Toward the understandability of an operating system*

R. C. Varney

Bell Laboratories 1C-406A Holmdel, N. J. 07733, USA†

Design considerations are presented for the construction of an operating system. (This approach emphasises the need to understand the interrelationships among all operating system components to facilitate the inevitable growth and development of a useful system). The concept of a process and a resource are defined in such a way that a resource-to-process concept is developed, which is later used to provide an 'individualised' virtual environment for a process. Terminology is introduced to distinguish between so-called type *S* and type *M* resources, which must be treated differently for resource sharing. The levels of abstraction are described for the tree structured operating system that evolved from the design considerations. Within the tree, resource sharing is allowed and controlled by a communication language and so-called resource tables. Since the system was designed for use on a PDP-11/45, the basic components of its tree are given. It is then suggested that the tree presented here is, in fact, a basic framework upon which different specific operating systems may be built.

(Received March 1974)

1. Introduction

Most of the operating system design efforts have been directed toward the efficient design of a specific portion of proposed operating systems. This kind of effort has led to very valuable advances such as those for paging, file organisation, scheduling, memory hierarchies, programming languages, etc. However, the design strategies for the interrelationships among these system components have received much less attention. Dijkstra attacked this problem in THE (Dijkstra, 1968) and others (e.g. Liskov, 1972; Atwood, 1972; and Varney, 1973a) have more or less utilised his approach. More recently a workshop was held in Texas (Workshop, 1973) which at least in part, was concerned with the question of operating system design. However, a well-specified and widely accepted design strategy has not yet emerged.

This paper presents one approach to the problem of operating system design, which is believed to increase the understandability of the system in such a way that it substantially improves the ease of implementation, debugging, maintenance, and development. The basic design is a tree of processes where the tree structure provides information (using so-called resource tables) indicating the interrelationships among these processes. The knowledge of process interrelationships is particularly important for the processes within the operating system itself, and is, in addition, useful for the design of subsystems.

2. Design considerations

A number of important points must be considered in the design of an operating system. These points have been divided into two groups: primary and secondary. In general, the primary considerations aim to provide for the ease of growth and development which are inevitable. The secondary considerations are to be taken as less important to the global aspect of a growing system, although they most certainly cannot be disregarded.

The primary considerations are as follows.

1. *Provide for system understandability*

(*a*) reduce and/or control complexity due to process interaction. Even though each process within an operating system may be (easily) understood, there are so many processes within a given system that their interaction is a significant factor in overall system complexity. It is assumed that reduced complexity implies increased understandability.

(*b*) keep the appearance of local complexity at a minimum. Since man often prefers to view a problem through local components, if even the appearance of these components can be simplified, then the resultant system may be more easily embraced.

2. *Provide for reliability*

It is difficult to provide system reliability a priori, although Dijkstra claimed system correctness for THE (Dijkstra, 1968) due to its structure. We expect that reliability is a function of understandability, so that improving the latter will improve the former.

3. *Provide for measurement tools*

Once again it is difficult to provide all the needed measurement tools a priori. Thus, it was decided that a structure should be designed to which measurement tools could be appended as needed.

The secondary considerations are as follows.

1. *Provide sufficient power to perform the usual operating system functions*

This system must either provide or allow such facilities as multiprogramming, time sharing, and/or real time; various scheduling disciplines; resource sharing; interprocess communication; debugging and/or monitoring of arbitrary processes; etc.

2. *Provide local operating efficiency*

The system must not prohibit efficient process execution so that throughput is noticeably degraded. Hence, system overhead must be kept at a reasonable level.

3. Operating system components

There are two basic types of components in any operating system: processes and resources. Let us define these not as absolute concepts, but as concepts which change with the frame of reference.

resource—a source of supply

process —an autonomous entity which, upon request, will provide a specific service

Although the definition of resource may seem to connote a static quantity, such a connotation is strictly due to tradition

*Portions of the work done at The Pennsylvania State University, University Park, Pennsylvania 16802, USA.

†The author is now with Systems Control Inc, 1911 North Ft. Myer Drive, Arlington, Virginia 22209, USA.

and convenience in discussion. For example, a read instruction is often envisioned as the active element which extracts a piece of information from a static resource—namely, storage. However, a read is actually a request to the storage controller to fetch work from one location and place it into another location. Extending the argument one level further, we see that the controller places a signal on lines in such a way that a message is sent to a given core, which in turn responds by inducing (or not inducing) a signal on another line.

The point to be made is the following: a process becomes a resource when viewed at the next higher level. The higher the level of the process (i.e. the greater its sophistication), the more choice it may have in the kind of response it makes to a given request for service. Since operating system complexity is a problem, it seems reasonable to suggest that this complexity could be reduced by viewing the entire system in terms of this resouce-to-process concept, i.e. the method by which a process requests service from another process (a resource) should be uniform over all levels. A useful construct for the representation of this hierarchy is a tree, an instance of which is discussed in the next section.

To satisfactorily complete this section on the resource-to-process concept, let us introduce some terminology. A single operation instruction can be defined as an instruction which corresponds to a single operation on a resource, i.e. the instruction is indivisible relative to the users of that resource. To complement that, a multiple operation instruction is defined as an instruction which corresponds to a set of operations on a resource. From this, two types of resources can be defined.

Type *S* resource—a resource used by a set of processes, where each process in that set issues *only* single operation instructions.

Type *M* resource—a resource used by a set of processes where at least *one* process in that set issues multiple operation instructions.

It can be seen that a type *S* resource requires no co-operation among its users, whereas a type *M* resource, to insure the integrity of the resource, requires mutually co-operating users. (Brinch Hansen, 1972 and Courtois, 1971).

4. A tree structured system

4.1. *The process selector tree*

In developing this model, let each level of abstraction (Dijkstra, 1968) be viewed as a resource relative to the processes at the next higher level. However, instead of building new levels which abstract from the entire level below, let us distinguish and identify the processes within a given level and build new (partial) levels which abstract from a given process in the lower level. Since a process becomes a resource at the next higher level, and since individual processes are identifiable, these levels of abstraction can be called local levels of abstraction. Such a construction is a tree. And these local levels of abstraction within the tree represent small, comprehendable units which facilitate basic understanding.

The specific tree structure is called the Process Selector Tree or PST (Varney, 1971 and Varney, 1973a). The levels of abstraction for the PST are shown in **Fig. 1.** Level 0 is the hardware. At level 1, the Message Co-ordinator (MC) provides a communication language which will allow processes to create, destroy, and monitor child processes; to send messages to other processes; to synchronise with other processes; to respond to events; etc. The tree structure and resource tables (to be discussed) are maintained by the MC. At level 2, the Selection Algorithm (SA) provides the abstraction that each process has a processor on which to execute, so that the actual number of processes is hidden. Thus far, each level has not been divided into separate processes. At level 3, however, a number of distinct processes are implemented; the abstraction

	Level
Hardware	0
Communications language	1
Virtual processors	2
Basic level of virtual resources	3
Local levels of abstraction	4

Fig. 1 Levels of abstraction

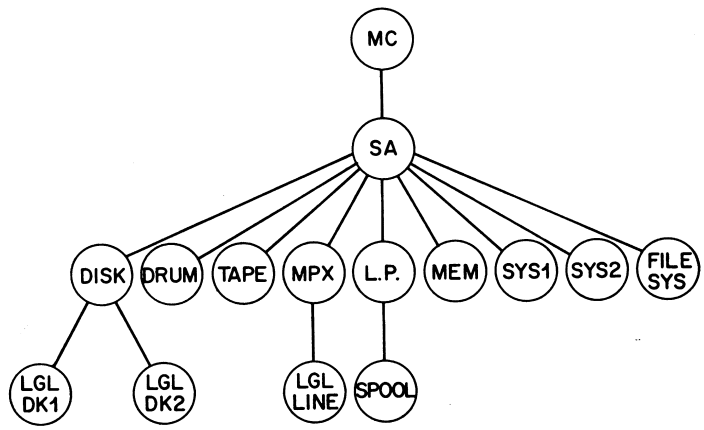

Fig. 2 Process selector tree

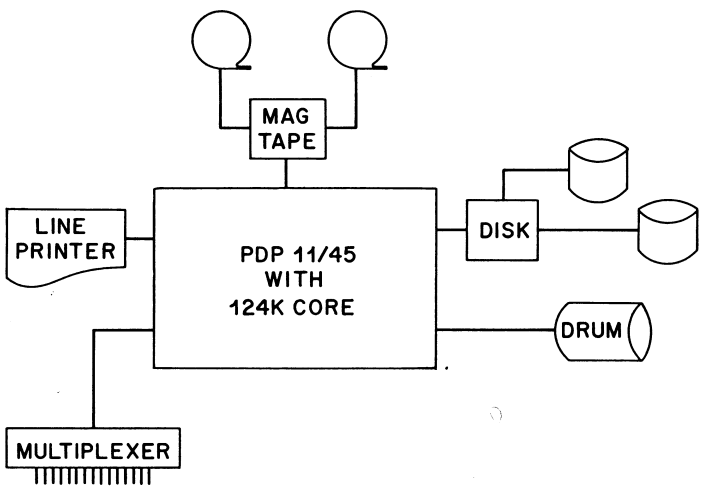

Fig. 3 Configuration

is called the basic level of virtual resources. At this level, one process is assigned to each physical I/O device to abstract a common message format which is capable of being transmitted to other processes in the system, i.e. these processes provide the first level of abstraction needed for I/O device independence. In addition, other processes appear at level 3, which also provide basic resources, such as a file system or a subsystem monitor. All processes at this level may communicate via the communication language with one another so that one process at level 3 may take advantage of the abstraction provided by another process at the same level. Above level 3, the tree can grow to provide the desired local levels of abstraction, e.g. two or more logical discs from one physical disc, a spooling process for the line printer, two distinct file subsystems for sequential and random files from a single basic file system, etc.

By providing any process with the ability to request service from any other process, a given process may operate on a virtual machine whose architecture is individually tailored to that process. Or, with the appropriate restrictions, a given process can be forced to accept a particular virtual machine, while

neighbouring processes operate on different virtual machines.

The specific PST system designed for a PDP 11/45 is shown in **Fig. 2.** The configuration is shown in **Fig. 3.**

4.2. *Resource tables and resource sharing*

Recall that one of the most important design goals was to reduce system complexity in order to increase its understandability. The tree built on the resource-to-process concept should provide the needed simplicity in conceptualisation; at the same time the tree may be used to impose restrictions on process interaction, thereby reducing much of the operating system complexity. A natural restriction is to force all interprocess communication along the branches of the tree. However, such a restriction is likely to significantly reduce operational efficiency (see Varney, 1973a). The resource table has been designed to provide a controlled means for relaxing or enforcing this restriction as desired.

When a parent process creates a child, the parent provides a resource table for the child as a subset of its own resource table. This table is basically a list of all other processes with which the child may communicate. Any interprocess communication must utilise the communication language provided by the MC, which, in turn, forces all communication through the process's own resource table.

A basic form of communication is the SEND MSG primitive. The process initiating the primitive is called the requestor and the target process is called the requestee. A given requestor, then has a resource table which contains a list of requestee names. The SEND MSG will be issued by the requestor for one of its known requestees. The resource table will map the requestee name either to the actual target process or to an intermediate process dependent on the mapping provided in the resource table which was prepared by the requestor's parent. That intermediate process can be the parent itself or any other process in its (the parent's) resource table. Such a mechanism is thus capable of enforcing or relaxing, to the desired degree, the restriction which forces all interprocess communication to follow the branches of the tree.

5. Concluding remarks

The PST structure forces the system designer to separate system functions into specific modules, and more importantly to organise the interrelationships among those modules. This structural restriction has much the same effect on an operating system as structured programming has on a single program, i.e. it is easier to understand, debug, and change. Restricting arbitrary process interaction is similar to restricting the use of the GOTO. Both force the designer to adopt different design habits, which, although initially uncomfortable, are usually superior to previous design habits (Dijkstra, 1972).

Another aspect of the PST concept is that it does not presuppose a specific set of operating system components. Rather, it is the structure upon which a specific operating system can be built. Although the PST concept does not necessarily exclude a general purpose operating system, it is designed as the structure to be used to build a specific (possibly special purpose) operating system, where the specifications are bound by the local designer. This implies that for almost every new installation (which differs from previous installations) some amount of system programming effort will be required, as opposed to the selection of parameters such as used in so-called general purpose operating systems. This programming effort is not regarded as a disadvantage for three reasons: (*a*) a new installation usually requires additional programming anyway; (*b*) the basic operating system structure is already provided, thereby requiring only the more local solutions to the configuration problems; and (*c*) the resulting system contains only what is required.

The PST concept, then, represents a framework which will accept various disciplines for handling system resources. It is a structure upon which a system may be built, while at the same time aiding the system implementor with improved system understandability.

References

ATWOOD, J. W. (ed.), *et al.* (1972). *Project Sue Status Report*, University of Toronto.

BRINCH HANSEN, P. (1972). A Comparison of Two Synchronizing Concepts, *Acta Informatica*, Vol. 1, pp. 190-199.

COURTOIS, P. J., HEYMANS, R., and PARNAS, D. L. (1971). Concurrent Control with 'Readers' and 'Writers', *CACM*, Vol. 14, No. 10, pp. 667-668.

DIJKSTRA, E. W. (1968). The Multiprogramming System, *CACM*, Vol. 11, No. 5, pp. 341-346.

DIJKSTRA, E. W. (1972). The Humble Programmer, *CACM*, Vol. 15, No. 10, pp. 859-866.

LISKOV, B. (1972). The Design of the VENUS Operating System, *CACM*, Vol. 15, No. 3, pp. 144-149.

VARNEY, R. C. (1971). Process Selection in a Hierarchical Operating System, *Proc. Third ACM Symposium on Operating System Principles*, pp. 106-108.

VARNEY, R. C. (1973a). *PST—A Tree Structured Operating System* (Ph.D. Thesis), The Pennsylvania State University.

VARNEY, R. C., and GOTTERER, M. H. (1973b). The Foundation for a Tree Structured Operating System, *The Computer Journal*, Vol. 16, No. 4, pp. 357-359.

Workshop on Operating Systems and Computer Architecture, Austin, Texas (January, 1973).