THE ROSCOE DISTRIBUTED OPERATING SYSTEM*

Marvin H. Solomon
Raphael A. Finkel
University of Wisconsin -- Madison

ABSTRACT

Roscoe is an operating system implemented at the University of Wisconsin that allows a network of microcomputers to cooperate to provide a general-purpose computing facility. After presenting an overview of the structure of Roscoe, this paper reports on experience with Roscoe and presents several problems currently being investigated by the Roscoe project.

INTRODUCTION

Roscoe [9, 10, 16, 17] is a distributed operating system designed for a network of microprocessors. The goal of the Roscoe network is to provide a general-purpose computation resource in which individual resources such as files and processors are shared among processes and control is distributed in a non-hierarchical fashion.

The essential features of Roscoe are:

1. All processors are identical. Similarly, all processors run the same operating system kernel. However, they may differ in the peripheral units connected to them.

2. No memory is shared between processors. All communication involves messages explicitly passed between physically connected processors.

3. No assumptions are made about the topology of interconnection except that the network is connected (that is, there is a path between each pair of processors). The connecting hardware is assumed to be sufficiently fast that concurrent processes can cooperate in performing tasks.

4. The network appears to the user to be a single powerful machine. A process runs on one machine, but communicating processes have no need to know if they are on the same processor and no way of finding out. (Migration of processes to improve performance is transparent to the processes involved.)

5. The network is constructed entirely from hardware components commercially available at the time of construction (January, 1978).

6. The software is all functional. Although Roscoe has undergone much revision, it has been working for over a year.

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. Message passing is the fundamental form of communication in several other networks, including Plits [7], Micronet [18], Technec [12], and DCN [14].

OVERVIEW

The current Roscoe implementation runs on five Digital Equipment Corporation LSI-11 computers.** Each has 28K words of memory, a programmable clock, extended instruction set, a bit-serial line (intended for a terminal), and word-parallel lines to one or

*This research was supported in part by the United States Army under contract #DAAG29-75-C-0024.

**This equipment was purchased with funds from National Science Foundation Research Grant #MCS77-08968.

more other LSI-11 machines. In addition, each LSI-11 has a word-parallel line to a PDP-11/40 running the Unix operating system [15]. The software is written in the C programming language [13] with the exception of a small amount of code written in assembler language. When Roscoe is running, the PDP-11/40 is not part of the network; its function is to assist in program development and initial loading.

The fundamental concepts of Roscoe are files, programs, core images, processes, links, and messages. The first four of these are roughly equivalent to similar concepts in other operating systems; links and messages are idiomatic to Roscoe.

The Roscoe kernel is a module that resides on all the machines of the network and provides various services for user programs. The services are requested by means of service calls, which appear as procedure invocations to the caller. Other services are provided by utility processes that reside on some machines, but not necessarily all. Library routines are available to facilitate communication with these processes.

Links

The link concept is central to Roscoe. It is inspired and heavily influenced by the concept of the same name in the Demos operating system for the Cray-1 computer [2]. Although the Cray-1 is a conventional uniprocessor, Baskett et al suggest that links might also be used in a multiprocessor environment. Links have proved to be a great success in insulating the writer of Roscoe processes from the peculiarities of a multiprocessor architecture. The demonstration of the usefulness of the link concept for this architecture is a major result of the project thus far.

A link combines the concepts of a communications path and a "capability" [6]. It represents a one-way logical connection between two processes, and should not be confused with a line, which is a physical connection between two processors. Each link connects two processes: the holder, which may send messages over the link, and the owner, which receives them. The holder may duplicate the link or give it to another process, subject to restrictions associated with the link itself. The owner of a link, on the other hand, never changes.

Links are created by their owners. When a link is created, the creator specifies a code and a channel. The kernel tags each incoming message with the code and channel of the link over which it was sent. Channels are used by a process to partition the links it owns into subsets; when a process wants to receive a message, it specifies an explicit set of channels. The process is blocked until a message arrives over a link corresponding to one of the specified channels. A link is named by its holder by a small positive integer called a link number, which is an index into a table of currently-held links maintained by the kernel for the holder. All information about a link is stored in this table. (No information about a link is stored in the tables of the owner.)

The creator may also specify certain properties of the link, for example, that it may not be copied, that it may be used only once, or that its destruction sends a notification to the owner.

Messages

A message may be sent by the holder to the owner of a link. In addition, certain messages, called notifications, are manufactured by the kernel to inform the owner of a link of changes in its status. For example, the creator of a link may ask to be informed when the link is destroyed or copied. Notifications are identified to the recipient by an unforgeable field.

A message may contain, in addition to arbitrary text, an enclosed link, which was previously held by the sender of the message. The kernel adds an entry to the link table of the destination process and gives its link number to the recipient of the message. In this way, the recipient becomes the holder of the enclosed link. If the sender does not destroy its copy, then both processes will hold identical copies of the link.

The holder of a link has no way to name or address the process at the other end. In particular, it does not know whether that process is on the same or a different machine.

Each kernel maintains a pool of buffers that are allocated among all local processes. Incoming messages that have not yet been received by their destination processes are queued in these buffers, as are outgoing messages that have not yet been delivered to a neighboring machine. A simple priority algorithm is used to reduce the chance of buffer deadlock.

Service Calls

The chief function of the kernel is to support links and messages by providing service calls to create and distroy links, and send and receive messages. Additional service calls create and destroy processes, read and set "wall-clock" and high-resolution interval timers, and establish interrupt handlers for processes that control peripheral devices. The service calls that create and destroy processes are normally issued by the resource manager utility process described below.

Utility Processes

Roscoe has been designed so that as many as possible of the traditional operating system functions are provided not by the kernel, but by ordinary processes. The terminal driver is an example. One terminal driver resides on each processor that has a terminal. All terminal input/output by other processes is performed through messages to this process. It understands and responds to all commands accepted by a file (see below), as well as a few extra ones, such as "set modes" (for example echo/noecho, hard-copy/soft-copy).

A file manager process resides on each processor that has mass storage. Each file manager controls a tree-structured directory. A process wishing to open a file sends a message to the appropriate file manager, which creates a link to represent the open file. To the user of a file, the open file behaves like a process that understands and responds to messages requesting read and write operations. The file is closed by destroying the link.

The most interesting utility process is the resource manager (RM). Resource managers reside on all processors and are connected by a network of links. Any process may request its local RM to create a new process. The local RM may create the process on its own machine or relay the request to another RM, based on local considerations such as availability of free memory and the possibility that the required program is already in memory. The next RM decides in a similar way whether to load the process or relay the request.

The new process is started with only a link to its local RM. It can use this link to request links to the process that requested its creation, to a file manager process, to a terminal driver, or to other resources. The RM can kill the process, or it can give a special link to another process (usually a terminal driver) that may be used to kill it.

Processes form a hierarchy in two senses. First, every process is started by another. However, since the RM usually performs the starting, this hierarchy is quite flat. Second, every process is started at the request of another to the RM. The RM maintains this hierarchy in order to satisfy requests to terminate processes.

Library Routines

Functions provided by service calls are rather primitive, and communication with utility processes can involve complicated protocols. An extensive library of routines has been provided to simplify writing of programs that use service calls and utility processes. For example, general message passing is simplified by "call", which creates a reply link, sends the message, waits for the reply, and decodes the returned message.

Other routines may be used to correspond with the file manager to request directory information or to open, create, or delete a file. For file or terminal I/O, one routine separates strings of text into message-sized chunks, and another accepts text in chunks and assembles it into a buffer. There is also a routine for formatted output.

EXPERIENCE

The first version of the Roscoe kernel was put into service in June 1978. Since that time, additions and enhancements have been added, and many parts of Roscoe have been modified or completely rewritten. Some of our initial decisions have turned out quite well; others have been revised. Some difficult problems that arose during implementation were temporarily resolved by ad hoc solutions, awaiting fuller study.

Links

The most gratifying result was the conceptual simplification due to the link concept. Most obviously, it isolates the programmer from details of communication. Sending a message to another process is the same whether or not the destination is on the same machine, even if the message needs to be relayed by an intermediate processor. Acknowledgement and retransmission, if needed, could also be hidden from the process, although we find that our transmission lines is sufficiently reliable not to need such measures.

The fact that a process sending a message names a link, not a destination process, has several advantages. First, it allows flexibility in allocating resources. A client requesting a service may be given a link to any process that provides the service. The client has no way of knowing to which server it has been connected, only that messages sent over the new link are understood and answered properly. Second, like abstract data types [4], links help to separate the behavior of a facility from its implementation. For example, to open a file, a process sends a message to a file manager process and gets back a link to an open file. To the user process, the file behaves like a process capable of storing or retrieving data. In fact, as currently implemented, the file manager does not create a new process, but only a new link to itself. I/O requests are satisfied by the file manager masquerading as a file process. If we decide to change the implementation, no changes are required in user processes.

A related advantage to hiding the identity of the owner of a link from the holder is

the ability to substitute a process that fulfills the expectations of the holder but does more. One possibility mentioned by Baskett et al [2] is to replace a standard process by one that not only satisfies requests but also monitors them to gather performance statistics. Another version of this idea is used in Roscoe to implement the UNIX "pipe" facility [15]. A "pipe" process is interposed between two processes. The pipe responds to one process as if it were an output file and to the other process as if it were an input file, buffering data between them. Similarly, a terminal driver is capable of behaving like a file, so a process needs no modification to read or write to a terminal rather than a file.

The link concept has also proved useful to solve other problems not so obviously related to communication. One example is synchronization. Since processes do not share address space, synchronization is not necessary to prevent interfering memory references. In situations where synchronization is required, the ability to accept messages only on a subset of all incoming links can be used to advantage. For example, the pipe process has two incoming links, one from the writing process and one from the reading process. When the pipe's buffer is empty, it stops accepting read requests; when it is full, it stops accepting write requests. Otherwise, it accepts whichever request comes first.

Another application of links involves interrupt handling. The terminal input driver performs a service call that establishes an entry point in itself to be invoked when an input interrupt occurs. This service call also names a link to the terminal driver process. When the input buffer is empty, the driver waits for a message along this link. When an interrupt occurs, the interrupt-handling routine puts the input character into a buffer shared by the "higher-level" part of the driver. If an input line is found, it invokes a service call in response to which the kernel manufactures a message and queues it for delivery to the "higher-level" part of the driver, which will eventually receive it as an ordinary process (not at interrupt level).

Bottlenecks

We have discovered that the most vexing aspect of the inexpensive processor we use is not its rather slow speed, but its lack of memory management facilities. This deficiency manifests itself in three ways. First, we have no way of preventing a runaway process form damaging other processes or the kernel on its machine. Second, a process image, once loaded, cannot be moved within a processor or to another processor except at the same address. This restriction has prevented us from experimentation with process migration. Third, the address space of 56K bytes must be shared by the kernel, utility processes and applications programs. Currently, a processor loaded with a kernel, resource manager, terminal driver, command interpreter, and file manager has no room for application programs. Luckily, the Roscoe design mitigates this problem; user programs running on other processors may use the services of such a "service-only" processor.

In attempting to live with our space problems, we have investigated why programs cannot be made smaller. One reason is that the general send/receive paradigm for communication is significantly more complicated than call/return. Library routines such as "call" hide this difficulty from the programmer, but separate copies must be loaded with each process, thus wasting space.

The memory limitations can be mitigated in two ways: One way is software memory management. We have designed a language and implemented a compiler for it that generates code in which all accesses are fully checked and are computed relative to fixed registers. The kernel can use this fact to relocate running programs, performing swapping or migration if necessary. This language also allows us to experiment with high level language features supporting parallelism. A more permanent solution is to use better hardware. Several microcomputers with memory management have recently been announced. We plan to switch to PDP-11/23 machines in the near future.

CURRENT WORK

As we mentioned earlier, several important issues have been sidestepped in an effort to get Roscoe running. We are currently working on solutions to some of these problems. These problems include message routing, resistance to and recovery from processor failure, flow control, distributed file systems, allocation of resources, and migration of processes.

Message Routing

Messages that are sent between processes must be delivered by the cooperation of several kernels. If a message generated on one machine must be directed to another machine, the source kernel must have algorithms for finding the recipient machine. Currently, routing tables are fixed and stored in each kernel. In larger networks, it would be better to arrange the nodes regularly so that routing information can be calculated when needed. Several such schemes for interconnection have been proposed [1, 5, 8]; some are suited to algorithmic routing. Another approach constructs routing tables dynamically [11].

Processor Failure

When a processor fails, the effect should be localized; otherwise the entire network is only as robust as the weakest part. Distribution of control provides advantages in the event of processor failure, since vital decisions about resource allocation can be made elsewhere. A hierarchical control structure would lose an entire subtree of the network when a processor fails. Roscoe currently will survive the loss of a processor only if certain critical processes (such as a file manager) are not lost. If a processor is "cold started", Roscoe is capable of accepting its reintegration into the network. However, the technique is clumsy and requires some manual intervention.

Flow Control

The object of flow control is to allocate the scarce resources of message buffers and use of inter-machine links in such a way that no process is continually blocked waiting for a message buffer, and lines are not left idle when messages are waiting to be sent.

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. Unfortunately, no fixed technique can guarantee that the buffer pool will not be depleted.

Roscoe follows a fairly simple strategy at present. Buffers are requested at one of three priorities, depending on the purpose. A high priority request is satisfied if any buffers are available; a medium priority request only if a fourth of the pool is available, and a low priority request only if a half of the pool is available. High priority is used for messages replying to others, medium priority for most messages, and low for messages arriving from other machines. This strategy avoids buffer depletion in most cases, but it can fail in pathological situations. A possible alternative is to institute per-process buffer quotas.

Files

At the moment, Roscoe files are managed by the PDP-11/40. We also have a floppy disk file driver, but have not yet achieved our goal of distributing a hierarchical file structure among the non-hierarchical machines.

Our plan is to link the independent file manager programs on various machines so that each one controls a local piece of the file hierarchy and knows which colleague controls adjacent pieces. File requests that cannot be handled locally can be forwarded through the hierarchy until they reach a suitable handler, which can then carry on a private communication with the client process.

The problems we expect are that the number of colleagues may be very high, and a disabled processor can disconnect a subtree of the file directory.

Resource Allocation

Nearly all the facilities are in place for the resource manager to do intelligent allocation of resources. Currently, however, the resource manager uses extremely simple-minded algorithms for resource allocation. An example is the allocation of main memory. When a resource manager is requested to load a program, it first attempts to load it locally: It looks for a usable image of the program already loaded, then for unused memory space, and finally for memory currently occupied by an unused program image. Failing to load locally, the resource manager asks a neighbor to load the program. The request is passed around the network in a predetermined circuit until it is satisfied or returns to the originator. If it cannot be satisfied, the original resource manager must report failure. This algorithm has three flaws. First, the predetermined circuit is too inflexible. Second, for larger networks a circuit of the entire network would be impractical. Third, the program may be loaded far from other processes with which it must communicate.

Migration of Processes

As mentioned above, we have been unable to experiment with process migration due to the lack of dynamic address translation. However, even with this problem removed, many others remain. Even though a process names its correspondent by link number rather than process name, its hidden link tables do contain names of processes. If a process can move, the message-routing function of the kernel must be able to discover its new location. One technique is to provide a forwarding address. As processes move, the chains of forwarding addresses are liable to get long, so a "change of address" notice might be sent to the sending kernel. However, in a distributed organization like Roscoe, in which processes are dynamically created and destroyed and in which there is no centralized control, "Flying Dutchman" messages can be created, forever seeking defunct processes, so a message "aging" technique may be necessary.

A more fundamental problem is how to decide when and where to move a process. At some point, the overhead involved in transmitting a program image is exceeded by the inefficiency of a large amount of communication between distant processes. Several heuristics come immediately to mind, but

this issue seems to be one that can only be resolved by experimentation.

CONCLUSIONS

The Roscoe operating system has been running for over a year. A major thrust in its design and development has been to decentralize control as much as possible. Each resource manager has local control over its own machine; the community of resource managers can solve larger problems if necessary.

One application that has been successfully implemented under Roscoe is a game-playing program that uses alpha-beta pruning in a fashion that takes advantage of the available parallelism [Fishburn 79].

The link concept appears to be very successful not only for insulating application programs from the details of communication implementation but also for organizing distributed control.

We have learned how to program with the underlying link structures that Roscoe provides; the link concept has undergone various refinements during the course of this study.

ACKNOWLEDGMENTS

The authors are pleased to acknowledge the assistance of the following graduate students who have been involved in the Roscoe project: Jonathan Dreyer, Jack Fishburn, Michael Horowitz, Will Leland, Paul Pierce, and Ronald Tischler. Their hard work has helped Roscoe to reach its current state and will be essential in continuing this research.

REFERENCES

[1] Arden, B., Lee, H., A Multi-Tree-Structured Network, Princeton University Electrical Engineering and Computer Science Department Technical Report #239, January 1978.

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

[3] Corbató, F. J., Vyssotsky, V. A., "Introduction and overview of the MULTICS system", Proceedings of the AFIPS 1965 Fall Joint Computer Conference, Vol. 27, Part 1. New York, Spartan Books, 1965, pp. 185-196.

[4] Dahl, O. J., Dijkstra, E. W., Hoare, C. A. R., Structured Programming, New York: Academic Press, 1972, pp 83-174.

[5] Despain, A. M., Patterson, D. A., "X-tree: a tree structured multi-processor computer architecture", Proceedings of the Fifth Annual Symposium on Computer Architecture, April, 1978, pp. 144-151.

[6] Fabry, R. S., "Capability-based addressing", Communications of the ACM Vol 17, No. 7, July 1974, pp. 403-412.

[7] Feldman, J. A., "High level programming for distributed conputing", Communication of the ACM Vol 22, No. 6, June 1979, pp. 353-368.

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

[9] Finkel, R. A., Solomon, M. H., The Roscoe Kernel, University of Wisconsin--Madison Computer Sciences Department Technical Report #337, September, 1978.

[10] Finkel, R. A., Solomon, M. H., Tischler, R., Roscoe User Guide, Version 1.1, University of Wisconsin--Madison Mathematics Research Center Technical Report #1930, March 1979.

[11] Finkel, R. A., Horowitz, M., Solomon, M., "Distributed Algorithms for Global Structuring", Proceedings of the 1979 NCC, June 1979.

[12] Fishburn, J. and Finkel, R. A. "A Parallel Implementation of Alpha-beta Pruning", To appear.

[13] Huen, W., Greene, P., Hochsprung, R., and El-Dessouki, D., "TECHNEC, a network computer for distributed task control", Proceedings of the First Rocky Mountain Symposium on Microcomputers, August, 1977.

[14] Kernighan, B. W., Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.

[15] Mills, D., "An Overview of the Distributed Computer Network", Proceedings of the National Computer Conference, Vol. 45, AFIPS Press, 1975, pp. 523-531.

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

[17] Solomon, M. H., Finkel, R. A., "ROSCOE: a multi-microcomputer operating system", Proceedings of the Second Rocky Mountain Symposium on

Microcomputers, August 1978, pp. 291-310.

[18] Tischler, R. L., Finkel, R. A., Solomon, M. H., Roscoe Utility Processes, University of Wisconsin--Madison Computer Sciences Department Technical Report #338, February 1979.

[19] Wittie, L. D., Micronet: A reconfigurable microcomputer network for distributed systems research, State University of New York at Buffalo Department of Computer Science Technical Report 143, April, 1978.