case history. In Proc. 4th Int. Conf. Software Engineering, Munich, Germany, Sept. 1979, pp. 94–99.
8. Internet Datagram Protocol, Version 4. Prepared by USC/Information Sciences Institute, for the Defense Advanced Research Projects Agency, Information Processing Techniques Office, Feb. 1979.
9. Lampson, B.W. Redundancy and robustness in memory protection. *Proc. IFIP 1974*, North Holland, Amsterdam, pp. 128–132.
10. Lampson, B.W., Mitchell, J.G., and Satterthwaite, E.H. On the transfer of control between contexts. In *Lecture Notes in Computer Science 19*, Springer-Verlag, New York, 1974, pp. 181–203.
11. Lampson, B.W., and Redell, D.D. Experience with processes and monitors in Mesa. *Comm. ACM 23*, 2 (Feb. 1980), 105–117.
12. Lampson, B.W., and Sproull, R.F. An open operating system for a single user machine. Presented at the ACM 7th Symp. Operating System Principles (*Operating Syst. Rev. 13*, 5), Dec. 1979, pp. 98–105.
13. Lauer, H.C., and Satterthwaite, E.H. The impact of Mesa on system design. In Proc. 4th Int. Conf. Software Engineering, Munich, Germany, Sept. 1979, pp. 174–182.
14. Lockemann, P.C., and Knutsen, W.D. Recovery of disk contents after system failure. *Comm. ACM 11*, 8 (Aug. 1968), 542.
15. Metcalfe, R.M., and Boggs, D.R. Ethernet: Distributed packet switching for local computer networks. *Comm. ACM 19*, 7 (July 1976), pp. 395–404.
16. Mitchell, J.G., Maybury, W., and Sweet, R. Mesa Language Manual. Tech. Rep., Xerox Palo Alto Res. Ctr., 1979.
17. Pouzin, L. Virtual circuits vs. datagrams—technical and political problems. Proc. 1976 NCC, AFIPS Press, Arlington, Va., pp. 483–494.
18. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.
19. Ross, D.T. The AED free storage package. *Comm. ACM 10*, 8 (Aug. 1967), 481–492.
20. Rotenberg, Leo J. Making computers keep secrets. Tech. Rep. MAC-TR-115, MIT Lab. for Computer Science.
21. Stern, J.A. Backup and recovery of on-line information in a computer utility. Tech. Rep. MAC-TR-116 (thesis), MIT Lab. for Computer Science, 1974.
22. Sunshine, C.A., and Dalal, Y.K. Connection management in transport protocol. *Comput. Networks 2*, 6 (Dec. 1978), 454–473.
23. Stoy, J.E., and Strachey, C. OS6—An experimental operating system for a small computer. *Comput. J. 15*, 2 and 3 (May, Aug. 1972).
24. Transmission Control Protocol, TCP, Version 4. Prepared by USC/Information Sciences Institute, for the Defense Advanced Research Projects Agency, Information Processing Techniques Office, Feb. 1979.
25. Wulf, W., et. al. HYDRA: The kernel of a multiprocessor operating system. *Comm. ACM 17*, 6 (June 1974), 337–345.

Operating Systems

R. Stockton Gaines Editor

Medusa: An Experiment in Distributed Operating System Structure

John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu
Carnegie-Mellon University

The design of Medusa, a distributed operating system for the Cm* multimicroprocessor, is discussed. The Cm* architecture combines distribution and sharing in a way that strongly impacts the organization of operating systems. Medusa is an attempt to capitalize on the architectural features to produce a system that is modular, robust, and efficient. To provide modularity and to make effective use of the distributed hardware, the operating system is partitioned into several disjoint *utilities* that communicate with each other via messages. To take advantage of the parallelism present in Cm* and to provide robustness, all programs, including the utilities, are *task forces* containing many concurrent, cooperating *activities*.

Key Words and Phrases: operating systems, distributed systems, message systems, task forces, deadlock, exception reporting
CR Categories: 4.32, 4.35

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This research was supported by the Department of Defense Advanced Research Projects Agency, ARPA Order 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 U.S. Government.

Authors' present addresses: J.K. Ousterhout and P.S. Sindhu, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213; D.A. Scelza, PRIME Computer, Inc., Old Connecticut Path, Framingham, MA 01701.

© 1980 ACM 0001-0782/80/0200–0092 $00.75.

1. Introduction

Medusa is a multi-user operating system under development for Cm*, a multimicroprocessor system recently completed at Carnegie-Mellon University [7, 24, 25, 26]. The project is an attempt to understand the effect on operating system structure of the distributed Cm* 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.

Medusa is the second operating system for Cm*. The first system, StarOS [10, 11, 12], has been concerned with making Cm* programmable at a high level by users. Thus the StarOS system embodies a general object addressing mechanism and a set of tools for manipulating parallel programs. For Medusa we have chosen to emphasize problems of structure, rather than facilities. The goal of the project has been to gain an understanding of how to build distributed operating systems and to exploit the hardware to produce a system organization with three attributes:

Modularity. The system should consist of a large number of small, cooperating subcomponents that may be built, modified, and measured independently. Such a structure complements the Cm* hardware, which consists of a large number of small processors.

Robustness. The system should be able to respond in a reasonable way to changes in its environment. These changes include an increase or decrease in workload and the failures of hardware, firmware, or software components. The multiplicity of processors should aid in achieving this goal.

Performance. Both the structure of the operating system and the abstractions it provides to user programs should reflect the underlying hardware. Application programs should be able to run with approximately the same efficiency using Medusa as they could on the bare hardware.

The software organization required to achieve the above goals has been strongly influenced by the Cm* architecture. Two general attributes of Cm* have been especially important. First, the physical arrangement of the hardware components is a distributed one. For each processor there is a small set of local resources that is accessible immediately and efficiently: the other facilities of the system are also accessible, but with greater overhead. The second fundamental attribute of the hardware is the power of the interprocessor communication facilities, which permits an efficient implementation of a variety of communication mechanisms ranging from shared memory to message systems. Section 2 discusses the Cm* hardware in more detail.

The combination of distribution and sharing in the hardware gives rise to two corresponding software issues: partitioning and communication. How should the operating 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 single logical entity? The remainder of this paper is a discussion of these issues and of the solutions embodied in Medusa.

Although the paper discusses the issues of partitioning and communication in a particular environment, that of Cm*, the issues have found relevance in many other areas of computer science. Research in structured programming has been concerned with reducing interactions within programs so that subportions of the programs can be designed and implemented separately [18]. Recent efforts in very large scale integrated (VLSI) circuit design are attempting to find ways to structure chip designs so as to reduce communication costs, since most of the area and complexity of the circuit stem from the interconnections [23]. Efforts to achieve fault tolerance must also be concerned with issues of partitioning and communication. For a system to be fault tolerant it must be subdivided into independent components with protected communication mechanisms so that (1) faults can be detected quickly, and (2) the occurrence of a fault in one portion of the system is not likely to destroy the other portions of the system [5].

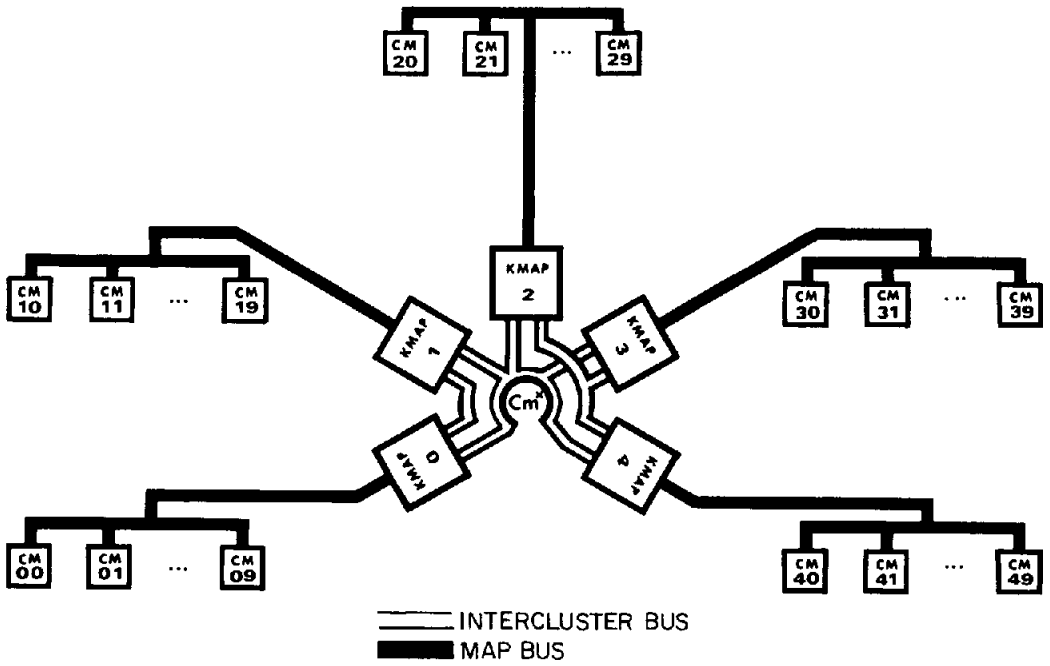
The structure of Medusa resulted from the interaction of the three general goals (modularity, robustness, and performance) with the architecture of Cm*. The system has two significant characteristics:

(1) The control structure of the operating system is distributed. The functionality of Medusa is divided into disjoint *utilities.* Each utility executes in a private protected environment and implements a single abstraction for the rest of the system. Utilities communicate using messages.

(2) Parallelism is implicit and expected in Medusa. All programs are organized as *task forces,* each of which is a set of cooperating *activities.* By making the task force the central unit of control, Medusa makes it possible for very fine-grain interactions to occur within a task force. Each Medusa utility is a single task force.

Section 2 gives a brief discussion of the Cm* hardware and develops the distributed structure of Medusa. In Section 3 the task force notion is introduced. Section 4 gives an overview of the actual Medusa organization. Sections 5–7 discuss several topics in the implementation of the system: the use of pipes for utility communication (Section 5); the special privileges granted to utilities so that they can implement operating system functions (Section 6); deadlock in utility communication and the internal utility organization used to eliminate it (Section 7). Section 8 shows how the distribution and sharing in the system have impacted the exception reporting mechanism and discusses the benefits gained thereby. Section 9 concludes by comparing the Medusa structure with that of other systems.

Fig. 1. The 50-Processor Cm* System.



2. The Distributed Structure

Figure 1 shows the organization of the Cm* hardware. It can be seen that the physical structure closely resembles that of a network. The system consists of a collection of 50 relatively autonomous processors called *computer modules* ("Cm"s in the PMS notation of Bell and Newell [2]), and five communication controllers (Kmaps). The Cm's are divided into *clusters*, with one Kmap presiding over each cluster. Each Kmap is responsible for the interprocessor communication to and from the Cm's in its cluster and must cooperate with the other Kmaps to handle intercluster transactions. All Kmaps are equal in stature; there is no central authority in the system.

Figure 2 gives a closer view of an individual computer module. Each Cm contains an LSI-11 processor, 64 or 128 kbytes of memory, and perhaps one or more I/O devices. A local switch, or Slocal, connects the computer module with the interprocessor communication structure. It is the presence of the Slocal as a switch in the computer module's addressing path (rather than as an I/O device on the Cm's bus) that distinguishes Cm* from networks. Tables in the Slocal allow it to decide on a reference-by-reference basis whether processor memory references are to proceed to local memory or be passed out to the Kmap of the cluster. Since Kmaps are general purpose processors with writable control stores, system designers have substantial freedom to choose their own methods of interprocessor communication. The Kmap microcode used by Medusa, which implements both shared memory and messages, is the third addressing structure to be implemented on Cm* (see [6] for a discussion of the others).

The ability to provide a variety of communication mechanisms efficiently, in spite of the distributed nature of the hardware, makes Cm* unique in the space of multicomputer systems and is fundamental to the development of Medusa.

Table I, taken from [19], shows the effect on program speed of making references to memory other than that of the processor executing the program. Although any processor may potentially access any memory location in the system, the cost of so doing varies by an order of magnitude, depending on the relative locations of processor and memory. The efficiency of a program running on Cm* is critically dependent on the "local hit ratio," or fraction of memory references that proceed from a processor directly to its local memory without involving a Kmap. A rule of thumb is that local hit ratios should generally be 90 percent or better in order to avoid serious performance degradation due to contention and/or the nonlocal access times.

Relatively high hit ratios can be achieved in a straightforward way by physically localizing information that need not be shared. Table II, also taken from [19], shows the dynamic distribution of memory references made by three multiprocess programs running on Cm*. Code references alone account for three-fourths of all memory references. Global information, that is, the information that must be accessible to all processes in a concurrent program, rarely accounts for more than a few percent of the references. Table II suggests that it makes relatively little difference where in the system global data is placed. On the other hand, code cannot be shared efficiently between concurrent processes; the primary problem facing system designers for Cm* is how to distribute the control structure.[1]

[1] This is in contrast to most of the past work in distributed systems, which has been concerned with distributed data structures.

Fig. 2. Organization of a Cm.

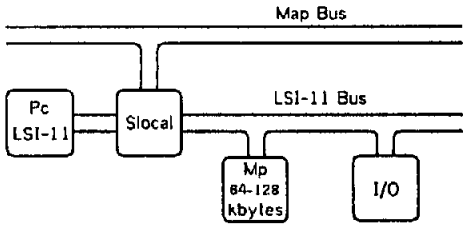


Table I. Average time per memory reference for a typical program making all code and data references to a single memory unit in a lightly loaded system

Memory position relative to Pc	Average reference time (microseconds)
Local	3.5
Nonlocal but within cluster	9.3
Cross-cluster	24

Fig. 3. One possible OS organization for Cm*: A single copy of the operating system code is shared by all processors.

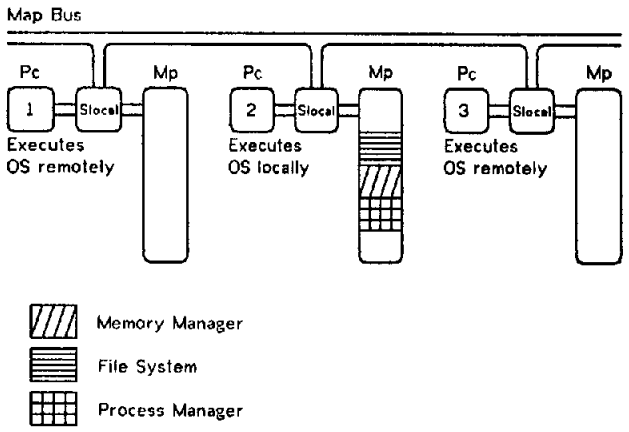


Table II. Dynamic distribution of memory references between four program components for three multiprocess programs

Program	Code (%)	Stack (%)	Owns (%)	Globals (%)
Partial differential equations	82	11.5	4	2.5
Quicksort	71	12.5	6	9.5
Set partitioning	71.5	23.5	4	1

2.1 Traditional Operating System Approaches

Using performance as a basis for judgment, this section shows why certain basic assumptions made by existing operating systems are inappropriate for Cm*.

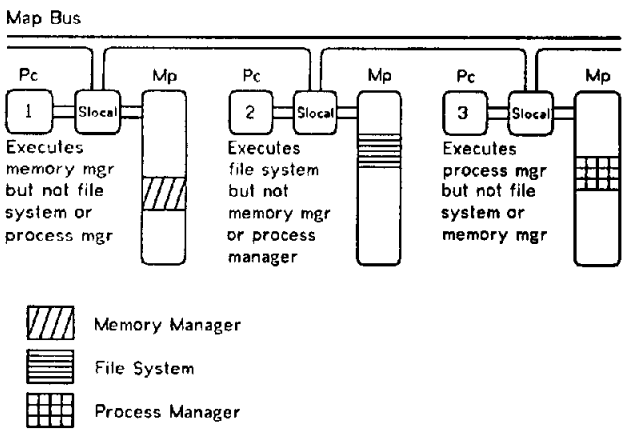
Consider the application of a traditional operating system structure to Cm*. It is an assumption of virtually all existing operating systems that all of the functionality of the OS is immediately available at every point within the system. Typically, a user program invokes the OS using a "trap" or "supervisor call" instruction. The processor switches execution environment to that of the kernel or supervisor, executes the function, then returns to the user program.

Figure 3 shows one possible operating system organization for Cm*. For purposes of this discussion it is assumed that the operating system must provide three functions: memory management, a file system, and process management. These functions must be accessible to all programs running anywhere on the system. The approach of Figure 3 is much like that of a traditional uniprocessor or multiprocessor operating system. A single copy of the operating system resides in one of the memories of the system. One processor is capable of executing the code directly, and the other processors have their memory mapping tables set up so that they can execute the code remotely. Unfortunately, the processors executing the operating system remotely will run at only about one-third speed because of the nonlocal access times. Processors executing the code from other clusters will suffer even more than this. In systems of any size, contention for the shared communication facilities will also degrade performance. In addition to the performance problems, the reliability of this system is limited by the reliability of the computer module containing the shared operating system code.

An alternative approach is to replicate the operating system code in each processor of the Cm* system. This is similar to the approach taken by networks, where each processor contains a copy of basic operating system functions; sharing occurs at a higher level, for example, via a shared file structure. Unfortunately, the size of local memories in Cm* is only 64 or 128 kbytes. A moderate sized operating system of 40–60 kbytes would consume most of the primary memory of the system. The utilization of this memory would be very low, especially when compared with the utilization of the shared memory in Figure 3. (Of course, even in the replicated system some global data still has to be shared between processors; Table II indicates that references to this memory do not substantially affect system performance.)

A static form of caching could be used to combine the approaches of the preceding paragraphs. Frequently executed portions of the operating system code could be replicated in each processor, while infrequently used code could be centralized in a shared memory. This scheme was used by the first version of the StarOS system. Unfortunately, the determination of which portions of the operating system are frequently executed is relatively difficult to make and likely to be application dependent. We decided not to attempt to make such decisions statically. The availability of hardware caches would make this scheme rather more attractive; given that there are no caches on Cm* we have not pursued it any further.

Fig. 4. The organization of Medusa: The operating system is divided into utilities that are distributed around the system. A processor executes a particular utility only if it can do so locally.



2.2 A Distributed Approach

The solution adopted for Medusa was to discard the assumption that all operating system code may be executed from any point in the system. A simplified example of this is depicted in Figure 4. The operating system is divided into disjoint *utilities*. Utilities are distributed among the available processors, with no guarantee that any particular processor contains a copy of the code for any particular utility. Furthermore, to avoid contention or memory access delays, a given processor is permitted to execute code for a particular utility only if it can do so locally.

Since no processor is guaranteed to be capable of executing any particular piece of utility code, it may be necessary for a program's flow of control to switch processors when it invokes utility functions. Trap or subroutine call instructions cannot be used for invocation because they are incapable of crossing processor boundaries. In Medusa, messages provide a simple mechanism for cross-processor function invocation. If one program wishes to invoke a function of another, it does so by sending a message to a particular *pipe* (see Section 5 for details of the pipe implementation). The invocation message contains parameters for the function invocation, as well as an indication of a *return pipe*; the return pipe is analogous to a return address for a subroutine call. The destination program receives the message, performs the requested operation, and sends a return message (containing result values) to the return pipe indicated in the invocation message. This message transaction is equivalent in power and effect to a call-by-value-result procedure invocation, with two major differences. First, the message crosses protection boundaries as well as processor boundaries, so that the invoker and server may exist in disjoint execution environments. Second, the invoker need not immediately suspend execution to await the requested service; if it has other functions to perform it may execute them concurrently with the execution of the server. Lauer and Needham discuss the duality between messages and procedure calls in more detail in [15].

There is an important difference of view between the Medusa system organization and the organizations presented in the previous sections. In the previous examples, a locus of control existed on a single processor. At the level of memory references, control and data information were transmitted to and from that processor by the Cm* hardware. In the Medusa paradigm, the control information (code) is fixed and is used by a single processor. As various loci of control (user programs and other utilities) need to access the control information, they must move to the processor containing the code. The message transaction is logically equivalent to a transfer of control from one processor to another.

2.3 Distribution for Modularity and Robustness

In the above development of Medusa's distributed structure, performance was used as the main motivation. However, the resulting structure is also desirable for other reasons; these may be even more compelling than the performance considerations. Given that the system must be partitioned into relatively autonomous subunits, the module scheme of Parnas [18] seems the most attractive. Each of the utilities encapsulates a single abstraction for the rest of the system. The use of messages for communication with utilities permits each utility to exist in its own protected execution environment. The boundaries between utilities are rigidly enforced and are crossed only by messages. Message communication is without side effects; that is, a message is received only when requested and the only effects it has on the receiver are those caused by the receiver. This is not true with shared writable memory, where either party may arbitrarily modify the memory without the other's knowledge. Although the utilities still depend on each other to perform certain services, Medusa has attempted to eliminate all interactions between utilities except those absolutely necessary for them to function. The strong separation between system components should simplify the tasks of building and maintaining the system.[2]

The distributed utility structure is also a first step toward robustness. Boundaries between utilities are firm enough that it is difficult for mishaps in one utility to contaminate the others. At the same time, message pipes do not restrict either the location or internal organization of the senders and receivers. Once the format of messages has been agreed upon and particular pipes have been selected for communication, utilities may migrate around the system without affecting their ability to communicate. A different pipe may be substituted for an existing one to change the handler for a particular class of requests. The use of pipes both limits damage and permits transparent system reconfiguration.

[2] The Family of Operating Systems project [9] has had favorable experiences with a system structure consisting of several isolated modules communicating only via a call-by-value-result mechanism.

System designers have traditionally resigned themselves to the "fact" that a performance penalty must be paid if a system is to have a manageable structure. Although the performance of Medusa has not yet been measured, it is encouraging to note that the same structure appears to be both efficient and modular. We speculate that the compatibility of good performance and good system structure is not just a statistical fluctuation in Medusa but a trend in large systems, where much of a system's cost stems from communication. For example, Sutherland and Mead suggest in [23] that the trend will hold for design in the domain of VLSI.

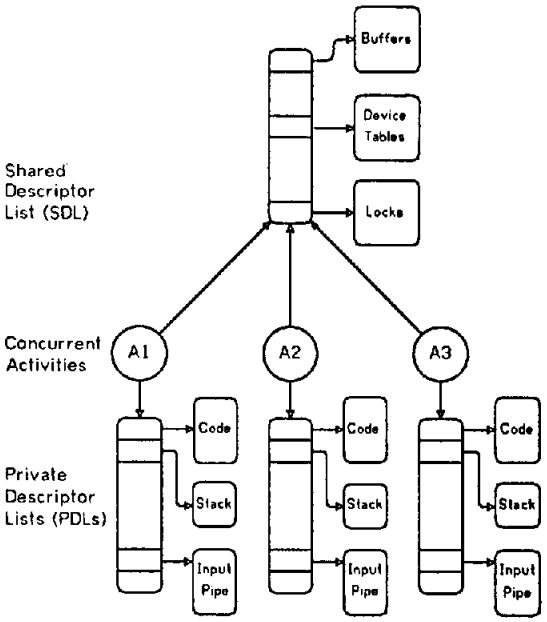
3. The Task Force Structure

LSI-11's are a relatively plentiful resource in Cm* but individually are not very powerful (16-bit virtual address space, 170,000 instructions per second). We have assumed in the design of Medusa that all interesting Cm* programs will be concurrent ones combining the computational power of several processors. This has led to the definition at the system's lowest level of a simple structure to support concurrency, called the *task force* after a similar structure of the same name used in the StarOS system [12, 13]. A task force is defined to be a collection of concurrent *activities* that cooperate closely in the execution of a single logical task. Activities are the entities that actually get scheduled for execution on processors. They are roughly analogous to processes or programs in other systems except that they exist only as part of a task force. A purely serial program is a degenerate task force with only one activity. All programs, including the operating system utilities, are task forces.

When a task force is created, its activities are allocated statically to individual processors (because of the importance of accessing code locally it makes little sense to execute an activity from any processor but the one containing its code). In general each of the activities of a task force will be allocated to a different processor. Individual processors may be multiplexed between several activities belonging to different task forces.

In addition to its activities a task force contains a collection of objects that may be manipulated by those activities. There are several types of objects in Medusa, each with its own set of type-specific operations. Access to objects is obtained through *descriptors* that are kept in protected objects called *descriptor lists.* The various types of objects may be divided into three general classes according to where the operations upon the objects are implemented. The simplest class consists of *page objects*; these may be associated with any of 16 portions of an activity's 64-kbyte virtual address space and may then be read or written using LSI-11 memory references. The second class of objects consists of *pipes* and *semaphores*, whose implementations are managed by Kmap microcode. Pipes and semaphores are protected so that LSI-11 programs cannot directly access their internal representations; requests must be made to a Kmap to manipulate them. The third class of objects, containing file control blocks, task forces, descriptor lists, and others, is implemented by the operating system utilities. User programs make requests to the utilities in order to perform operations upon these objects. Plans exist for a fourth class of objects, whose implementation would be managed by other user-level programs; the initial system will not provide for objects of this class.

Fig. 5. The organization of the Medusa file system utility. All activities are identical; they serve different requests in parallel.

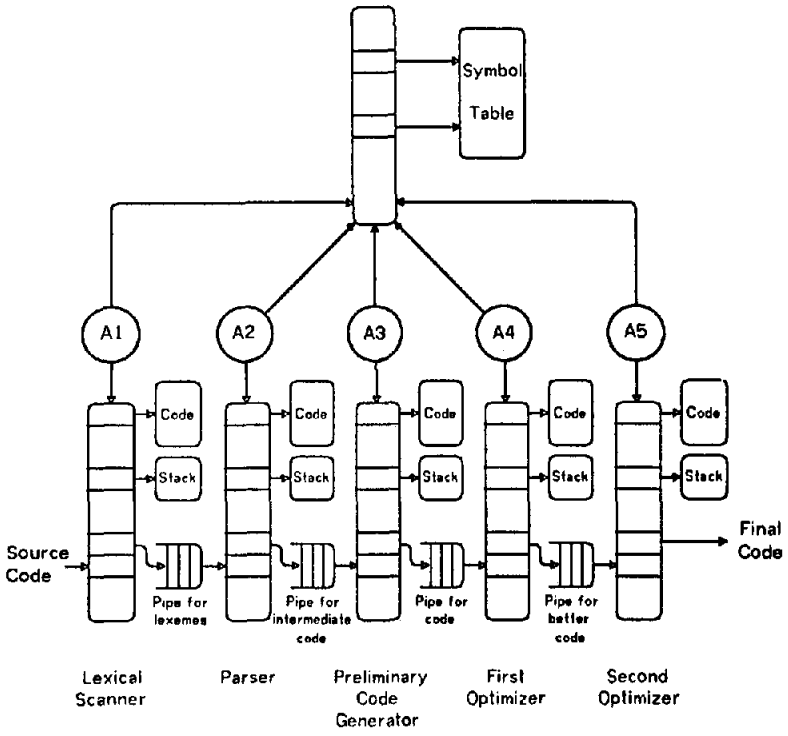


The descriptors used by a task force to access objects are held in two kinds of descriptor lists. Each activity has exclusive access to a *private descriptor list* (PDL). A *shared descriptor list* (SDL) contains descriptors that may be used by all of the activities of the task force. The activities of a task force generally share a common abstract function; the SDL is the encapsulation of that shared task. In general, descriptors may be moved and/or copied, subject to the restriction that the total number of descriptors that may simultaneously exist for each object may not exceed a limit specified when the object was created. Page objects are special in that only a single descriptor may exist. Although a page may be shared within a task force by placing the descriptor in the SDL of the task force, it cannot be shared between task forces.[3]

As an example of a task force, the Medusa file system utility is shown in Figure 5. The several activities are identical. Each contains in its private descriptor list (PDL) a copy of the file system code and a stack page; both are allocated in the memory of the Cm executing

[3] We felt that activities that are not closely related in their abstract functions should not interact in uncontrolled ways. Since the task force is the unit of abstraction, we decided to restrict read/write memory sharing to within a single task force.

Fig. 6. A Concurrent Compiler. Each activity of the task force implements one stage of the compilation process.



the activity, so that access may be made locally. In addition, each PDL contains descriptors for one or more pipes over which service requests are made to that activity. The shared descriptor list contains information that must be accessed by all of the file server activities in order to use I/O devices in a consistent way. This includes the physical device registers, buffer space, device descriptors, and locks used to synchronize file access. The size of the file system task force varies dynamically to provide load-balancing and fault tolerance. If the existing activities become overloaded, they create new activities to assume some of the workload. When activities become underloaded, they give their work back to other activities and delete themselves. For reliability purposes, there will always be at least two activities in each utility task force. In this simple example, global data is centralized in the SDL while control information, particularly that associated with error handling, is replicated.

An alternative task force organization is shown in Figure 6. In this example we have speculated about the structure an optimizing compiler might take. The organization proposed here is a pipelined one. The program being compiled passes through every activity of the compiler, undergoing a transformation at each step. For example, the lexical scan activity reads the source code, generates symbol table entries, and outputs lexemes to the parser. The parser activity generates intermediate code from the lexemes and so on. The symbol table is the only piece of information needed by all of the activities; it is referenced through descriptors in the SDL and could probably be stored in any processor in the system without degrading the performance of the compiler (see Table II). The task force structure is used in different ways in these two examples. For the file system, the activities are used to serve multiple independent requests; the multiplicity of activities provides robustness and increased throughput. In the compiler the many activities serve to encapsulate subtasks of the compilation process and to speed up the processing of a single request.

The SDL/PDL task force organization has levels of locality that reflect the Cm* hardware organization. The PDLs are roughly analogous to individual computer modules, and the SDLs are roughly analogous to clusters. Of course, Medusa does not force all information in a PDL/SDL to be in the same Cm/cluster, but we hope that this structure will encourage programmers to organize their programs in ways that map cleanly onto Cm*. Each PDL is always allocated in the processor that will execute the corresponding activity, as is the *activity control block* that contains saved registers and other execution state of the activity. If an activity allocates its critical information in its own processor and keeps descriptors for that information in its PDL, then the activity is not likely to be destroyed by hardware failures in other processors in the system.

The Medusa structures are organized so as not to associate unrelated information arbitrarily. The strong separation between the utilities is one example of this design goal. The descriptor organization is another. In many operating systems all of the descriptor information for all user programs is kept in one large array; a small failure in the array, or in a program that manipulates the array, may affect many otherwise unrelated portions

of the system. In contrast, Medusa places descriptor information for each activity in a separate structure and places that structure in the processor that will execute the activity. The distribution and reliability of the system are roughly those of the user programs; the operating system imposes no hidden centralities.

There are two reasons for implementing the task force notion as a low-level system structure. First, most of the operating system functions are provided by task forces, hence the structure must exist at the operating system level. Second, for activities within a task force to interact on a fine grain, certain guarantees must be made to the task force. Foremost among these is the notion of *coscheduling* which, if not provided by the operating system, cannot be implemented using higher level protocols. A task force is said to be coscheduled if all of its runnable activities are simultaneously scheduled for execution on their respective processors. In large task forces whose activities interact frequently, it is often the case that the descheduling of a single activity can cause the whole task force to block on locks held by that activity. If the task force is not coscheduled, a form of thrashing occurs that is very similar to the thrashing in early demand paging systems. The set of activities that must execute together for a task force to make progress on its task is analogous to a process' working set in a paging system. If the operating system does not provide for coscheduling of this "activity working set," there is no way to simulate it at user level short of disabling time sharing. Simple algorithms have been developed for task force allocation and scheduling in Medusa to maximize coscheduling.

4. Overview of the Current System

The complete Medusa operating system consists of three parts:

The several Kmaps, each using about 4,000 80-bit words of microcode, cooperate to provide the basic interprocessor communication mechanisms. These include references to page objects, simple operations on descriptors that help to guarantee the consistency of descriptor space, block transfers, and operations upon message pipes and semaphore objects. A relatively powerful event system has been implemented, partially in Kmap microcode and partially by the utilities: it provides for events to be associated with many different kinds of objects in a similar way so that programs may signal and await events in a type-independent fashion.

A small *kernel* is replicated in each processor to respond to interrupts generated by that Cm. The kernels supply simple multiplexing and device handling code, currently amounting to about 400 words of basic kernel code and 250 additional words for each device type present on the processor's LSI-11 bus (most of the Cm's have no devices at all). The multiplexing code makes no policy decisions about which of the activities allocated to a processor should be executed; it merely provides a mechanism for switching execution contexts. Device routines translate interrupts into messages to the various utilities. In addition the device routines chain device commands to maximize throughput of the devices.

Almost all of the operating system functions are provided by a collection of five utilities. Each utility is a task force that implements one or a few abstractions for the rest of the system. The *memory manager* is responsible for the allocation of primary memory and also aids the Kmap microcode in descriptor list manipulation. The *file system* task force acts as a controller for all the input and output devices of the system and implements a hierarchical file system nearly identical to that of Unix [20]. The *task force manager* creates and deletes task forces and activities and provides some simple debugging functions. Two often overlooked portions of operating systems have been formalized in Medusa by making them utilities: the *exception reporter* and a *debugger/tracer.* When an exception occurs on a processor, the kernel of that processor reports the exceptional condition to the exception reporter. It is the responsibility of the exception reporter to communicate information about the exception to relevant parties. The debugger/tracer holds symbol tables and performance measurement information for all of the utilities. It provides facilities for on-line debugging of the system and for gathering performance data.

The activities of the utility task forces are spread among the available processors, typically with no more than one utility activity per processor. Although no memory is shared between utilities, central tables and synchronization locks for a given utility are shared between all the activities of that utility. The Kmaps route messages from user activities to utilities when the user activities request services from the utilities (see below); the same scheme is also used for service requests made by one utility of another.

5. Pipes and Utility Communication

5.1 The Implementation of Pipes

Pipes in Medusa are similar to those of Unix [20] in that they hold uninterpreted strings of bytes. The operation of Medusa pipes differs from that of Unix pipes in two important ways: (1) the integrity of messages in Medusa is insured by maintaining byte counts for each message and by permitting only whole messages to be read from pipes; (2) the identity of the sender of each message is made available to the receiver of the message. Descriptors may not be passed using the pipe mechanism; instead, descriptors are moved by the memory manager utility in response to requests issued via pipes.

In a system that uses message transactions as a procedure invocation mechanism, the overhead involved in such transactions is of fundamental importance. The degree to which the system can be partitioned is limited

from an efficiency standpoint by the cost of an interaction between components. The time required for a 5–10 word message interaction in Medusa is approximately the same as the time used in executing 30 LSI-11 instructions; that is, assume that a server has attempted to receive a message from an empty pipe and hence has been suspended; the time between the initiation of a "send" operation by an invoker (executing on a different processor) and the fetch of the first instruction by the server after receiving the message is roughly 250 microseconds, or about 30 average LSI-11 instruction times. This is about the same time overhead as a high-level language procedure call in which registers must be saved and a display updated.

A primary reason for the efficiency of the message mechanism is the treatment of the "pipe empty" and "pipe full" conditions. In a producer–consumer relationship between activities, it is unlikely that the producer and consumer will operate at exactly the same speed. The communication pipe will almost always be either empty or full and one of the activities will usually have to wait for the other. Virtually all existing operating systems reschedule a processor as soon as the current activity blocks on an empty or full pipe; this means that when the pipe becomes nonempty or nonfull a second context swap must be executed to reactivate the now-runnable activity. Thus almost every interaction between activities results in processor context swaps. In uniprocessor systems the context swaps are necessary since only a single activity can execute at a time; however, in a multiprocessor system the context swaps may become the efficiency bottleneck in the interprocess communication mechanism.

For Medusa, we wished to provide a mechanism that would permit interactions to occur on a substantially finer grain than that of a context swap. When an activity attempts to send to a full pipe or receive from an empty pipe, its execution is suspended until the operation can proceed. However, the activity does not relinquish its processor immediately. For a small interval of time, called the *pause time* and specifiable by the activity, the activity's processor remains idle with the activity loaded. If the activity becomes runnable within the pause time it is reactivated without incurring any context swaps. If the pause time is exceeded, then the processor reschedules itself to another activity. Note that if the time for which the activity is blocked is less than two context swap times, then the lost processor time due to the pause is less than the time that would otherwise have been wasted in context swaps.

5.2 Utility Descriptor Lists

Communication between user activities and utilities, and also between utilities and other utilities, is accomplished via a set of message pipes using the message-passing mechanism described above. The collection of pipes used for utility communication is a critical system resource whose organization affects both the reliability and modularity of the system.

Every processor contains a special descriptor list, called a *utility descriptor list* (UDL), that is used by the activities of that processor to communicate with utilities. To invoke a utility operation a user activity specifies to its Kmap the index of a pipe descriptor in its processor's UDL and a message to be placed into the selected pipe. Each descriptor in a UDL corresponds to a set of operating system functions. The assignment of particular slots of utility descriptor lists was made at system design time, and is analogous to the assignment of trap numbers to particular functions in other operating systems. The descriptors in the UDLs are divided into two groups: the first group may be used by either user or utility activities; the second group is visible only to utility activities and is used to invoke functions that are privileged to utilities.

If there were only a single utility descriptor list, then both the reliability and performance of Medusa would be jeopardized. The assignment of UDL slots to functions is the same for all UDLs; however, having several UDLs makes it easier to share the implementation of a function between several activities. For example, consider the pipes used to invoke the OPEN operation provided by the file system. Normally, the file system task force contains two activities. Half of the UDLs in the system will contain descriptors for a pipe serviced by the first activity, and the other half will contain descriptors for a pipe serviced by the second. If one of the file system activities becomes overloaded, it will spawn a new activity in the file system task force and replace several of the OPEN descriptors in UDLs with descriptors for a new pipe served by the new activity. If at some later time the new activity becomes underloaded, it may overwrite its UDL descriptors with descriptors for the pipes serviced by the other file system activities and kill itself. Similarly, one activity may take over for another activity that has failed. Having one UDL per processor makes it relatively easy for workload to be exchanged between utility activities in a way that is transparent to the rest of the system. Note that it is also important that utility communication pipes are kept exclusively in the UDLs; if user activities were given private copies of these descriptors and allowed to move them around it would be extremely difficult for the utilities to find all the relevant descriptors during reconfiguration.

6. Utility Privileges

In order to guarantee the integrity of protected types such as descriptor lists, there must exist mechanisms to insure that the utility that implements each type may access its internal representation while other, less-trusted, programs may not. Thus, each utility activity executes with a status bit set to grant it several privileges not given to user activities. The privileges allow utilities to (1) share address space with other activities, and (2) perform certain special operations denied to user programs.

The sharing of address space is accomplished through an *external descriptor list* (XDL) accessible to each utility activity in addition to its PDL, the SDL of its task force, and the UDL of its processor. The XDL is not a distinct descriptor list, but just a mechanism to give a utility shared access to the PDL or SDL of another activity in the system. By presenting the absolute system name of a descriptor list to its Kmap, a utility activity maps its XDL onto that descriptor list such that references to the activity's XDL refer to the named descriptor list.

A simple form of *amplification* is provided to the utilities so that they can manipulate the representations of protected types of objects.[4] This is accomplished by allowing utilities to make special requests to the Kmaps, wherein an object is made readable and writable by the activity as if it were a page object, even though it is really of a protected type. The amplification affects only the utility activity that requested it: if other activities attempt to make read or write references to the object, then type violations will occur.

The combination of XDL usage and amplification provides a simple *seal* on objects. Thus objects can be stored with their owners yet be manipulated only by the utilities that implement them. Without this seal, each object would have to be stored with the utility that manipulated it and would consequently be more vulnerable to failures in the utility. Having some sort of seal mechanism is critical to our ability to distribute and reconfigure the system. Medusa's protection mechanism is a relatively simple one; readers are referred to [21] and the protection literature for a more detailed discussion of these issues.

The XDL and amplification privileges are used extensively by the utilities. For example, when a user activity requests that information be written into a file, the file system utility amplifies a *file control block* (accessible through the user's SDL or PDL) to determine where on disk to write the information. Upon completion of the write, the information in the file control block is updated. The file system uses its XDL to read information from the user activity's pages into the actual disk buffers. Task force creation also illustrates how the XDL and amplification privileges are used: the memory manager uses its utility privileges to access allocation tables and thereby create the descriptor lists for the new task force; the task force manager then amplifies the kernel areas of relevant processors to enter the new activities into scheduling tables.

During the design of Medusa a tradeoff was made between the number of special utility privileges and the degree of distribution of the system. The replicated portion of the system consists of 400 words of kernel code in each processor and 4,000 microinstructions in each Kmap. The kernel and Kmap code provide what we perceive to be the minimum functionality needed to tie together a distributed system in an efficient and protected way. The largest portion of the operating system (thirty to forty thousand words) is distributed among the utilities. It would have been desirable to provide a general enough object addressing mechanism that utilities could get all the privileges they needed via the object mechanism. For example, a mechanism like that of Hydra [4] would have provided a substantially finer grain of protection than we have with the single utility status bit, would have enabled a closer enforcement of the "need-to-know" principle, and would have eliminated the need for any special utility privileges. Unfortunately, the implementation of such a general mechanism would have enormously increased the complexity of the replicated kernel and Kmap code. This is not to say that such general mechanisms are undesirable. Rather, we contend that for a system to be distributed it must be *layered*; if all mechanisms are implemented at the system's lowest level, then they must be replicated everywhere. Since one of the principal design goals of Medusa was to distribute its functionality, we decided that the proper place to implement a general object mechanism would be *in* the utilities rather than underneath them.

7. The Internal Structure of Utilities

7.1 Deadlock

The potential for deadlock arises as a direct consequence of the low-level distribution of operating system functions among disjoint utilities. This section introduces the deadlock problem and the restrictions placed on utility interactions in order to eliminate it.

Consider the OPEN operation implemented by the file system utility. In order to open a file, storage must be allocated for the file control block that will be used during transfers to and from the file. Thus a file system activity executing the OPEN operation must invoke the memory manager utility to allocate the storage. It might turn out that the memory manager was unable to meet the allocation request without swapping information between primary and secondary memory, so it would have to request one or more I/O transfers from the file system. This circularity of requests between the file system and the memory manager could result in a deadlock situation with all the file system activities awaiting results from the memory manager and all of the memory manager activities awaiting I/O transfers from the file system; there would be no file system activities to service the I/O requests.

Another example of an operating system containing circular dependencies between system modules is described by Schroeder at al. in [22]. Habermann et al. [9] have argued that in general there is no correspondence

[4] Amplification is a mechanism that permits users to hold tokens for objects without being able to access their representations. When the token is passed to the subsystem responsible for the object's implementation, the subsystem *amplifies* the token to obtain access to the information that represents the object. Thus tokens for objects may be passed around the system at will, yet operations upon the object may be performed only by the subsystem with the ability to amplify the tokens [14].

between the notions of *module* and *invocation level*; even though the invocation dependencies of a system are hierarchical, the dependencies between modules may still be circular. Figure 7 depicts the example of the previous paragraph. There are three invocation levels, consisting of the OPEN, ALLOC, and I/O operations. There are two modules, the file system and memory manager, and two invocation dependencies, represented by arrows. Deadlock could occur if all of the service resources of the file system become allocated to OPEN requests, leaving none to service I/O requests.

The standard implementation used in operating systems has been to allocate service resources dynamically from a large central pool to each invocation level of an operation. For example, many uniprocessor systems allocate an independent kernel stack for each user process. When a kernel operation is invoked, frames on the user's kernel stack are allocated for each invocation of a procedure within the kernel. As long as the total depth of the stack is not exceeded, circularities may exist in the interactions between kernel modules.

The distribution of the Medusa utilities makes it impossible to share service resources between invocation levels. An invocation of one utility by another crosses execution environments such that a common call stack is infeasible. An alternative solution would be to pass all of the execution state of a transaction from each utility to the next one that it calls, thus freeing up the resources of the caller. Unfortunately this would be intolerably inefficient, and would make the execution state of each utility vulnerable to the whims of other utilities that it invokes. Another approach would be to allocate new resources at the time of each call, for example by creating a new utility activity to serve each request. In addition to being rather inefficient, the creation of a new activity, or even the allocation of memory, would itself require a utility call and hence could not be done as part of the calling sequence. The potential for deadlock is a result of the fact that even the low-level system functions are distributed among several disjoint environments.

Two conditions must be met for a distributed system like Medusa to be deadlock free. First, all of the functions provided by utilities must be divided up into *service classes* such that (1) a single utility provides all of the services in each class and (2) there are no circularities in the dependencies between service classes. A single utility may implement more than one service class. In the terminology of [9] each service class is the collection of functions provided by a single module at a single invocation level. In the example of Figure 7 the OPEN and CLOSE functions, as well as several others, are included in one service class while I/O functions are contained in another and memory allocation functions in a third. The second condition for deadlock avoidance is that each utility must contain separate (statically allocated) pools of resources so that it can provide independent service to each of its service classes. In the current Medusa system the utilities provide about five service classes apiece, and

Fig. 7. A hierarchical set of calls that leads to circular utility interactions.

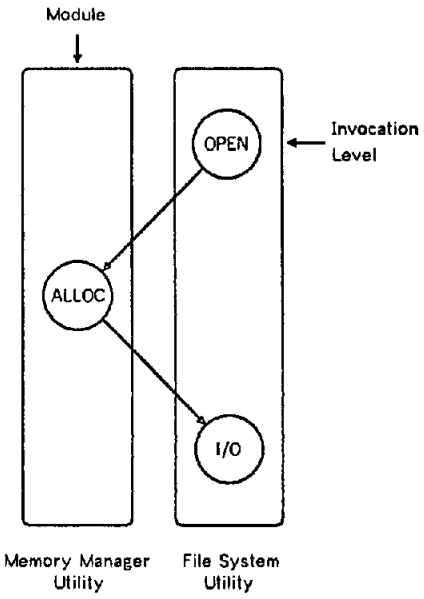
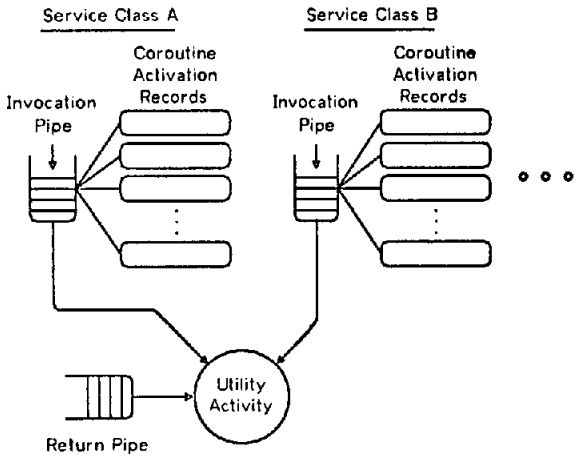


Fig. 8. The internal structure of a utility activity.



the invocation hierarchy of the system as a whole contains eight levels.

7.2 Coroutines and Multi-Eventing

One way to satisfy the requirement that independent facilities be provided for each class of utility service is to dedicate particular activities of a utility to provide each of the classes of service supported by the utility. This eliminates the possibility of deadlock, but makes relatively poor use of the utility activities since some service classes are rarely invoked. It also precludes load-sharing between utility activities that are responsible for different service classes.

The solution used in Medusa is to structure utility activities as collections of coroutines or processes, as shown in Figure 8. Every activity in a utility task force is capable of providing any and all of the classes of

service implemented by the utility. For each class of service provided by the utility there is a separate invocation pipe and a pool of coroutines that serve requests received via the pipe. Descriptors with receive privilege for the invocation pipes are held in the PDL of the activity, and descriptors with send privilege are present in one or more UDLs around the system. Each coroutine has its own activation record containing a stack and other local information. The coroutines are allocated statically to the service of particular invocation pipes, so that the overloading of any one pipe cannot affect the activity's ability to service the others.

When a message arrives from an invocation pipe, one of that pipe's coroutines is allocated to serve the message. The coroutine remains allocated to the request until it completes, at which time the coroutine's activation record is returned to the free pool associated with the pipe. If the coroutine needs to request a service from another utility, then the activity executes other runnable coroutines while the remote service is being provided. When a reply is received for the coroutine it is reactivated and continues service. To the coroutine the remote service request appears like a procedure call. In addition to eliminating the deadlock problem, the use of coroutines increases the throughput of each utility activity by allowing it to process one transaction while another waits for remote service.

A single return pipe is shared by all of the coroutines of the activity to receive acknowledgments from other utilities. When an invocation message is sent from one utility to another, as in the case of the ALLOC request of Figure 7, a *transaction identifier* is sent in the message. The invoked utility must return the transaction identifier in the acknowledging message. The identifier indicates which coroutine is to be reactivated and provides a simple validity check on the acknowledging message.

At any given time a utility activity could potentially receive messages over any of several pipes. The Kmap microcode used by Medusa implements a *multi-event* system whereby an activity may wait for the first message to arrive from any of a set of pipes. For a utility this normally includes all of its invocation pipes and its return pipe. When the current coroutine of the activity cannot continue execution, either because it completed its current operation or because it invoked another utility, the *MultiEventWait* Kmap operation is invoked. This causes the activity to suspend until a message is received on one of the pipes in the set. When a message is received, the activity is reactivated with an indication of the pipe from which the message was received. If the pipe is an invocation pipe, then a coroutine is allocated to service the message and the activity begins executing the coroutine at a standard startup address. If the pipe is the return pipe, then the transaction identifier in the return message is verified and a return is made to the coroutine that invoked the transaction. If all the activation records for a particular invocation pipe become allocated, the pipe is removed from the activity's multi-event set so that requests via the pipe will be stored in the pipe until coroutines become available to service them.

The coroutine structure of the utilities is visible only to a few routines that are used to invoke other utilities and recycle coroutines after they have completed requests. The vast majority of utility code is completely unaffected by the internal multiplexing of each activity. Since the utilities already contain multiple concurrent activities, there are no additional synchronization constraints imposed by the internal multiplexing except that busy-waiting on locks cannot be permitted (busy-waiting prevents another coroutine of the same activity from executing code to release the lock).

8. Exception Reporting

The combination in Medusa of distribution and cooperation has had interesting effects on several portions of the system. Section 7 discussed some of the difficulties encountered in the design of the utilities because of their distribution. This section briefly describes two ways in which the distribution and sharing have impacted the exception reporting mechanism. The sharing of objects has led to two different views of exceptions, *internal* and *external*. The sharing of a control function within a task force, combined with the distribution of the activities of the task force, has resulted in the notion of a *buddy* that can handle exceptions remotely.

For a discussion of what constitutes an exception, readers are referred to [16]; our informal definition is that an exception is just an unusual occurrence, which may *or may not* be an error. Several exception classes are defined in Medusa, ranging from parity errors to the execution of unimplemented instructions. Each activity is allowed to specify for each class a *handler* to be invoked when an exception from that class is generated by the activity.

When an object is shared, it is reasonable for all the co-owners of that object to participate in the handling of exceptions that occur for the object. However, the exception may manifest itself in two ways to the different co-owners. For each detection of an exceptional condition there is exactly one activity whose explicit action uncovered the exception. To this activity the exception is manifested as an *internal condition*; the execution state of the activity is directly involved with the exception, and the handler may have to unwind operations in progress. The other co-owners, although interested in being notified of the exception, were not directly involved in its detection. The exception is seen by them as an *external condition* whose occurrence is not necessarily related to the current actions of the co-owners. The handler for an external condition is generally concerned with the state of the shared abstraction but not with the execution state of its activity. An exception may be reported internally without being reported externally if no shared abstraction is involved; however, all external conditions also

have corresponding internal conditions.[5] The double view of exceptions arises only because it is possible to share objects between independent activities.

While the distinction between internal and external conditions arose because of data sharing, the notion of a buddy has arisen because of control sharing in a distributed environment. It is often easier to handle an exceptional condition at a point that is physically and logically distinct from the site of the exception's discovery (see the next paragraph for examples). Medusa allows an activity to present the name of a *buddy* activity within the same task force as handler for any exception class. When the exception occurs the buddy activity is notified. While handling the exception the buddy receives access to all the private information of the exception-generating activity, which is suspended. Buddies are only useful if the buddy is relatively distinct from the exception-generating activity, yet has enough knowledge of the other activity's function to take reasonable actions on its behalf; fortunately, this is almost exactly the relationship between the activities of a task force.

We believe the notion of a buddy to be an especially useful one: two examples follow. A failure in a code page is conceptually a simple one to recover, since virtually all modern-day code is pure. However, it may be extremely difficult for the activity whose code page is lost to recover itself, since recovery code may have been contained in the page. A solution is for the activity to specify a buddy handler for hard errors in the page. The buddy need merely locate a duplicate copy of the code and replace the bad page with a fresh copy (in the case of Medusa utilities, each activity contains an identical copy of the code, making this task especially simple). The activity that generated the exception then repeats the instruction that failed and goes on its way with no explicit knowledge that an error occurred. Another potential use of the buddy mechanism is in emulation. Programs written for an operating system that use TRAP instructions for invocation can be run as one activity of a task force, with TRAP instruction exceptions being handled by a buddy. The buddy activity uses Medusa calls to emulate the desired effects of the TRAP instruction and then allows the trapping activity to proceed. In this way programs written for other operating systems may be run on Medusa with no changes to their code.

9. Comparisons and Conclusions

Many similarities exist between Medusa and the StarOS system. Both systems are object oriented and both provide for task forces of cooperating activities. However, tradeoffs in StarOS generally were made in favor of providing an extremely flexible set of facilities to high-level users; tradeoffs in Medusa generally were made in favor of producing a highly distributed operating system structure that closely matched the hardware. For example, StarOS task forces are much more general than Medusa task forces. The precisely defined structure of Medusa task forces may occasionally constrain users, but allows us to support the structure with coscheduling and with buddy exception handling; these facilities would be much more difficult to support in the more general StarOS task force framework. The StarOS object mechanism provides a finer grain of protection than Medusa's mechanism, but has resulted in a more centralized structure.

Message communication in Medusa is similar to the usage of links in DEMOS to communicate with the operating system [1], while task forces are somewhat similar to teams of processes in Thoth [3]. A substantial difference between the systems is the presence of concurrency in Medusa. Thus issues not present in DEMOS arise concerning the sharing of work between the activities of a utility task force and the dynamic reconfiguration of the system. Memory sharing in Thoth is restricted to processes in the same team, just as it is restricted to activities of the same task force in Medusa. However, Thoth guarantees that only a single process of a team executes at any one time and that it is not interrupted by any other processes of the team; Medusa attempts to insure that ALL activities of a task force execute simultaneously.

Several designs for exception reporting mechanisms have been proposed in the literature, among them [8], [16], and [17]. All of these are based in programming languages; the reporting mechanism for Medusa is not intended to replace them. Rather it provides an entry mechanism into the reporting schemes of individual languages, leaving the languages to propagate exception information among the abstract levels of a program. Medusa's scheme also provides facilities for passing exception information between programs.

The design of Medusa was begun in the late fall of 1977, and coding started in the summer of 1978. The status of Medusa at the time of this writing (November 1979) is as follows: all of the Kmap microcode and utility code has been written and microcode testing is nearly complete. The Medusa system has been successfully loaded onto the machine and the debugger utility is operational. Testing of the remaining utilities has just begun, with roughly two-thirds of all the utility code yet to be tested. We expect preliminary utility testing to be complete by early 1980, with a Unix emulator running shortly thereafter.

In summary, Medusa has attempted to achieve modularity, robustness, and performance in a Cm* operating system using a structure that embodies at its lowest level both distribution and concurrency. No processor of the system is self-sufficient, yet no processor should be critical to the operation of the system. We intend to evaluate our success by measuring how well the system can deal with changes in its environment, including changes in user load as well as failures in hardware, firmware, and software.

[5] This is just another way of saying that exceptional conditions do not arise spontaneously, but must be explicitly checked for.

Acknowledgments. The authors wish to thank Prof. Nico Habermann, whose comments concerning both the operating system and this paper have been extremely helpful.

Received June 1979; accepted September 1979; revised November 1979

References

1. Baskett, F., Howard, J.H., and Montague, J.T. Task communication in DEMOS. Proc. 6th Symp. Operating Systems Principles, SIGOPS, 1977, pp. 23–32.
2. Bell, G.C., and Newell A. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1971.
3. Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R. Thoth, a portable real-time operating system. *Comm. ACM 22,* 2 (Feb. 1979), 105–114.
4. Cohen, E., and Jefferson, D. Protection in the Hydra operating system. Proc. 5th Symp. Operating Systems Principles, SIGOPS, 1975, pp. 141–160.
5. Denning, P.J. Fault tolerant operating systems. *Comput. Surv. 8,* 4 (Dec. 1976), 359–389.
6. Fuller, S.H., Jones, A.K., Durham, I., Eds. Cm* Review, June 1977. Carnegie-Mellon Univ., June 1977.
7. Fuller, S.H., Ousterhout, J.K., Raskin, L., Rubinfeld, P., Sindhu, P.S., and Swan, R.J. Multi-microprocessors: An overview and working example. *Proc. IEEE 66,* 2 (1978), 216–228.
8. Goodenough, J.B. Exception handling: Issues and a proposed notation. *Comm. ACM 18,* 12 (Dec. 1975), 683–696.
9. Habermann, A.N., Flon, L., and Cooprider, L. Modularization and hierarchy in a family of operating systems. *Comm. ACM 19,* 5 (May 1976), 266–272.
10. Jones, A.K., Chansler, R.J. Jr., Durham, I., Feiler, P., and Schwans, K. Software management of Cm*—A distributed multiprocessor. Proc. AFIPS 1977 NCC, Vol. 46, AFIPS Press, Arlington, Va., 1977, pp. 657–663.
11. Jones, A.K., Chansler, R.J. Jr., Durham, I., Feiler, P., Scelza, D.A., Schwans, K., and Vegdahl, S.R. Programming issues raised by a multiprocessor. *Proc. IEEE 66,* 2 (1978), 229–237.
12. Jones, A.K., et al. StarOS, a multiprocessor operating system for the support of task forces. Proc. 7th Symp. Operating Systems Principles, SIGOPS, 1979, pp. 117–127.
13. Jones, A.K., and Schwans, K. TASK forces: Distributed software for solving problems of substantial size. 4th Int. Conf. Software Eng., SIGSOFT, 1979, pp. 315–330.
14. Jones, A.K. Protection in programmed systems. Ph.D. Th., Carnegie-Mellon Univ., Pittsburgh, Pa., 1973.
15. Lauer, H.C., and Needham, R.M. On the duality of operating system structures. Proc. 2nd Int. Symp. Operating Systems, IRIA, 1978; Reprinted in *Operating Syst. Rev. 13,* 2 (April 1979), 3–19.
16. Levin, R. Program structures for exceptional condition handling. Ph.D Th., Carnegie-Mellon Univ., Pittsburgh, Pa., June 1977.
17. Liskov, B., and Snyder, A. Structured exception handling. Lab. for Computer Science, M.I.T., Cambridge, Mass., March 1979.
18. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM 15,* 12 (Dec. 1972), 1053–1058.
19. Raskin, L. Performance evaluation of multiple processor systems. Ph.D. Th., Carnegie-Mellon Univ., Pittsburgh, Pa., Aug. 1978.
20. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Comm. ACM 17,* 7 (July 1974), 365–375.
21. Saltzer, J.H., and Schroeder, M.D. The protection of information in computer systems. *Proc. IEEE 63,* 9 (1975), 1278–1308.
22. Schroeder, M.D., Clark, D.D., and Saltzer, J.H. The Multics kernel design project. Proc. 6th Symp. Operating Systems Principles, SIGOPS, 1977, pp. 43–56.
23. Sutherland, I.E., and Mead, C.A. Microelectronics and computer science. *Sci. Amer. 237,* 3 (Sept. 1977), 210–229.
24. Swan, R.J. The switching structure and addressing architecture of an extensible multiprocessor: Cm*. Ph.D. Th., Carnegie-Mellon Univ., Pittsburgh, Pa., Aug. 1978.
25. Swan, R.J., Bechtolsheim, A., Lai, K., and Ousterhout, J.K. The implementation of the Cm* multi-microprocessor. Proc. AFIPS 1977 NCC, Vol. 46, AFIPS Press, Arlington, Va. 1977, pp. 645–655.
26. Swan, R.J., Fuller, S.H., and Siewiorek, D.P. Cm*—A modular, multi-microprocessor. Proc. AFIPS 1977 NCC, Vol. 46, AFIPS Press, Arlington Va., 1977, pp. 637–644.

Operating Systems

R. Stockton Gaines Editor

Experience with Processes and Monitors in Mesa

Butler W. Lampson
Xerox Palo Alto Research Center

David D. Redell
Xerox Business Systems

The use of monitors for describing concurrency has been much discussed in the literature. When monitors are used in real systems of any size, however, a number of problems arise which have not been adequately dealt with: the semantics of nested monitor calls; the various ways of defining the meaning of WAIT; priority scheduling; handling of timeouts, aborts and other exceptional conditions; interactions with process creation and destruction; monitoring large numbers of small objects. These problems are addressed by the facilities described here for concurrent programming in Mesa. Experience with several substantial applications gives us some confidence in the validity of our solutions.

Key Words and Phrases: concurrency, condition variable, deadlock, module, monitor, operating system, process, synchronization, task

CR Categories: 4.32, 4.35, 5.24

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

A version of this paper was presented at the 7th ACM Symposium on Operating Systems Principles, Pacific Grove, Calif., Dec. 10-12, 1979.

Authors' present address: B. W. Lampson and D. D. Redell, Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, CA 94304.

© 1980 ACM 0001-0782/80/0200-0105 $00.75.