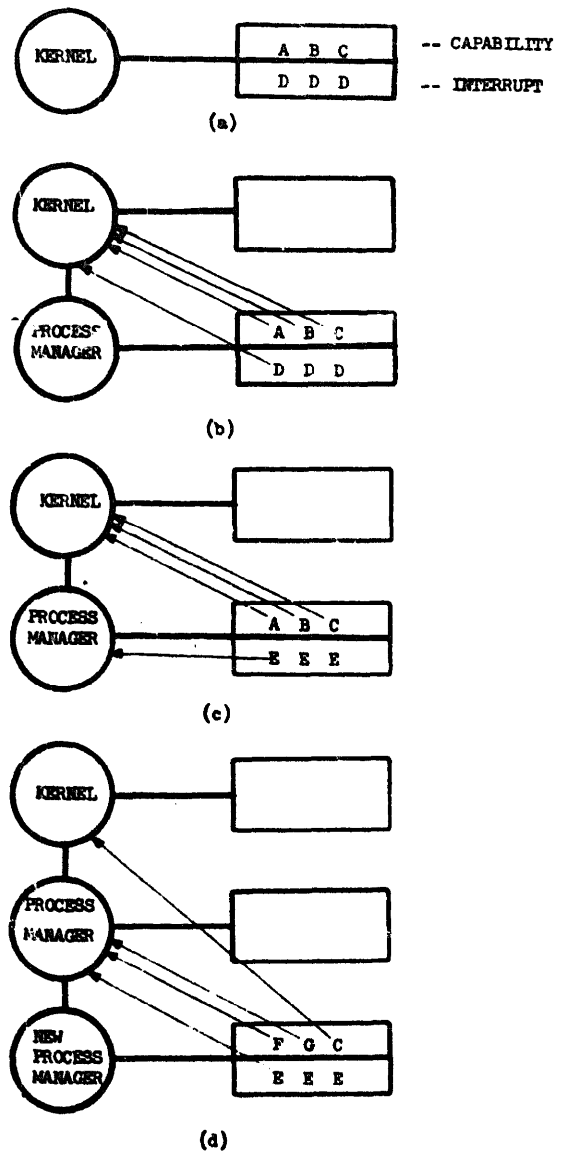

Fig. 1. Using capability vectors to create virtual machines.

overhead in message transmission; however, it will probably incur some overhead in generating error responses. The problem with errors is that the error condition may not effect the process causing the error because the interrupt vector may actually be associated with some other process. This would cause the kernel to generate an additional message to be sent, and thus more overhead would result.

An example of this organization may make this operation clearer. In fig. 1a the kernel of a simple operating system provides 3 capabilities – process creation (port A), process communication (port B) and the ability to alter the interrupt vector (port C). Port D is an error port for all 3 functions. The kernel creates the process manager and provides it with a capability vector (fig. 1b). The process manager enables its own interrupts by changing the interrupt vector entries (fig. 1c) using its capability to port C. Finally the process manager tests a new version of the process manager in a controlled virtual environment. It creates a new process manager process with modified capability vector (fig. 1d). Requests for process creation and process communication will be intercepted by ports F and G so that the old process manager can monitor the new process' behavior, and can simulate the requests by using its own capabilities to ports A and B. In this example the tested process is still given capability C, the ability to change its interrupt vector. In this example, the operation of the new process is under complete control of the original process manager, and proper reliable operation of the system can continue.

3. System reliability

An investigation into some of the ideas of structured programming shows that the above system organization uses attributes that result in well-structured systems. The following are some of these design considerations.

3.1. Module and data independence

Systems should be designed with minimal sharing of data structures across independent modules since basic data structures can be altered more easily if they are referenced in only one routine [6,9]. Only that routine need be altered should the structure change.

Modules should be explicitly defined as a set of input/output relationships [5]. No module should assume any implied structure within another module. This allows one module to be updated and changed independently from all other modules. It also prevents certain errors since a module cannot assume knowledge of the structure of some other module. If the input/output relationships of any module do not change, then any change to its data structures will be transparent to the rest of the system.

These attributes are preserved by the preceding system design, since all communication is via messages to ports. Notice that data shared in a common address space may not necessarily preserve these attributes since a module may be influenced not only by its input/output relationships, but also by the state of another module due to its shared data. Via shared memory it is often possible to (inadventently) monitor the progress of one routine by another, and thus make assumptions about its behavior, which may later be false if one of the modules is altered.

Messages can also have predetermined formats that are independent of the algorithms (or machines) used to create the messages. This enhances the independent nature of processes, and possibly allows processes to easily communicate between two different computers as in a computer network.

3.2. Systems are hierarchically defined

Systems are hierarchical using the proposed capability vector since each process in the system is created by some other process with the creator process having control of this capability vector. Only processes that have been passed a process' name can communicate with it, and thus the communications path is secure. This allows for certain functions to be implemented at one level of the system by being added to the capability vector, and to be deleted at another level by being deleted from the capability vector.

4. Summary

The design of a system using ports for all communications and a capability and interrupt vector to control communications paths can lead to a well-structured hierarchically designed operating system. Each process is effectively isolated from all other processes, yet the design allows for a reliable protection system where some but not all processes have access to certain functions.

This type of organization will become more significant with increased use of computer networks. With the low cost of minicomputers and the increased reliability of a set of small machines over a single large one, many applications will be distributed over networks of computers. Since processes need to know only a port name, rather than the location of a process, it is possible to design a distributed operating system where processes execute on several different machines, and can easily communicate [3,4,7]. This organization avoids the problems associated with trying to make an operating system that uses shared memory operate in a distributed manner across several machines. It is felt that the preceding will more readily allow for this type of implementation with such ideas as load sharing and resource sharing networks as the major beneficiaries of such design.

Acknowledgement

Some of the ideas expressed in this paper grew out of discussions held with W. Michael Lay during the initial design phases of the distributed computer network project at the University of Maryland.

References

[1] P. Brinch Hanson, The Nucleus of a Multiprogramming System, C. ACM 13 (1970) 238–241.

[2] E. Dijkstra, The structure of the THE multiprogramming system, C. ACM 11 (1968) 341–356.

[3] D.J. Farber and K.C. Larson, The system architecture of the distributed computer system. Symp. Computer networks, Polytechnic Inst. of Brooklyn (April, 1972).

[4] W.M. Lay, D.L. Mills and M.V. Zelkowitz, Operating systems architecture for a distributed computer network. Computer Networks: Trends and Applications, IEEE Computer Society Washington chapter and National Bureau of Standards, Gaithersburg, Md. (May, 1974) pp. 39–44.

[5] D. Parnas, A technique for software module specification with examples, C. ACM 15 (1972) 330–336.

[6] D. Parnas, On the criteria to be used in decomposing systems into modules, C. ACM 15 (1972) 1053–1058.

[7] R.H. Thomas and D.A. Henderson, McRoss – a multi-computer programming system. AFIPS Spring Joint Computer Conf. 40 (1972) 281–293.

[8] D. Walden, A system for interprocess communication in resource sharing computer networks. C. ACM 15 (1972) 221–230.

[9] W. Wulf and M. Shaw, Global variable considered harmful. SIGPLAN Notices 8 (1973) 28–34.