INTEL/311848-001

(1)

DTIC FILE COPY

Touchstone Project Mach Port Special Technical Report

AD-A224 057

Milestone Event Q4

Don Cameron
Joel Clark

Intel Corporation

December 28, 1989

Defense Advanced Research Projects Agency
Information Science and Technology Office

~~Limited Rights Legend~~

Contract No. MDA972-89-C-0034

Contractor: Intel Corporation

Case #90-2828

~~Limited rights are not subject to an expiration date.~~

~~The restrictions governing the use and disclosure of technical data marked with this legend are set forth in the definition of "limited rights" in paragraph (a)(15) of the clause at 252.277-7013 of the contract listed above.~~

~~This legend, together with the indications of the portions of this data which are subject to limited rights, shall be included on any reproduction hereof which includes any part of the portions subject to such limitations. This technical data will remain subject to limited rights only so long as it remains "unpublished" as defined in paragraph (a) above.~~

90 07 16 ~~249~~ 002

INTEL/311848-001

Mach Port Special Technical Report

Milestone Event Q4

Prepared By:

Don Cameron
Joel Clark

Intel Corporation
Intel Scientific Computers
15201 NW Greenbrier Parkway
Beaverton, OR 97006

December 28, 1989

Sponsored by

Defense Advanced Research Projects Agency
Information Science and Technology Office
Research in Concurrent Computer Systems
ARPA Order No. 6402, 6402-1; Program Code No. 8E20 & 9E20
Issued by DARPA/CMO under Contract #MDA972-89-C-0034

Abstract

The purpose of this ~~technical~~ report is to describe the porting of the Mach operating system to the Touchstone IOTA prototype. The report describes the major activities associated with the port and the kernel design issues that they brought to our attention.

Mach was ported to a small cube of integrated I/O nodes. Drivers for the SCSI disk drive, serial line, and the Direct-Connect hypercube routing hardware were written. Serial communication to the nodes was multiplexed through a terminal connected to an Intel SYP-301 host platform.

We found that to extend the Mach thread model onto a multicomputer architecture such as Touchstone, a compatible form of "shared virtual memory" must be provided. Alternately, supporting the multicomputing model of high-speed inter-node message passing will require better support from the Mach kernel. For both of these usage models, the speed of inter-node message passing is critical. Further research is required into how to best provide this in the Mach environment. (KP)

Contents

Figures

1. Summary

This report describes the porting of the Mach operating system to the Touchstone IOTA prototype. Our objectives were to implement the Mach operating system on a small IOTA prototype and to pass messages between at least two nodes via the Direct-Connect hypercube routing modules.

To achieve these goals, the porting activities included the development of mechanisms for bootstrapping, interfacing to the node processor's memory management unit, and developing several device drivers. The later task, device drivers, required the largest single effort since drivers were written for the node's SCSI disk controller, Com/TTY function, and Direct-Connect Module (routing hardware). A small IOTA system was used to debug the port consisting of two integrated I/O nodes, two 760 MB Winchester disks, a System Resource Manager (SRM) for program loading, and a second SRM for holding the Mach source code and building Mach binaries.

Mach was successfully ported to the IOTA prototype and messages were passed between two I/O nodes; this will now serve as a base from which we can compare and contrast Mach with NX/2 and the Reactive Kernel.

Overall, this development process has indicated that to extend the Mach thread model onto multicomputer architectures a compatible form of "shared virtual memory" must be provided. Alternately, supporting the multicomputing model of high-speed inter-node message passing will require better support from the Mach kernel.

For both of these usage models the speed of inter-node message passing is critical. Further research is required how to best provide this in the Mach environment.

2. Introduction

Intel Scientific Computers, in cooperation with Carnegie Mellon University (CMU), and Intel's Software Technology Operation, has completed an initial porting of CMU's Mach operating system to the Touchstone IOTA prototype. This report describes the results of this project and key issues raised in its porting to the IOTA prototype.

This report is organized as follows. The first section reviews the goals of the project and provides an overview of the Mach operating system. The next section covers the implementation details for Mach on the IOTA prototype. The following section discusses various design and implementation issues that arose during the course of the project. Finally, the last two sections conclude the report and offer recommendations for further study.

2.1 Project Goals

The goals of the port of Mach to the Touchstone IOTA prototype were to:

- Implement the Mach operating system on a small IOTA prototype containing integrated I/O nodes.
- Pass messages between the nodes via Direct-Connect Module (DCM) routing hardware.

2.2 Overview of Mach

The Mach operating system was developed at CMU to address the increased complexity of the UNIX kernel as it was stretched to handle characteristics of modern host hardware systems—such as multiprocessors, memory management units, and networking—that were absent when UNIX was first developed[1].

Mach features that support these new characteristics are:

- multiple threads of control.
- machine independent virtual memory management.
- interprocess communication that is transparently extensible across a network.

In addition, the Mach kernel is intended to be small but flexible enough to allow a variety of operating systems interfaces to be implemented on top of the kernel. Mach is binary compatible with 4.3 BSD UNIX[2].

2.2.1 Tasks and Threads

In Mach, the traditional process model of UNIX is replaced by tasks and threads. A UNIX process is represented by a task with a single thread of control. Tasks are the basic unit of resource allocation; e.g., virtual address space and port rights. Threads are the basic unit of scheduling and execution. In essence, a thread is a program counter and a set of registers. Threads belong to exactly one task. A task consists of one or more threads.

Threads can run in parallel. In a system with shared memory and multiple processors, more than one thread may execute at a time; either from a single task or from multiple tasks.

Tasks are related to one another in a tree structure determined by task creation operations. The virtual memory management interface allows regions of memory to be inherited in a number of ways.

2.2.2 Virtual Memory Management

Mach virtual memory management is architecture independent. This is a significant advantage when porting the kernel because only a small machine dependent module needs to be written[3].

More importantly, the architecture independent memory model allows the advanced features of Mach's virtual memory management to be available to users writing application programs.

The advanced features of Mach virtual memory management are:

- flexible inheritance of memory from parent tasks: copy-on-write, shared, or not-inherited.
- shared libraries.
- memory mapped files.
- the ability to specify an associated paging server when allocating regions of memory.
- the ability to create paging servers outside of the kernel.

These last two features are important in a distributed environment because they should allow a mechanism such as shared virtual memory to be implemented as a simple extension to Mach[4].

2.2.3 Interprocess Communications

Within a network node, Mach provides a secure Inter-Process Communication (IPC) facility based on ports. Ports are kernel objects on which messages can be queued and dequeued. A task can access a port only if it has rights to do so[5].

The rights associated with a port are:

Receive	Only one task at a time may hold receive rights to a port. A task with this right can dequeue messages from a port.
Ownership	Only one task at a time may hold ownership rights to a port. A task with this right will acquire receive rights to the port in the case where the receiver terminates.
Send	Many tasks may simultaneously hold send rights to a port. Send rights allow a task to queue messages on a port.

A task gains access rights to a port only from the kernel or from another task that already has access.

The Mach kernel, by itself, does not contain a mechanism for passing messages over a network. However, hooks are provided for support of a user level task known as a network server which can be used to transparently extend IPC across a network. At one time, Mach also contained a mechanism known as *netports* which for efficiency placed network servers in the kernel; this facility is no longer actively supported by CMU.

3. Methods, Assumptions, and Procedures

This section describes the strategy used to port Mach to the IOTA prototype and the scope of this initial implementation. It also describes the design of key elements of the port.

3.1 Implementation Strategy and Scope

The version of Mach ported to the IOTA prototype is internal release X95 from July, 1989. This release is in between external release 2.0 and the current release of 2.5. While source code more recent than X95 became available as the project proceeded, it was safer to continue with a single stable release.

Ultimately, we want to have the "pure kernel" version of Mach on the DELTA prototype. This version, known as Mach 3.0, will implement the UNIX interface outside of the kernel, leaving a small and flexible kernel with a well defined and generic set of services. This version of the kernel was not available in time for the initial port to IOTA.

Mach requires a file system; this is both to provide virtual memory swap space and to allow the normal UNIX program loading model. For this reason Mach was ported to the IOTA prototype integrated node. The integrated node contains a SCSI interface that can be connected to disk and/or tape drives. We require a disk to be present on each integrated node. We did not implement NFS or other mechanisms to support diskless nodes. Similarly, no support is provided for tape drives.

An autonomous copy of the Mach operating system is in place on each node. In many ways this resembles a network of workstations running Mach as would be typical in a more loosely connected environment such as a L\N. We did not make any changes to Mach to make it more of a distributed operating system.

3.2 IOTA Prototype System

Mach was ported to an IOTA I/O node with an eight-channel Direct-Connect Module and not the normal single-channel module used in non-integrated I/O nodes. This approach enables one I/O node to communicate with the other and the System Resource Manager (SRM). The latter is used to down-load programs into the nodes.

In addition, these I/O nodes contained a SCSI interface to Winchester disks because each copy of Mach requires a disk drive for its local file system and virtual memory swap space.

The entire prototype system consisted of two IOTA I/O nodes connected to two 760 MB Winchester disks all driven by an Intel SYP-301 system resource manager (386-based workstation). A second SYP-301 was used to hold the Mach source code and to perform software builds. Mach binaries were then written onto cartridge tape and transferred from one SYP-301 to the other. Finally, a *light pen* and terminal were used to decode the optical messages generated by the firmware and emitted by the LED's on the I/O node board.

3.3 Design

The work necessary to port Mach to the IOTA prototype can be broken in to the following areas:

- bootstrapping.
- interface to the memory management unit.
- device drivers.

Key design issues in each of these areas is discussed below.

3.3.1 Bootstrapping

Typically, Mach is booted from disk in several steps. First a PROM boot monitor loads a bootstrap loader into memory from the boot device. This bootstrap loader reads the Mach executable file into memory and starts execution.

For Mach on the IOTA prototype, we chose to implement the method used by NX/2. [6]

1. A bootloader is downloaded onto each node from the SRM via the unit services module (USM) diagnostic line.
2. The bootloader initializes the DCM channels to the topology specified in a configuration file.
3. The OS image is read onto node 0 (the node connected to the SRM) from the DCM channel which then broadcasts it to each of the other nodes.
4. The bootloader transfers control to the OS image.

Bootstrapping in this manner was helpful for several reasons: first, it allowed us to load and begin to debug Mach before we had a reliable disk driver; second, it made it easier during the development cycle to test new versions of the kernel without having to build a new file system image for each of the disks.

Mach is significantly larger than NX/2 and the kernel virtual address space is in high memory (see Figure 1). This made it necessary to change several of the NX/2 tools. These were:

bootld This is the bootloader that runs on the node. Because the i386™ microprocessor comes up in real-mode, this program needs to be located in the first megabyte of physical memory; NX/2 is then loaded below *bootld.* Since Mach is larger, there was not enough room to load the text segment unless *bootld* was modified so that it was located at a higher address. We also changed *bootld* to load the data and other segments above 1 megabyte.

Finally, we modified *bootld* to build a i386 task state segment entry for Mach, and to transfer control to Mach's new starting address.

mkimage This is a program run on the SRM that takes a.out format files and creates an image file suitable to be loaded by *bootcube*. For NX, *mkimage* creates a single image where the byte position in the image file corresponds to the addresses specified in the a.out file. Because Mach kernel virtual addresses do not correspond to the physical addresses where Mach will be loaded, we had to subtract an offset when creating the image file.

As shown in Figure 1, the loadable portion of Mach is not contiguous, and if we attempted to load it as a single image it would overwrite *bootld.* We therefore changed *mkimage* to create two image files: one with the text segment, and another with the data and other segments.

bootcube This SRM program sends the bootloader to the nodes across the USM line and then sends the OS image over the DCM channel. For Mach, this program needed to be modified to send the additional image file mentioned above.

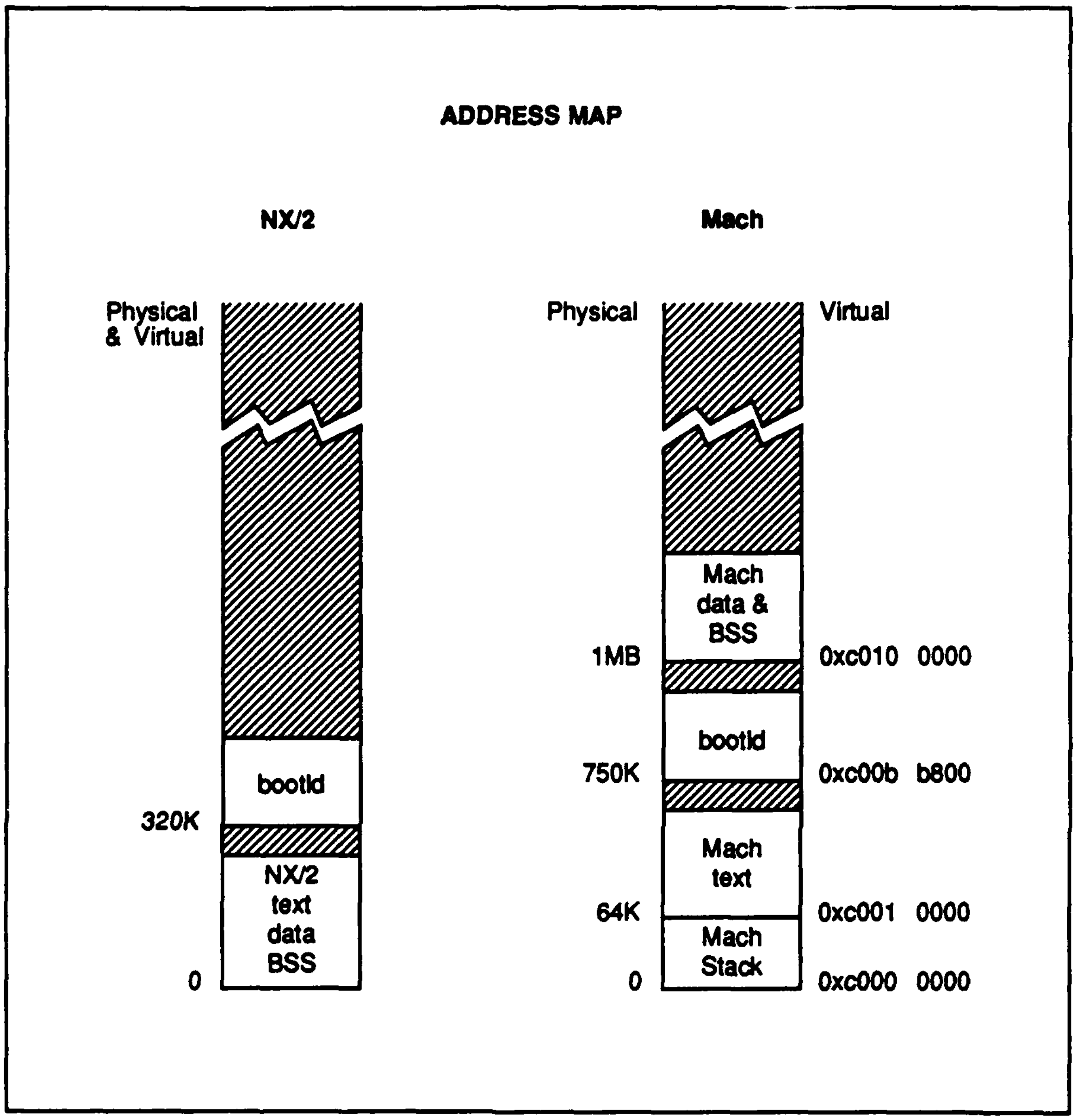

Figure 1
Mach Kernel Virtual Address Space

3.3.2 Memory Management Interface

A key part of porting Mach to a new architecture is providing the architecture specific interface to the memory management unit. Mach had already been ported to the i386 microprocessor by the consortium of Intel, Prime, Olivetti, and AT&T; and we were able to use their code without modification.

Mach allows logical page sizes to be any power of 2 of the physical page size. We chose 4-Kbytes as the virtual memory logical page size. This corresponds to the i386's physical page size and to the largest block that could be transferred from disk in a single operation.

3.3.3 Device Drivers

A major part of the porting effort was developing the device drivers. These are described below.

SCSI disk driver

The NX/2-CFS SCSI disk driver was modified to conform to the BSD UNIX style driver; this was a significant effort. Because transferring data blocks larger than 4-Kbytes can not be done in a single operation on the IOTA prototype SCSI interface, we chose this value as the file system block size. Normally, file system accesses that exceed a single block are broken up by the file system routines in to a series of smaller requests. When Mach is initializing, however, a request to read an 8-Kbyte block is made; the driver was modified to handle this case.

Com/TTY driver

Mach requires access to a console. Lacking any direct support on an I/O node for such a connection, we chose to provide this service the USM diagnostic line. Because the USM line is connected to each node in the cube—*kt*, an NX/2 program on the SRM, implements a protocol that allows the USM line to be shared by many nodes. A node views the USM as being in one of three states:

1. IOSELECT - the node is free to read and write the USM line.
2. GSELECT - global selection, the node can read but not write the USM line.
3. NOSELECT - the node can not read or write the USM line.

The nodes are connected to the USM via an asynchronous serial controller (UART). We were able to take a Mach driver for the UART and modify it to recognize the *kt* protocol. The driver contains entry points for both polled IO (used by *kdb* the kernel debugger) and interrupt driven IO.

The polled output routine buffers characters sent to it when it is not IOSELECT'ed, and dumps them when it is next selected. The interrupt-driven output routine uses the UNIX character buffering scheme.

Another change to the UART driver was needed to handle flow control. Normally, a UART driver knows when the next character can be sent by examining a bit in the line status register that indicates a transmission is complete. Polled IO routines poll this register, while interrupt driven routines interrupt on the transition of this bit. The design of the USM precludes using the line status register to indicate that transmission is complete. Instead, *kt* sends a special character to indicate that a character has been received. The Com/TTY driver waits for receipt of this character before sending the next character.

DCM driver

The driver for the IOTA prototype DCM hardware is implemented as a UNIX character device. Since the read and write driver entry points do not contain parameters that will allow us to indicate the node to write to or the node read from, the *ioctl* entry point is used to perform input and output. The device specific structure in the *ioctl* call contains fields for the address of a buffer, the length of the buffer, and a node number; the ioctl command parameter indicates whether to do a read or write.

It is the responsibility of the DCM driver to perform flow control; i.e., to regulate the rate at which messages are sent so that the receiving node is not overrun. On a network such as Ethernet a receiving node may simply ignore the network when it no longer has the resources to accept more messages. In contrast, if a node using the IOTA prototype DCM hardware stops reading from a channel it blocks the sending node and blocks other nodes from utilizing the portion of the network in use. Thus, nodes using the DCM hardware should only send a message when it is known that the receiving node is in a state to accept it.

The DCM driver for Mach implements a modified version of the protocol used for NX/2. There is a queue of message buffers on each node used for receiving messages. The queue is logically partitioned so that a set number of buffers "belong" to each node in the cube. Nodes keep a table that indicates how many buffers they have available at each other node. This count is decremented when a message is sent to a node. When the count for a particular node reaches zero, no further messages can be sent to that node until an indication is received that a buffer on the receiving node is once again available (i.e., the contents of the buffer have been copied to user memory).

This indication is put in the *give* field of the message header. The *give* field is a one byte integer that indicates the number of buffers that have been made available; this field is in the header of every message sent and is set to zero when not used. In the best case, the *give* information can be sent in the normal flow of messages between nodes. It is possible, however, to reach a situation where a node's count of buffers available on the receiving node reaches zero, while the receiving node has buffers available but has not had the opportunity to send a message with the *gives*. To handle this situation the DCM driver notices when the number of pending *gives* is within a threshold of the number of buffers per node and sends a control message to carry the necessary *give* information.

The size of the buffers, the *give* threshold, and the number of buffers reserved for each node are all configurable. For this first implementation, we are using a buffer size of 4-Kbytes, a *give* threshold of zero, and eight buffers per node.

4. Results and Discussion

This section presents the results of the research and interprets the significance of each result.

4.1 Com/TTY driver

A console input/output device is required by Mach. The console is used to display warnings and errors, it is used by the kernel debugger, and (unless some other device is provided) it is where the user logs in and executes programs.

We chose to multiplex the USM serial line between nodes (see discussion of com/TTY driver in previous section). Using *kt* the user can have a terminal on the SRM serve as the console device for one node at a time on the IOTA prototype.

We expected this scenario to be awkward for large numbers of nodes. In practice, it was burdensome even when using just two nodes. A better method for console communication needs to be found. For example, polling each node in turn to collect error and warning messages.

4.2 DCM Driver

Because Mach provides virtual memory, message data being sent between nodes cannot be assumed to be in physical memory; in fact, the message may be larger than available physical memory. This prompted several differences between the message passing protocols for Mach and NX/2.

1. For Mach we did not implement the NX/2 long-message protocol. This is because there is no guarantee that a buffer of sufficient size will ever be made available on the receiving node. This also avoids having to lock into physical memory a large number of pages.
2. The small-message packet size has been dramatically increased from 100 bytes to 4-Kbytes which is the virtual memory page size. We use a larger packet because a page is the minimum amount of memory that may be locked at a time.

   Note that in a system with a large number of nodes the packet size may need to be reduced to avoid reserving excessive amounts of physical memory for receive buffers.

3. User message data longer than the message packet size must be broken up into a series of transfers. This is done by the *ioctl* routine.

On the sending node, message data is transferred directly from user buffers to the DCM. On the receiving node, message data is moved from the DCM into driver buffers and then copied into user space when the application calls the driver to perform a receive.

4.3 Mach and Distributed Memory

Mach offers two ways of structuring programs for parallelism, multiple tasks and multiple threads within a task; and two ways to coordinate these parallel activities, message passing and shared memory. These correspond, roughly, to the multi-computer (Intel iPSC/2 and NCUBE 2) and multiprocessor (Encore and Sequent) models of parallelism.

4.3.1 Multicomputing Model

Operating systems for multicomputers are geared towards high-speed inter-node message passing. While we do not yet have measurements of Mach message passing performance between nodes, analysis indicates that it will be significantly slower than that provided by NX/2. This is because of context switches, data copies, and other overhead associated with the user level process network message server. At one time Mach provided a facility known as "netports" which placed support for inter-node message passing in the kernel. This facility is no longer supported by CMU.

4.3.2 Multiprocessor Model

The Mach kernel does not allow tasks to span physical memory boundaries. Nor does the kernel itself provide a way for tasks to share access to virtual memory across physical memory boundaries. To allow applications written for shared memory multiprocessors to work on distributed memory multicomputers these limitations must be lifted.

A technique known as "shared virtual memory" [4] can be used on distributed memory machines to allow nodes to share access to virtual memory objects. The Mach virtual memory implementation allows this technique to be supported outside of the kernel. Shared virtual memory would support load balancing and task propagation.[8]

The support of shared virtual memory is necessary to allow tasks to span physical memory boundaries, but it is not sufficient; work by Joseph Barrera at CMU is under way to address this issue within the Mach context.

For both of the above usage models the speed of inter-node message passing is critical. Further research is required how to best provide this in the Mach environment.

5. Conclusions

The goals of the activities reported in this document were to implement the Mach operating system on a small IOTA prototype system and to successfully pass messages between two nodes via the Direct-Connect routing modules. Our major results are as follows:

- We identified and completed the tasks necessary to port Mach to the IOTA prototype including: bootstrapping, an interface to the memory management unit, and the device drivers.
- Bootstrapping was accomplished using the techniques developed for NX/2 that involve the RS-422 serial line and Direct-Connect routing modules.
- The virtual memory logical page size of 4 Kbytes was selected because it corresponds to the i386 physical page size and to the largest disk block that can be transferred in a single operation.
- Three device drivers were written: SCSI disk driver, Com/TTY driver, and DCM driver. These tasks represented a major part of the porting effort.
- Finally, a program was written and executed which successfully passes messages between two nodes,

Mach was successfully ported to the IOTA prototype; this will now serve as a base from which we can compare and contrast Mach with NX/2 and RK.

We found that in order to extend the Mach thread model onto multicomputer architectures shared virtual memory must be provided. Alternately, supporting the multicomputing model of high-speed inter-node message passing will require better support from the Mach kernel. For both of these usage models the speed of inter-node message passing is critical.

6. Recommendations

As we continue our research, several of the activities reported in this document may benefit from additional study. This section discusses several of these possible extensions to the Mach port.

6.1 Future work

There are several areas that would be worth pursuing in evaluating Mach on the IOTA and DELTA prototypes:

- Implementation of a network message server transport module for the DCM.
- Implementing shared virtual memory.
- Examining ways to improve Mach message passing performance between nodes.
- Eliminating the requirement of a local disk.
- Finding a better way to display console messages.
- Providing a pseudo-terminal facility through the DCM or Ethernet.

In the future, we would like to work cooperatively with CMU to investigate improvements to Mach in support of distributed memory, message passing, multicomputers. It would be very useful to expand Barrera's efforts in shared virtual memory to include Touchstone architectures.

6.2 Changes and Extensions to Mach

Early releases of Mach provided a mechanism known as *netports* which for efficiency placed network servers in the kernel; this facility is no longer actively supported by CMU.

As the speed of inter-node message passing is critical in multicomputer architectures such as the IOTA and DELTA prototypes, the support and/or structure of the netport mechanism should be re-examined.

7. Acknowledgments

We gratefully acknowledge the support and cooperation of the Computer Science Department at Carnegie Mellon University and, in particular, of Robert Baron. We also thank Intel's Software Technology Operation for providing Mach sources and a version of Mach that ran on the Intel SYP-301 platform.

8. References

1. Tevanian and Rashid, *Mach: A Basis for Future UNIX Development*, CMU-CS-87-139, June, 1987.
2. UNIX is a registered trademark of AT&T.
3. Tevanian, *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach*, CMU-CS-88-106, December, 1987.
4. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, YALEU/DCS/RR--492, September, 1986.
5. Sansom, Julin, and Rashid, *Extending a Capability Based System into a Network Environment*, 1986 ACM 0-89791-201-2/86/0800-0256.
6. Pierce, P., *The NX/2 Operating System*, Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications, ACM, New York, 1988.
7. Seizovic, J, *The Reactive Kernel*, Caltech CS-TR-88-10, 1988.
8. Forin, Barrera, Young, and Rashid, *Design, Implementation, and Performance Evaluation of a Distributed Memory Server for Mach*, CMU-CS-88-165, August, 1988.

9. Distribution

Delivery of this report has been made to the following:

DARPA/ISTO
Attn: Stephen L. Squires
1400 Wilson Boulevard
Arlington, VA 22209-2308
(one copy)

DARPA/RMO/Retrieval Services
1400 Wilson Boulevard
Arlington, VA 22209-2308
(one copy)

Defense Technical Information Services
Building 5, Cameron Station
Attn: Selections
Alexandria, VA 22304
(two copies)

~~FURTHER DISSEMINATION ONLY AS DIRECTED BY DARPA/ISTO, 31 JANUARY 1989, OR HIGHER DoD AUTHORITY.~~