ROSCOE: A MULTI-MICROCOMPUTER OPERATING SYSTEM

by

Marvin H. Solomon

and

Raphael A. Finkel

Computer Sciences Technical Report #321

May 1978

ACKNOWLEDGEMENTS

The authors would like to acknowledge the assistance of the following graduate students who have been involved in the Roscoe project: Jonathan Dreyer, Jack Fishburn, James Gish, Frank Horn, Michael Horowitz, Wil Leland, Hossein Malek, Paul Pierce, Ronald Tischler, and Milo Velimirovic. Their hard work has helped Roscoe to reach its current level of development and will be essential in completing its design and implementation.

Roscoe: A Multi-microcomputer Operating System

Marvin H. Solomon

Raphael A. Finkel

Technical Report 321

Abstract

The Roscoe project at the University of Wisconsin is an experimental network of microcomputers running a common operating system. The purpose of the project is to develop techniques and software needed to create a distributed system that can run on loosely coupled identical processors without shared memeory. This paper gives an overview of Roscoe, introducing some of the concepts on which it is based, and discussing some problems currently under investigation.

ROSCOE: A MULTI-MINICOMPUTER OPERATING SYSTEM

1. INTRODUCTION

Parallelism in computer architectures has taken many forms. At one extreme are architectures using tightly coupled, special-purpose functional units responsible for the high performance of computers like the CRAY-1. At the other extreme are very loosely coupled, general-purpose machines that form a community like the ARPANET, spread across large physical distances. The Roscoe project investigates a middle ground: a network of identical, physically close, small, general-purpose machines. The goal of such networks is to create a computer resource that can serve many users simultaneously and will devote as much available power as necessary and appropriate simultaneously to each task. Individual users are divorced from the physical realization of the computer; the size of the network affects only the throughput, not the design, of individual programs.

The technical problems that this research must attack are twofold. First, suitable topologies for interconnecting large numbers of small machines must be found. Constraints of simplicity and generality demand that the connection strategy be as uniform as possible and that the local complexity of the network be small. Second, the operating system must be designed and implemented so that the component machines are organized in a useful way. Each of these problems can be addressed to a large extent independently of the other. Since it is quite expensive at

current hardware prices to build a network large enough for interconnection problems to become acute, these problems are best investigated by analytic models and computer simulations. On the other hand, operating system design issues can only be evaluated by actual experience on real machines. Methodologies can be developed on a small pilot system (on the order of five machines) and extrapolated to larger systems.

In this paper we begin by classifying current computer network research, in order to establish our terminology and identify how Roscoe fits into this classification. Next, we survey current research, showing how it fits into our classification. Finally, we describe Roscoe itself, indicating its relation to other work.

2. A TAXONOMY OF PARALLELISM

Several excellent general discussions of recent software developments based on parallel architectures are available in the literature; see, for example, [Whitby-Strevens 1976] and [Baer 1976]. We begin by distinguishing between tightly and loosely coupled functional units. Large scientific computers, such as the CRAY-1 [Baskett 1977], the CDC Cyber series and the Illiac IV, owe much of their speed to internal parallelism. Functional units operate simultaneously and to a great degree independently. However, the entire machine executes a single instruction stream, and therefore the functional units are tightly coupled in their interactions. The other extreme is typified by the hosts on the ARPANET, where completely independent computations are the rule

and interactions are the exceptions. In short, tightly coupled architectures are usually considered to form a single machine, whereas loosely-coupled ones form many machines. The following terms will only apply to architectures that are not tightly coupled.

A centralized architecture is one in which operating system functions for a community of processors are concentrated in one central processor. If the management of tasks is spread among all the processors, the network is termed distributed.

If the network is used as a device for allowing users to communicate with distant machines, but they must follow the local conventions of that host's operating system, then the network is properly speaking a computer communications network. If the entire ensemble of machines appears to the user as a single entity, and the physical nature of the machines is transparent to the user, then the network is called a computer network or a multi-processor.

All the machines in a homogeneous network are identical. Otherwise the network is heterogeneous. Likewise, a network may use identical machines for specialized functions, in which case the network is functionally heterogeneous.

Close networks have all their machines near each other, in the same room or rack. Distant networks have machines geographically distributed, perhaps across thousands of miles.

Almost all distributed networks use packet switching to accomplish message transfer. Large messages are split into shorter packets, each of which travels through the network to be

reassembled at the destination. In contrast, telephone networks traditionally use some form of circuit switching: the pathway from source to destination is reserved during the course of the connection.

A closely related issue is whether communication is achieved through explicit messages, shared memory, or some combination of both.

A last difference involves the components used in the network. Most network designs employ significant amounts of special equipment designed specifically for the network. Other designs use more standard components.

3. ADVANTAGES OF NETWORKS OVER LARGE MACHINES

Development of networks, especially locally distributed computer networks of minicomputers, has four major expected advantages. First, networks allow modularity. This feature not only helps to organize software, but also allows for easy incremental upgrading of capacity.

Reliability is another special potential of computer networks. Redundancy, both of computation and data storage, is often natural to networks. If part of the hardware or software should fail, other copies of important information can be made available and the surviving hardware can recover from the failure. At the worst, failing hardware can destroy only local computation, while independent computation can proceed unhindered. Reliability has been one of the goals of several distributed computer networks ([Mills 1976], [Ornstein 1975]).

Perhaps the most important feature of computer networks is their potential for speed, either measured by response time or by total system throughput. In order to get quick response, tasks must be subdivided into portions that can be simultaneously executed on independent machines. Since the individual machines may not be very fast, the appearance of speed is gained through appropriate use of parallelism. High overall throughput can be achieved by simultaneously running many independent tasks (such as separate users) on individual machines that only cooperate with respect to shared or limited resources.

The last advantage is cost. Large number-crunchers have not become cheaper (although their power has certainly increased over the past years). However, minicomputers and individual logic chips are becoming less expensive. The result is that large networks of minicomputers do not cost more than individual large scientific computers.

4. RESEARCH IN NETWORK OPERATING SYSTEMS

Although the field of networking is fairly young, many various types of networks have already been investigated. In this section we briefly summarize some of this work and discuss how it fits into the classification developed earlier.

The most famous project to date has been the development of the Arpanet, a heterogeneous computer communications network that has more than fifty member computers across the United States and Europe. This project has encountered and solved many of the interesting problems stemming from packet switching, particularly

discovery of lost messages, duplication of messages, and messages that arrive out of proper order. This work has been aimed primarily at low-level IMP-to-IMP protocols (An IMP is an interface message processor), and much less attention has been given to host-to host protocols that allow many computers to be used as one large resource. To an extent this failure is due to the wide variation of constituent hosts. One notable effort at creating a computer network (as opposed to a computer communications network alone) from the Arpanet is the Rsexec project, as described in [Thomas 1973]. This system of programs spans several hosts, all running the same basic operating system, Tenex. The principal feature of Rsexec (and McRoss, which uses Rsexec) is that it provides a file system that allows the user to refer to a file without needing to know on which of the hosts it resides. The existence of the network is not completely transparent, however, as the user must specify at which sites copies of files are to be stored. Distributed file storage with the possibility of multiple copies gives rise to problems of consistency and protection that are still unsolved.

Many other networks of computers with properties similar to the Arpanet exist. For example, the Telenet is based on a practically identical scheme. The Aloha network in Hawaii is special in its use of radio as one of the communications media. Problems of contention for the scarce radio resource introduce interesting complexity in this kind of network. Another network in which contention for the solitary communications link is important is the ETHERNET at Xerox Palo Alto Research Center. Here many

identical minicomputers are attached to a single high-speed cable. This network is used for sharing expensive resources such as a printer and mass storage; the overall traffic on the ether turns out to be well within its bandwith. A distributed operating system could make use of such a topology, but that is not the current use of the network; rather, each computer serves a solitary user directly connected through a keyboard, and tasks are never given to currently inactive processors.

An important attempt to use many minicomputers in an organized whole is the C.mmp project at Carnegie-Mellon. (See, for instance, [Wulf 1974].) Here up to 16 PDP-11 minicomputers and up to 16 memories are connected together in such a way that any processor can access any memory. All interprocessor communication is accomplished through shared memory. This architecture requires a very expensive crossbar switch, which becomes more expensive quadratically with the size of the network. In addition, it turns out that up to a large number of the memory references result in contention, forcing one processor to wait while another is serviced. For this reason, the expected performance of the network is not very close to the theoretical optimum, in which all processors and all memories are fully active. The expense and special hardware required for the connection are also severe drawbacks.

If the application for which the network is intended is well-enough understood, it is possible to make a special-purpose network with very high performance. A good example of this approach is the Pluribus machine. (See [Ornstein 1975].) The

principal goals of this network are to provide extreme reliability through redundancy and mutual suspicion, and at the same time provide adequate throughput for a time-critical application. (The principal application of the Pluribus machine is message switching in the Arpanet.) The individual processors are all identical, and a simple bus architecture connects processors and memories. Some common memory is used for communication, but in case of failure a consensus of the machines can press a new area of memory into service for this task. The application program is encoded in independent strips, each of which takes a short time to execute. Each task is then accomplished by assigning component strips to available processors. Reliablility is enhanced by the fact that each strip first checks to see if its actions are appropriate in the current context; no harm results from invoking a strip accidentally. The allocation of strips to processors depends on a special hardware device called a Pseudo Interrupt Device, but most of the other hardware in the Pluribus is standard. Although the performance of this machine is impressive, much of it depends on the careful hand-encoding of the particular application into strips. It is not clear how generally applicable this technique is, nor whether it can be automatically done by a compiler.

A sequel to the C.mmp project is currently underway at Carnegie Mellon. Cm* is a network of clusters, or computer modules. Each cluster contains up to 14 microcomputers (LSI-11s) under the control of a special-purpose microcoded processor known as the K.map. Each microcomputer has its own special hardware as well,

called the S.local. The function of the K.map and the S.local is resolving memory references. Although each processor has its own local memory, the logical memory for the ensemble is composed of all the individual memories. Each memory reference must be decoded and directed to the appropriate place. If the reference is local, the S.local will direct the request to the local memory. If the reference can be found elsewhere in the same cluster, the K.map redirects the reference. If another cluster must be accessed, the K.map routes the request (perhaps indirectly through other K.maps) to the right cluster. If most references are in fact local, then the overhead is not severe. However, the more distant the references are, the farther the information must travel. A program loop that continually requires distant data will run very slowly indeed. This architecture is a hybrid: It uses message transfer for communication on the physical level, but employs shared memory on the logical level for all operating system functions.

An extremely important contribution to the field of distributed computer networks has been made by the Distributed Computer System (DCS) at the University of California at Irvine. (See [Farber 1972, 1973].) Several computers (on the order of five) reside on a high-speed communications ring, something like the Ethernet. There are two ways in which DCS differs from the Ethernet: the computers are heterogeneous, and more important, a single operating system links them all. The kernel of this operating system resides in each individual processor, and its primary purpose is to deliver messages among the processes both

on its machine and on other machines of the network. The file system is distributed among all the machines. In order to read a file, a process sends a message to the catalogue process, which finds the correct physical volume process. The physical volume process creates a new process for the desired file, and the requesting process then carries on all negotiations with this file process. This strategy makes the physical location of files invisible to the user and all user programs. In addition, since all mail is directed to processes, not machines, it is not necessary to keep track globally of process-processor bindings.

The ring architecture allows resource sharing based on bidding. Messages may be broadcast to every processor soliciting bids for a desired resource. The customer for the resource can then choose the processor that offers the best price. The bids are based on such considerations as scarcity of the resource, current demand, and current load on the bidding processor. The ring structure may be used to enhance reliability. If any processor should malfunction, that error cannot damage the connection among all the other machines. If several other processors notice consistent malfunction in one of the ring processors, they can cause it to be disconnected from the ring. This amputation is similar to the action that can be taken in the Pluribus architecture to remove a failing component. However, if a ring connection fails, it could disconnect the entire network, since each connection must actively retransmit all information that travels through it.

Using a ring mechanism has a price. As the number of proces-

sors increases, the contention on the ring becomes more severe. Some special-purpose hardware is necessary to provide the interface between the ring and each processor. This hardware contains a small associative memory to determine if an incoming message is intended for a process on the local machine. A subtler failing of a ring mechanism is that it encourages the use of broadcast messages. These messages must be considered by each processor. In a very large network, it might cause unacceptable overhead to require that every processor receive such messages; they begin to appear like junk mail. In a large network in which the cost of a message varies with the distance it must travel (unlike the situation in the ring), a more local broadcast might be less costly to perform and more useful in its yield.

The Distributed Computer Network (DCN; see [Mills 1976]) is a close relative to Roscoe. This network contains about five PDP-11s of various sizes. They are interconnected by a variety of data paths with a variety of characteristics. A single operating system connects the entire network. All communication is performed by sending messages between processes, and as with DCS, these messages are sent to ports, several of which may be owned by any process. The file structure is very much like that of DCS, except that once a file is opened, it is read and placed in the logical address space of the requesting process. (Four of the processors have memory management.) The network has several strategies for error recovery. These strategies are also used to re-establish connections with processes that migrate (which happens only on explicit user request). The operating system is

distributed much like that of DCS. Each processor (called a hostel) has a resident program in charge of message buffering, process scheduling, and interrupt management. Individual "system" processes are provided for all file and device manipulation. One of these system processes, the stream demon, makes sure that files are read as necessary when processes refer to them in their logical memory. The Distributed Computer Network does not provide dynamic resource allocation, in the sense that DCS sends out requests for bids. Each hostel operates independently of the others except when required to receive or send a message. No dynamic load balancing is attempted. Since the network is not regular (it is neither a ring nor any particular pattern), each processor must contain routing tables that associate not only processes with hostels, but also associate routes with hostels.

The Micronet project at SUNY-Buffalo (see [Wittie 78]) is another relative of Roscoe. On the order of 20 LSI-11 microcomputers are being assembled into a network. A special-purpose frontend microcomputer handles communications in the network; each machine-frontend pair is connected to at most two high-speed busses. The operating system makes communication and file transfer appear very similar from the point of view of the user program. This project is still in the formative stage, so it is not clear how the operating system will make use of the communications paths.

The Unix operating system is being modified (see [Chesson 75]) to introduce a networking feature. The user can specify on what machine any program should execute. A communications link

appears like a Unix pipe, which can be read from or written to in the same manner as a file. If many processes all have entry to the same group, as this link is called, then whatever is written by any can be read by the others. There is no way to select messages from the group; any read operation receives all current messages from the group multiplexed together. Some files in the directory are entries into groups. Any process that has permission to write in a special group file can request permission to join the group. The group owner can grant or refuse that permission. Much of the effort in network Unix has been directed to making inter-machine data flow as efficient as possible and to keep such communication as much within the spirit of Unix as possible.

An important contribution to message-based program design is the module concept of Plits, as described in [Feldman 1977]. Each message appears somewhat like a procedure call, with formal arguments bound to actual arguments at the time the message is constructed. Each argument fits into a slot, whose name is mentioned by the caller. These names are not generally known, and therefore only those processes that know the slot names can read the message. If the message passes through several intermediate hands before arriving at its destination, it is safe from tampering, since the intermediaries will only look at the slots that pertain to routing, and will not even be aware of slots that do not concern them.

5. THE ROSCOE OPERATING SYSTEM

The Roscoe project investigates operating systems for physically and functionally homogeneous, close, distributed computer networks. We do not want to use a simple ring structure such as DCS ([Farber 72, 73]), because it cannot be extended without introducing increased contention. Furthermore, we wish to use standard available equipment as much as possible, so that our efforts can be directed toward software, and not hardware, design and implementation. The goal of the Roscoe network is to provide a computation resource in which individual resources such as files and processors are shared among processes. Our current work is centered around construction of an operating system that allows a cluster of initially five LSI-11 computers to provide such a computation resource. Although we intend to produce a working prototype system, the design is strongly influenced by the idea that Roscoe is a testbed for techniques that should be applicable to much larger collections of processors.

It is clear that distributed computing offers the potential for extremely cost-effective implementations of large operating systems. But before this potential may be realized, many problems of software design must be overcome: transmission and routing of messages, sharing of resources (including the processor resource and the file system), protection and recovery from failure.

In Roscoe, communication among processes is implemented exclusively by explicit messages. The decision not to use logical

or physical sharing of memory for communication is influenced both by the constraints of currently available hardware and by our perception of cost bottlenecks likely to arise as the number of processors increases. Physical sharing leads to complicated crossbar switches, whose cost and complexity would be prohibitive for large numbers of processors. Logical sharing hides the cost of communication between physically distant processes. Explicit message passing is used by several other projects, including Plits [Feldman 77] and Micronet [Wittie 78].

5.1 Processes

The fundamental unit of execution is a process. Each processor runs one or more processes. Each process is constrained to run on one processor at a time, but it may migrate to another. The purpose of this migration is to ease contention for message pathways and to allow a heavily-loaded processor to divest itself of some of its load to neighboring machines. Algorithms for load-balancing are a major goal of this research. Since the LSI-11 does not have dynamic address translation, swapping of processes onto secondary store and migration of processes are restricted in that the core image must eventually be placed into memory at exactly the addresses from which it came (although potentially on a different machine). Special hardware (like the S.local of the Cm* project or revisions to the LSI microcode) may alleviate this restriction in the future. Another possibility is to translate user programs into a location-independent intermediate code with an interpreter resident in each machine within the Roscoe net-

work.

Migration introduces some difficult problems. In fact, some researchers consider it infeasible; [Feldman 1977] calls it one of the "current fantasies of distributed computing." It must be possible to direct messages to a process that has moved. If forwarding tables are kept, then they will eventually overflow. This catastrophe can be postponed by distributing "change-of-address" notices. Since process-host associations will eventually become unreliable (and may be destroyed due to a component failure), messages may be forced to actively seek their destination by wandering through the network. Some research into this concept is currently underway at Xerox PARC (E. McCreight, personal communication). One result is the "Flying Dutchman" problem, in which the destination process has in fact disappeared, and the message that is seeking it is doomed to wander forever.

5.2 Messages

All communication among processes takes the form of messages. The concept of ports, as defined in DCN [Mills 1976] and also employed by Farber [Farber 1972, 1973], allows each process to distinguish the various messages it might receive. This idea has been recently combined with the notion of capabilities [Fabry 1974] in the Demos operating system designed by Baskett for the CRAY-1 computer. (See [Baskett 1977].) This operating system employs links, which are both permissions and pathways for messages. Only the recipient process (called the owner of the link) can create a link pointing to itself. This link can then be

presented to another process (across another, pre-existing link) to enable the second process (called the holder of the link) to direct messages to the first. As a process creates a link, it specifies which channel (analogous to Mills' port) incoming messages across that link will enter. The receive command specifies a set of channels that the receiving process wishes to become sensitive to. (It is interesting to note that Feldman's Plits language and the Thoth microcomputer project [Cheriton 1977] both follow similar policies that allow a program to enable receipt of any of a set of messages.) The link concept provides a means of protection, since a process will only be able to send messages to processes that have given permission. (The Plits language, in contrast, allows messages to be directed to any process whose name is known. This set always includes those processes from which messages have been received.) Links come in several flavors, which enhance the protection. For example, a reply link may only be used once; it is a permission to send exactly one message. The owner of a link can specify certain restrictions that will apply to any holder of that link. The holder can be prevented from duplicating the link or giving it away by enclosing it in a message. In addition, the owner can specify that it be notified whenever the holder copies, gives away, or destroys the link. The notification is not forgeable by a user. These notifications allow the owner of a resource to keep track of how many processes hold links to it. When the last of the links is destroyed, the resource is free to be re-used.

We make heavy use of links in Roscoe. One goal of our

research is to investigate the utility of links as a formulation on which to build a robust and capable distributed operating system.

5.3 Flow Control

The object of flow control is to allocate the scarce system resources of message buffers and use of inter-machine links in such a way that no process is perpetually starved waiting for a message buffer, and the links are not left idle when messages are waiting to be sent. These problems can be solved in a single-machine system by instituting scheduling rules. For example, the Thoth operating system, under which processes communicate exclusively through messages, insists that the sender of a message be blocked until the message is received. In this way, only one buffer need be allocated for each active process; that buffer holds the one message that the process might currently have outstanding.

The problems of allocation of buffers becomes much more difficult when resource control is distributed. Each machine has a pool of message buffers. These may be requested in several ways: A local user may wish to send a message, an external message may arrive for local delivery, and an external message may arrive for forwarding. If the allocator decides not to honor the request for a buffer, the local user is blocked, or the external message is refused. There are two ways that a message buffer can be released: a local user accepts its message, or a neighboring site accepts its message. Until one of these events occurs, the

local operating system must hold on to the buffer.

The task of the flow control algorithm is to decide whether to honor each request for a buffer. If the allocator is too stingy, very little communication will take place. If it is too generous, the buffer supply may be exhausted, blocking all further work. One of the research goals of Roscoe is to investigate buffer allocation strategies.

Some theoretical results can be achieved. The first question is how many buffers are absolutely necessary to prevent a buffer deadlock, which is the situation in which all processes are blocked, but if more message buffers were available, computation could proceed and eventually terminate. This situation must be distinguished from a process deadlock, which in its simplest form consists of two processes, each of which is waiting for a message from the other, with no messages currently sitting in buffers. Here no number of extra buffers can unblock either process.

Even in a very simple case an unbounded number of buffers is necessary. Consider two processes, each of which runs this program:

```
send n messages to the other process.
receive n messages from the other process.
```

Neither process can receive a single message until it has placed n messages in message buffers. This algorithm requires n+1 buffers, n for the messages sent by the first process, and 1 for all the messages that the second might send after the first has finished. This result assumes an optimal scheduling of the processes that allows the first process to send all its messages before starting the second one. A pessimal schedule would have

each send n-1 messages and then finally send the last one. Here 2n-1 buffers would be needed.

Since examples of this form can be made arbitrarily bad, it is hopeless to provide enough buffers for all possible situations. However, there is some hope that an allocation scheme exists that will avoid deadlock in those cases where there are enough buffers. Even if there are plenty of buffers, allocation decisions can be made that cause deadlock. In particular, if we decide that each of the two processes above must only engage one buffer at a time for sending, then even if plenty of buffers are available, the processes will reach deadlock if n is greater than one.

Let us avoid the problem that the previous example caused by prohibiting a second message from being sent between the processes until the first is received. We will call this rule the propriety restriction. The algorithm still has a deadlock, but now we will call it a process deadlock, since the process failed to run a proper program. Now the worst case that can be constructed for p processes is this:

```
send a message to each other process
receive all messages
```

The optimal schedule in this instance turns out to let process 1 send all its p-1 messages, enabling it to receive all messages without queueing them. Then let process 2 send all its p-1 messages. One of them is accepted immediately by process 1, and the other p-2 must be queued. Then process 2 can accept the message awaiting it from process 1. Each process in turn is then allowed to send all its messages. The ith process will send p-1 mes-

sages, i-1 of which are immediately accepted (therefore requiring a total of only one transient buffer), and p-i of which must be queued and require buffer space. Then the ith process can accept the i-1 messages awaiting it. The execution of the ith process causes an increase in the number of buffers needed of

$$(p-i) - (i-1) = p - 2i + 1.$$

The number of buffers in use after k processes have completed is

$$\text{summation}_{1 \leq k \leq 1} (p - 2i + 1) = k\,(p-k).$$

This expression reaches a maximum after half the processes have been scheduled, when about $p^2/4$ buffers are needed. A pessimal schedule will allow each of the processes to send p-2 messages before any process may send the last message. In this case, almost p^2 buffers are needed.

The problem is compounded if processes can selectively receive messages. For example, a Plits process can restrict its attention to incoming messages relating to a particular transaction. A Thoth process can restrict its attention to incoming messages from one other process. Roscoe processes (and Demos processes) can select some combination of 16 channels to activate for message reception. Thus an algorithm can be written in which a process demands input on channel 1 before it will relieve the system of the message currently waiting on channel 2.

In this case, the propriety restriction can be recast to require that a process cannot send a message along a channel to a destination until all previous messages on that same channel to that same destination have been received. Now a very bad user program is this:

```
send a message on each channel to each other process
for each channel do
     receive p-1 messages on that channel
```

The number of required buffers approaches 16 p^2, assuming that 16 channels are available. The situation in Roscoe cannot get quite that bad, since the number of links held by any one process is limited (at the moment, to 20). However, Roscoe does not enforce the propriety restriction.

One other source of complexity is the fact that a free buffer on one machine does not necessarily help a buffer depletion on a neighboring machine. The buffer pool is local to each machine, and even if enough system-wide buffers exist, problems can arise. One can imagine flow control algorithms that try to relieve local buffer depletion by remote storage to other machines.

Since an enormous number of buffers is needed to begin to solve the problem properly, some other approach must be found. Plits makes sure that one buffer is available for each possible communication path, but strongly restricts the number of such paths. In Roscoe, we cannot afford enough buffers, and we do not want to restrict user programs to the extent necessary to be able to afford buffers. Therefore, the buffers that are available must be carefully allocated to try to prevent buffer deadlock in the common cases, and only to infinitely block user programs that are behaving unreasonably (by sending many messages to recipients that are unwilling to receive them).

When buffers are plentiful, practically any request should be honored. When buffers become scarce, some requests should be denied. We expect the research into proper denial strategies to

be quite fruitful. One simple idea that may prove worthwhile is when buffers are scarce to refuse to accept any messages over external links that are not directed to a process currently waiting for a message on the channel of the incoming message. Another possible heuristic is to prevent local processes from sending any local message that will have to be queued or any foreign message that the neighboring machine is willing to take immediately.

5.4 Files

DCS, DCN, and Demos all use interprocess communication to manage files. This approach seems the only reasonable one in an environment where interprocess communication is the fundamental operation and data may reside on a distant machine. Unlike DCN, we do not map the files into logical memory, principally because the LSI-11 does not have the necessary memory management facility. Unlike Demos, we do not use data windows in links to access distant memory directly. Not only does such usage lead to protection problems, it also is difficult to implement on a multiprocessor machine.

Operating systems like Multics and Unix that have multileveled hierarchical file structures have shown that this organization has extremely nice properties for the user, who can arrange personal files according to logical groupings. We arrange Roscoe files in a similar fashion, so that directories are files that can be read like other files, with all files ultimately linked at the root of the directory.

However, it is desirable to place the physical disk drives on

various machines scattered throughout Roscoe. In this way, requests for file access do not have to be funnelled through one processor, tying up all local link bandwith in transferring file contents. The problem is to coordinate use of files on different machines so the user sees one universal file system capable of opening any file, no matter where it might be located.

If the user program has no control over where files are stored, then random fluctuations will eventually place some often-needed files quite far physically from the machine in which a user process is running. This situation argues for some user control. However, any reasonable algorithms that can be found for the allocation of file space with respect to the location of the frequent users ought to be embedded in the part of the file system that the ordinary user does not control. Then there is no danger that the user will persistently make poor choices of placement, to the detriment of all other users of the network.

Every machine that has a disk must have a device driver for it. A file system process exists in each such machine as well. This file system process knows the full path name of each file in the local disk. If a local user requests a file that resides on that disk, the file system process can open the file and provide the user program with a file access link that it can use to read and write the file. The user program sends read/write requests on that link and encloses a reply link that the file access process can use to return data and success codes. When the user program wishes to close the file, it destroys its link to the file access process, which receives a notification that the link

was destroyed and closes the file.

If a local process requests a file that is higher in the file hierarchy or in another branch, then the local file system must pass the request on to a distant file system process that may be able to help. One such distant process is the owner of the root of the whole directory. If the local file process has a link to this root, then it is easy to relay the request to it. However, consistent use of this strategy will result in high use of the root process, which may cause bottlenecks. A different approach is for the local file process to pass the request to the file process that owns the parent disk, in whose directory the entire local disk volume resides. Either that process can satisfy the request or it can relay the request yet another level up the hierarchy.

The inverse problem occurs when a user requests a file that sits below the local disk: one of the directories in the path is on the local disk, but the path extends to another disk. In this case, the local file process should direct the request to the distant file process that owns the child disk that holds that directory.

These considerations imply that each local file process must have links to the parent of its disk and to each of the possible children of its disk. In addition, it must know the full path name of the top-level directory on the local disk. The number of necessary links may be very high. Much of this information may be stored on the disk itself. In particular, each disk has (in a globally known area) a description that includes the path name of

its top-level directory and an entry that can be translated to a link to the owner of the parent disk. Each lowest-level directory that leads to another disk has a special entry that can be translated to a link to the owner of the child disk.

Initialization of this scheme is worth considering in some detail. Initialization must occur not only when Roscoe is first brought up on a machine, but also whenever it is restarted after any failure. Let us assume that a fresh Roscoe has just started to execute on a machine. One of the first jobs is to determine if there is a disk on the machine. If so, a file handler process (whose code must be either part of the Roscoe kernel or easily found on the disk) must be instantiated. This process will read the information in the root of the disk to find the path name and a link to the parent. A message should be sent to the parent file process that informs it that the local files are now available again. The distant file process that receives that message should install the return link as the appropriate child in its disk. Furthermore, any lowest-level directory entries that extend to other machines require that confirmatory messages be sent so that the file system becomes linked together properly. This scheme should be able to survive unplugging a disk from one machine and installing it on another.

Other aspects of distributed file systems will also be investigated by Roscoe. A method of access hints, as suggested in [Lampson 74], may make some directory searches much faster, since fewer messages would have to be sent. The situation under which a new disk pack is initialized must be considered, as must the

situation when a disk pack runs out of room. Standard operations such as checking the disk for consistency must also be designed; to an extent they can be modelled after the Unix pattern. Roscoe machines that do not have disks must still have access to the file system. One method would be to use the nearest file process that can be found. In this case, some arrangement must be made to find that file process.

Another issue is restricting rights to files. Most modern operating systems associate protection information with each file to indicate which users are to have what rights to that file. Typical rights include read, write, and execute access. The fundamental protection mechanism in Roscoe is the link, which is intended to be given selectively only to processes that are to have the type of access that link provides. In order to mesh the link mechanism with file protection, we need to be able to prevent indiscriminate distribution of file-access links. The question is where the responsibility should lie for making the decision. If the file process is to make the decision, then it must know something about the requesting process, which would violate the Roscoe philosophy that insists that no process know any identifying feature of any other. If the requesting process is expected to identify itself in some manner to the file handler, then there is the danger that it might forge a name and acquire illegal privileges. The kernel might append an identifying source on each message in such a way that a recipient can only compare it with other sources, but now if the file process must relay the request to another file process elsewhere, the source field has

been obliterated. The reply link supplied by the user may contain identification placed there by the kernel, which would avoid that problem.

An alternate approach is to have the file process always form the appropriate file access link and include it in the response. It also will include conditions under which this link may actually be given to the recipient process. The kernel that governs the requesting process can check those conditions against the identity of that process and possibly destroy the link instead of giving it to the user. The user sees a failure return and cannot distinguish between the file not existing and permission being denied to access it. The file system sees the destruction of the link as if the user had closed the file. This scheme will work even if the request must be relayed several steps before a file process is found that can respond to it, because the response travels directly back to the original requestor. However, it causes files to be opened and immediately closed again.

The problem of contention for files gives rise to some interesting research questions. If files are duplicated, then how can updates be performed? If files are not duplicated, then how can many users read simultaneously without overloading the network locally? Some of these issues have been discussed in [Lampson 76].

6. CURRENT STATUS

Roscoe is being implemented on five LSI-11 machines. Each has 28K words of memory, a programmable clock, extended instruction set, a serial link (intended for the console teletype), and parallel-word links to one or more other LSI-11 machines. In addition, each LSI-11 has a parallel-word link to a PDP-11/40 running Unix. The serial links are directed to a manual switch that can connect any one of them to the Unix machine.

All software for Roscoe is being prepared under Unix. User programs are written in C (the Unix systems implementation language), as is most of the kernel. A very few parts of the kernel (currently less than 30 lines of code) are written in assembler; these pieces deal with interrupt and user call dispatching and stack modifications necessary for scheduling. A modification has been made to the C compiler so that procedure entry code checks stack limits. In this way, bugs caused by stack overflow (which is not detected by the hardware) can be discovered and corrected.

A prototype kernel has been implemented. It has four functions. It manages links for each user, allowing new links to be made, messages to be sent and received, and links to be destroyed. It schedules the users, employing a non-preemptive technique in which a user process is only blocked if it tries to receive a message that is not yet available or to send a message when no buffers are available, or if it explicitly yields control or dies. It provides a relocating loader that can accept a load

module from the PDP-11/40 and load it into memory. Finally, the kernel routes messages along the parallel-word interfaces to other machines and accepts messages from other machines.

Several utility programs have been written that operate like user processes and communicate via links. One is a teletype handler, and another is a primitive file system that treats the Unix machine as a backing store.

7. SUMMARY AND CONCLUSIONS

Roscoe is an operating system that allows a cluster of five LSI-11 computers to co-operate in providing a computing service similar to that provided by a single, much more expensive, machine. Although we intend to produce a working prototype system, the design will be strongly influenced by the idea that Roscoe is a testbed for techniques that should be applicable to much larger collections of processors. As we have indicated above, we borrow extensively from operating system concepts recently advanced by other researchers, with the goal of applying their ideas to the problem of creating a multi-microprocessor based on communication through message-passing rather than shared memory. Before the potential of distributed computing can be realized, many problems of software design must be overcome. Some of these are detailed above: sharing of resources, including the processor resource and the file system; transmission and routing of messages; and protection and recovery from failure. No doubt other unforeseen problems will arise in the course of implementation. Further research associated with this project will include

adaptation of specific problems to the structure of our system; other researchers are investigating the possibility of a multiprocessor-based compiler, automatic partitioning of problems into communicating processes, and the problems of managing a distributed data base.

REFERENCES

Anderson, G. A. and Jensen, E. D., "Computer Interconnection Strategies: Taxonomy, Characteristics, and Examples", Computing Surveys, Vol. 7, No. 4, pp. 197-213, Dec., 1975.

Arden, B. E., Berenbaum, A. D., "A Multi-Microprocessor Computer System Architecture", Proceedings of the Fifth Symposium on Operating Systems Principles, (In Operating Systems Review, Vol. 9, No. 5) pp. 114-121, November, 1975.

Baer, J-L., "Multiprocessing Systems" IEEE Transactions on Computers, Vol. C-25 No. 12, pp. 1271-1277, December, 1976.

Baskett, F. and Keller, T. W., An Evaluation of the CRAY-1 Computer, Los Alamos Scientific Laboraroty Reprot, 1977.

Baskett, F., Howard, J. H., Montague J. T., "Task Communication in Demos", Proceedings of the Sixth Symposium on Operating Systems Principles, pp. 23-31, November 1977.

Beneš., V.E., Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, 1965.

Cheriton, D. R., Malcolm, M. A., Melen, L. S., and Sager, G. R., Thoth, a Portable Real-Time Operating System, Report CS-77-11, University of Waterloo Computer Science Department, October, 1977.

Chesson, G. L., "The Network Unix System", Proceedings of the Fifth Symposium on Operating Systems Principles, (In Operating Systems Review, Vol. 9, No. 5) pp. 60-66, November, 1975.

Enslow, P. H., "Multiprocessor Organization -- A Survey", Computing Surveys, Vol. 9, No. 1, pp. 103-129, March, 1977.

Enslow, P. H., Multiprocessors and Parallel Processing, John Wiley and Sons, New York, 1974.

Farber, D. J., Heinrich, F. R., "The Structure of a Distributed Computer System -- The File System", Proceedings of the International Conference on Computer Communications, pp. 364-370, October, 1972.

Farber, D. J., Feldman, J., Heinrich, F. R., Hopwood, M. D., Larson, K. C., Loomis, D. C., Rowee, L. A., "The Distributed Computing System", Proceedings of the Seeventh Annual IEEE Computer Society International Conference, pp. 31-34, February, 1973.

Feldman, J. A. A Programming Methodology for Distributed Computing (among other things) TR9, Computer Sciences

Department, University of Rochester, September 1976.

Finkel, R. A., Solomon, M. H., Processor Interconnection Strategies, University of Wisconsin -- Madison Computer Sciences Technical Report #301, July 1977.

Jones, A. K., Chansler, R. J. Jr., Durham, I., Feiler, P., Schwans, K., "Software Management of Cm* -- A Distributed Multiprocessor", Proceedings of the National Computer Conference, Vol 46, pp. 657-663, AFIPS Press, 1977.

Kleinrock, L., "On Communications and Networks", IEEE Transactions on Computers, Vol. C-25 No. 12, pp. 1326-1335, December, 1976.

Lampson, B. W., An Open Operating System for a Single User Machine, Xerox Palo Alto Research Center report, 1974 (approximately).

Lampson, B. W., Sturgis, H. Crash Recovery in a Distributed Storage System, Xerox Palo Alto Research Center reprot, 1976 (approximately).

McQuillan, J. M. "Graph Theory Applied to Optimal Connectivity in Computer Networks," Computer Communication Review (SIGCOMM), 7,2 April, 1977.

Mills, D., "An Overview of the Distributed Computer Network", Proceedings of the National Computer Conference, Vol. 45, pp. 523-531, AFIPS Press, 1976.

Ornstein, S. M., Crowther, W. R., Kraley, M. F., Bressler, R. D, Michel, A., and Heart, F. E. "Pluribus -- A Reliable Multiprocessor," Proceedings of the National Computer Conference, Vol. 45, AFIPS Press, 1975.

Ritchie, D. M., Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM, Vol. 17, No 7, pp. 365-375, July 1974.

Swan, R. J., Fuller, S. H., Siewiorek, D. P., "Cm* -- A Modular Multi-processor", Proceedings of the National Computer Conference, Vol. 46, pp. 637-644, AFIPS Press, 1977.

Swan, R. J., Bechtolsheim, A., Lai, K-W., Ousterhout, J. K., "The Implementation of the Cm* Multi-Microprocessor", Proceedings of the National Computer Conference, Vol. 46, pp. 645-655, AFIPS Press, 1977.

Thomas, R. H., "A Resource Sharing Executive for the ARPAnet", Proceedings of the National Computer Conference, Vol. 42, AFIPS Press, 1973.

Whitby-Strevens, C., "Current Research in Computer Networks",

Computer Communication Review, Vol. 6, No. 2, April 1976.

Widdoes, L. C., "The Minerva Multimicroprocessor", Proceedings of the Third Symposium on Computer Architecture, pp. 34-39, 1976.

Wittie, L. D. Efficient Message Routing in Mega-Micro-Computer Networks, State University of New York at Buffalo Technical Report, 1976.

Wittie, L. D., Micronet: A reconfigurable microcomputer network for distributed systems research, State University of New York at Buffalo Technical Report TR 143, April, 1978.

Wulf, W., et al, "HYDRA: The Kernel of a Multiprocessor Operating System," Communications of the ACM 17, 6 June, 1974.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the assistance of the following graduate students who have been involved in the Roscoe project: Jonathan Dreyer, Jack Fishburn, James Gish, Frank Horn, Michael Horowitz, Wil Leland, Hossein Malek, Paul Pierce, Ronald Tischler, and Milo Velimirovic. Their hard work has helped Roscoe to reach its current level of development and will be essential in completing its design and implementation.