AN OPERATING SYSTEM MODEL FEATURING DEMAND SCHEDULING OF VIRTUAL RESOURCES *

Michael J. Spier
Massachusetts Institute of Technology
Project MAC

and

Elliott I. Organick
University of Houston
Department of Computer Science

DRAFT
10.24.69

SUMMARY

An operating system is a functional buffer to allow the programmer to manipulate the computer hardware via an idealized interface. High level languages have long been accepted as a desired interface which offers the programmer partial independence of the hardware. However, many hardware associated constraints, primarily the limitations of physical resources, which may affect the logic of a programmer's computation have not been systematically resolved. Various mechanisms have been implemented to override some of the more obvious constraints (e.g., paging, segmentation, processor stack, logical I/O devices etc.). In this paper we present a model for a computing utility which provides the programmer with a systematically idealized programming environment.

We think of a process as being an ordered sequence of requests for elementary computational operations, which are satisfied through the interaction of two or more independent virtual resources. A virtual resource is a logically distinct entity which has to interact with some other virtual resource in order to produce some meaningful effect. Examples of virtual resources are a segment with distinct access attributes, or a virtual I/O device which is capable of performing a single elementary I/O task (e.g., printing a file, typing out a line) in a single indivisible logical operation. We define the operating system to be a collection of independently allocatable virtual resources. For the purpose of this paper, we assume that it is generally known how to implement individual virtual resources, and consequently take such virtual resources for granted. We concentrate on the issues of virtual resource management, i.e., allocation, protection, accounting and retrieval.

---

* Work reported herein was supported in part by project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

In order to be able to develop the desired model, it is essential to introduce the following two concepts: a) the virtual processor which is a single sequential execution agent that, in itself, is incapable of performing any meaningful computation and which is an independently allocatable resource, and b) the protection sphere which is the set of all virtual resources capable of mutual interaction. To substantiate the proposed model, the authors suggest (in the body of the paper) possible implementations for both concepts.

A certain controversy exists in regards to the usefulness of a system, such as our proposed model, in which the problems of system throughput optimization (i.e., multiprogramming and scheduling) are solved by the system itself, as opposed to systems which provide the programmer with control handles and which expect the programmer to make decisions and apply multiprogramming control from his level. Our model hides all multiprogramming and hardware I/O management behind a façade of virtual resources and, by allocating a dedicated virtual resource to a programmer's computation upon demand (even though such resource may not necessarily map into an actual hardware resource at the time of allocation) it resolves the problems of hardware resource multiplexing by way of demand scheduling. We suggest that an operating system in which virtual resources capable of independent execution are awarded to competing customers on the basis of demand scheduling (as opposed to predetermined scheduling) provides optimum system throughput under all circumstances; that is, under the worst possible environmental conditions the system throughput is no worse than that of a system featuring predetermined scheduling (namely because the worst possible environmental conditions make it behave like a predetermined scheduling system). This seems reasonable because predetermined scheduling decisions are logically dependent upon unpredictable and irreproducible run time conditions. The appendix to this paper carries a more formal justification to this effect.

Central to our resource management scheme is the concept of the protection sphere. Every virtual resource in the system is associated with at least one or perhaps several (shared resource) protection spheres. The operating system is the universal protection sphere. A protection sphere is identified by a unique access tag which is meaningful to the system. Every virtual resource is associated with at least one access tag which the system is capable

of examining. Access tags are internal to the system and inaccessible to non-supervisor code. The system does not tolerate the interaction of virtual resources which do not feature matching access tags; any such attempted interaction is intercepted by the system and causes an access violation condition to be established.

All protection spheres in the system are administratively organized into a hierarchical tree structure. The root of the tree consists of the "system sphere" whose system tag is a predetermined value that is associated with at least one predetermined person identity which we name "system administrator". In other words, all system resources are, by definition, preallocated to the system administrator who may access them as he wishes. Resources may be recursively suballocated into smaller protection spheres.

. A virtual resource may be associated with several access tags, one of which is an owner tag. The system recognizes the owner tag as such, and only a resource's owner may suballocate that resource or grant access to it. In as far as access to a shared resource is concerned, all access tags are treated in an equivalent manner. The action of resource suballocation causes the resource's owner tag to be redefined. The execution of a process may require the services of a privileged subsystem which operates within a distinct protection sphere. In order to allow this, we associate a virtual processor with an access stack which permits the stacking of access tags. The system inspects the two topmost access tags upon each interaction; the second tag from the top is deprived of its execution attribute. A process may switch into another protection sphere by entering the new sphere in a specific control entry which pushes down the virtual processor's current access tag and placing the subsystem's owner tag on top of the access stack. Thus the virtual processor is dynamically allocated into the subsystem's protection sphere while it still maintains restricted (no execute) access in its previous protection sphere.

The implementation of a model as described above may be approached via a Dijkstra-like design that hierarchically orders a set of system processes so as to provide the operating system with the desired virtual resources. The lowest-level system process, named "process exchange", manages the virtual processors. Each virtual processor is associated with a set of virtual processor registers

which include, among others, the execution stack. By organizing the virtual processors into a hierarchical tree structure, and by giving a descendent virtual processor restricted access to his parent process' execution stack, it is possible to implement PL/1 type tasking on our model.

APPENDIX

Suppose that a computation requires a cpu and one or more I/O devices, that all steps in the computation are executed sequentially (i.e., no multiprogramming), and that only a single computation at a time may be executed on the computer (i.e., no multiprocessing). Let c be the amount of cpu time required by the computation, and let d_i be the amount of I/O device time required by the computation for a given device i. The duration t_j of a computation j using a cpu and n devices i is

$$t_j = c + d_1 + d_2 + \ldots + d_i + \ldots + d_n \qquad [1$$

and the duration T necessary to throughput m computations sequentially (i.e., batch processing) is

$$T = t_1 + t_2 + \ldots + t_j + \ldots + t_m \qquad [2$$

In order to increase the throughput of the system described above, we must decrease T by decreasing one or more of its component t_j. Supposing that a) all devices are logically independent of one another, and b) that the amount of cpu time spent c is very small compared to any d_i (assumption based upon the large speed discrepancies between cpu and I/O devices, as well as upon the observation of actual computer usage examples), we may optimize [1] through the application of multiprogramming techniques. Associated with the optimization is a certain overhead c_o measured in cpu time. We obtain an optimized time t_j^* for the duration of a computation j

$$t_j^* = \max [\, c, d_1, d_2, \ldots, d_i, \ldots, d_n \,] \qquad [3$$

by maintaining the assumption that $(c + c_o) \ll d_i$ we derive, by using t_j^* components for [2], an optimized T^* which is less than T.

We observe the similiarity of [1] and [2], and may further optimize [2] by applying to it multiprocessing techniques, in a way analogous to the multiprogramming technique applied to [1]. We assume that our m computations consist of q computations which may be run concurrently, and m-q computations which are logically dependent and must be run sequentially. Our optimized T^* corresponds to the throughput time of a multiprocessing system featuring predetermined scheduling,

$$T^* = \max [\, t_1^*, t_2^*, \ldots, t_q^* \,] + t_{m-q}^* + \ldots + t_{m-1}^* + t_m^* + c_o \qquad [4$$

It is obvious that the mutual independence of computations depends upon their degree of competition for any given device i; however, by preallocating devices in step [3] decisions were made which do not take into account run-time competition in step [4].

We achieve further optimization by the introduction of demand scheduling; we do this by simply skipping step [3] and expressing the optimized T_v^* directly in terms of device usage time d_i. We use the notation D_i^p to express the total amount of time spent by some device i on independent operation, and notation D_i^s to express the amount of time spent by the same device i on sequential operation. The throughput time T_v^* for a multiprocessing system featuring demand scheduling is

$$T_v^* = \max [D_1^p, D_2^p, \ldots, D_i^p, \ldots, D_n^p] + D_1^s + D_2^s + \ldots + D_i^s + \ldots + D_n^s + c$$