Medusa: An Experiment in Distributed Operating System Structure (Summary)

John K. Ousterhout, Donald A. Scelza[1], and Pradeep S. Sindhu
Carnegie-Mellon University
Pittsburgh, Pennsylvania

The paper is a discussion of the issues that arose in the design of an operating system for a distributed multiprocessor, Cm*. Medusa is an attempt to understand the effect on operating system structure of distributed hardware, and to produce a system that capitalizes on and reflects the underlying architecture. The resulting system combines several structural features that make it unique among existing operating systems.

Cm* consists of fifty processor-memory pairs arranged in five clusters, with one communication controller (Kmap) managing interprocessor communication for each cluster. The Kmaps are microprogrammable processors and have been used to implement several interprocessor communication mechanisms ranging from shared memory to message systems. Thus each processor may potentially access all of the system resources in a uniform manner. However, because of the distributed hardware organization the cost of accessing a given memory location from a given processor varies by a factor of ten depending on the relative locations of processor and memory. For a program to execute efficiently, it must make most of its references to the local memory of the processor on which it is executing.

The combination of distribution and sharing in the hardware has a strong impact on software design, and raises two operating system issues. How should the system be partitioned in order to enhance its modularity and make use of the distributed hardware? How should the separate subunits communicate so as to function together in a robust way as a cohesive unit?

The paper shows why traditional operating system organizations are unsuitable for Cm*. Centralized systems like those used for current uniprocessors and multiprocessors would lead to contention for communication resources and serious performance degradation. Replicated systems like those used in networks would waste most of the primary memory of Cm*.

The functions provided by Medusa have been divided into several *utilities* that are distributed around the Cm* system. Each utility executes in a private protected environment and implements a single abstraction for the rest of the system. Messages are used by user programs to communicate with utilities, and by utilities to request services from other utilities. The message system provides an efficient mechanism for passing control between processors, encourages strong logical separation between utilities, and makes it possible to reconfigure the system transparently.

Because Cm* consists of a relatively large number of small processors, it was assumed in the

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

[1]Author's current address: PRIME Computer, Inc. Old Connecticut Path, Framingham, MA 10171.

design of Medusa that all interesting programs would be concurrent ones combining the computational powers of several processors. Thus the fundamental control structure in Medusa is a *task force*, defined to be a collection of *activities* that cooperate closely in the execution of a single logical task. In general the activities of a task force run concurrently on different processors. All programs, including the operating system utilities, are task forces.

The task force structure permits very close cooperation to occur within a task force. Activities may share memory pages and other objects by placing descriptors for them in a shared descriptor list accessible to all activities of the task force. Medusa attempts to *coschedule* task forces so that when one activity is executing, all the activities of its task force are executing.

Medusa is made up of three pieces: Kmap microcode, a replicated kernel, and distributed utilities. Approximately 4000 microinstructions in each Kmap implement memory sharing, synchronization, and a fast message system. Each processor contains 400 words of kernel code providing activity multiplexing and interrupt response. The bulk of the operating system (about 30,000 words of code) is distributed between five utility task forces: a memory manager, a file system, a task force manager, an exception reporter, and a debugger/tracer.

Each utility task force contains at least two activities in different clusters. Each activity is capable of providing all of the functions of the utility. If the utility's workload increases, additional activities may be created in the task force to assume some of the utility's load; if the workload decreases again the additional activities will delete themselves. Within a utility, tables and synchronization variables are kept in shared memory where all the activities of the utility may access them. The only communication between utilities occurs using the message system.

Utilities are given several privileges so that they may access the implementations of protected objects such as file control blocks. The paper presents the special privileges given to utilities and discusses the tradeoff between the grain of those privileges and the distribution of the system.

The potential for deadlock is a natural consequence of the distribution of low-level system functions such as memory management and process management. The paper shows why simple-minded utility implementations could result in deadlock even though the calling structure within the operating system is hierarchical. To eliminate the deadlock problem, each utility activity is structured as a collection of coroutines dedicated to particular classes of service.

Finally, the paper discusses the effects of the distribution and concurrency of the system on exception reporting. The sharing of objects has resulted in a dual view of exceptions, internal and external, depending on whether or not the activity to which an exception is being reported is the one that detected the exception (internal) or just an interested bystander (external). The sharing of control within a task force has led to the notion of a *buddy*, by means of which one activity may handle an exception on behalf of another activity in its task force.